ProtoGen-MLIR V2: An Optimizing Compiler for Cache Coherence Protocols

Petr Vesely

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2022

Abstract

This report presents ProtoGen-MLIR v2, an optimizing compiler for cache coherence protocols using MLIR compiler infrastructure. The compiler provides the user with a Domain Specific Language to specify a cache coherence protocol with atomic transactions and generates a optimized concurrent version of the protocol without this requirement. This project builds upon the previous version presented in Part 1 of this project, and implements new features that increase the overall optimization capability.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Petr Vesely*)

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. Additionally, I would like to thank Nicolai Oswald for sharing his detailed knowledge of ProtoGen to allow me to complete this project. Finally I would like to thank my family and my friends for supporting and motivating me to complete this project.

Table of Contents

Chapter 1

Introduction

Designing custom silicon is hard. Hardware, unlike software, cannot be 'fixed' or 'patched' after it is produced. This results in a huge verification effort to ensure hardware is verified on multiple fronts, including correctness, safety and security [12]. Due to the strictness in these requirements, it has been observed that verification of hardware can often become the bottleneck in chip design [8]. Moreover, chips will inevitably become more complex, which will likely exaggerate this problem further. Hardware is verified in many ways, including simulation, testing on pre-production chips and even some formal verification techniques are used [12]. Yet, bugs and defects in hardware still occur regularly and sometimes with grave consequences. For example, issues with the implementation of Intel's speculative execution led to the Spectre and Meltdown vulnerabilities [9]. The 'fix' for the vulnerabilities was either to completely disable Hyper-Threading or replace the CPU entirely, resulting in significantly degraded performance or high cost respectively [4].

What is concerning, is that over the past two decades Dennard Scaling (the inability for CPUs to continue increasing clock speeds due to power and thermal limits) and the decline of Moore's law in recent years has resulted in a move to ever more parallel and heterogeneous architectures [19]. This means that specific problems will likely require custom silicon or accelerators to achieve their desired performance. Moreover, custom silicon for domain specific purposes will likely have very unique computing requirements and might not be able to use the general purpose industry tooling available. This could make it prohibitively expensive for small to medium sized companies to be able to develop their own silicon chips and have to rely on off-the-shelf designs, which may not yield the best performance possible.

The solution may however lie with compilers. A compiler is in effect a program which transforms an input of one form to an output of another form. But this ability has proven to be very powerful, not only in the classical sense of compiling static languages to machine code, but also in other more specialised domains, including hardware [11]. Compilers have steadily moved forward in their capability over the past several decades. In fact compiler technology has proved very effective in solving many challenges as the community shift to parallel and heterogeneous systems. SYCL is a ISO C++ language standard that allows a programmer to write code for heterogeneous architectures in

a single source file using standard C++ [3]. Previous attempts at this like OpenCL required the programmer to explicitly write accelerated code in separate source file and orchestrate their interactions using the required DSL. Another exciting development is the Circt project [11], which uses MLIR for hardware synthesis, and implements many industry standard backend targets such as Verilog and RTTL. This could prove essential in reducing the development effort to engineer custom silicon.

This project attempts to use compiler development techniques to engineer a compiler for cache coherence protocols. Cache coherence protocols are an essential component of all multi-core shared memory multiprocessors. On the surface, a cache coherence protocol can seem deceptively simple, their function can be described in a few states and transitions, and their correctness and be easily reasoned about. However, such specifications usually assume atomic transactions, where the caches and memory behave atomically. However, most hardware implementations relax this constraint for performance and thus require many additional transient states and transitions to handle all possible race conditions that can occur. As a result an MSI protocol with only three stable states, results in a protocol with eighteen total states (including transient states) when the atomic transaction requirement is relaxed [14].

1.1 Previous Work

1.1.1 Circt

Circt is an existing open-source project to address the lack of open source tooling for electronics design, including processor architecture [11]. It's popularity has grown steadily in recent years and has moved to be part of the LLVM umbrella project, which saw many new contributions from the industry, including from Chris Lattner (the original creator of LLVM). Circt itself exposes a diverse, modular and reusable library of intermediate representations for hardware synthesis. Then, through it's many sophisticated optimization pipelines, it can generate industry standard outputs like Verilog or RTTL [11]. The key to it's potential is that verification efforts can be reduced, since the compiled outputs are *correct-by-construction* as a consequence. The project uses MLIR's compiler infrastructure for its internal representations as well as its optimization pipelines. The success shown by this project was a big motivating factor in attempting to use MLIR to model cache coherence protocols.

1.1.2 ProtoGen

ProtoGen is an existing tool developed by Nicolai Oswald at the University of Edinburgh [16]. It's purpose is to generate highly concurrent and optimized protocols with non-atomic transactions from an atomic protocol specification. To achieve this ProtoGen comes with a Domain Specific Language (DSL) named PCC, which allows the user to specify a protocol's function in code. ProtoGen then uses this protocol description, and applies its sophisticated optimizations which allows the transaction atomicity constraint to be relaxed. ProtoGen can relax this constraint by discovering when race conditions occur in the protocol and adding additional transient states and transitions to handle

them. ProtoGen can also generate MurΦ (pronounced Murphy) from the generated protocol, which is a verification tool that allows the protocol to be verified formally for deadlocks and other constraints. This project is heavily influenced by this previous work, in fact we borrow the DSL and reason about the optimizations in a similar way in ProtoGen-MLIR, although we present an entirely novel architecture leveraging compiling techniques.

1.1.3 ProtoGen-MLIR

ProtoGen-MLIR was presented in part 1 of this project and from now on we will refer to it as ProtoGen-MLIR v1 to make the distinction between ProtoGen-MLIR v2 presented in this report. ProtoGen-MLIR v1 was a full compiler implementation which successfully proved that MLIR and compiling techniques in general could be used to optimize cache coherence protocols [20]. V1 presented a full frontend implementation for the PCC DSL into a custom IR (declared using MLIR), which parsed in the input PCC protocol description and generated the MLIR operations as input into the optimization pipeline. V1 also included an optimization tool using MLIR pass infrastructure which transformed the input IR by applying a small subset of the optimizations from ProtoGen. Finally, it also included a full backend implementation which targeted MurΦ to verify the generated protocols for correctness (similarly to ProtoGen).

ProtoGen-MLIR is fundamentally a re-implementation of the ProtoGen algorithm. However, ProtoGen implements its own internal representation as in-memory data structures, which limit its development and interactions with other tools. ProtoGen-MLIR instead focuses on bringing well established compiler development techniques, to develop a well specified text-based internal representation. This effectively provides a generic interface and allows for development of specific tools, which can interact together using the IR. This improves the modularity and can allow for pipelined execution of the compiler.

1.1.4 Teapot

Teapot is a domain specific language and compiler for specifying the behaviour of cache coherence protocols [5]. 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 includes backend targets for C and MurΦ, which made it appealing to study for parallels to ProtoGen and ProtoGen-MLIR. However, we found that its use of functional programming did not fit into the model required for ProtoGen-MLIR. In order to perform automatic optimization of protocols, we require additional information about the execution of the protocol to discover race conditions, which proved difficult using their model. Moreover, PCC uses its own `await` syntax which provides a similar

functionality to continuations, and we felt that there was no advantage in pursuing this model.

1.2 Aims

The aim of this project was to build upon the previous work of ProtoGen-MLIR v1 and to develop an MLIR compiler for cache coherence protocols. We aim to address the main challenges which plagued its original design and to extend the optimization capability towards ProtoGen and support additional language features and protocols.

ProtoGen-MLIR v1 proved altogether unmaintainable for several reasons. Firstly, the IR presented was basic and didn't correctly encode the operation of the protocol. This meant that the optimizations presented were limited in their capability and ad-hoc techniques were used to manipulate the IR, which became difficult to implement and reason about. In this project we aim to primarily address this main shortcoming, by developing a more suitable IR, which correctly expresses the Finite State Machine (FSM) nature of the cache and directory controllers.

From the challenges of implementing optimizations in ProtoGen-MLIR v1 we also aim to develop a declarative specification of how optimizations are to be performed. This specification is presented as a set of declarative steps, which can be reasoned about and then guide the implementation. This means that we have a strong logical and reasoned foundation that applies to all protocols, meaning we do not have to revert to ad-hoc techniques.

Lastly we aim to develop a modular and maintainable compiler backend implementation. ProtoGen-MLIR v1 suffered from significant coupling between the IR and the MurΦ CodeGen implementation, meaning that to support the new IR, this component would have to be almost completely rewritten. As part of this report we also aim to develop a generic implementation of this component, which will work for any custom IR developed in the future, and will make future contributions much simpler and straightforward.

1.3 Contributions

This report presents ProtoGen-MLIR v2, which is a greatly improved (re)implementation of the original ProtoGen-MLIR presented in Part 1 of this project.

Key Improvements:

- Improved overall optimization potential over v1, especially with the addition of non-stalling optimizations.
- A new Intermediate Representation (IR) which abstracts a coherence protocol as a Finite State Machine (FSM). The representation also includes a strong type system and improved diagnostics.

- A detailed declarative specification for each class of coherence optimization which is based on the sound reasoning of coherence protocols from ProtoGen.
- A generic MurΦ CodeGen backend component, designed for maintainability and compatibility with any MLIR representation.
- Extended support for PCC language features, which allow us to express more complex protocols than v1.
- Improved testing and verification of IR & optimization pipelines, which provides confidence in the overall implementation of the compiler.

Chapter 2

Background

2.1 Cache Coherence

Most modern CPUs can best be described as shared-memory multiprocessors, which consist of a number of discrete processor cores, each of which has direct access to the full address space. This has the consequence, that any of the processors on the chip can read and write data to addresses that may be being accessed by another processor. This is sometimes also called a Uniform Memory Access (UMA) architecture, and enables communication and synchronization between processes through the shared-memory interface [18].

To complicate matters, such systems rely heavily on caching, which is a technique to store frequently accessed memory addresses on a fast-access chip that is physically close to the processor. Caching is crucial for performance as main memory is prohibitively slow in comparison to the processor and access to memory locations are not randomly distributed. A typical program execution will issue reads and writes to the memory system which exhibit certain patterns, namely *temporal locality* (where a recently accessed address is likely to accessed again) and *spacial locality* (addresses close to the previously accessed address are likely to be accessed) [18]. Thus, by using a cache to store blocks of recently accessed addresses we can expect a large percentage of requests for an address can be fulfilled by accessing the cache instead of main memory. For comparison, an access to the cache can be completed in a handful of CPU cycles, while access to main memory will be more than a 100 cycles [13]. Thus having an effective caching system is critical for performance. Other caching performance techniques such as replacement policies or cache associativity are not considered as part of this report.

