ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications

Nicolai Alexander Oswald

Master of Science by Research
Institute of Computing Systems Architecture
School of Informatics
The University of Edinburgh
2018

Abstract

Non-atomicity of cache coherence transactions in modern multicore processors leads to messages interleaving with other conflicting cache coherence transactions. Hence, designing correct directory cache coherence protocols is a complex and difficult task architects face. ProtoGen is an automated tool that overcomes this design challenge. It takes the description of a directory cache coherence protocol with atomic transactions and generates the corresponding protocol for a multicore processor with non-atomic transactions. The output of ProtoGen are finite state machines for the cache and directory controllers, including all the transient states required to handle concurrent transactions.

Using ProtoGen, the complete MSI, MESI, and MOSI protocols have been generated based on their stable state protocol specifications. The correctness of the generated protocols for safety and deadlock freedom was verified by a model checker. By reducing the number of stalls and merging logically identical states, ProtoGen generates protocols that are identical or better than manually generated protocols. Furthermore, to prove its generality, ProtoGen has been used to generate a non-atomic implementation of the relatively unconventional TSO-CC protocol.

Acknowledgements

I want to thank Professor Vijay Nagarajan and Professor Daniel J. Sorin for supervising my thesis. Furthermore, I want to express my deepest gratitude to my family and friends.

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.

(*Nicolai Alexander Oswald*)

Preface

For my master thesis I have developed the ProtoGen tool. The ProtoGen tool is based on my own description language and algorithm, which automatically generates correct non-atomic cache coherence protocols from an atomic cache coherence protocol specification. The correctness of the generated protocols is verified using the Murφ model checker. Based on my work, I published a paper together with my supervisor Professor Vijay Nagarajan and Professor Daniel J. Sorin. The title of the paper is ”ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications” and it was published in the ”The 45th International Symposium on Computer Architecture, ISCA 2018” [1]. The chapters 3, 4, 5 and 6 in this thesis are taken from this publication.

Table of Contents

Chapter 1

Introduction

It is notoriously hard to design correct directory cache coherence protocols for multicore processors in the absence of atomic cache coherence transactions. Cache coherence protocols are often presented using only atomic transactions. Although this representation makes it easy to understand a protocol's basic behaviour and ideas, it also makes the protocol seem misleadingly simple, as cache coherence protocols have only a small number of stable states.

Due to prior analysis related to the scalability of different directory protocols this work focuses on cache coherence protocols based on the reply-forwarding directory protocol [2]. Directory based cache coherence protocols scale better in large systems as they require less bandwidth than snooping based protocols [3]. The available bandwidth of modern network-on-chip interconnects scales worse than linear with an increasing number of participants [4].

1.1 Atomic Transaction Cache Coherence Protocols

In literature cache coherence protocols are often described based on atomic transactions. An atomic cache coherence transaction is triggered by an access (load, store or replacement). It encompasses all state transitions that have to be performed by the controllers that are affected by a cache block access. As shown in Figure 1.1 for the example of a store access, the cache controller, depending on its current state, must send a request to the directory, which then either responds to the request or forwards it to one or multiple remote caches. Whether the directory responds to the requestor or forwards the request to one or more remote caches depends on the directory's auxiliary states and the type of

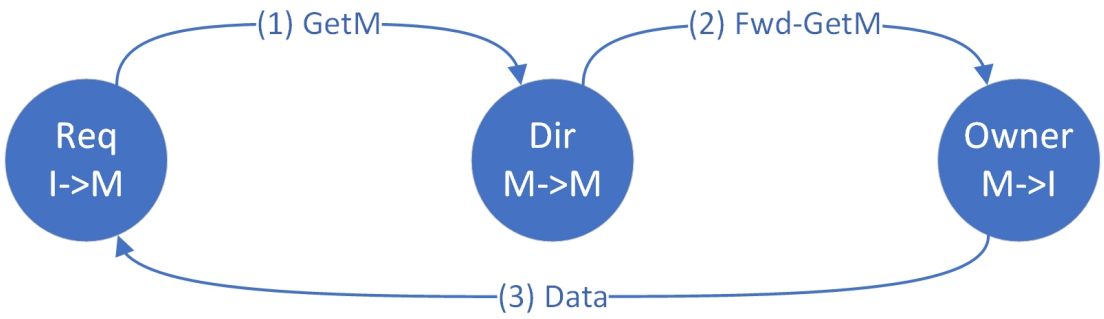

Figure 1.1: Store access cache miss at requestor triggers a GetM request to be sent to the directory. Based on its auxiliary states the directory determines the current owner of the cache block and forwards the request. The remote owner responds to the requestor with data.

request. Auxiliary states are used by the directory to track the state of a cache block in the caches. If a request has been forwarded the receiving cache has to serve it.

Tables 1.1 and 1.2 show the atomic stable state directory cache coherence protocol (SSP) specification of the MSI protocol. For the cache controller the behaviour in case of an access or an incoming remote request (e.g. GetM, Invalidate) has to be defined. The directory controller has to serve an incoming request resulting from an access miss at a cache controller by either responding or forwarding it. Furthermore, based on the request, it updates its auxiliary states tracking the state of the cache controllers.

For most textbook protocols only the stable states are described. Although this representation is minimal with respect to the protocol's functionality, these protocols cannot be directly implemented in systems that do not support atomic transactions such as modern multicore processors. An atomic transaction requires all transitions at the involved controllers to complete before a further transaction can be performed. Therefore, a cache coherence transition at a controller appears instantaneous ruling out any interference between multiple cache controllers when accessing a cache block.

For example, to support atomic transactions, a cache can lock the entire interconnect until the transaction has completed. By locking the interconnect, races of different cache coherence requests to the directory can be prevented. The SSP relies on the blocking of concurrent transactions. Disallowing races and therefore multiple concurrently pending cache coherence transactions has a negative effect on the overall system performance as modern systems use scalable non-atomic interconnection networks to leverage possible concurrency.

Table 1.1: Specification of Cache in Atomic MSI Protocol

	Load	Store	Eviction	Receive Forw. GetS	Receive Forw. GetM	Receive Invalidation
I	send GetS to dir, receive Data / S	send GetM to dir, receive DataNoAck or Data and Acks / M				
S	hit	send GetM to dir, receive Data and Acks / M	send PutS to dir, receive PutAck / I			send Ack to requestor / I
M	hit	hit	send PutM to dir, receive PutAck / I	send Data to requestor and Dir / S	send Data to requestor / I	

Table 1.2: Specification of Directory in Atomic MSI Protocol

	Receive GetS	Receive GetM	Receive PutS	Receive PutM
I	send Data to requestor, add requestor to Sharers / S	send Data to requestor, set Owner=requestor / M		
S	send Data to requestor, add requestor to Sharers	send Data to requestor, send Invalidations to Sharers, set Owner=requestor, clear Sharers / M	send PutAck to requestor, remove requestor from Sharers	
M	forward GetS to Owner, receive Data from Owner, add requestor and Owner to Sharers / S	forward GetM to Owner, set Owner=requestor		send PutAck to requestor / I

1.2 Non Atomic Transactions

An SSP as given in Tables 1.1 and 1.2 assumes atomic transactions. However, to leverage the concurrency allowed by modern scalable interconnects, directory cache coherence protocols have to be designed for a non-atomic system model. An access miss in one stable state requires the cache to perform a cache coherence transaction. By performing the cache coherence transaction, the cache eventually acquires the cache block in another stable state that allows the access to complete once the cache coherence transaction has finished. A cache coherence transaction consists of multiple steps such as sending a request, possibly serving a forwarded request of another transaction and waiting for responses. For each step interleaving cache coherence transactions have to be considered when designing a protocol to avoid potential conflicts. Handling the concurrency correctly is challenging. Herein many publications from recent years agree:

- ”The coherence problem is difficult, because it requires coordinating events across nodes” [5]
- ”... coherence protocols are notoriously difficult to design and implement correctly” [6]
- ”... directory-based cache coherence protocols are notoriously complex” [7]
- ”... designing and verifying a new hardware coherence protocol is difficult” [8]

1.2.1 Racing Transaction Example

To understand the race between concurrent transactions better, Figure 1.2 shows a possible interleaving of two concurrent cache coherence transactions. At step **(0)** both CPU cores want to store to the same cache block. However, in both caches the cache block is maintained in the I(nvalid) state. In the I state a cache does not possess a valid copy of the cache block. Therefore, the store accesses of the cores miss in their local caches at **(0)**. As defined in Table 1.1 for the store to complete, a cache has to perform a cache coherence transaction from the I state to the M(odified) state. The transaction encompasses the sending of a GetM request to the directory and the reception of one or more responses. The number of responses required for a transaction to complete depends on the state of the cache block when the request is received by the directory.

At **(1)** each cache sends a GetM request to the directory. Both requests race in the interconnect to the directory. The directory orders the transactions based on the

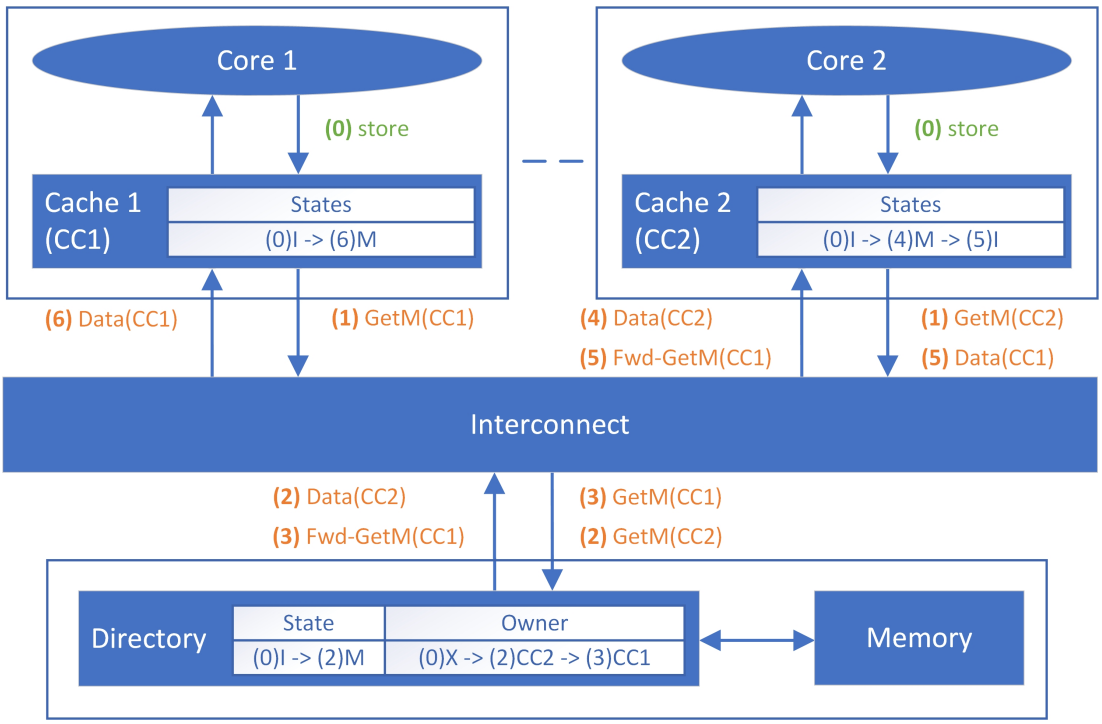

Figure 1.2: Racing Transactions

observed sequence of requests. The cache coherence protocol ensures that the ordering of transactions related to the same cache block performed at the directory is globally obeyed. This ensures that although the transactions are pending concurrently and are not atomic, they complete in a logically atomic way. The observation that transactions are serialized at the directory lead to the development of the ProtoGen algorithm, which is described in Chapter 5.

