THE UNIVERSITY
of EDINBURGH

Automatic Generation of Highly Concurrent, Hierarchical and Heterogeneous Cache Coherence Protocols from Atomic Specifications

Nicolai Alexander Oswald

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
The University of Edinburgh
2023

Abstract

Cache coherence protocols are often specified using only stable states and atomic transactions for a single cache hierarchy level. Designing highly-concurrent, hierarchical and heterogeneous directory cache coherence protocols from these atomic specifications for modern multicore architectures is a complicated task. To overcome these design challenges we have developed the novel *Gen algorithms (ProtoGen, HieraGen and HeteroGen). Using the *Gen algorithms highly-concurrent, hierarchical and heterogeneous cache coherence protocols can be automatically generated for a wide range of atomic input stable state protocol (SSP) specifications - including the MOESI variants, as well as for protocols that are targeted towards Total Store Order and Release Consistency. In addition, for each *Gen algorithm we have developed and published an eponymous tool.

The ProtoGen tool takes as input a single SSP (i.e., no concurrency) generating the corresponding protocol for a multicore architecture with non-atomic transactions. The ProtoGen algorithm automatically enforces the correct interleaving of conflicting coherence transactions for a given atomic coherence protocol specification.

HieraGen is a tool for automatically generating hierarchical cache coherence protocols. Its inputs are SSPs for each level of the hierarchy and its output is a highly concurrent hierarchical protocol. HieraGen thus reduces the complexity that architects face by offloading the challenging task of composing protocols and managing concurrency.

HeteroGen is a tool for automatically generating heterogeneous protocols that adhere to precise consistency models. As input, HeteroGen takes SSPs of the per-cluster coherence protocols, each of which satisfies its own per-cluster consistency model. The output is a concurrent (i.e., with transient states) heterogeneous protocol that satisfies a precisely defined consistency model that we refer to as a *compound consistency model*.

To validate the correctness of the *Gen algorithms, the generated output protocols were verified for safety and deadlock freedom using a model checker. To verify the correctness of protocols that need to adhere to a specific compound consistency model generated by HeteroGen, novel litmus tests for multiple compound consistency models were developed.

The protocols automatically generated using the *Gen tools have a comparable or better performance than manually generated cache coherence protocols, often discovering opportunities to reduce stalls. Thus, the *Gen tools reduce the complexity that architects face by offloading the challenging tasks of composing protocols and managing concurrency.

Acknowledgements

My PhD and past career in academia and industry alike have been wonderful experiences, because I was given the opportunity to do research and to work on a topics I love collaborating with many amazing and outstanding people whom I want to thank. First, I want to express my deepest gratitude to my supervisors Professor Vijay Nagarajan and Professor Daniel J. Sorin for all their guidance and feedback. It was always exciting to work with them and I cannot remember a single week that I did not impatiently awaited our regular meetings to discuss new ideas. They have been the best supervisors I could desire and I am eternally grateful for their mentorship.

Throughout my career I have been blessed with many great supervisors and mentors. I want to express my sincere thanks to Professor Frauke Steinhagen, Wilfried Saur, Ralf Ohmberger, Dr. Alessandro Bernardini, Dr. Todor Mladenov and Professor José Cano Reyes for all their support and inspiration.

Furthermore, I would also like to express my special thanks to Google for supporting me with a Google PhD Fellowship and my mentor Dr. Milo Martin.

I also want to thank all the friends I made during my time at the University of Edinburgh, TU Munich and DHBW Lörrach. You have enriched my life and made me feel at home in all the places I have lived.

Additionally, I extend a heartfelt thanks to my wonderful fiancée Aishwarya, who supported me during my PhD journey and showed understanding towards my sometimes extended working hours.

Finally I want to express my deepest gratitude to my parents for their endless love and support throughout all these years.

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. Some of the material used in this thesis has been published in the following publications:

- N. Oswald, V. Nagarajan and D. J. Sorin, "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018, pp. 247-260, doi: 10.1109/ISCA.2018.00030.
- N. Oswald, "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications,", Master of Science by Research Thesis
- N. Oswald, V. Nagarajan and D. J. Sorin, "HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 888-899, doi: 10.1109/ISCA45697.2020.00077.
- N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson and R. Carr, "HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols," 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2022, pp. 756-771, doi: 10.1109/HPCA53966.2022.00061.