In Figure 2.1 we present the baseline system model used throughout this report. We show that each *core* issues its read and write requests to a *cache controller*. Each cache controller is then connected to a single *private data cache* which is only accessible by the controller. Access to main memory is controlled by the *LLC/Memory controller* (throughout this report we usually refer to this component as the *directory*) and it is connected to the LLC which is shared by all cores. Each controller is then connected together through an *Interconnection Network* (Interconnect for short) which allows messages to be sent and received by different controllers. Note that we do not specify

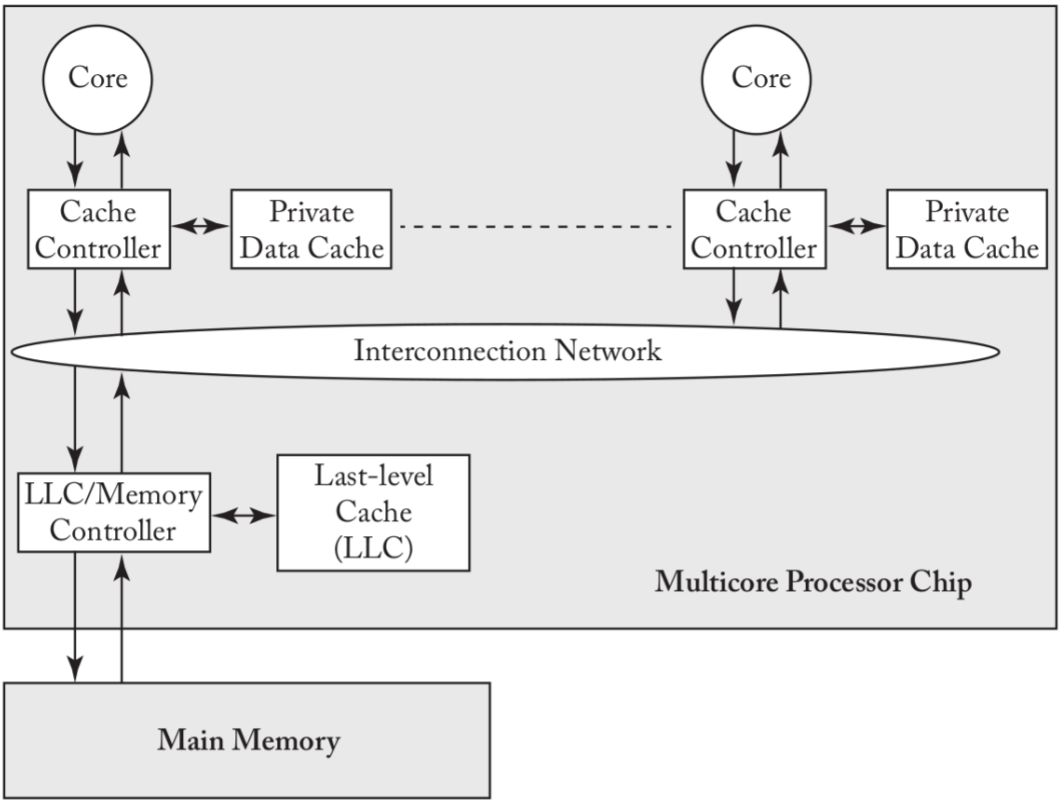

Figure 2.1: Baseline system model used throughout this report [14]

any specific hardware implementation for any of these components. This is a deliberate choice as reasoning about cache coherence protocols can be done at high-level without considering hardware directly. However, some hardware elements can be specified using PCC, such as the structure of messages and if the interconnect enforces ordering.

With the baseline architecture from Figure 2.1 we can quickly show how *cache incoherence* could arise. Suppose that a core *C0* performs a load of address *A*. The cache controller will issue a request to the directory to load the data and the directory will fetch it from the LLC or main memory and sent it back to *C0*, where it will be stored in its private cache. Next, another core *C1* will perform the same action resulting that both cores *C0* & *C1* having address *A* in their private data caches respectively. Now suppose that *C1* issues a write to address A (changing its value). Since the address *A* is present in the cache, this request will be fulfilled locally and is not updated in *C0*'s cache nor the LLC. Any future loads and stores to address *A* from *C0* (or any other core other than *C1* for that matter) will not reflect the change made by *C1*.

For cache coherence to be maintained, meaning that every cache holds the most up-to-date value in its private cache, it is required that a cache coherence protocol maintains *Write Propagation* & *Transaction Serialization* [2]. Write Propagation means that when a write is performed to an address by some core, the result of that write will *eventually* become visible to all other cores. This requirement was violated in the previous example, which clearly led to incoherence. Transaction Serialization states that there will exist some ordering of read/write requests, but that order will be identical for every core. To

see why this is a requirement consider the following scenario. Suppose that C0 & C1 both hold address A in their private data cache and its value is 0. C0 issues a write to change the value to 1, but C1 simultaneously issues a request to change the value to 2. If the caches do not observe the ordering of writes happening in the same order, then the caches will still remain incoherent, even if write propagation is maintained.

When designing real-world protocols, it is often easier to reason about if the protocol maintains specific invariants during its execution, from which we can then infer that Write Propagation and Transaction Serialization are enforced. Consider the following invariants:

- *Single-Writer, Multiple-Reader* (SWMR)[14] - This invariant states that at any point during the protocols execution, there either exists a single core which can write (and also read) an address or there can exist many cores that can read an address.
- *Data Value Invariant*[14] - states that the result of a write operation will be reflected in the next read/write operation.

We can show that when these two invariants are maintained, Write Propagation and Transaction Serialization are also enforced [14]. As a consequence, we need only to consider enforcing the above two invariants to ensure that any protocol maintains cache coherence. This is an important result, as it means that we can use a model checker like MurΦ to explore every possible state in the protocol and ensure that these invariants are enforced, and thus verify the protocol.

2.2 Memory Consistency

Memory consistency or consistency model is another important component in a shared-memory multiprocessor. In essence a consistency model is a contract between the processors as to what is allowed behaviour when executing multi-threaded programs. This may seem like a non-issue, but in the endless quest for greater performance, processor pipelines will re-order instructions and buffer writes to main memory in order to improve throughput, which can cause unexpected problems when writing multi-threaded programs.

When considering memory consistency we can divide coherence protocols into two groups. A *Consistency-agnostic* coherence protocol is one where reads and writes are completed synchronously, i.e. the value of the read or write is returned only after it has been completed. This means that the processor pipeline can interact with the memory system as if it is atomic and the protocol provides the illusion of making the caches invisible. This is a nice property as the coherence protocol need not concern itself with the consistency model and vice versa, which provides an elegant separation of concerns [14]. A *Consistency-directed* coherence protocol is one where writes can return before the value has been propagated to all other caches. These kinds of protocols must ensure that when writes are performed their result will *eventually* become visible to all other processors in accordance with the consistency model [14]. Throughout this report we will only consider *consistency agnostic* coherence protocols.

2.3 MOESI Protocols

The MOESI protocols (pronounced ”MO-sey” or ”MO-EE-see”) are a standard group of coherence protocols that use a combination of **M**odified, **O**wned, **E**xclusive, **S**hared & **I**nvalid states to encode the execution of the protocol. Each of these states has a specific meaning, and the protocol is designed to transition between these states in order to maintain the required invariants.

- **I**nvalid - the address is either not present in the cache or the value is no longer up-to-date
- **S**hared - the cache can read the address, and it may be shared by zero or more other caches
- **M**odified - the cache can read and write the address, every other cache is in state **I**.
- **E**xclusive - the first cache to issue a read request is issued with **E** state, and can silently upgrade with write permissions without further communication with the directory.
- **O**wned - a cache in this state holds the most up-to-date value of the address and is responsible for fulfilling coherence requests when they arrive.

MOESI protocols are well understood and common in the literature. We will use a subset of these protocols as input to ProtoGen-MLIR v2 to verify that it is functioning correctly.

2.4 MLIR

MLIR (Multi-Level Intermediate Representation) is a novel approach to building reusable and extensible compiler infrastructure and is part of the LLVM umbrella project [10]. In the past, each programming language required their own full compiler implementations. However, more recently, projects like LLVM provide modular and reusable compiler infrastructure components. This significantly reduces the cost of implementing programming languages since significant components of the compiler pipeline are provided ’out-of-the-box’. However, these components rely on LLVM-IR which is a very low-level representation and resembles a pseudo CPU instruction set. The consequence of this is that many higher level languages cannot implement certain optimizations once the program is transformed to LLVM-IR. Instead such compilers will use custom frontend implementations, where the program is transformed through higher-level representations before eventually targeting LLVM-IR. Rust for example is a language with a very strong type system and has strong memory safety requirements. Its compiler processes the language through two custom intermediate representations before finally targeting LLVM-IR, in order to achieve its safety guarantees and performance [17].

MLIR addresses this problem by allowing the programmer to declaratively specify custom representations. This allows different representations to be used for different levels

of abstraction and semantics, unlike the fixed instruction set of LLVM-IR. Programs can then be compiled through multiple levels of IR, while reusing all the well implemented validation, transformation and optimization components provided by MLIR in each stage.

MLIR is particularly exciting in the context of Domain Specific Languages (DSLs), which are still largely difficult to implement. DSLs tend to be niche and do not have many users, which results in many aspects of their compiler implementations being overlooked or performing inadequately. DSL compilers often suffer from slow compilation times, bugs, poor debugging and error messages, and an overall disappointing user experience. MLIR is designed to provide a robust supporting infrastructure, which allows language designers to create high quality, efficient and maintainable domain specific compilers. As computing architectures become ever more specialised, we expect that highly optimized DSLs will become common. Thus, MLIR is an important technology which will allow language designers to create high quality and efficient optimizing compilers.

Representations in MLIR are all expressed in Single Static Assignment (SSA) form [10]. This means that the result of any variable is assigned only once and every variable must be defined before it is used. Using this form has significant advantages in implementing many compiler optimizations like constant propagation and register allocation, because the origin of values and their use can be traced exactly due to the SSA properties.