Since the directory receives the GetM request of cache 2 **(2)** before to the request of cache 1 **(3)**, the transaction of cache 2 is ordered before the transaction of cache 1. As defined in Table 1.2 the directory responds to cache 2 with the cache block, updates the owner field and changes its local state of the cache block to the M state. Upon reception of the second GetM message at time **(3)** the directory remains in the M state, forwards the GetM request to the current owner and updates the owner field. The owner field is a pointer, that always points to the last cache that eventually will have the cache block in the M state and therefore will have the most recent copy of the cache block. The ordering among all prior transactions that are still pending is maintained through the forwarded request messages. A cache can only serve a forwarded request, after its own pending cache coherence transaction has completed.

After cache 2 receives the response data message containing the cache block from the directory at **(4)** the state of the cache line changes to the M state and the pending transaction related to the store miss at **(0)** can complete. At **(5)** cache 2 receives the GetM request of cache 1 that has been forwarded by the directory. Cache 2 responds directly to cache 1 with a data message containing the cache block, before invalidating its local copy.

The response data message sent by cache 2 arrives at cache 1 at **(6)**. Since the data message contains the most recent copy of the cache block and no further responses are required to complete the transaction, the cache changes the state of the cache entry to M and updates the cache entry with the cache block from the message. This allows the pending transaction related to the store miss at cache 1 at **(0)** to complete.

The transactions of the two caches in this example cannot perform atomically, but also do not violate the SSP specification given in Table 1.1 and Table 1.2. However, if for example at the cache 2 the data response at **(4)** and the forwarded request at **(5)** arrive in reverse order, so that the forwarded request arrives before the data response, the SSP specification is violated. At the arrival of the forwarded request, the cache would be in the I state. As the SSP specification assumes transactions to complete atomically, the behaviour for a forwarded GetM request arriving in state I is not defined in Table 1.1. To consider this behaviour it is necessary to introduce transient states that monitor the completion of a pending transaction. In these transient states the arrival and handling of forwarded requests has to be considered. As it is complex and therefore error prone, to consider all forwarded requests that can possibly arrive in a transient state, automating this process is desirable.

1.2.2 Transient States

To consider all possible interleavings of concurrent transactions in relation to the number of stable states, a large number of transient states have to be introduced. The cache controller changes the state or at least a local auxiliary state, e.g. a counter, of a cache entry for every step in a coherence transaction. A single transaction step consists of the reception or the sending of a cache coherence message. As the reception or sending of a cache coherence message has to be tracked by the cache controller to monitor the completion of the transaction it transitions into a different cache coherence state. As a transaction step corresponds to a state transition, a transaction encompasses one or more state transitions.

If in the example described in Chapter 1.2.1 the store misses in cache 1, because the cache block is held in state I, it has to perform a cache coherence transaction into state M by sending a GetM request. After sending the request the cache controller transitions into a transient state waiting for the response. When the response is received the cache controller transitions into state M. The number of transient states traversed by a transaction depends on the number and type of responses. Additional transient states are required to cover transaction interleavings. For example, the relatively simple MSI directory cache coherence protocol given in Table 6.1 has seventeen transient states but only three stable states.

The large number of transient states required makes the manual protocol design error prone. It is very easy for an architect to forget a possible transaction interleaving pattern, resulting into the arrival of an, in the current state, unexpected forwarded request. Furthermore, the architect can make mistakes in handling incoming messages correctly by for example transitioning into a logically wrong state. Finally, the architect might simply forget to update an auxiliary state at one of the many transitions. Such a bug can affect the correctness of the cache coherence protocol. Furthermore, finding bugs in cache coherence protocols is a non-trivial problem. Model checkers struggle to verify highly concurrent cache coherence protocol implementations, because of the state space explosion problem [9]. Cache coherence bugs are a real world problem. Due to a bug in the CCI-400 cache coherent interconnect interface, Samsung had to deactivate cache coherence in their processors used in the Samsung Galaxy S4. This resulted into significant performance losses and increased energy consumption [10].

In addition to the previously discussed safety problems the performance gap between a manually and an automatically generated protocol must be considered. Due to the large number of possible transaction interleavings, the architect might decide to stall an incoming cache coherence request rather than serving it, because this would require even more transient states and might result into further possible transaction interleaving patterns. While stalling a forwarded cache coherence request, depending on the transaction ordering, does not necessary result into a violation of the protocol safety, it can have a negative performance impact. The stalled message is eventually served when the cache controller enters the pending transaction's terminal stable state, e.g. state M for an I to M transaction. It is tempting for an architect to conservatively restrict concurrency in some cases of transactions interleavings as the protocol complexity would grow significantly otherwise. For an architect it can be difficult and time consuming to reason about how a specific cache coherence message must be handled in the affected transient states.

While it is easy to understand the SSP specifications, it is a complex task to generate the transient states required to implement the cache coherence protocol correctly for non-atomic systems like most current multicore systems.

1.3 Contribution

ProtoGen is an automated tool that takes a SSP as input. The SSP is described in a custom domain-specific language (Chapter 4.1, Input DSL) that has been developed to enable the designers to express concurrency of cache coherence transactions accurately. From the SSP description ProtoGen generates finite state machines for the cache and the directory controller using the ProtoGen algorithm. The ProtoGen algorithm generates all transient states needed to maximize the cache coherence protocol concurrency, while properties like deadlock freedom and correctness of the SSP are preserved (Chapter 5, ProtoGen). The ProtoGen output finite state machines encompass the stable states defined in the SSP and the generated transient states. In addition, a back end has been developed that generates an input file for the Murφ model checker based on the ProtoGen output (Chapter 4.2, Output format). Figure 1.3 shows the ProtoGen tool workflow.

ProtoGen simplifies the cache coherence protocol design significantly. From the SSP description the cache and directory controllers are generated handling all incoming messages related to the racing transactions correctly. This is achieved by exploiting the insight that in case of directory cache coherence protocols the transactions related to the same cache block are serialized at the directory. Caches can infer the ordering of their own transaction in relation to other transactions from the received requests forwarded by the directory. Therefore, in every stable state the message types of the forwarded requests are required to be unique as otherwise the caches cannot infer the transaction serialization order. If in two stable states forwarded requests of the same type can arrive at a cache, ProtoGen introduces a new message type for one of the conflicting states to ensure that the sequence of transactions can again be inferred by the cache from the message type. ProtoGen implements these modifications of the SSP at the cache and at the directory controllers before generating the transient states required for concurrency. By guaranteeing consensus among the caches and directory about the order of racing transactions, ProtoGen can generate highly-concurrent and non-blocking cache coherence protocol implementations.

Prior work looked into high level synthesis (Chapter 2, Background and Related Work). These works have some similarities with ProtoGen with respect to the idea of

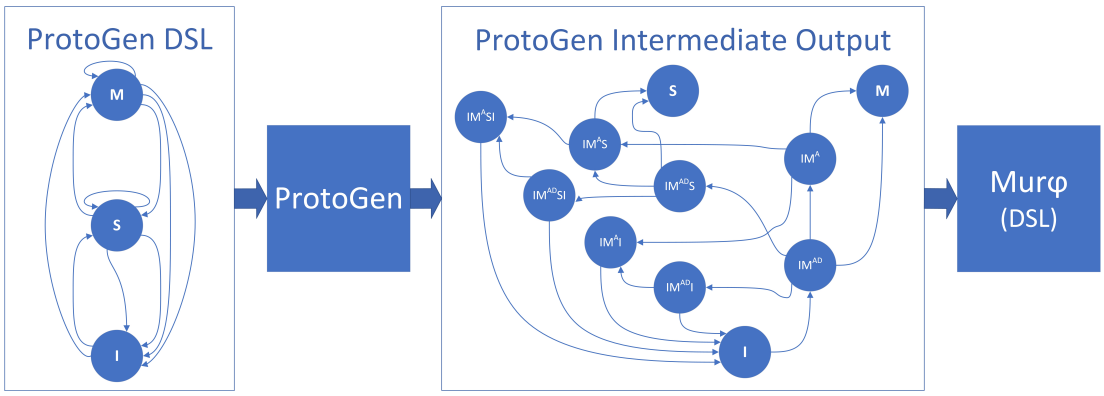

Figure 1.3: ProtoGen Workflow

high-level synthesis, but follow different approaches. To the best of my knowledge, my ProtoGen tool[1] is the first tool that can generate a complete non-atomic cache coherence protocol from an SSP specification. Prior approaches either only aid the design process or fail to generate a solution that in terms of optimization and performance can compete with ProtoGen. This thesis explains the ProtoGen tool, focusing on the ProtoGen algorithm and demonstrating its correctness. The correctness is verified using the Murφ model checker [11] for the generated directory cache coherence protocols.
Summarizing the contributions of my work:

- ProtoGen is an automated open-source tool that generates a complete directory cache coherence protocol maximizing its concurrency, which does not require atomic transactions, from an SSP that requires atomic transactions. Correctness is preserved, when transforming the atomic into an non-atomic transaction based cache coherence protocol.
- ProtoGen has been used to generate non-stalling MSI, MESI and MOSI protocols from SSP specifications. The correctness and deadlock freedom of the generated protocol has been verified by Murφ model checker. The input for the model checker is automatically generated from the ProtoGen output finite state machines.
- Furthermore, ProtoGen has been used to generate a concurrent implementation of the TSO-CC protocol [12] from its SSP specification. The TSO-CC protocol is a protocol that in contrast to the MSI, MESI and MOSI protocols by design only satisfies the TSO memory consistency model. The ability of ProtoGen to generate different types of cache coherence protocols shows the versatility of the approach.