(*Nicolai Alexander Oswald*)

To Mimi

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Most modern computer systems support shared memory on a hardware level, as it simplifies the programming model of parallel programs [1]. While early parallel architectures often had a single physical memory resulting into uniform memory accesses (UMA) [2], with the increasing number of computing units - that can be homogeneous as well as heterogeneous - the system memory has become distributed resulting into non-uniform memory access (NUMA) latencies. To reduce access latencies to spatially distant memory, local copies of memory addresses previously accessed or speculatively fetched by a computing unit are kept in caches [1]. In Figure 1.1 an example of a heterogeneous and hierarchical NUMA architecture [3; 4] is presented.

Even though the system has two CPU and a GPU cluster using different instruction set architectures (ISAs) and a large number of hierarchical caches, the complexity of its programming model does not increase, as the hardware continues to provide the impression of a single shared memory handling all the complexity resulting from data replication in the caches across the system on behalf of the programmer enforcing correct behaviour.

But what does correct behaviour even mean and how is it enforced? The secret sauce to enforcing correct behaviour among multiple data copies across a system is a cache coherence protocol. While being invisible to the programmer, it keeps the data of a single address location consistent.

While cache coherence protocols allow the programming model to remain largely unchanged they are challenging to design and to verify. Cache coherence protocol specifications like the MSI protocol specification shown in Tables 1.1 and 1.2 look fairly intuitive and straightforward when presented in textbooks. For each of a small number of states, the

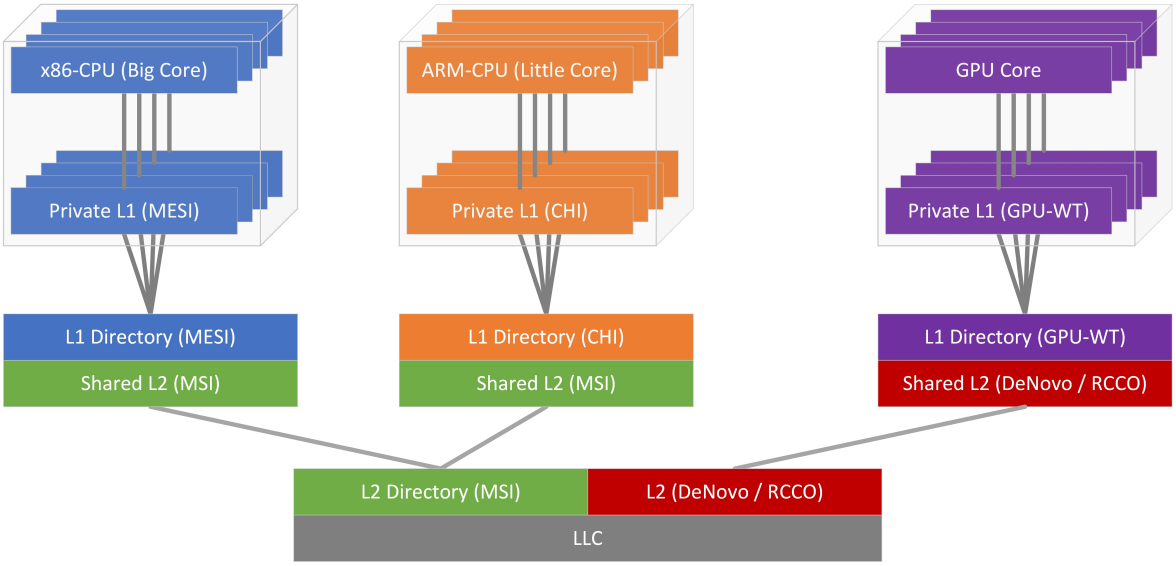


Figure 1.1: Heterogeneous and hierarchical cache coherent NUMA architecture. The system has two CPU and one GPU cluster. The cache hierarchy uses a wide range of different cache coherence protocols. The MESI[1], CHI [5] and MSI[1] cache coherence protocols are used to implement a cache hierarchy on the CPU cluster side. To enforce coherence between the CPU clusters and the GPU clusters a heterogeneous MSI and DeNovo [6] cache coherence protocol implementation is used

specification describes what happens: (a) when an access (load, store, or replacement) takes place and (b) when an incoming coherence message (e.g., an Invalidate) arrives.