ProtoGen-MLIR v2 is implemented using MLIR due to its advantages for creating domain specific compilers. We use MLIR to define our custom representation, which we can generate from our frontend. Once we have transformed the program into our representation we use MLIR's optimization infrastructure by modifying the IR in-place through our declarative transformations. Once the optimized IR is obtained, we use it to generate MurΦ code, which will verify our protocol for correctness.

2.5 MurΦ

MurΦ (also written as *Murphi* and pronounced *Murphy*) consists of the Murphi Compiler and the Murphi description language which was originally developed at the University of Utah to evaluate cache coherence protocols [6].

A Murphi description is a high-level program which consists of declarations of constants, types, global variables, procedures and rules. The Murphi Compiler then uses this to generates a special purpose verifier. This special purpose verifier is then executed to check properties of the system, such as error assertions, invariants and deadlocks. A Murphi program is then executed using the following algorithm.

> 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. [7]

Its important to note that Murphi descriptions are non-deterministic, because of the arbitrary choice in step b). This is a good property, because a coherence protocol must do the 'right thing' regardless of which rule is chosen to be executed.

Due to the large number of states and execution paths that can exist in a coherence protocol, Murphi uses symmetry reduction in order to reduce the state space search. For example, a Murphi description that uses the `scalarset` type, allows it to consider every element of the set as symmetrical. If for example we define the caches in the system to be part of a `scalarset` then a large number of states can be eliminated due to symmetry. If, for example, C_1 was in state S_1 & C_2 was in state S_2, Murphi would consider this equivalent to the scenario C_1 in S_2 and C_2 in S_1.

2.6 PCC Language

The PCC Language is a DSL originally introduced as part of the original ProtoGen project. Its purpose is to allow a coherence protocol designer to specify a protocol using atomic transactions.

As part of the specification PCC requires some basic architectural descriptions such as the networks required for the protocol, the structure of the cache controller and the structure of the messages sent onto the networks. To specify the execution of the protocol, PCC provides a `Process` block, which expresses a transaction between stable states. In Listing 2.1 we show and example of the I→M transaction from the MSI protocol. PCCs simple syntax and declarative style, makes following the execution of a transaction straightforward, especially with the use of `await` syntax to halt execution until a specific message arrives from the directory or another cache. From Listing 2.1 we can still see that the protocol designer must still take care to handle some race conditions, since we must handle Inv_Ack messages arriving before the response from the directory. However, only messages that relate to the currently executing transaction must be handled by the protocol specification and the designer can still assume that the transaction is occurring atomically i.e. without any other transaction executing simultaneously.

```
Architecture cache {
    ...

    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;
        }
    }

    // ... additional transactions
}
```

Listing 2.1: PCC Specification of I→M Transaction

Chapter 3

ProtoGen-MLIR v2

In this chapter we present the significant changes implemented as part of ProtoGen-MLIR v2. In Section 3.1 we first present the new Intermediate Representation used in v2 and outline many of its significant improvements compared to v1. In Section 3.2 we present how we can optimize a cache coherence protocol from its atomic specification, and present a declarative specification for each class of optimization. Lastly in Section 3.3 we present a new and improved CodeGen backend designed for modularity and extensibility.

3.1 Defining the new FSM dialect

As mentioned previously one of the great shortcomings of v1 was the design of its internal representation (IR). In MLIR we can specify a *dialect* which is a collection of related operations, types and attributes that collectively form the IR. In v1 we presented the *PCC dialect*, which was designed to express the language features of PCC in MLIR. However, partly due to time constraints and relatively weak understanding of IR design the resulting IR had many critical flaws. As a consequence, through several iterations, we present a new *FSM dialect*, which is designed as a direct replacement for the PCC dialect and implements the following key improvements.

- Strongly typed operations and interfaces (Section 3.1.1)
- Much more useful abstraction as an FSM (Section 3.1.2)
- Symbolic links to express transitions between states (Section 3.1.2)
- Custom assembly format for improved readability (Section 3.1.3)

To support this new dialect we had to re-implement the frontend tool which parses the input PCC file and generates the correct operations in the FSM dialect. This tool is broadly similar to the tool presented for the PCC dialect in v1, however it much more capable in terms of diagnostics and error reporting. In Section 3.1.1 we show how the much improved type system of the FSM dialect allows our new frontend to deduce types of values used in PCC and report errors when values are used incorrectly.

3.1.1 Strong Types

The original PCC dialect from v1 performed no type checking, and enforced no constraints on operands or arguments of operations. In Listing 3.2 we present an example of the `cache_definition` operation from the PCC dialect, which is used to specify the internal storage of the cache controller from its definition in PCC shown in Listing 3.1. From the definition we can see that the cache controller should have two internal state variables: (1) a **State** variable with initial value **I** and (2) a **cl** variable of type **Data** and no initial value, which represents a cache line stored in the cache. The operation in Listing 3.2 simply maintains two string arrays of the field names and their types. Not only does this encoding not preserve any real type information, it is actually incorrect since the initial state 'I' is not a type at all. Moreover, the operation does not return any SSA results nor have any symbolic references, which means that operations which wish to interact with the internal state cannot reference it directly.

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

Listing 3.1: PCC language cache definition

```
"pcc.cache_definition" ()
    {fields=["State", "cl"], types=["I", "Data"]}
: () -> ()
```

Listing 3.2: PCC dialect cache definition

```
fsm.machine @cache{
    %State = fsm.variable "State" {initValue="I"} : !fsm.state
    %cl = fsm.variable "cl" : !fsm.data
    ...
}
```

Listing 3.3: FSM dialect cache definition

In Listing 3.3 we present how the cache internal state is represented using the FSM Dialect. Firstly observe that we define a generic operation called `machine` which is used to specify any finite state machine in the system, in our case we will use this operation to specify the controllers for the cache and directory. The operation is also declared as a **Symbol** and assigned the symbolic name *cache* (indicated by the `@` character), which means that it can be referenced by future operations. The `machine` op contains a single region (a list of operations nested within the parent operation) in which we specify the internal state. Note that the prefix `fsm` is used to indicate which operations and types belong to the FSM dialect.

Once within the region of the `machine` op we can specify the internal state of the controller using the `variable` operation. This operation contains a reference to the original variable name as well as allowing an optional initial value attribute. Each `variable` operation also produces a single SSA value as a result, which is used as

```
/home/petr/dev/protogen-mlir-v2/cmake-build-debug/bin/protogen-translate --pcc-to-fsm-dialect ../../protocols/MESI.pcc
../../protocols/MESI.pcc:47:25: error: Cannot assign Type 'i64' to '!fsm.state'
                State = 5;
                        ^
```

Figure 3.1: FSM Dialect Diagnostic Error

input to other operations which will update the state. Furthermore, the result is strongly typed with custom types that are also specified as part of the representation. In this case we use the `fsm.state` and `fsm.data` types which map directly to the types used in PCC. The strong types used here become important when considering operations that update the state, because we can enforce that the types match. This means that if a PCC programmer attempts perform updates to the internal variable with invalid values, our compiler will issue errors and show exactly where in the code the error is occurring. This kind of diagnostic ability was simply not possible with the string based representation used in v1. In Figure 3.1 we present an example of such an error, when we attempt to assign and invalid value to the machine variable ”State”. This same style of IR design was replicated in the other hardware specifications including the message type interface and the network type interface.

With the success of this design we further pushed to improve other areas where the PCC dialect suffered from weak or no type system. In Listing 3.4 we show in PCC code for how a message is constructed. In this case were are creating a Request message and the arguments to it represent that we constructing a *GetM* message, which has source address *ID* (meaning the cache it was sent from) and destination address of the directory. In Listing 3.5 we show how this construct is transformed into the PCC dialect in MLIR. You can see that we again use string attributes to represent the data and arguments passed to the message constructor. This means that even references to constructs within the IR aren't expressed, such as the reference to `directory.ID` is simply a string and preserves no meaning. Although the operation does yield an SSA result, which can be used by other operations, it is still however incorrectly typed. Since MLIR requires that results have types, we chose to type this as i64 (even though its not an integer) just as a placeholder since the PCC dialect had not types of its own. The consequence of this decision meant that we lost all information about the context of what this result meant.

```
msg = Request(GetM, ID, directory.ID);
```

Listing 3.4: PCC message construction

```
%msg = ”murphi.msg_constr”(){msgType=”Request”, parameters=[”
    GetM”, ”ID”, ”directory.ID”]} : () -> i64
```

Listing 3.5: PCC dialect message construction

```
%src = fsm.ref @cache
%dst = fsm.ref @directory
%msg = fsm.message @Request ”GetS” %src, %dst : !fsm.id, !fsm.id -> !fsm.msg
```

Listing 3.6: FSM dialect message construction

In Listing 3.6 we show how the construct from Listing 3.4 is expressed in the FSM dialect. Firstly, you can observe that the same construct is expressed with several

operations. The two `ref` operations construct references to the source and destination addresses that we will then use in the message constructor. This is possible because our `machine` operations are symbols and can be referenced by other operations. The `ref` operation is declared to always return an SSA result of the type `id`, hence why the type definition is omitted here, but the result is still strongly typed (see Section 3.1.3). By having strongly typed SSA values for these references means that we can use them as inputs (or operands) with the message constructor operation. The last operation `message` (which constructs the message) has a symbolic reference to the type definition of the message we are constructing (in this case Request), next is the string of the message name followed by the remaining arguments to constructor. As a result we obtain a strongly typed `msg` SSA value, which is then later used by other operations, such as sending the message onto one of the interconnects. Leveraging this strong typing, we can again provide diagnostic and error reporting capability to the user, in the case when the message construction is written incorrectly, such as entering too many arguments or of the wrong type.

This strong type system is enforced throughout the FSM dialect, with all the required custom types implemented from PCC, however we felt that these two examples most strongly highlight it benefits.