[1] https://github.com/icsa-caps/ProtoGen

Chapter 2

Background and Related Work

In this chapter prior work related to ProtoGen is discussed. Firstly, schemes for simplifying the specification or implementation of complicated protocols or systems that are related to the ProtoGen domain specific language (DSL) are introduced. Since, the quintessence of ProtoGen is an algorithm that converts an atomic cache coherence protocol specification into an non-atomic implementation for non-atomic systems, prior work with similar objectives following different approaches is discussed next. Afterwards, techniques for synthesizing complete cache coherence protocols from incomplete specifications are discussed. Finally, new cache coherence protocols are discussed that have been developed to reduce the design and verification effort.

2.1 From Atomic Specification to Implementation

A number of areas like programming languages (e.g. transactional memory) and hardware synthesis(e.g. Bluespec [13] have put significant effort in developing techniques that aim to derive concurrent implementations from an atomic specifications.

A SSP consists of multiple atomic transactions transferring the system from one stable state into another. A transaction encompasses a sequence of steps that all complete without interfering with another transaction. Concurrent implementation of SSPs can be generated either with a blocking approach, using locks, or a non-blocking approach, that takes transaction interleavings into consideration. In general, non-blocking approaches permit a higher degree of concurrency [14]. Therefore, it should be the preferred method. However, the greater complexity, arising from the transaction interleavings, may lead to architects using blocking approaches to reduce the necessary design, implementation and verification efforts.

2.2 Description Languages

Teapot is a DSL for writing cache coherence protocols [15]. It was designed to address the complexity of implementing a cache coherence protocol. The complexity of engineering coherence protocols is considered to be discouraging for users to experiment with new and possibly more efficient protocols. Teapot features language constructs that are similar to hardware description languages like Verilog and are better suited to express protocol state machines than those of typical system programming languages like C. The ability to model a protocol using an event driven approach that is dependent on the current system state simplifies the design and the debug process significantly.

Teapot has been proven to considerably simplify the description of cache coherence protocols [16]. Furthermore, due to its language design, it allows to couple implementation and formal verification closely, increasing the confidence of architects in their protocol implementations. The ProtoGen DSL is conceptually very similar to Teapot. However, in contrast to the ProtoGen DSL, Teapot does not allow the user to directly describe varying event driven control paths within the same state description. Therefore, using Teapot, it is not possible to describe the SSP protocols on a transaction granularity. The user has to introduce transient states manually and describe the transactions step by step.

2.3 General Hardware Synthesis

Following the idea of having a custom DSL to describe atomic specifications, some work have used these for hardware synthesis [17; 18]. The Bluespec line of work [18; 19; 20; 21] uses a DSL to specify the behaviour of modules as a set of guarded atomic state transitions. An atomic state transition is a rule that is triggered by a specific condition, updating registers [19]. The Bluespec compiler generates a concurrent register-transfer level implementation for the given rules managing interactions between these. Races are prevented by inserting hardware arbitration and scheduling logic that ensures that conflicting rules perform mutually exclusive. The compiler ensures that a rule completes atomically, by producing a blocking implementation. Due to practical reasons, the compiler ensures that a rule completes in a single cycle. There is some work that focuses on lifting these restrictions [20; 21], but it is not documented whether these approaches are implemented in the Bluespec compiler.

2.4 Blocking Protocol Synthesis

By expressing every transaction of an SSP as a Bluespec rule, Bluespec can potentially be used to implement an SSP. The Bluespec compiler ensures that a transaction completes atomically by blocking the execution of any other transaction that would cause a conflict. For a cache coherence protocol using atomic transactions, Bluespec identifies a conflict among all transactions using the same directory in disregard of the affected cache block. While, by this approach, all transactions are completely serialized, it can limit the system performance significantly.

Atomic coherence [22] follows a similar approach as Bluespec, by using a hardware mutex to serialize accesses to the same cache blocks. Before a coherence request can be issued, the mutex has to be acquired. Like in case of Bluespec the implementation relies on blocking, making the coherence protocols become simpler and protocol extensions straightforward. However, to minimize the performance disadvantages resulting from the blocking implementation, low-latency optical mutexes were used.

2.5 Non-Blocking Protocol Synthesis

A case study shows that non-blocking cache coherence protocols can be naturally expressed in Bluespec [23]. The study shows that when using Bluespec, little verification effort is required for the hardware implementation, if the protocol has been previously verified at a rule level. Furthermore, it shows that the synthesized cache coherence controllers were able to meet the timing constraints. However, the input cache coherence protocol for Bluespec was a correct and complete non-atomic MSI protocol. Transient states and transitions required for concurrency were part of the Bluespec input description. While ProtoGen generates a correct non-atomic protocol implementation from the SSP as input, Bluespec was given a non-atomic protocol. The output generated by Bluespec has always the same degree of concurrency as the input.

While Bluespec is a general synthesis tool, ProtoGen is limited to synthesizing directory based cache coherence protocols. However, ProtoGen is the first tool that can generate a non-blocking and non-atomic cache coherence protocol from an SSP. ProtoGen achieves this, by exploiting domain specific knowledge about the functionality of directory protocols.

2.6 Coherence Protocols via Program Synthesis

TRANSIT [24; 25], which is inspired by program synthesis approaches like sketching [26], uses so called concolic snippets, which are execution fragments, to describe the system partially. A concolic snippet can contain concrete and symbolic values. The synthesis engine completes holes in an extended finite-state-machines skeleton by inferring guards and updates from the given snippets. By synthesizing the holes independently, TRANSIT avoids the state space explosion that would result from synthesizing all holes at once. A model checker analyzes the generated protocol with respect to given invariants. If an invariant is violated, the model checker produces a counter example. The architect then creates a new snippet that describes the correct system behaviour for the counter example scenario. Afterwards, TRANSIT again synthesizes a new protocol and tests it. The iterative protocol specification refinement by providing new concolic snippet continues until the model checker does not find any violated invariant.

The goal of VerC3 [27] is also to synthesize holes in a protocol skeleton. Given only correctness specifications VerC3 applies a dynamic programming based method to automatically synthesize the holes. VerC3 does not require the architect to provide any other input such as example traces. The dynamic programming method prunes inferred failure candidates, exploiting the fact that typically only few transitions are required to reach an erroneous state. The pruning of failed candidates significantly reduces the methods search space. However, due to state space explosion, the number of holes VerC3 managed to synthesize for a given MSI SSP description was limited to 12 out of 35 possible.

While the above approaches try to complete holes in a protocol skeleton, ProtoGen refines a given SSP specification to an equivalent concurrent implementation. Designing an SSP and verifying its correctness is less complex than designing a concurrent implementation of the SSP. Furthermore, due to the SSP's atomic transactions the state space that needs to be explored by the model checker is small in comparison to a concurrent protocol implementation. ProtoGen exploits domain knowledge to generate a high-performance non-blocking protocol from the SSP. Therefore, ProtoGen avoids the state explosion problem of VerC3 and the description of concolic snippets required by TRANSIT.

2.7 Complexity-Aware Coherence Protocols

Several new approaches have been developed to reduce the design and verification effort of cache coherence protocols. One approach used by DeNovo [7] and VIPS [28] is to exploit data-race-free cache coherence protocol models to minimize the number of transient states. Another approach is Fractal coherence [29; 30], which is a methodology to design formally verifiable cache coherence protocols. Protocols that have been designed to be fractal in behaviour can be formally verified, while avoiding the state space explosion problem.

In contrast to the these approaches that introduce new protocols, ProtoGen is an algorithm that can generate a non-atomic and non-blocking protocol from a given SSP specification.

Chapter 3

Intuition for ProtoGen [1]

Before delving into the low-level details of ProtoGen, we first present a high-level explanation of how it works. As part of this explanation, we introduce some background material. Where possible, we adopt the terminology and notation used in Sorin et al.'s primer on consistency and coherence [31].

3.1 System Model and Terminology

ProtoGen is designed for multicore processors with flat directory cache coherence protocols. Thus far, we assume that the protocols use a subset of the well-known MOESIF states. Furthermore, we assume typical coherence requests to either increase permissions (with a "Get" or "Upgrade") or decrease permissions (with a "Put"). These protocol assumptions are likely more conservative than necessary, but we have not yet explored protocols with other states or request types.

Each core can have one or more levels of private cache, and there is a last-level cache (LLC) that is shared by all cores. For simplicity, we describe systems with one level of private cache, but more levels are possible. The directory state is colocated with the LLC. We make no assumptions about the interconnection network, regarding either topology or whether point-to-point ordering is enforced.

For clarity, we now define several terms that we use throughout this paper. We consider every cache block and every directory entry to have a *state* that consists of its *coherence permission state* and possible *auxiliary state*. Example coherence permission states are S(hared) and M(odified), and they reflect a cache's ability to access a block or a directory's knowledge about a block. Auxiliary state is any state that does not describe permissions, but is often necessary to correctly transition between

two coherence permission states. A directory entry's auxiliary state might include the ID of the current owner of the block and the set of caches currently sharing the block (the sharer list). A cache block's auxiliary state might include a counter used to keep track of incoming acknowledgments.