Unfortunately, these textbook protocols with just a handful of stable states are overly simplistic and cannot directly be applied to a state-of-the-art system architecture like shown in Figure 1.1.

Table 1.1: Specification of Cache in Atomic MSI Protocol

	Load	Store	Eviction	Fwd_GetS	Fwd_GetM	Inv
I	send GetS to Dir, recv Data, load / S	send GetM to Dir, recv Data or DataAck plus Acks, store / M				
S	hit	send Upgrade, recv Data or DataAck plus Acks, store / M	send Put, recv Put_Ack / I			send InvAck to, Requestor / I
M	hit	hit	send Put, recv Put_Ack / I	send Data to Requestor, and Dir / S	send Data to Requestor, / I	

Table 1.2: Specification of Directory in Atomic MSI Protocol

	GetS	GetM	Put
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	remove Requestor from Sharers, send Inv to Sharers, send DataAck (containing Data and #Sharer) to Requestor, set Owner=Requestor, clear Sharers / M	send Put_Ack to Requestor, remove Requestor from Sharers, if (#Sharers == 0) then / I
M	send Fwd_GetS to Owner, receive Data from Owner, add Requestor and Owner to Sharers / S	send Fwd_GetM to Owner, set Owner=Requestor	send Put_Ack to Requestor / I

When designers use the textbook protocols specifications to create a state-of-the-art cache hierarchy they face three key challenges namely Concurrency, Hierarchy and Heterogeneity:

- Concurrency: The textbook protocols assume that a transition from one stable state to another stable state is atomic, i.e., the transition happens or appears to happen instantaneously. Only a simplistic system model (e.g., cores connected to a simple atomic bus and a centralized coherence controller) can provide this atomicity, yet high-performance modern system architectures like multicore processors employ multi-hop interconnection networks and distributed directory protocols. Typical multicore directory protocols are complicated because of their non-atomic system model. When a cache performs a coherence transaction to change a block from one stable state to another, the transaction typically involves multiple steps (issuing a request, waiting for a response, etc.) that could potentially interleave with other conflicting coherence transactions initiated by other caches. Designing protocols to correctly handle this concurrency is challenging.

  The protocol complexity introduced by concurrency is revealed in the large number of possible *transient* coherence states, in addition to the handful of stable states. At each step in a coherence transaction, the cache usually changes the state of the block to a different transient state that reflects that step in the transaction. For example, consider a cache using the MSI protocol shown in table 1.1 that has a block in state I(nvalid) and issues a request for state S(hared). In the interim—between issuing the request and when a response arrives to complete the transition from I to S—the cache block will be in an at least one transient state. Moreover, in addition to the transient states between a current state and a requested state, there are often transient states that arise due to coherence requests from other caches that arrive in this window of time.

  In typical coherence protocols with dozens of states, it is easy for architects to make mistakes. In every coherence state of a block, an architect may forget about a possible coherence message arriving while the block resides in that state or they might introduce a bug in handling ab incoming message. By failing to consider a possible situation an architect can end up with too few states to correctly handle protocol concurrency. Subtle bugs in the coherence protocols can seriously affect end users.

  In addition to these safety problems, a protocol can also fail to achieve its maximum performance because an architect can conservatively restrict concurrency in overly complicated situations. Rather than reason about how to handle a certain message that

arrives for a block in a certain state, it can be tempting to simply stall that request until the block is in a stable state.

Thus we have a situation in which we can easily understand protocol specifications with just the stable states, yet current multicore system models require that we have many transient states and the resultant complexity.

- Hierarchy: Hierarchy is a time-tested design strategy for scalable systems [7; 8; 9; 10; 11; 12; 13; 14], and it is an attractive approach to multicore processor design as the number of cores continues to increase. Hierarchy can also enable coherence protocols that are more scalable [15]. In Figure 1.1 the MESI[1], CHI [5] and MSI[1] cache coherence protocols are used to implement a cache hierarchy on the CPU multicore cluster side. While hierarchy has many desirable features, it also greatly complicates the design of the coherence protocol. There are more states, more transitions, and more possible concurrency. Crucially, communication between levels of the hierarchy must preserve coherence invariants. In addition to the design complexity, verification is also more challenging with hierarchy, due primarily to the much larger state space [16; 13].