3.1.2 FSM abstraction

The PCC dialect had (rather crudely and naively) modeled its operations too closely to the language structures of PCC and provided no useful abstractions to the problem we were actually trying to solve. A cache or directory controller is effectively a finite state machine (FSM): in each state it can receive messages from either the interconnect sent by other controllers or from read/write requests issued by the processor pipeline. The pair of (state, event) will always uniquely identify a transition in a FSM, and each pair is also associated with a set of actions that will be performed when this transition is triggered, and a destination state. Thus, we can reason about a transition in a FSM to be a 4-tuple of the form (state, event, destination state, actions).

```
fsm.machine @cache(){
    %State = fsm.variable "State" {initValue = "I"} : !fsm.state
    %cl = fsm.variable "cl" : !fsm.data
    %acksReceived = fsm.variable "acksReceived" {initValue = 0} : !fsm.range<0, 3>
    %acksExpected = fsm.variable "acksExpected" {initValue = 0} : !fsm.range<0, 3>

    fsm.state @I transitions {

        fsm.transition @load() attributes {nextState=@I_load} {
            %src = fsm.ref @cache
            %dst = fsm.ref @directory
            %msg = fsm.message @Request "GetS" %src, %dst : !fsm.id, !fsm.id -> !fsm.msg
            fsm.send %req %msg
        }

        fsm.transition @store() attributes {nextState=@I_store} {
            %src = fsm.ref @cache
            %dst = fsm.ref @directory
            %msg = fsm.message @Request "GetM" %src, %dst : !fsm.id, !fsm.id -> !fsm.msg
            fsm.send %req %msg
            %n_cnt = fsm.constant {value = "0"} : i64
            fsm.update %acksReceived, %n_cnt : !fsm.range<0, 3>, i64
        }
    }
    // ... more states and transitions ... //
}
```

Listing 3.7: Sample from the FSM dialect

In Listing 3.7 we show a snippet of IR in the FSM dialect obtained from a PCC definition of the MSI protocol. We can again see the definitions of the internal state of the cache controller, as well as a new `range` type which restricts integers between two limits. In this example we show a single `state` operation, which has symbolic name `I`. Nested within its region, we can then specify which events (as a symbolic reference) can take place in this state with the `transition` operation. Inside the region of the transition operation we have nested the actions that will be performed when this transition is triggered. A transition operation is also assigned a symbolic `nextState` attribute, which links to the destination state. This provides an elegant abstraction to the 4-tuple model of an FSM. In the first level of hierarchy we can find the current *state*, within the state region we can the find the specific *event* which is triggered, and lastly within the transition we can find the *actions* that need to be performed. This allows us to elegantly express the coherence protocol as a set of finite state machines.

Because every state and transition operation in the FSM dialect is a symbolic reference, we can use these to specify the links between states. In each transition operation we specify a symbolic reference to the new state of the cache after the transition is completed. This may seem like an unimportant consequence of the IR, but is in fact quite deliberate. When performing optimizations, it is necessary to trace part of the previous execution of the state machine, which can now be naturally done by following the path with the `nextState` attribute.

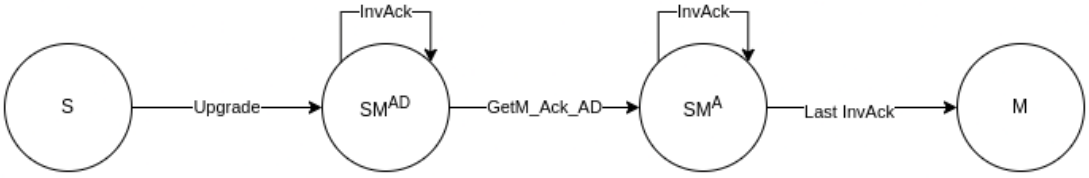

Figure 3.2: S→M transaction converted to use transient states

3.1.3 Custom Assembly Format

One nice improvement that the new FSM dialect implements is a *Custom Assembly Format*. MLIR has a built-in parser and printer which will work for any operation defined in MLIR and is referred to as the *generic syntax*. This is useful for an IR designer as they do not need to perform any additional work to parse and print operations. However, since it is designed to work with any operation, all components of the operation are printed, which makes them less readable and more verbose than is necessary. With the FSM dialect we chose to define a custom assembly format for each operation, along with the necessary parser and printer implementations which hook into MLIR to read and write the representation correctly. This change is only syntactic and has no real effect on the performance or expressiveness of the IR, but it makes the IR much more readable than when using the generic syntax (See Listing 3.8).

```
%3 = fsm.add (%1, %2) : (i64, i64) -> i64    // generic syntax
%3 = fsm.add %1, %2                          // custom assembly format
```

Listing 3.8: Generic Syntax vs Custom Assembly Format

3.2 Defining declarative optimizations

Once we have successfully converted the PCC protocol specification into the FSM dialect, we can begin performing transformations on the IR which implement our optimizations. The first step in our optimization pipeline is to generate an equivalent stable state protocol, which removes the `await` syntax. Protocols in PCC are designed to be specified as a series of transactions between stable states, each of which are executed atomically. However, a transaction between two stable states states sometimes involves multiple steps. Consider the transaction from S→M in the MSI protocol: (1) Send request to the directory, (2) wait for data from directory, (3) wait for all invalidation acknowledgements to arrive before completing transaction. PCC provides a nice abstraction for the protocol designer with the `await` statement, which is a blocking operation that waits for another event to occur before continuing the execution of the protocol. However, we can easily express this structure as an FSM by transitioning to an appropriate transient state and introducing the appropriate transitions between them to follow the execution of the protocol. In Figure 3.2 we show how the S→M transaction is encoded into a FSM by introducing the appropriate transient states and transitions. We consider this step as more of a pre-processing step rather than an optimization.

3.2.1 Stalling Optimizations

Now that we have a FSM protocol without any awaits, we still however require that we enforce atomic transactions. In order to correctly relax this constraint we must be able to handle all race conditions that can occur. In essence, we must answer the question: How should a cache controller respond when it receives a message from the directory or another cache, which is unrelated to the currently executing transaction? To be able to answer this question we must first understand which messages might be received in any particular transient state and also to which ones we must respond immediately and which ones we can delay.

In directory based protocols, the directory acts as the serialization point for all messages: even if two messages arrive simultaneously, the directory will break the tie and order them. What this means is that when two caches issue transactions simultaneously the directory will chose a *winner* and a *loser*. However, since our protocol assumes atomic transactions we do not currently handle any unexpected messages that will arrive due to race conditions. The questions we must now answer are: ”How can I (as a cache) determine that I have lost a race to the directory?” & ”If the race is lost, how should I continue with the execution of the protocol?”.

To answer the first question, it may seem reasonable that any unexpected message received from directory would indicate that the cache lost the race, but this is not entirely true. Suppose that two caches C1 & C2, issue transaction requests for I→M and they then both send GetM messages to the directory. The directory serializes the requests such that C1 wins the race and responds to C1 with the data. Next suppose that this message becomes delayed on the interconnect and in the meantime the directory processes C2’s request. Since the directory has issued the data to C1, it forwards the request to C1 to send the data to C2. But C1’s data message has been delayed, so it receives the forwarded message from C2 first. This message is not expected by the specification of the protocol, but C1 has still won the race.

Transactions within our protocol always occur between two stable state i.e. S→M, even through they are executed by transitioning through transient states. What this means is that any transient state always has a **logical start state** (the stable state from which the transaction began) and a **logical end state** (the resulting stable state after the transaction is completed). The directory will always *see* a cache as being in one of these two stable states and will therefore issue messages to the cache which are related to these states. Therefore, if a cache receives a message that is expected by its logical end state (like in the previous example) it can deduce that it won the race, conversely if it is expected by the logical start state then it lost the race.

If a cache receives a message from the directory and deduces that it won the race, it can safely delay responding to this message. This is because it knows that the original response from the directory must still be present on the interconnect and will eventually arrive and thus will not result in a deadlock. Moreover, in some cases it is unable to respond since it may have not yet received the data that needs to be forwarded. However, if the cache deduces that it lost the race, it cannot delay its response as this could lead to deadlock. For example, suppose that two caches C1 & C2 both currently is State S, simultaneously initiate the S→M Transaction (see Figure 3.2), 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 it lost the race as this request is handled by its logical start state S. If C2 were to delay 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, since C2 has not yet responded with the invalidate message, which causes a circular dependency and therefore a deadlock. Thus, to be able to relax the atomic transactions constraint, caches must respond to messages when their transaction lost the race and they can delay (or stall) all other unexpected messages. We refer to protocols of this class as *Stalling Coherence Protocols*.

Now that we have this insight we present a declarative specification of how to perform these necessary optimizations. In order to do this we will use the following notation.

$$FSM = \{S, T\}$$

Represents a FSM controller in the protocol, where S is a set of all states (including transient states) in the controller, and T is a set of transitions.

In our specification we say that a transition $t \in T$ is a 4-tuple of the form: $t = (S_{start}, e, S_{end}, \alpha)$. With this syntax S_{start} & S_{end} represents the start and end states, e is the event i.e. (load/store/GetM ...) & α represents the set of actions to be performed. Furthermore, we define some shorthand statements, which allow us to more clearly express express how our optimizations are implemented.

$$\begin{aligned}
Transient(s)_{s\in S} &: -\text{returns true if } s \text{ is a transient state} \\
StableStart(s)_{s\in S} &: -\text{returns the stable start state of } s \\
StableEnd(s)_{s\in S} &: -\text{returns the stable end state of } s \\
Events(s)_{s\in S} &: -\text{returns the set of all events handled in the state } s
\end{aligned}$$

In Listing 3.9 we present a declarative specification of the function *Optimize*, which implements the stalling optimizations described above. To optimize the entire protocol, we simply repeatedly scan through every state in the protocol and execute the optimize function with that state. We stop execution once no additional change is detected.

```
Optimize(S) :-
    if Transient(S)
        let ss = StableStart(S)
        ∀t' ∈ ss ∧ t'.e ∉ Events(S)
            HandleRaceCondition(S, ss, t')
```

Listing 3.9: Specification of Optimize Function

```
HandleRaceCondition(S, ss, t') :-
    let S_new = GetNextState(S, ss, t')
    let t_new = new Transition(t.S_start, t'.e, S_new, t'.α)
    FSM ⋃ t_new
```

Listing 3.10: Specification of HandleRaceCondition Function

```
GetNextState(S, ss, t') :-
    let t_prev = ∃e,α Transition(ss, e, S, α)
    if ∃S_new Transition(t'.S_end, t_prev.e, S_new, t_prev.α)
        return S_new
    let S_new = new State()
    let t_dir = ∃e_dir Transition(S, e_dir, t'.S_end, α)
    let t_new = new Transition(S_new, t_dir.e, t'.S_end, {})
    FSM ⋃ t_new
    return S_new
```

Listing 3.11: Specification of GetNextState Function

The Optimize function in Listing 3.9 first checks if the state we are considering is transient (since stable states cannot be optimized). Next we obtain its logical start state, and then, from our reasoning before, we know that any event that can be handled by the logical start state must also be handled in our transient state. Once these conditions are met we know we must handle this case and cannot simply stall. Therefore, we issue a call to the `HandleRaceCondition` function defined in Listing 3.10.