If a core wants to perform an *access* (load, store, or replacement) that cannot be satisfied by its cache, its cache initiates a coherence *transaction* to obtain the desired cache block in the appropriate coherence state or evict it. A transaction consists of:

- an initial *request* message such as GetShared (GetS), GetModified (GetM), or PutModified (PutM);
- zero or more *forwarded* requests—such as Forwarded-GetM or Invalidation messages—that are sent by the directory in response to incoming requests;
- one or more data *responses* or *acknowledgments*. Both the directory and the cache controller may respond to incoming coherence messages with data and acknowledgment messages. For example, if a directory with a block in state I(nvalid) receives a GetS request, it will respond with Data; and
- state *transitions* in response to incoming coherence messages. A cache may change its block state and a directory may change the state of its entry; these state changes can include the coherence permissions and/or the auxiliary state. For example, if a directory receives a GetS from a cache for a block in state I, it will change the directory entry's coherence permission state and its auxiliary state for tracking sharers.

In our examples, we often consider a transaction from the point of view of a given cache that initiates the transaction with a request. We refer to that transaction as the cache's "own" transaction. If that cache receives a message that is part of a transaction initiated by another cache, we refer to that as an "other" transaction. Similarly we refer to coherence messages as "own" (e.g., own GetM) and "other" (e.g., other PutS).

Lastly, we consider each transaction to start or end a *coherence epoch*, a window of time during which a cache has coherence permission to a block. For example, a cache issues a GetS request to start its own transaction that, when complete, will begin a Shared epoch at that cache. This epoch will end either when: (a) the cache completes another transaction to change its permissions, or (b) another cache performs a transaction that affects the cache's permissions.

3.2 Definition of Coherence

ProtoGen generates coherence protocols that adhere to certain safety properties. We use the common definition of coherence, which consists of two invariants. First, for any memory location at any given time, there is either a single writer or zero or more readers. This invariant is often referred to as SWMR, and it means we can divide a block's lifetime into epochs, during each of which there is either a single writer or zero or more (multiple) readers. Second, the value of a location at the start of an epoch is the same as the value of the location at the end of its last write epoch. This invariant is often referred to as the data-value invariant.

Intuitively, these epochs can be in physical time, but it is also correct for them to be in logical time, in which the two invariants combine into one single invariant, i.e., sequential consistency for every memory location [32]. A discussion of logical time correctness is beyond the scope of this paper, but numerous protocols have been developed that rely upon it [33; 34; 35]; the benefit of per-location sequential consistency is that it can enable greater concurrency.

3.3 Big Picture

ProtoGen starts with a SSP and creates a complete protocol with transient states—generating both coherence permission and auxiliary state—without requiring an atomic system model that can guarantee physically atomic transactions. ProtoGen requires that the SSP descriptions for the cache and directory are correct and complete for an atomic system model; for example, the SSP must correctly enforce SWMR.

In this discussion, we focus on the generation of transient coherence permission states, because the issues involved in generating transient auxiliary state are either the same or far simpler. Intuitively, auxiliary state is either intimately tied to the coherence state (e.g., a directory entry in S keeps auxiliary state to track the sharers) or is "bookkeeping" state that is uninvolved in races (e.g., a counter for a cache block to track how many acknowledgments it has received).

To understand how ProtoGen works, first consider a simplistic protocol without physically atomic transactions but with no concurrency either and hence logically atomic transactions (i.e., for any given block of memory, only one coherence transaction can be active at a time). Generating transient state specifications for such a protocol is simple; each transient state just corresponds to a step in the transaction. For example,

a cache issuing a GetS request to transition from I to S has a single transient state, which we will call IS, that reflects it has issued the request but has not yet received the response; once it receives a Data response, it will transition to S.

The challenge for ProtoGen is handling multiple concurrent transactions for a given block. What happens when a cache in the middle of a transaction receives a forwarded request belonging to a concurrent transaction to the same block? In our above example, what happens when the cache that issued the GetS receives an Invalidation from another cache (via the directory) while in state IS? In directory protocols, the directory is the serialization point, so the key is for the cache to be able to deduce the ordering of transactions at the directory. In our example, the cache must deduce whether its own GetS was ordered at the directory before or after the other cache's transaction that led to the incoming Invalidation. Here, the cache can infer that its own GetS was ordered at the directory first; otherwise there is no reason for the directory to have forwarded the Invalidation. Thus, by simply looking at the incoming forwarded request, ProtoGen is able to deduce ordering.

But the SSP could have been written such that the same forwarded message could arrive in two stable states. To deal with this, ProtoGen preprocesses the input SSP to ensure that a given forwarded request can arrive at exactly one stable state (if the input SSP uses the same name for two forwarded request messages, ProtoGen creates a new name for one of them). This invariant allows for caches to reliably deduce the order in which transactions are serialized at the directory by simply looking at incoming forwarded requests. ProtoGen uses this ordering information to generate the cache and directory state machines as described in detail later.

Intuitively, ProtoGen creates transient states for the caches and directory so that these finite state machines always respect the ordering of transactions at the directory. By doing so, ProtoGen creates protocols that are guaranteed to enforce coherence.

Chapter 4

Using ProtoGen [1]

In this section, we discuss the input, output, and limitations of ProtoGen.

4.1 Input

The primary input to ProtoGen is a high-level specification of a stable state protocol (SSP) described in our domain specific language (DSL). Our DSL is similar in spirit to other previously proposed ones for coherence protocols, including Teapot [**?**] and SLICC [36]. The specification describes the protocol with respect to a single cache block, because the behavior of all blocks is identical, and it essentially contains the information in Tables 1.1 and 1.2, including:

- a list of stable coherence permission states, often a subset of the MOESIF states;
- the stable auxiliary state at each cache block and each directory entry;
- a list of accesses (loads, stores, replacements) and the requests they trigger;
- a list of possible coherence messages (requests, forwarded requests, responses, and acknowledgments) that can arrive in each stable state; and
- the transitions from one stable state to another stable state—including both coherence permission and auxiliary state—that occur as a result of incoming coherence messages.

The DSL enables the designer to define the structure of any machine (e.g. caches and directories) as well as its architectural behavioral specifications. To this end, the DSL supports standard data types like bool, int, and set, but there are also protocol-specific

data types. In particular, the DSL provides a special type called Data (to represent actual data in a block of memory). Every machine has two predefined internal variables: State (denoting the state of the machine) and ID (the ID of the machine).

Listing 1 shows a snippet of code that uses many of these data types in defining the cache structure (lines 2 through 7). In line 3, we have initialized State to I. In lines 5 and 6, we have defined and initialized two auxiliary states: *acksReceived* and *acksExpected*.

Our DSL also allows us to specify the behavior of the caches and directories in response to internal accesses and incoming coherence messages. Most of these transactions consist of a request and a response, and specifying these transactions is straightforward. However, there are two specification issues that are worth discussing.

First, some requests can lead to multiple possible transaction routes depending on the state of the block in other caches; our DSL allows the user to specify this. Consider the S to M transition (lines 14 through 45), in which a cache—upon receiving a store to a block in shared state—requests Modified access to the block. If the block is exclusively held by the requestor and not cached elsewhere, the requestor simply receives a single response from the directory to complete its transaction (lines 21 through 23). If, however, the block is held by one or more other caches, the requestor must wait for both the response from the directory (containing a count of the sharers) and the acknowledgments from all caches that had the block in state S (lines 25 through 44).

This example also leads us to the second issue: some transactions require the initiating cache to receive multiple *types* of messages to complete the transaction. Our DSL allows the user to specify that multiple messages must arrive. Coming back to our example, the two relevant messages are *GetM_Ack* (the message from the directory with the count), and *Inv_Ack* (invalidation acknowledgment). When the former arrives (line 25), our DSL allows us to set the auxiliary state *acksExpected*; when the latter arrives (line 34), *acksReceived* is incremented; when they are found to be equal, the transaction completes. [1]

[1] Due to races, an *Inv_Ack* can actually arrive before the *GetM_Ack*, which is the situation handled by lines 43-44.

```
// Machine definition
Cache {
    State = I;          // Cache initial state
    Data block;
    int[0..NumCaches] acksReceived = 0;
    int[0..NumCaches] acksExpected = 0;
} set[NumCaches] cache;
...

Architecture cache {
    ...
    // S to M transition
    Process(S, store){
        msg = Request(GetM, ID, dir.ID);
        reqNet.send(msg);
        acksReceived = 0;

        await{
            when GetM_NoAck:
                State = M;
                break;

            when GetM_Ack:
                acksExpected = GetM_Ack.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;
        }
    }
    ...
}
```

Listing 4.1: Snippets from ProtoGen DSL

Configuration parameters. ProtoGen also has inputs that control the nature of the protocols that it generates. One parameter controls whether the generated protocol is *stalling* or *non-stalling*. With the former, cache and directory controllers stall when they receive potentially racing requests, at the cost of performance (while still preventing deadlocks). With the latter, the generated protocol avoids stalling whenever possible at the expense of an increase in the number of transient states.

Another ProtoGen parameter controls whether the generated protocol allows for loads or stores to access a block in a transient state (e.g., a load to a block in a transient state between S and M), and this input affects the coherence invariant that the generated protocol guarantees. Allowing accesses to cache blocks in transient states can preclude a protocol from enforcing SWMR in physical time but is compatible with enforcing per-location sequential consistency.

4.2 Output

ProtoGen produces fine state machines (FSMs) for the caches and the directory including the transient states (containing information similar to Table 6.1). These FSMs are expressed in the same DSL and can be translated to any other format for specifying FSMs. Thus far, we have implemented a backend to the language of the Murφ model checker [11] and translation to other outputs like SLICC [36] or Verilog is future work.

4.3 Limitations

First, ProtoGen requires a correctly specified SSP as its input; it cannot automatically "correct" bugs in the SSP. Second, ProtoGen cannot generate new protocol actions not explicitly specified in the SSP. For example, it cannot infer how atomic read-modify-writes must be implemented without it being specified in the SSP; likewise, it cannot automatically deduce the protocol for an unordered network given an SSP for an ordered network. Third, ProtoGen does not specify how protocol actions must interact with the interconnect; it requires the user to manually define virtual channels and assign messages to channels. Finally, ProtoGen is limited to directory based protocols.

Chapter 5

ProtoGen [1]

Given an SSP, ProtoGen generates a highly concurrent directory protocol including the transient states. This process is explained step by step in this section, and we use a running example of an MSI protocol to illustrate each step. We first explain this process for generating the cache controller; at the end of this section, we discuss the minor differences involved in the process of generating the directory controller.

Unless otherwise noted, we focus on the coherence permission state, because incorporating the stable auxiliary state is usually trivial. We only discuss the auxiliary state when ProtoGen has to generate transient auxiliary state (e.g., state for a cache block that records to which other cache to send the data when its own transaction completes).

5.1 Preprocessing the SSP

Before generating any transient states, ProtoGen first preprocesses the SSP specification to ensure the invariant that a given forwarded request can arrive at exactly one stable state. If, in an input SSP specification, the same forwarded request can arrive in two stable states, ProtoGen creates a new name for one of the forwarded requests.

For some directory protocols, architects might find it natural to specify their SSPs in a manner that already satisfies the invariant. For example consider the SSP of the MSI protocol specified in Table 1. As we can see, each of the three forwarded requests—Fwd_GetM, Fwd_GetS and Invalidate—arrive at exactly one stable state: M, M, and S respectively.

On the other hand, consider a MOSI protocol. Architects might find it natural to specify its SSP such that a Fwd_GetS can arrive at both the M and O states (the relevant

Table 5.1: MOSI SSP: Before preprocessing

	Fwd_GetS
M	send Data to requestor / O
O	send Data to requestor

Table 5.2: MOSI SSP: After preprocessing

	Fwd_GetS	O_Fwd_GetS
M	send Data to requestor / O	
O		send Data to requestor

snippet of the SSP is shown in Table 5.1). In such a case, ProtoGen would rename one of the messages as shown in Table 5.2. If the directory receives a GetS and finds the block in O state, the directory would forward the new O_Fwd_GetS message.

To see *why* this renaming is necessary for ProtoGen, let us consider the following scenario. Consider a cache C_0 that holds a block in O state and wants to write to the block. Accordingly, it would send a GetM request to the directory and wait for a response. In the meantime, say there is a concurrent transaction to the same block: another cache C_1 wanting to read the same block issues a GetS to the directory. The key point to note here is that the directory will forward the GetS to C_0 irrespective of the order in which the two transactions serialize at the directory. But for ProtoGen to work, C_0 needs to somehow discover the order in which the racing transactions have serialized at the directory. It is for this very reason that ProtoGen performs the renaming: if C_0 were to receive a Fwd_GetS message it can now infer that its own GetM request must have "won the race"; if C_0 were to receive a O_Fwd_GetS on the other hand, it can infer that the other GetS request must have been serialized before its own GetM.

5.2 Step 1: Generate Initial State Sets

The key challenge that ProtoGen addresses is to generate cache controllers that correctly respond to incoming forwarded requests for a block in a transient state. But for this, ProtoGen must first know what forwarded requests can *potentially* arrive in a transient state.

It is worth noting here that transient states are local to a cache and not visible to the directory. The directory always sees any block in a cache as being in a stable state at all times; it forwards requests to a cache based on the state of the cache block as it sees it. Therefore, the set of forwarded requests that can arrive at a transient state is determined by the set of possible stable states in which the block can be seen at the directory (while the cache block is in the transient state).

ProtoGen keeps track of this information with a data structure called a State Set. ProtoGen creates one State Set for each stable state, and initially each State Set contains just its stable state. For an MSI protocol, the State Sets are initially {I}, {S}, and {M}, and we refer to them as the ***I***, ***S***, and ***M*** State Sets, respectively. (We use **bold** to distinguish a State Set from a stable state.) As ProtoGen generates new transient states, it adds them to one or more State Sets, as described below.

5.3 Step 2: Add Transient States in Absence of Concurrency

ProtoGen next adds the transient states required for non-atomic protocols in the absence of concurrency. If a transaction from stable state S_i to stable state S_j involves issuing a request and waiting for a single response, ProtoGen would add a single transient state that reflects the situation in which the request has been issued but the response has not yet been received. If the transaction involves multiple responses, ProtoGen would add a transient state for each response.

In our MSI protocol, a cache can initiate one of five transactions types, $I \rightarrow S$, $I \rightarrow M$, $S \rightarrow M$, $S \rightarrow I$, and $M \rightarrow I$. The transaction from I to S would lead to the creation of a transient state that we call IS, that reflects the situation when the cache has issued the GetS request to the directory and is awaiting a data response. When the data is received from the directory, the cache will transition from IS to S.

Recall that the transaction from I (or S) to M could happen via two routes depending on the state of the block at the directory: one route with a single data response (if the block is in I or M at the directory) and the second route with multiple responses. To account for this, ProtoGen creates transient states as shown in Table 5.3. The transaction from I to M would first lead to the creation of a transient state that we call IM^{AD} (potentially waiting for data as well as acknowledgments). If the response received consists of only data, the cache block will directly transition to M state. If

the response includes both data and a count of acknowledgments, then the block will transition to another transient state called IM^A (waiting for acknowledgments). When the last of the acknowledgments is received, the block will transition to M.

Table 5.3: Adding transient states (no concurrency)

	Store	DataNoAcks	Data + #Acks	Last Ack
I	send GetM to Dir / IM^{AD}			
IM^{AD}		M	IM^A	
IM^A				M

When ProtoGen creates a new transient state for a transaction from stable state S_i to stable state S_j, it adds it to the State Set for both S_i and S_j. This "duality" is because the transient state can behave like either S_i or S_j, depending on what message arrives (i.e., whether the message is one that could arrive in S_i or one that could arrive in S_j). In our example, state IS thus belongs to the State Set $\boldsymbol{I}$ and the State Set $\boldsymbol{S}$. At the end of this step, the State Sets for our MSI protocol are as follows:
$\boldsymbol{I}$=$\{I, IS, IM^{AD}, SI, MI\}$,
$\boldsymbol{S}$=$\{S, IS, SM^{AD}, SI\}$, and
$\boldsymbol{M}$=$\{M, IM^{AD}, IM^{A}, SM^{AD}, SM^{A}, MI\}$.

5.4 Step 3: Accommodating Concurrency

We now explain how ProtoGen accommodates concurrency by describing the generated cache controller behavior for when a forwarded request arrives in one of the transient states produced in Step 2. In this section, we explain how this is done for any given transient state. Later in Section 5.7, we present a global picture of how ProtoGen does this for *all* transient states.

A cache that has a block in any state in State Set $\boldsymbol{S_i}$, can receive any forwarded request that the SSP specifies is possible to arrive in stable state S_i. If the message arrives in that stable state, the block's state will immediately change to a stable state (or perhaps remain in S_i) as specified in the SSP. The more challenging scenario is if the forwarded request arrives in a transient state (belonging to State Set $\boldsymbol{S_i}$). In our example, an Invalidation arriving in stable state S is easy to handle; the block immediately transitions to stable state I. But what if the Invalidation arrives in transient

state IS? Because ProtoGen's preprocessing step ensures that any type of forwarded request can arrive only in a single stable state, ProtoGen can generate the new transient states without confusion.

Consider a transient state that is part of cache C_0's transaction T_{own} (triggered by an access A_{own}) from stable state S_i to stable state S_j. This transient state, which we call S_{ij}, is part of State Sets $\boldsymbol{S_i}$ and $\boldsymbol{S_j}$, and thus any forwarded request that can arrive at C_0 in state S_i or S_j can also arrive in S_{ij}. ProtoGen considers every one of these possible forwarded messages and how a cache in S_{ij} would respond to it. As explained in Section 3, the key is knowing in which stable state that forwarded request can arrive, and thus inferring the transaction ordering at the directory.

We now consider the two possible scenarios in which a forwarded request arrives at C_0 while it is in a transient state: either the forwarded request was ordered earlier or later than C_0's transaction T_{own}.

5.4.1 Case 1: Other Transaction Ordered Earlier

If the arriving forwarded request (R_{other}) is one that is associated with state S_i, C_0 can infer that the directory must have seen the other transaction before its own request (R_{own}), i.e., $T_{other} \rightarrow T_{own}$ at the directory. Moreover, C_0 can infer that when T_{other} arrived at the directory, the directory must have seen C_0 in state S_i. (At this instant C_0 is unable to infer whether or not its own request has reached the directory, but this is not needed.)

Let us assume that in the SSP the forwarded request R_{other} causes C_0 to transition from S_i to S_l. Upon receiving R_{other}, C_0 must: (a) respond to this request immediately; (b) transition to a transient state and logically restart its own transaction T_{own} as if starting from the stable state S_l. Let us now discuss the two issues in more detail.

Responding to forwarded request immediately. Once C_0 infers that R_{other} is part of an earlier transaction than T_{own}, it is critical that C_0 respond immediately to R_{other}. In particular C_0 must not wait for a response for its own request, as this could potentially lead to a deadlock.

To see why, consider two caches C_0 and C_1 both wanting to transition from S to M state, so both caches issue a GetM to the directory; let us refer to the two racing transactions as T_0 and T_1 respectively. Say the GetM from C_1 "won the race" and reached the directory first—i.e., $T_1 \rightarrow T_0$—but the response from the directory was delayed and instead C_1 received the forwarded GetM request that is part of T_0 first. C_1,

upon seeing the forwarded GetM can infer that its own request won the race (otherwise the directory would not have forwarded the GetM), and so it can choose to stall the incoming forwarded GetM request (more about stalling in Section 5.4.2). However, if C_0 also chose to delay the incoming Invalidate (that is part of T_1), this will lead to a circular dependency between T_0 and T_1 and hence a deadlock. This explains why C_0 must respond to the Invalidate immediately once it knows that the Invalidate is part of the earlier transaction.

Transitioning to a suitable transient state. Once C_0 responds to the incoming forwarded request, what state must it transition to? Logically, C_0 must appear as if it first transitioned to S_l and then performed the access A_{own}. But the problem here is that C_0 had already sent a request R_{own} (for access A_{own}) to the directory when in stable state S_i. Technically, the earlier request must be rescinded and a fresh request must be sent. However, for most directory protocols the same memory access in two stable states triggers the same request to the directory. In such a case—i.e., if the access A_{own} triggers the same request R_{own} from S_l also—there is no need for C_0 to rescind the earlier request and send a new request from S_l. It can simply move to a transient state that logically corresponds to the situation in which it has issued R_{own} from S_l and is waiting for a response. If such a transient state does not already exist, ProtoGen creates it. Say in the SSP the request R_{own} causes a transition from S_l to S_m. The transient state to enter would be S_{lm} between these two stable states, and it would have been identified in Step 2 (Section 5.3).