To handle a race condition we first obtain a new state to which the cache will transition to after handling the race condition (shown in Listing 3.11). Next we construct a new transition from the original transient state to the newly discovered state which will occur when the unexpected message arrives. Since the cache must behave as if being in the stable state from where it originated, it uses the actions that are used by the unexpected message in the stable state. This new transition is then added to the FSM.

The trickiest part of specifying this optimization is deciding what state the cache should transition to after handling the unexpected message. What's important to understand is that when a cache loses the race, the nature of its original transaction changes. For example, if a cache initiates a S→M transaction, but while waiting for its response it receives an invalidate. This means that another cache won the race at the directory since invalidates are handled in the logical start state **S**. However, when this invalidate arrives when the cache in state **S** it downgrades its permissions to **I** (since its been invalidated). What this means is that when the directory processes the losing transaction it will observe it to be an I→M transaction rather than a S→M one. In Listing 3.11 we first attempt to find a transient state which is part of the now modified transaction i.e. in the previous case it will transition to the transient state *IM* which is part of the I→M transaction. However, sometimes there is no existing transaction which represents the current execution. Consider the transient state in the M→I transaction: the cache has sent an eviction request to the directory and is waiting for and acknowledgement that it has been received before it can complete and transition to **I**. However, the cache can

receive a forwarded request for the data to which it must respond causing itself to self invalidate to state **I**. In the protocol there is no I→I transaction and so we must create a new transient state to handle this case. Finally, since we know what state the directory will be in when it eventually receives our message, we know which response will be sent back. Once we receive this response we can then silently complete our transaction without any further actions.

Having this declarative specification significantly affected to design of the FSM dialect too. This is because in the specification we require a method of tracing specific executing paths of the protocol to be able to find the logical start state or to discover a potential end state to transition to. This requirement lead us to develop symbolic links between states, which allowed for complete knowledge of the execution of the protocol and be able to fulfil the requirements successfully. The PCC dialect was significantly limited in this regard as links to previous states were encoded as simple strings, and to gain complete knowledge required significant processing of the IR, which was error prone and missing links between states were common.

3.2.2 Non-Stalling Optimizations

Once we have obtained a stalling version of the protocol, we can now safely remove the atomic transactions constraint and verify the protocol for correctness. However, stalling messages is not ideal since the incoming queue of messages is blocked, preventing the cache controller from processing other forwarded messages that sit behind it in the queue. In order to solve this we can remove the message from the queue and defer its response until we have completed our original transaction, thereby unblocking the queue of incoming messages and increasing the overall throughput of the cache controller. We refer to protocols of this type as non-stalling cache coherence protocols.

The disadvantage to this technique is that it will introduce a large number of transient states and additional transitions, as the coherence protocol must track (with these new transient states) what actions have been deferred and when they can be responded to. However, now that we have fully relaxed the atomic transaction constraint we can slowly add these optimizations over time, since we can always fall back to stalling incoming messages. This makes them much easier to test and verify, since we can verify the protocol after each change.

With the success we observed in the stalling case we again wanted to provide a declarative specification for performing non-stalling optimizations. In Listing 3.12 we show the *Optimize* function, which is identical to Listing 3.9 except now we are considering messages that can arrive in the logical end state instead.

```
Optimize(S):
    if Transient(S)
        let se = StableEnd(S)
        ∀t′ ∈ se ∧ t′.e ∉ Events(S)
            NonStall(S, se, t′)
```

Listing 3.12: Non-Stalling Optimization Specification

```
NonStall(S, se, t'):
    t_next = ∃e,α Transition(S, e, se, α)
    S_new = new State()
    t_race = new Transition(S, t'.e, S_new, {})
    t_eventual = new Transition(S_new, t_next.e, t'.S_end, {t_next.α, t'.α})
    FSM ⋃ S_new, t_race, t_eventual
```

Listing 3.13: Non-Stalling Specification

In Listing 3.13 we show how we can implement the non-stalling optimization, when we receive an unexpected message which is handled by the logical end state. Firstly, we must find the transition in the cache controller which resumes the execution of the transaction when it receives the correct response from the directory (labeled here as t_{next}). We then create a new transient state (S_{new}) and a new transition from the current state S to the new state s_{new} when we receive the unexpected message. Note, that we do not perform any actions yet as part of this transition since we have not completed our original transaction. In the new transient state S_{new} we need to add a new transition for when our originally expected message arrives. This transition will perform its original actions t_{next}, followed by the actions of the racing transition t'. The protocol, functions identically to the non-stalling one, except that we have increased the throughput of the cache controller by not blocking the queue of incoming messages.

As an example, consider the I→S transition in Figure 3.3 where we show the implemented non-stalling optimization. In the state IS^A, instead of stalling the Inv message (from the logical end state S), we instead transition to the newly created II^A state, where we continue to wait for the original GetS_Ack message. When this message arrives, we then perform the actions as specified in Listing 3.13 and transition to state I, since when the GetS_Ack message arrives, we have effectively completed two transactions: I→S immediately followed by S→I.

However, by implementing this optimization there is a possibility of livelock, meaning that two caches can seemingly be able to execute their protocol correctly without making any progress. In this example, if a cache initiates a I→S transaction, but when waiting for the data from the directory it receives an Inv messages, causing it to yield its transaction and return to I state and restart the transaction. If this situation happens repeatedly, the cache will fail to make progress to transition to state S and fulfil its read request. We can solve this problem, because when the GetS_Ack response arrives in state II^A we can allow the cache to complete a single read of the data before transitioning back to I. In effect, the cache logically briefly transitions to state S to complete its read request before immediately being invalidated. In this way, all caches will continue to make progress and will not become livelocked by forwarded messages which are handled by the logical end state.

3.2.3 Optimizing the Directory Controller

Up to now we have only been discussing optimizations in terms of the cache controllers. However, we can optimize the directory controller in an almost identical way. In fact, optimizing the directory is far simpler, since the directory controller has complete

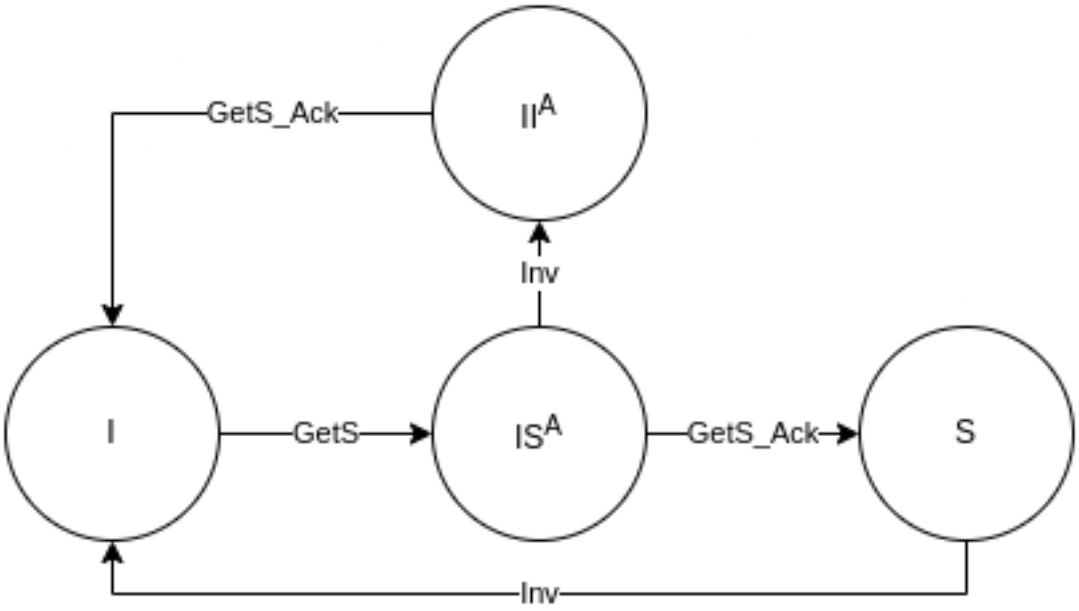

Figure 3.3: Non-Stalling I→S Transaction

knowledge of how requests have been serialized, and knows exactly what messages can arrive. Although it is possible to generate a non-stalling directory controller, the optimization potential is limited and introduces a large number of additional transient states and transitions. We therefore, only perform non-stalling optimizations on the directory for simplicity.

One technique we must perform at the directory is *message reinterpretation*. Consider an MSI protocol that uses an Upgrade message when initiating the S→M transaction instead of a GetM. The difference being that an Upgrade response does not require the data being sent across the network, which is more efficient as it reduces the bandwidth on the interconnect. Suppose that a cache initiates a S→M transaction and sends an Upgrade message to the directory, but later receives an Invalidate and deduces that it lost the race and transitions to the appropriate state according to our optimization specification. However, after the directory has processed the winning transaction and invalidated all sharers, it then attempts to processes the invalidated cache's Upgrade message, which is invalid as the directory only expects to receive this message from caches in state S. Instead of discarding this message, we know that this message must have been sent from a cache in state S, but has since been invalidated, so we can reinterpret this message as a GetM. For us to be able to correctly reinterpret messages at the directory, it is essential that a unique message is sent in each stable state, so we can deduce from which state the cache originally sent the message. This requirement however, is naturally fulfilled by most directory protocols.

At the directory there is an interesting consequence, that stale Put message (PutM, PutS, ...) can arrive in any state [16]. Consider the scenario where a cache issues a PutS message to the directory, but loses the race to a GetM from another cache. The directory processes the GetM by invalidating all sharers and sending back the data and transitions to state M. However, now in state M the directory does not expect to receive a PutS

because it cannot occur with atomic transactions and is thus not part of the protocol specification. To be able to handle this scenario, we have to leverage domain knowledge of directory based coherence protocols. We know that if a stale Put message is received, it signals that the cache that initially sent the message lost the race to the directory. Thus, we can safely acknowledge the message and allow the cache to complete its stale transaction. Note, this approach only applies to MOESI protocols, other protocols that use Nacks (Negative Acknowledgements) will need to be handled differently. However, ProtoGen-MLIR is designed to work only with MOESI protocols, therefore this is the correct solution.

3.3 Modularizing and Generalizing the CodeGen backend

Once we have performed all optimizations on the IR in the FSM dialect, we then scan over the operations in the IR to generate a Murphi description, which we can then compile (with the Murphi compiler) and validate that our protocol enforces cache coherence correctly. V1 had a very simple implementation of this component, it scanned through the operations and performed some basic string manipulations to generate the Murphi description. This approach was flawed as it introduced deep coupling between the IR and the CodeGen component. Therefore, since v2 uses the much improved FSM dialect, the entire CodeGen component would require significant rework to function with the new representation. Instead of this we chose to re-implement this component entirely with a modular and general design, that eliminates the coupling between the IR and the CodeGen backend component.

We were motivated in this decision too by the unimpressive performance of the v1 GodeGen implementation. Firstly, the generated code wasn't particularly robust and suffered from many bugs, which were influenced by the weak type system of the PCC dialect, which meant ad-hoc techniques were used to deduce the correct types for use in Murphi. Moreover, nice features like code indentation and code formatting we not present and proved difficult to implement with the previous design. This made analyzing and debugging the generated code difficult.

Instead, to address these problems, in Figure 3.4 we present the architecture of the CodeGen component implemented in v2. To allow us to support any future or existing dialect, we introduce the *Dialect Interpreter* abstract class. The interpreter provides has a natural interface to the essential information required to produce a Murphi description and does not rely on any features of a particular dialect. The *CodeGen* component then interacts with the specific interpreter for the dialect in question and constructs the input to the Inja template rending engine as a large JSON object.

To be able to produce a robust and readable Murphi description, we chose to use a package called Inja [1], which is a C++ implementation of the Jinja templating engine used in Python. A templating engine allows us to define a template for every structure we wish to utilise in our Murphi description. A template can then be combined with JSON data, which allows us to render out specific versions of our templates as we require. Moreover, templates can also include other templates, which allows us to

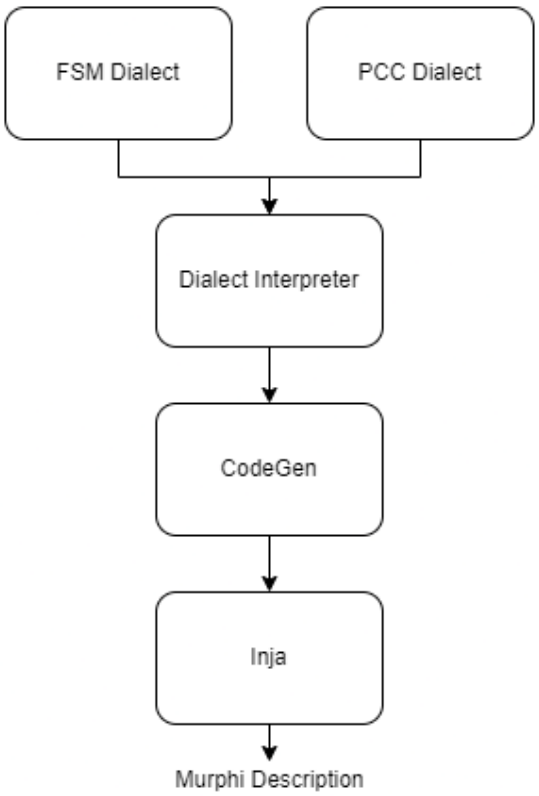

Figure 3.4: Murphi Backend Implementation in v2

construct our Murphi description from set of individual templates each of which will render out the specific structure we desire. When the *Inja* component receives the data from *CodeGen*, it verifies that it is of the correct format, parses the required templates and renders them. The resulting output is a well formatted, correct and readable Murphi description.

This component was difficult and time consuming to implement, however its design proved invaluable due to the frequent changes made during the design of the FSM dialect. When the IR was changed, only a few additional changes were required, all contained within in the *Dialect Interpreter* class, to convert the dialect to Murphi. Moreover, as the project progressed, we extended support for additional PCC language features required for more complex protocols, and experiences very little friction when implementing their translation.

3.4 Testing and Quality Assurance

When implementing v1, little consideration was spent on good design and coding practices. As the project grew it proved increasingly difficult to maintain and implement new features and changes would often have unforeseen affects throughout the project. As part of this project we focused heavily on writing automated tests, especially on the IR and optimization components of the compiler. Having high quality and reliable tests proved essential in the long run, as it enabled us to iterate on the design of the FSM

dialect and our optimization transformations, since many components relied on this functionality to work correctly.

To address this v2 has over 200 automated unit tests, which extensively test all operations of the IR and other important functions used throughout the project. Moreover, we also incorporated *FileCheck* tests for our optimizations passes. FileCheck is a simple tool provided with MLIR, which loads in a snipped of user supplier IR, and runs it through a specified optimization pass. This allows us to check the result of each optimization pass individually for a range of inputs and verify that optimization is functioning correctly. Finally, we created a CI build for the project using GitHub actions, which built the project from scratch and executed all unit and FileCheck tests. We believe that focusing on this infrastructure in such a large and complex project increased productivity overall, as it provided confidence in the implementation of new features.

Chapter 4

Testing

In order to thoroughly test our compiler implementation, we need to compile a variety of different MOESI protocols (MI, MSI & MESI) with ProtoGen-MLIR. After each successful compilation, we then compile the generate Murphi description (with the Murphi compiler) to verify that the optimized protocol correctly enforces SWMR and is free from deadlocks. For each protocol specification we can also compile it with different options to verify each optimization level. Firstly, we can compile the protocol without any any optimizations and enforcing atomic transactions, which ensures that the high level specification is correct. Next we can compile with our stalling optimization pipeline, which relaxes the atomic transactions constraint by handling all messages related to the logical start state. Finally, we can compile with stalling & non-stalling optimizations enabled and generate the most concurrent version of the protocol.

4.1 MI Protocol

The MI protocol is the most basic MOESI protocol, and incorporates only two stable states: (M)odified for read/write & I(nvalid) for evicted or non-cached blocks. In Figure 4.1 we present an FSM transition diagram of the atomic specification of the MI protocol and the PCC protocol specification in Appendix A. With MI, in order to fulfil any read/write request, the cache must first obtain M state and return back to I when evicting or receiving a request from another cache. After pre-processing the protocol by removing the `await` syntax we obtain the FSM cache controller presented in Figure 4.2 with the additional necessary transient states. From this specification we successfully generated the Murphi description while enforcing atomic transactions and successfully verified the protocol.

Now that we have shown that the protocol is correct as an atomic protocol, we can apply stalling optimizations, which will allow us to relax this constraint. In Figure 4.3, we show the generated protocol after applying the stalling optimization pipeline. We can see that in the MI protocol, there is only a single transient state which requires us to handle a race condition 'M_evict': the other states 'I_load' & 'I_store' have logical start state I, which does not handle any messages. We can see that the state 'M_evict' has logical start state M and thus must handle the forwarded message 'Fwd_GetM'. We

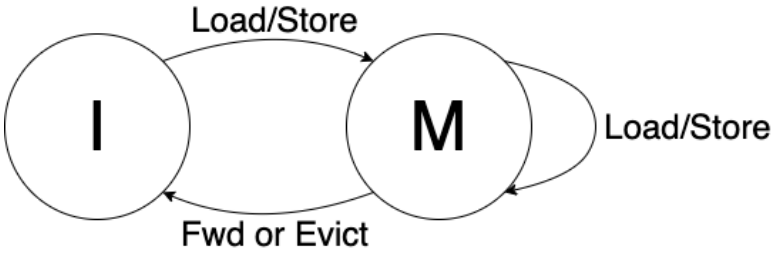

Figure 4.1: Stable MI Protocol

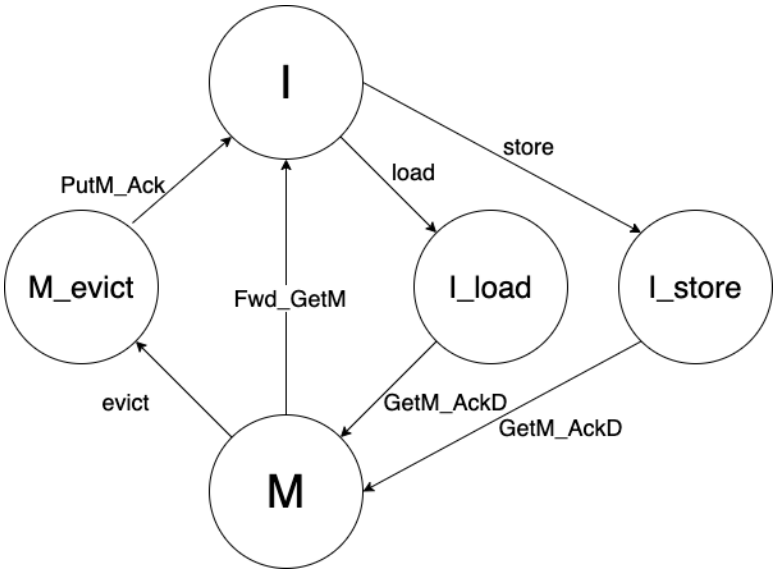

Figure 4.2: Pre-Processed MI Protocol

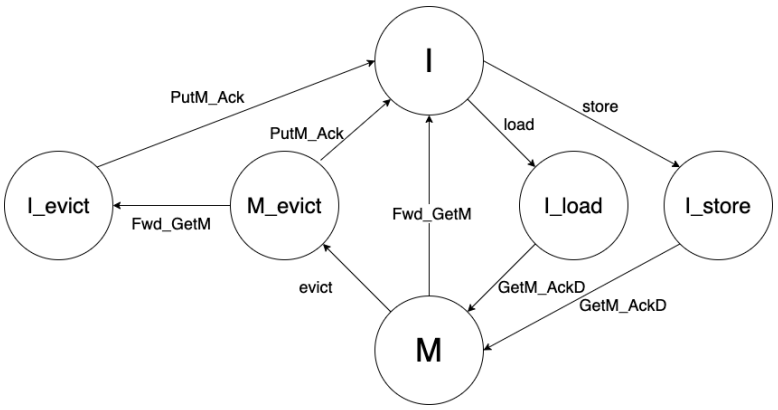

Figure 4.3: MI Stalling Protocol

know that the state 'M_evict' is part of the M→I transaction, but when the 'Fwd_GetM' arrives the transaction changes to I→I. Therefore, since no state exists in the protocol which is part of this transaction, we must create it ('I_evict') as discussed in Section 3.2. However, to complete the transaction the cache still requires a response from the directory and we know that the directory will continue to acknowledge stale Put messages. Thus, in state 'I_evict' the cache continues to wait for the PutM_Ack before exiting. We managed to successfully compile and verify the generated protocol with Murphi for both deadlock freedom and the SWMR invariant.