In our MSI protocol, consider the transient state SM^{AD} shown in Figure 5.1, which denotes that the cache has issued a GetM request to transition from S to M but has not yet received a response. If an Invalidation arrives, which is only possible in stable state S, the cache infers that the other transaction (involving the Invalidate) was ordered before its own transaction. ProtoGen looks at the SSP and discovers that an Invalidate received in state S would send the block to state I. Logically, ProtoGen must make the cache appear as if it first went to state I and then performed a store. But recall that the cache had already issued a GetM request to the directory, in response to a store in state S. Fortunately, the SSP reveals that a store in state I results in sending the same request, i.e., GetM. Thus, there is no need to rescind the earlier GetM. Further, ProtoGen would check the complete protocol generated until now to look for a transient state that denotes the situation in which the cache has issued a GetM in I and is waiting for a response; it is able to find IM^{AD}, and so ProtoGen transitions from SM^{AD} to IM^{AD}. However, in some situations the same

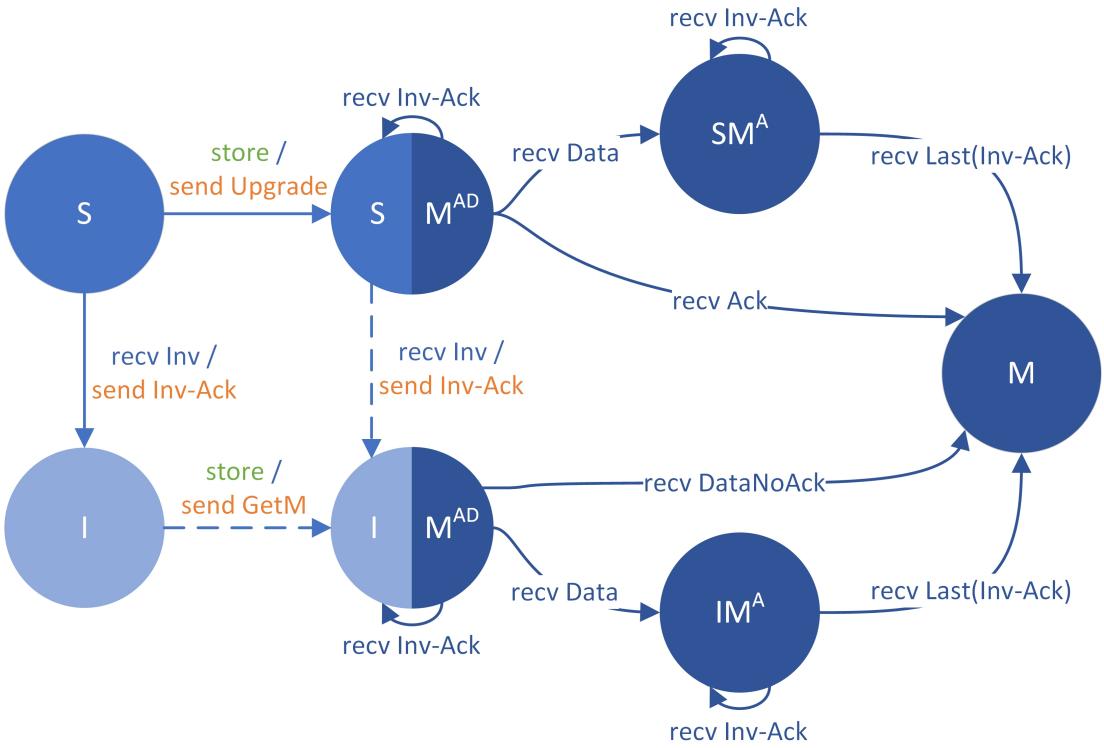

Figure 5.1: Cache S to M Transaction with $T_{other} \rightarrow T_{own}$. The shade of each state denotes the State Set(s) it belongs to. E.g., IM^{AD} is part of I and M State Sets

access could lead to two different requests when in two different stable states. Consider an MSI directory protocol that uses *Upgrade* requests. An Upgrade is a special type of request for transitioning from the S to M state. The difference between an Upgrade and a GetM is that the former does not need the data from the directory whereas the latter requires it. Consider two caches C_0 and C_1, both wanting to transition from S to M state, so both caches issue an Upgrade to the directory; let us refer to the two racing transactions as T_0 and T_1 respectively. Suppose C_1 won the race and reached the directory first. Logically, C_0 must now issue a GetM as the data it holds in its cache is invalid; it must obtain the new data written by C_1. Thus, this is an example where the same access (store) can lead to two different requests: Upgrade (if block in S) or GetM (if block in I). ProtoGen deals with this issue as follows. When the directory receives an Upgrade from C_0, it infers what happened because it knows that an Upgrade request cannot possibly arrive in state I. The directory reinterprets the Upgrade as a GetM, the request triggered by the same access (store) in state I.

5.4.2 Case 2: Other Transaction Ordered After

Once again, consider a cache C_0 that has issued a request to transition from S_i to S_j and is in a transient state that reflects that it has issued the request but has not received the response. If an arriving forwarded request (R_{other}) is one that is associated with state S_j, C_0 can infer that the directory must have seen C_0's own request (R_{own}) before the other transaction (i.e., $T_{own} \rightarrow T_{other}$ at the directory). Moreover, C_0 can infer that when T_{other} arrived at the directory, the directory must have seen C_0 in state S_j, which is why it forwarded R_{other} to C_0 in the first place.

Let us assume that in the SSP the forwarded request R_{other} causes C_0 to transition from S_j to S_k. Upon receiving R_{other}, C_0 must logically transition to S_k, but C_0 is unable to enter the stable state S_k yet because it is still waiting for a response to its own earlier request R_{own}. C_0 must honor the ordering $T_{own} \rightarrow T_{other}$, and there are three approaches to doing so.

Stalling. The most straightforward way to honor this ordering is by *stalling* C_0 and not having it respond to the forwarded request R_{other} until a response to its own request R_{own} has been received. Because the later transaction is the one that is stalled, there is no risk of a deadlock. However, stalling degrades performance in two ways. First, stalling will delay the start of the coherence permission epoch that R_{other} seeks to initiate. Second, stalling the controller will also block incoming coherence messages for other cache blocks.

Immediate Transition, Deferred Responses. ProtoGen can generate cache controllers that achieve greater performance by *not* stalling when the forwarded request R_{other} arrives. The key observation is that C_0 can process R_{other}—and avoid having it block its incoming queue—and any subsequent forwarded requests that arrive for the same block, knowing they are all ordered after T_{own}. C_0 enters a new transient state S_{new}. Because R_{other} causes a transition from S_j to S_k, the new transient state S_{new} is inserted into the State Set of $\boldsymbol{S_k}$. If the arrival of R_{other} in S_j would cause the sending of one or more responses, then C_0 *defers* the sending of these responses until it has completed its own transaction.

In a similar vein, if any subsequent forwarded request for the same block (say R_{other2}) arrives at C_0 in S_{new}, C_0 can infer that R_{other2} is part of a transaction that is ordered after T_{other}.[1] Therefore, upon receiving a forwarded request while in S_{new}, C_0

[1] If the network has point-to-point ordering then this is trivially guaranteed; forwarded requests will arrive at C_0 in the order the respective transactions were ordered at the directory. If point-to-point

behaves analogously to how it behaved when R_{other} arrived. It transitions to another transient state S_{new2} and, if the arrival of R_{other2} in S_k would cause the sending of one or more responses, then C_0 also defers sending those responses.

A cache that processes incoming forwarded requests instead of stalling may need transient auxiliary state to remember where to send the responses it defers. ProtoGen generates this transient auxiliary state when it generates cache controllers. It may appear that this state is unbounded, because each subsequent forwarded request could require some amount of state. In most directory protocols, however, the number of forwarded requests that a cache can receive is limited to three or fewer, before the cache block will reach a state (e.g., Invalid) in which it cannot possibly receive any new forwarded requests. If that is not the case, ProtoGen can limit the number of transactions that the cache can observe before its own transaction completes (in effect limiting the size of the transient auxiliary state) and simply stall the controller when this *pending transaction limit (L)* is reached.

Immediate Transition and Responses. The solution above, with deferred sending of responses, preserves the SWMR invariant in physical time, but more aggressive designs are possible. A design in which response sending is immediate, and not deferred, still preserves per-location sequential consistency and is compatible with common consistency models, including SC. As each forwarded request arrives, C_0 observes that a new coherence permission epoch for the block begins in *logical time*. (And all of these epochs are after the cache's permission epoch that resulted from T_{own}.)

This design is otherwise identical to the one above with immediate transitions. The only subtlety is that there are situations in which an immediate response is impossible because it requires sending a message whose contents depend on completing T_{own}. For example, if C_0 is awaiting data for T_{own} and the forwarded request demands the same data from C_0, then C_0 must defer sending the response until it has the data to send. In this case, C_0 uses auxiliary state to remember where to send what messages when sending becomes possible. ProtoGen creates that transient auxiliary state for the cache controller.

An Example with Immediate Transitions and Responses. Consider cache C_0 and the I to S transition of our MSI protocol. A load to a cache block in state I (a cache miss), leads to C_0 sending a GetS request to the directory and changing the block state

ordering is not guaranteed, the directory will have to serialize racing transactions by stalling the second until the first one completes, which again ensures this invariant.