Finally, we attempt to generate the most concurrent protocol by applying stalling optimizations followed by non-stalling optimizations. The generated protocol contained the optimization presented in Figure 4.4, with an identical optimization implemented for the state 'I_load'. In the state 'I_store' we handle the 'Fwd_GetM' message from the logical end state M and transition to the new state 'I_store_Fwd_GetM_I'. In this new state, we then continue to wait for our original message (since we know we won the race) before completing both transactions simultaneously. We again compiled this specification into a Murphi description and successfully verified it.

Its important to note that the newly generated transient states are then also passed through the optimization pipeline, but because both 'I_store_Fwd_GetM_I' & 'I_evict' both have logical start and end states I, there is no further optimization potential. Thus, the generated protocol we obtained is an optimal non-stalling MI protocol and indeed this result matches exactly with the output generated from ProtoGen.

4.2 MSI Protocol

The MSI protocol is an extension of the MI protocol which incorporates an additional S(hared) stable state. The shared state allows for multiple caches to read simultaneously and is issued to caches upon completion of a read request. We also use the version of the MSI protocol with Upgrade messages to complete the S→M transaction to indicate that we do not require a data response from the directory. We provide the full protocol specification in Appendix B. We managed to successfully compile and verify the protocol without any optimizations and enforcing atomic transactions, which indicates

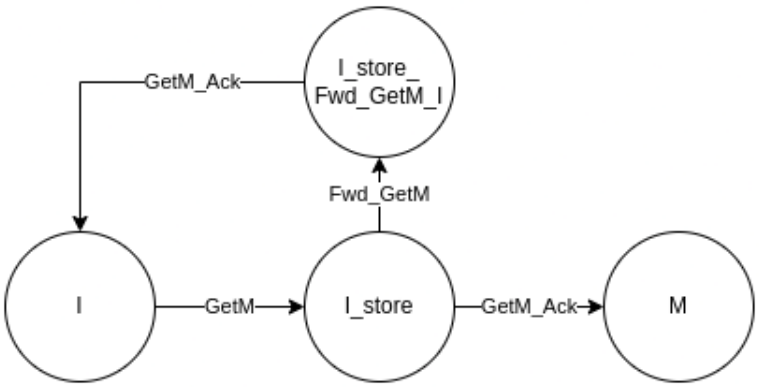

Figure 4.4: I→M non-stalling transaction

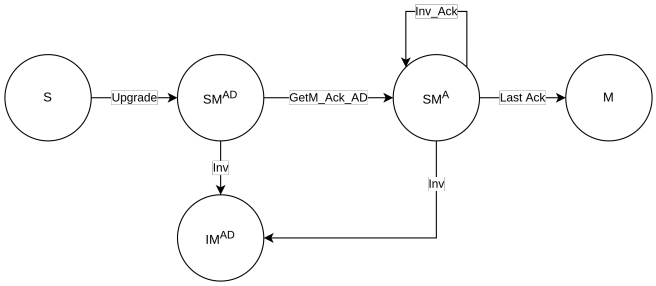

Figure 4.5: MSI Optimized S→M Transaction

that the protocol was functioning correctly.

Next we applied the stalling optimization pipeline to the MSI protocol and obtained a successful compilation. In Figure 4.5 we show the stalling optimization generated to the S→M transaction in the MSI protocol. In the diagram we present the transaction using the state naming conventions from *A Primer in Memory Consistency and Cache Coherence*[14] for clarity, but ProtoGen-MLIR generated an identical transaction albeit with different (and much longer) names. We can see that in the state SM^{AD} we generated the transition to handle the `Inv` message from the logical start state S. When an Inv message is received, the original transaction changes from S→M to I→M and it correctly transitions to the existing state IM^{AD}, which will resume the transaction from state I. Interestingly however, we found the same transition from the state SM^{A}. This transition is not strictly necessary because once the cache has received the response from the directory, the directory will begin issuing messages related to M state. This means, that when the cache is in state SM^{A} it will never receive an invalidate message. The transition was added because ProtoGen-MLIR believes that the state SM^{A} has logical start state S (which is true), but is no longer seen as such in the context of the protocol. However, the cache can continue to stall messages from the directory without risk of deadlock because the directory has already issued all invalidation messages and thus all acknowledgments of invalidation will eventually arrive and the cache can complete the transaction safely. So although the transition is redundant it does not negatively affect the operation of the protocol.

Next we present the remaining stalling optimizations generated in Figure 4.6, with

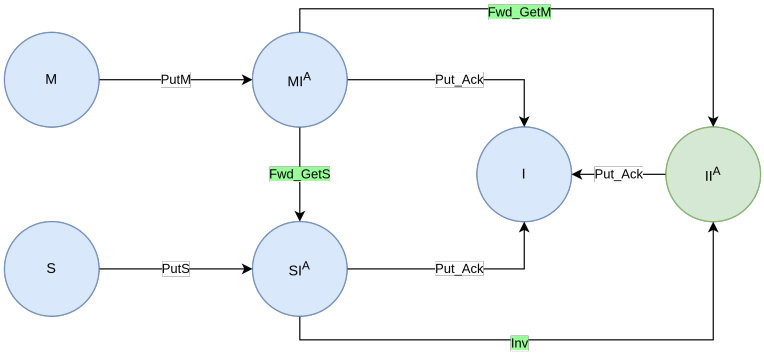

Figure 4.6: S→ I & M→I stalling transactions

the existing states and transitions shown in blue and the newly generated states and transitions shown in green. Firstly, we can see that the state MI^A needs to respond to messages handled by its logical start state M, which are Fwd_GetS and Fwd_GetM. With a Fwd_GetS the transaction is altered to S→I and therefore transitions to state SI^A, but with Fwd_GetM a new state is generated (II^A) to accept the Put_Ack message. Also the SI^A with logical start state S, still needs to handle Inv messages, upon which it also transitions to II^A. With this resulting stalling protocol, we again generated the corresponding Murphi description and verified it for both deadlock freedom and SWMR.

Lastly we applied both stalling & non-stalling optimizations to the protocol and generated a correct and verifiable protocol specification. However, inspecting the generated protocol, ProtoGen-MLIR failed to discover all potential optimizations. The only optimization discovered was for the I→S transaction, which is functionally identical to the case in MI (see Figure 4.4), except that we handle a racing Inv message. The reason behind this is that from our non-stalling optimization specification (see Section 3.2.2) we require that there is only a single intermediate transient state between the logical start and logical end states. For transactions that go through multiple transient states i.e. S→M & I→M our optimization specification does not consider this. ProtoGen uses a different technique to optimize non-stalling protocols based on distributing states into *StateSets* from which it can determine the logical start and end states, and is thus not limited in this way.

4.3 MESI Protocol

The MESI protocol extends the MSI protocol with an additional E(xclusive) stable state, which is issued when no other cache is currently reading the block (i.e. it becomes the exclusive user). When a cache is issued with exclusive state from a read request, it can complete a subsequent write request silently, without issuing coherence requests to the directory or other caches. This reduces the overall latency of a coherence transaction, but also reduces the bandwidth on the network as fewer messages are sent. Many applications are single-threaded and do not require data being shared across cores, therefore they are always issued with E state and do not require another coherence transaction with the directory. We again provide a full PCC specification of MESI (see

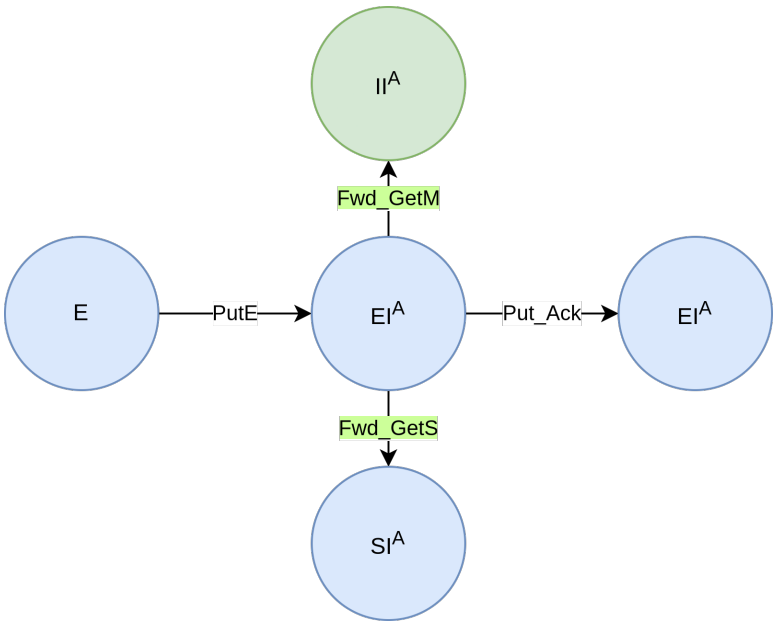

Figure 4.7: MESI: E→I Stalling Transaction

Appendix C), which we successfully compiled and verified without optimizations and with atomic transactions, to ensure a working stable protocol.

When we attempted to compile the MESI protocol with stalling optimization we generated a nearly identical protocol to MSI. This is not surprising as the MESI protocol introduces only a single additional transient state EI^A, which is transitioned to during the E→I transaction (see Figure 4.7). In this transient state, we see that it behaves identically to the state MI^A from MSI, which is expected as E state is essentially a special case of M. We again generated the Murphi description for the generated MESI protocol and successfully verified it for correctness.

We had perform a slight modification to the generated Murphi description to test the MESI protocol, because we now have two possible states with write permissions (M & E). By default, the Murphi code we generate checks (on each iteration) that when one cache is in state M there is no other cache also in state M. However, we could encounter a scenario where once cache is in state M and another in state E, which would violate the SWMR invariant. ProtoGen handles protocols with multiple write states elegantly by inspecting the protocol to determine which states have read and write permissions [16]. It can then assign a permission to every state (including transient states) and impose a generic invariant to assert that only a single cache can have write permissions. We unfortunately did not have the time to implement this feature, and thus chose instead to implemented it manually. However, with this manual change we were still able to verify correctness of the generated protocol.

Finally we tested the MESI protocol with stalling and non-stalling optimizations. We again experienced the same limitation like in MSI, however we did discover an interesting optimization relating to the I→E transaction (see Figure 4.8). In state IS^A, we can optimize messages relating to the logical end states E and S(Fwd_GetM & Fwd_GetS). When we receive a Fwd_GetM we know that we must have been issued with E state

from the directory and thus can transition to state IS^AI and wait for the GetM_Ack_D. Understanding the Fwd_GetS is more complicated, because we don't yet know if the Fwd_GetS is received from the directory from state S or E, thus in state IS^AS we must handle both responses from the directory GetS_Ack and GetM_AckD to transition to state S. The transient state IS^AS also has logical end state S, and thus we could potentially handle racing Inv messages, but due to the limitation in the specification, we are unable to realise this optimization. Yet, we still managed to successfully verify the optimized non-stalling protocol with Murphi.

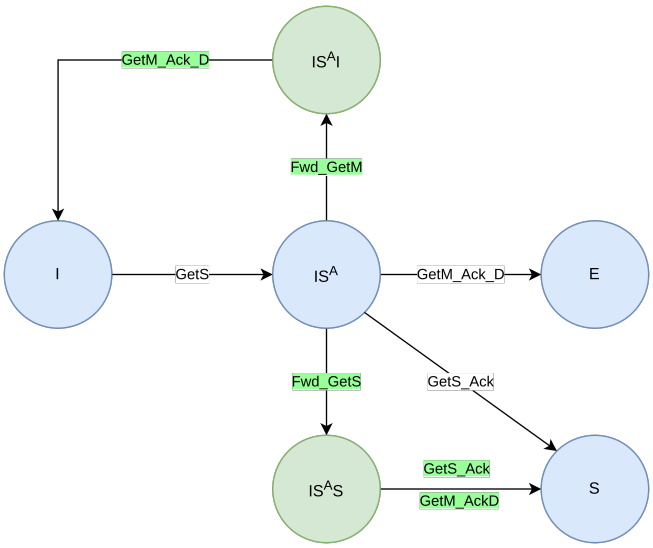

Figure 4.8: MESI: I→E non-stalling transaction

4.4 Summary

Although ProtoGen-MLIR v2 continues to lack some optimization potential when compared to ProtoGen, it remains a significant step forward when compared to v1. In Figure 4.9 we present a comparison of the optimization capability of v1 with v2, which highlights v2's significant advance in its optimization capability. With v2, we can now compile any MOESI protocol into an equivalent stalling protocol without the need of atomic transactions, while v1 was only capable of this with MI & partially with MSI. But by far the greatest improvement to v2 is the addition of non-stalling optimizations, which were completely absent in v1. However, our specification and implementation of non-stalling optimizations in v2 is still not as capable as the optimizations in ProtoGen, hence we have marked them as *Partial Compilation*.

	Successful Compilation		
Key	Partial Compilation		
	Unsuccessful Compilation		
	Optimization	**v1**	**v2**
MI	Atomic		
	Stall		
	Non-Stall		
MSI	Atomic		
	Stall		
	Non-Stall		
MESI	Atomic		
	Stall		
	Non-Stall		

Figure 4.9: Comparison of ProtoGen-MLIR v1 to v2

Lastly we also compared compilation times of ProtoGen-MLIR with ProtoGen and observed a significant performance improvement. With all protocols compiled with ProtoGen-MLIR we observed compilation times of $<$100ms, while ProtoGen executes in approximately 1-2s. With ProtoGen-MLIR we actually do much more processing and manipulation of the IR than ProtoGen, but since our implementation is written in C++ compared to Python we see as significant speed increase. Although our performance is better, the compiling performance of ProtoGen is more than adequate, thus the increase in our performance is largely inconsequential.

Chapter 5

Conclusions

5.1 Summary

This report presents ProtoGen-MLIR v2 which is a significantly improved (re)implementation of ProtoGen-MLIR v1 that we presented in Part 1 of this project. We aimed to significantly extend the optimization capability of v1 by fully completing stalling optimizations and by implementing additional non-stalling optimizations. During this process, we encountered many critical challenges which led us to re-implement several key components of the compiler in order to proceed, including the IR, frontend & CodeGen components.

In Chapter 3 we outlined the significant changes that were implemented as part of ProtoGen-MLIR v2 to allow us to optimize cache coherence protocols. In Section 3.1 we discussed the new FSM dialect and its significantly improved type system, expressiveness and abstractions. We further go on to explain how these features assists us in implementing optimizations. In Section 3.2 we detailed our reasoning for how we can implement protocol optimizations by understanding how a cache can deduce the ordering of requests based on what messages it receives. We then applied this reasoning to produce a declarative specification for each class of optimization (stalling & non-stalling), which we then used to assist our implementation for each optimization. Finally, in Section 3.3 we presented a generic implementation for our Murphi backend, which is designed to be generic, modular and easily extensible which allows us easily support additional dialects and language features.

In Chapter 4 we evaluated the implementation of our compiler using a subset of MOESI protocols (MI, MSI & MESI). For each protocol, we compiled it through three different levels of optimization (atomic, stalling & non-stalling) and for each compilation we successfully verified the optimized protocol with the generated Murphi description. Our results showed that when compiled as an atomic or as a stalling protocol, we were functionally identical to ProtoGen. However, our non-stalling optimizations were limited to transactions with only a single intermediate transient state, and thus we did not obtain the most optimal non-stalling protocol when compared to ProtoGen.

5.2 Reflections

As it turns out, implementing a compiler is a difficult task. While completing this project we encountered many significant challenges and delays that influenced its completion. In this section we outline the areas which were particularly challenging or complex and reflect on how we could have planned better to complete the project in time.

The primary challenge of this project, was designing and implementing a suitable representation with MLIR. MLIR has grown into an extremely large and complex project, yet it lacks advanced developer guides or 'best practices' from which to learn. Instead, to gain an understanding of the project we analyzed other MLIR compiler implementations (Circt & Tensorflow) to discover how to design and implement a suitable IR. Moreover, this project was written entirely using C++, which is a language we had not used previously and proved to be uniquely challenging to work with. We often struggled with building and linking issues, which would sometimes take days to resolve.

Much of the delays encountered while implementing this project stemmed from poor decisions taken during the development of v1. In order to have a working version of the project ready in time for the deadline many shortcuts were taken, such as the design of the IR. However, when we resumed the project it was clear that many aspects of the previous implementation were inadequate and which resulted in an almost complete rewrite of the entire ProtoGen-MLIR compiler. Some components, such as the frontend, were broadly similar and weren't challenging to implement. However, significant time was spent on the optimization and CodeGen components, which were entirely novel to ProtoGen-MLIR v2.

In reality it would have been difficult to predict and plan for these delays, since we only realised the problem until (at least partly) the project had been implemented. Yet we still feel that the project could have been planned better if we:

- Spent more time familiarizing ourselves with MLIR and good IR design principles, instead of creating an ad-hoc representation.
- Planed for extensibility and maintenance initially, rather than have to re-design or re-implement components to accommodate new features.
- Worked on components of the compiler in a more logical order. For example, we began implementing the CodeGen component before the IR was finished resulting in many further changes.

5.3 Future Work

ProtoGen-MLIR v2 has taken a big step forward compared to v1 and we believe has conclusively proved that using compiling techniques paired with MLIR are capable of optimizing cache coherence protocols from stable state definitions. We also believe strongly that we have laid significant and strong foundations upon which we can develop and extend the potential of ProtoGen-MLIR further. The key areas for future development are:

Completing Non-Stalling Optimizations. As discussed earlier, ProtoGen-MLIR does not implement the full non-stalling optimization potential shown in ProtoGen and is limited to just transactions with a single transient state. As future work, we aim to develop a new optimization specification, which accounts for multi-hop transactions and implement this optimization into the compiler.

Support for consistency directed protocols. ProtoGen-MLIR is designed exclusively to work with *consistency agnostic* MOESI protocols, which need not consider the consistency model of the CPU. However, heterogeneous architectures typically use *consistency directed* protocols, that do not necessarily maintain SWMR, but will propagate writes based on the consistency model. These kinds of protocols can also be specified atomically, and optimized similarly to MOESI protocols, although the reasoning will be different. Since adoption of heterogeneous architectures is rapidly expanding, it is important to support this class of protocol.

Implement additional backend targets. ProtoGen-MLIR supports only a single backend (Murphi), which is used to verify the correctness of the generated protocol. However, other backends could be added to evaluate the performance of the generated protocol. Gem5 is a processor simulator, which can simulate the execution of a protocol and measure valuable metrics about a protocol's performance, such as latency, bandwidth, throughput etc. Theo Olausson successfully implemented a Gem5 backend for ProtoGen [15] and gained significant insight into the characteristics of cache coherence protocols. Thus, the addition of such a backend would greatly improve the usability of ProtoGen-MLIR.

Bibliography

[1] Inja: A template engine for modern c++.

[2] A general introduction to cache coherence, Oct 2020.

[3] Alexey Bader, James Brodman, and Michael Kinsner. A sycl compiler and runtime architecture. *Proceedings of the International Workshop on OpenCL*, 2019.

[4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, and et al. Fallout. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[5] 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.

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

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

[8] 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.

[9] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, and et al. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[10] 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.

[11] Llvm. llvm/circt.

[12] Sharad Malik. Hardware verification: Techniques, methodology and solutions. *Tools and Algorithms for the Construction and Analysis of Systems*, page 1–1.

[13] Atif Memon. *Advances in computers*. Academic Press, 2013.

[14] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David Allen Wood. *A Primer on memory consistency and cache coherence*. Morgan & Claypool Publishers, 2020.

[15] Theo Olausson. Generating gem5 cache coherence controllers with protogen, 2021.

[16] 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.

[17] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. Understanding memory and thread safety practices and issues in real-world rust programs. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[18] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. *High performance computing: Modern systems and practices*. Morgan Kaufmann, 2018.

[19] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *C/C++ Users Journal*, 2005.

[20] Petr Vesely. Protogen-mlir: an mlir compiler for cache coherence protocols, 2021.

Appendix A

PCC Specification of MI Protocol

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

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

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

}
```

Appendix B

PCC Specification of MSI Protocol

```
# 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 = Ack(Put_Ack, ID, PutS.src);
       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

PCC Specification of MESI Protocol

```
# 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;
        }
    }

}
```