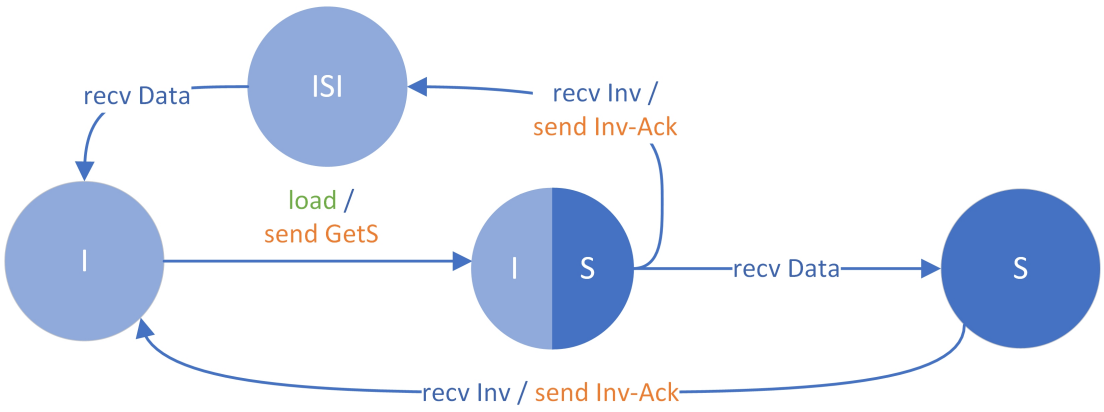

Figure 5.2: The I to S transition of the MSI protocol. The IS state belongs to the State Sets of ***I*** as well as ***S***, which is why it is shown in two shades. The ISI state, on the other hand, belongs to only the State Set ***I***.

to IS. Recall that state IS corresponds to the situation in which a GetS has been sent but the cache is awaiting a response. Hence, it belongs to both ***I*** and ***S*** State Sets because the cache block can be seen by the directory to be in I or S, depending on whether the GetS has reached the directory.

As shown in Figure 5.2, if an Invalidation arrives at C_0 in IS, then C_0 knows its own transaction was ordered before the transaction that triggered the Invalidation, because an Invalidation is only possible in stable state S. A stalling protocol would defer processing the Invalidation until C_0 received the response to its own GetS, but we consider the more aggressive protocol with immediate transitions and responses. To avoid stalling, C_0 must process the incoming Invalidation. ProtoGen generates a new transient state which we call ISI. This transient state gets added to State Set ***I***, because at this point it is clear to C_0 that its block is logically in state I (but still awaiting a data response for its request). C_0's own permission epoch has logically ended (even before the data arrived), and the other cache's permission epoch has logically begun. Also, C_0 immediately sends an acknowledgment to the cache that initiated the transaction that led to the Invalidation. When the data response eventually arrives to complete C_0's transaction, C_0 first serves its stalled load—which is logically part of its *own* epoch—and then transitions to state I.

5.5 Step 4: Assigning Access Permissions to States

For every state in the protocol, ProtoGen assigns which accesses (load, store, replacement) are allowed in that state. For stable states, this assignment is given in the SSP.

For transient states, this assignment is a function of the transient state's initial and final stable states (e.g., a transient state in a transaction from S to M). If the initial and final states both have sufficient permissions for an access, then the access can be performed in the transient state. ProtoGen also has an input parameter that determines whether to permit *any* loads and stores in transient states; recall that permitting this could lead to violations of SWMR in physical time but is still compatible with per-location sequential consistency.

5.6 Generating Directory Controller

ProtoGen has been described until now from the perspective of generating the cache controller. Although the process of generating the directory controller is quite similar, we now discuss a few issues specific to generating the directory controller.

In general, generating the directory is easier, because the directory has perfect knowledge about the order in which requests are serialized. Even if a directory entry is in a transient state (e.g., in an MSI protocol, a GetS that arrives in state M causes the directory to have to wait for data from the owner), the directory is able to trivially deduce that a subsequent request has to be ordered after the current transaction. Unlike with cache controllers, there is no possibility of an arriving request being ordered before one the directory has already seen.

Directory generation, however, poses one unique challenge. Because our directory controller is non-stalling, there can be as much concurrency at the directory as there are caches in the system. (Concurrency at a cache is constrained by its limit of one outstanding transaction per block.) This concurrency at the directory raises the possibility of observing requests in states that would not be possible in atomic protocols. Specifically, our directories can receive Put requests (PutS, PutM, etc.) in *any* state. For example, consider a block in state S at cache C_0 that issues a PutS to the directory. Before that PutS reaches the directory, cache C_1 issues a GetM that reaches the directory. The GetM changes the directory state to M, and the directory sends an Invalidation to C_0. Later, C_0's now-stale PutS arrives at the directory which is in state M; this situation is not possible in an atomic protocol and thus would not appear in any SSP specification. Furthermore, there are scenarios like this for every combination of Put and every state at the directory.

Because there is no good way to specify these scenarios in an SSP—because they do not occur in an SSP—we leverage knowledge of how directory protocols work. A stale

Put request “lost” in its race to the directory and the directory knows that the issuer of the Put had its epoch ended by another transaction that was ordered before its Put. For protocols using a subset of the common MOESIF states, it is correct for the directory to simply acknowledge any stale Put request, so that the issuer of the Put can complete its stale transaction. It might also be possible, in some protocols, to update the directory’s auxiliary state (e.g., by removing a cache from the Sharer list), but we do not pursue this option; it is a possible optimization, but not required.

5.7 Putting It All Together

In this section, we describe how the aforementioned steps are put together to generate cache and directory controllers. ProtoGen first preprocesses the SSP and initializes the State Sets (Step 1). Then, for every stable-to-stable transition in the SSP, ProtoGen generates a transient state for each intermediate step in the transition (Step 2). Then, for each of these transient states, ProtoGen performs the process in Step 3 to accommodate the possible concurrency in that transient state. As part of that process, ProtoGen may generate new transient states. ProtoGen repeats Step 3 on all newly generated transient states until either no new ones remain or we reach the pending transaction limit. After all states are generated, ProtoGen assigns access permissions to every state (Step 4).

Chapter 6

Evaluation: Protocols Generated with ProtoGen [1]

To experimentally evaluate ProtoGen, we have used it to generate several different protocols with different features and different system model assumptions. Unlike traditional architectural evaluations that seek to show improvements in performance, power, etc., this evaluation seeks rather to show that ProtoGen can successfully generate protocols that are identical–and, in some cases, arguably superior—to existing protocols.

6.1 Stalling Protocols

In our first set of experiments, we used ProtoGen to generate several stalling protocols from Sorin et al.'s primer [31]. The primer includes specifications of concurrent MSI, MESI, and MOSI protocols, all of which are stalling. We developed an SSP for each of these protocols—SSPs that are are vastly simpler than the specifications in the primer—and ProtoGen produced concurrent versions of these protocols. For all three of these protocols, ProtoGen generated the same cache controller specifications as in the primer, and the directory controllers were also identical except for one trivial difference for the MSI and MESI directories.[1] All of the protocols passed Murφ verification of SWMR and deadlock freedom with three caches, which is the most caches that Murφ can handle without exhausting memory. The results in this section are perhaps unsurprising but they are reassuring.

[1] ProtoGen split a state to more precisely track the sharer list in one rare situation.

6.2 Non-Stalling Protocols

To test ProtoGen's ability to generate new directory protocols with even more concurrency, we used ProtoGen to generate non-stalling versions of the MSI, MESI, and MOSI protocols from the previous section. It is worth noting that the protocols generated were fairly non-trivial with 18-20 states and 46-60 transitions. There are no non-stalling MESI and MOSI protocols in the primer (or specified completely elsewhere), so there are no comparisons to be made. There is, however, a non-stalling MSI protocol in the primer, and we compare our generated protocol to it.

In Table 6.1, we highlight some differences between our generated protocol and the protocol in the primer. Entries in **bold** are related to the protocol generated by ProtoGen. Where ProtoGen shows a different behavior than the primer's protocol, the transition related to the primer's protocol is crossed out. Two interesting differences are observable. First, the generated protocol is more aggressive, i.e., stalls less often. Even though the primer's protocol is "non-stalling", it still stalls in some complicated situations (e.g., if a *Fwd-GetS* or *Fwd-GetM* arrives in the states IM^{AD} and SM^{AD}); our generated protocol does not, since it possesses the additional transient states $IM^{AD}S$, $IM^{AD}I$, $IM^{AD}SI$ *and* $SM^{AD}S$. Second, ProtoGen was able to merge some states that were kept separate in the primer like $IM^{A}S = SM^{A}S$, $IM^{A}SI = SM^{A}SI$, and $IM^{A}I = SM^{A}I$.

Verifying non-stalling protocols with a model checker is difficult, because non-stalling protocols tend to enforce SWMR in logical time and not physical time. A model checker seeks to determine whether an invariant is true in the entire reachable state space of the system, and specifying a logical time invariant can be difficult. (State space here refers to all possible states of the entire system, not just the possible coherence states of a given block of memory.) We use Murφ to verify that our protocols do enforce physical time SWMR except in one well-known situation, which is when they perform a single load or store for a transaction whose epoch ended before the data arrived. For example, if a cache issues a GetS to go from I to S so it can perform a load, there is the possibility of an Invalidation arriving while the block is still in state IS; the block transitions to ISI and the cache seemingly fails to perform its load. This is a well-known livelock pitfall, and the common and correct solution is to allow one access (load or store) in physical time after the invalidation [37] (as in the protocol generated by ProtoGen). This access *logically* occurs before the block is invalidated.

Table 6.1: MSI Non-Stalling Primer vs. ProtoGen

	Load	Store	Evict	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data (ack=0)	Data (ack>0)	Inv-Ack	Last Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}									
IS^D	stall	stall	stall			send Inv-Ack to Req/IS^DI		-/S			
IS^DI	*stall*	*stall*	*stall*					*-/I*			
IM^{AD}	stall	stall	stall	~~stall~~ **-/$IM^{AD}S$**	~~stall~~ **-/$IM^{AD}I$**			-/M	-/IM^A	ack–	
IM^A	stall	stall	stall	-/IM^AS	-/IM^AI					ack–	-/M
IM^AS	stall	stall	stall			send Inv-Ack to Req/IM^ASI				ack–	send Data to Req and Dir/S
IM^ASI = **SM^ASI**	stall	stall	stall							ack–	send Data to Req and Dir/I
IM^AI = **SM^AI**	stall	stall	stall							ack–	send Data to Req/I
SM^AS	~~stall~~ **hit**	stall	stall			send Inv-Ack to Req/IM^ASI				ack–	send Data to Req and Dir/S
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I					
SM^{AD}	hit	stall	stall	~~stall~~ **-/$SM^{AD}S$**	~~stall~~ **-/$IM^{AD}I$**	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	ack–	
SM^A	hit	stall	stall	~~-/SM^AS~~ **-/IM^AS**	~~-/SM^AI~~ **-/IM^AI**					ack–	-/M
M	hit	hit	send PutM + Data to Dir/MI^A	send Data to Req and Dir/S	send Data to Req/I						
$IM^{AD}S$	*stall*	*stall*	*stall*			*send Inv-Ack to Req/$IM^{AD}SI$*		*send Data to Req and Dir/S*	*-/IM^AS*	*ack–*	
$IM^{AD}I$	*stall*	*stall*	*stall*					*send Data to Req/I*	*-/IM^AI*	*ack–*	
$IM^{AD}SI$	*stall*	*stall*	*stall*					*send Data to Req and Dir/I*	*-/IM^ASI*	*ack–*	
$SM^{AD}S$	*hit*	*stall*	*stall*			*send Inv-Ack to Req/$IM^{AD}SI$*		*send Data to Req and Dir/S*	*-/IM^AS*	*ack–*	
MI^A	stall	stall	stall	send Data to Req and Dir/SI^A	send Data to Req/II^A		-/I				
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I				
II^A	stall	stall	stall				-/I				

6.3 An MSI Protocol for an Unordered Network

The MSI protocols we have discussed already were designed to work correctly on interconnection networks with point-to-point order. Point-to-point order—which means that, if node A sends two messages to node B, they arrive in the order in which they were sent—makes protocol design easier by eliminating several possible race conditions that could otherwise occur.

To test ProtoGen, we developed the SSP for an MSI protocol that does not rely upon point-to-point order. This protocol adds extra "handshaking" messages to handle the races that arise. Specifying the SSP for this protocol was not much more difficult than for the MSI protocol that relies upon ordering, even with the extra handshaking messages. ProtoGen generated the concurrent protocol from the SSP and thus saved us from having to manually deal with this complexity.

6.4 TSO-CC

TSO-CC [12] is a recently developed coherence protocol that is tailored for use in systems that support the TSO memory consistency model. Conventional protocols are designed to support *any* consistency model and thus conservatively avoid any behavior that could violate sequential consistency (SC), e.g., by enforcing SWMR in physical time. TSO-CC, by contrast, exploits the relaxed nature of TSO to avoid sharer tracking. In doing so, it breaks physical time SWMR but honors TSO.

The TSO-CC paper is accompanied by a complete protocol specification with concurrency that is designed to work correctly even if the network is unordered. We wanted to see if we could use ProtoGen to generate a complete TSO-CC protocol with concurrency but leveraging point-to-point ordering.

The first step was to develop an SSP specification that can leverage point-to-point ordering. This was reasonably straightforward given the complete TSO-CC specification; it was a question of selecting the stable state transitions and eliminating "handshakes" to exploit point-to-point ordering.

We then used ProtoGen to generate the complete protocol with concurrency. Using the verification methodology of Banks et al. [38], we verified that our complete protocol correctly enforces TSO. The key takeaways from this study are twofold. First, ProtoGen can be used to generate unconventional protocols such as TSO-CC. Second, it also showcases ProtoGen's utility in transforming a complex protocol and making it work

for a different system model. Our protocol modification was easy to make at the SSP level, whereas it would have been much more difficult to generate a complete TSO-CC protocol at the concurrent protocol level that is able to leverage point-to-point ordering.

6.5 Discussion

Our current ProtoGen implementation has not been optimized for performance, but was designed for flexibility during the development process. Nevertheless, runtimes are always well less than one second on an Intel i5.

Although one could argue that many of the protocols in this section already existed and generating them is not practically useful, automatic generation is still far faster and less error-prone than designing protocols by hand. Moreover, it is exciting to observe that ProtoGen could handle an unconventional protocol like TSO and uncover additional concurrency in non-stalling protocols.

Chapter 7

Conclusion

ProtoGen has been developed to ease the design of cache coherence protocols. By exploiting domain specific knowledge about the functionality of directory based cache coherence protocols, ProtoGen generates correct, highly-concurrent, non-blocking cache coherence protocol implementations from a given atomic stable state directory cache coherence protocol (SSP) specification. This simplifies the protocol design process significantly. An architect can focus on designing a correct SSP protocol instead of worrying about races between transactions due to concurrency. In contrast to other techniques focusing on the automatic generation of cache coherence protocols, ProtoGen is a method that refines an SSP specification consisting of atomic transactions into a non-atomic implementation.

For a variety of protocols, it has been shown that ProtoGen successfully generates correct finite state machines for the cache and directory controllers. The generated protocols possess at least as much concurrency as the existing reference protocols. Furthermore, ProtoGen merges states to optimize the finite state machines of the controllers. This suggests that protocols generated by ProtoGen do not sacrifice performance in comparison to manually designed protocols.

ProtoGen was presented at ”The 45th International Symposium on Computer Architecture, ISCA 2018”. The architecture community appreciated the potential of ProtoGen, with the gem5 team asking for an integration of ProtoGen into the gem5 simulation ecosystem.

With planned extensions, ProtoGen is expected to automate the design of hierachical and heterogenous cache coherence protocols. ProtoGen has already become a foundation for several new design automation projects.

Bibliography

[1] Nicolai Oswald, Vijay Nagarajan, and Daniel Sorin. Protogen: Automatically generating directory cache coherence protocols from atomic specifications. In *The 45th International Symposium on Computer Architecture, ISCA, Los Angeles, United States, June 2-6*, 2018.

[2] Nicolai Oswald. Memory consistency and cache coherency in network-on-chip. *Master Thesis, TU Munich*, 2016.

[3] Milo M. K. Martin, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Bandwidth adaptive snooping. In *International Symposium on High-Performance Computer Architecture, HPCA, 2010*, 2010.

[4] Daniel Sanchez, George Michelogiannakis, and Christos Kozyrakis. An analysis of on-chip interconnection networks for large-scale chip multiprocessors. In *ACM Transactions on Architecture and Code Optimization, April 2010*, 2010.

[5] Craig Williams, Paul F. Reynolds Jr., and Bronis R. de Supinski. Delta coherence protocols. *IEEE Concurrency 2000*, 2016.

[6] Satish Chandra, Michael Dahlin, Bradley Richards, Randolph Y. Wang, Thomas E. Anderson, and James R. Larus. Experience with a language for writing coherence protocols. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2017*, 2017.

[7] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 155–166, 2011.

[8] Johnathan Alsop, Matthew D. Sinclair, and Sarita V. Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *The 45th International Symposium on Computer Architecture, ISCA, Los Angeles, United States, June 2-6*, 2018.

[9] Dennis Abts, Steve Scott, and David J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2003*, 2003.

[10] Anand Lal Shimpi and Brian Klug. Samsung updates exynos 5 octa (5420), switches back to arm gpu. Accessed: 2018-05-24.

[11] David L Dill. The Murphi Verification System. In *Computer-Aided Verification, CAV*, volume 1102, pages 390–393, 1996.

[12] Marco Elver and Vijay Nagarajan. TSO-CC: Consistency directed cache coherence for TSO. In *Proceedings - International Symposium on High-Performance Computer Architecture, HPCA*, pages 165–176, 2014.

[13] Bluespec system verilog. `http://bluespec.com/`, note = Accessed: 2018-03-30,.

[14] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[15] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 25, NO. 3, MAY/JUNE 1999*, 1999.

[16] Satish Chandra, Michael Dahlin, Bradley Richards, Randolph Y. Wang, Thomas E. Anderson, and James R. Larus. Experience with a language for writing coherence protocols. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages, DSL, 1997*, 1997.

[17] Jørgen Staunstrup and Mark R. Greenstreet. From high-level descriptions to VLSI circuits. *BIT*, 28(3):620–638, 1988.

[18] Arvind and Xiaowei Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1999.

[19] BluespecTM SystemVerilog Reference Guide. 2014.

[20] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as rule composition. In *5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), May 30 - June 1st, Nice, France*, pages 51–60, 2007.

[21] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *International Conference on Computer-Aided Design, ICCAD, San Jose, CA, USA, November 10-13, 2008*, pages 24–31, 2008.

[22] Dana Vantrease, Mikko H. Lipasti, and Nathan Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA, pages 132–143, Washington, DC, USA, 2011. IEEE Computer Society.

[23] Nirav Dave, Man Cheuk Ng, and Arvind. Automatic synthesis of cache-coherence protocol processors using bluespec. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), 11-14 July*, pages 25–34, 2005.

[24] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296, 2013.

[25] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Automatic completion of distributed protocols with symmetry. In *International Conference on Computer Aided Verification (CAV)*, pages 395–412, 2015.

[26] Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, University of California, Berkeley, 2008.

[27] Christopher J. Banks, Marco Elver, Ruth Hoffmann, Susmit Sarkar, Paul Jackson, and Vijay Nagarajan. VerC3: A library for explicit state synthesis of concurrent systems. In *Design Automation and Test in Europe, DATE, Dresden, Germany, March 19-23*, 2018.

[28] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 241–252, New York, NY, USA, 2012. ACM.

[29] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 471–482, Washington, DC, USA, 2010. IEEE Computer Society.

[30] Gwendolyn Voskuilen and T.N. Vijaykumar. High-performance fractal coherence. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 701–714, New York, NY, USA, 2014. ACM.

[31] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[32] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[33] Milo M. K. Martin, Daniel J. Sorin, Anatassia Ailamaki, Alaa R. Alameldeen, Ross M. Dickson, Carl J. Mauer, Kevin E. Moore, Manoj Plakal, Mark D. Hill, and David A. Wood. Timestamp snooping: An approach for extending smps. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 25–36, New York, NY, USA, 2000. ACM.

[34] Mieszko Lis, Keun Sup Shim, Myong Hyon Cho, and Srinivas Devadas. Memory coherence in the age of multicores. In *Proceedings of the 2011 IEEE 29th International Conference on Computer Design*, ICCD, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[35] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache coherence for gpu architectures. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA, pages 578–590, Washington, DC, USA, 2013. IEEE Computer Society.

[36] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[37] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 274–284, New York, NY, USA, 1992. ACM.

[38] Christopher J. Banks, Marco Elver, Ruth Hoffmann, Susmit Sarkar, Paul Jackson, and Vijay Nagarajan. Verification of a lazy cache coherence protocol against a weak memory model. In *Formal Methods in Computer Aided Design, FMCAD, Vienna, Austria, October 2-6*, pages 60–67, 2017.