- Heterogeneity: Heterogeneity is a persistent trend in modern system architectures as the annual growth in transistor count continues to decrease and with it the previously provided computing performance gains [17]. To continue to increase performance and energy efficiency, modern systems use a wide range of different general and specialized compute units resulting into an increasingly heterogeneous architecture. Current chips from Apple, Qualcomm, and Samsung have many different cores, including CPUs, GPUs, digital signal processors (DSPs), and camera processing cores.

  Architects face two challenges when designing heterogeneous processors with cache coherent shared memory. First, designing heterogeneous coherence protocols is difficult. Designing a coherence protocol for a *homogeneous* processor is already a notoriously challenging task and introducing heterogeneity multiplies the design complexity. This is because communication patterns within a CPU, GPU or an accelerator are very different, often mandating bespoke coherence protocols for each case. Conventional protocols that use writer-initiated invalidations for enforcing the Single-Writer-Multiple-Reader (SWMR) invariant work well for CPUs. But they are ill-suited for GPUs [18], which tend to employ self-invalidating protocols that directly enforce relaxed consistency models instead of SWMR [1]. To compose these very different protocols into a unified heterogeneous whole requires designing a bridge between them. While recent

academic [19; 20] and industrial works [5; 21; 22] have developed interfaces or wrappers that facilitate this process, it is still quite challenging to manually compose the protocols.

The second challenge faced by architects is reasoning about the memory consistency model provided by the heterogeneous processor. Recall that a memory consistency model defines the software-visible orderings of loads and stores across all of the threads in a shared memory system [1]. Consider a system architecture like shown in Figure 1.1 that consists of several clusters of cores, each with its own per-cluster coherence protocol and consistency model. Now assume that the first challenge of composing the cluster-level protocols together into a single protocol has been overcome. What consistency model does this heterogeneous processor provide? How does one ensure that the composed protocol adheres to the intended consistency model?

Due to aforementioned challenges, designing cache coherence protocols is perceived as notoriously difficult by academia and industry alike. Herein many publications from recent years agree:

- ”The coherence problem is difficult, because it requires coordinating events across nodes” [23]
- ”... directory-based cache coherence protocols are notoriously complex” [6]
- ”... coherence protocols are notoriously difficult to design and implement correctly” [24]
- ”... designing and verifying a new hardware coherence protocol is difficult” [19]

In industry a bug in the CCI-400 cache coherent interface that was not discovered during the product design and verification phase, forced Samsung to ship its Galaxy S4 smartphone with coherence disabled, having serious performance and power implications [25].

1.2 Thesis Contributions

This thesis solves the problem of designing highly-concurrent, hierarchical and heterogeneous cache coherence protocols, by automatically generating them from atomic stable state protocol (SSP) specifications. The *Gen tools apply a sequence of different algorithms presented in the following subsections to compose stable state coherence protocols generating, depending on the requirements of the designer, hierarchical (HieraGen) as well as heterogeneous (HeteroGen) coherence controllers that are highly-concurrent (ProtoGen). The algorithms operate

alone on the input SSP specification described in the ProtoGen Protocol Language (PPL). For HieraGen and HeteroGen the designer must provide additional information specifying the position of the SSP protocol caches in the system cache hierarchy and providing the memory consistency model assumed by the coherence protocols.

The ProtoGen[1], HieraGen[2] and HeteroGen[3] tool source code versions have been published at different conferences and are available on GitHub. While the conferences at which ProtoGen and HieraGen were published did not perform artifact evaluation, the conference at which HeteroGen was published had artifact evaluation with the HeteroGen tool earning all badges.

Throughout this thesis we use the terminology of Nagarajan et al. [1]. This terminology includes coherence states (e.g., transient state names like *IS* that denote the block is in a transient state between states I and S) and coherence requests (e.g., GetShared or GetS).

1.2.1 ProtoGen

The ProtoGen tool presented in this thesis is an automated tool for taking a stable state protocol (SSP) specification of a directory coherence protocol and generating a directory protocol with those same stable states and all of the transient states needed to maximize protocol concurrency, while preserving safety (correctness) and preventing deadlocks. The protocol concurrency is maximized by immediately serving any message received and thus avoiding stalls causing potential input buffer head-of-line blocking. ProtoGen accepts the SSP specification in the domain-specific PPL and generates finite state machines for the cache controller and directory controller.

ProtoGen overcomes the key challenge of protocol generation—creating cache and directory controllers that correctly handle incoming coherence messages when transactions are racing—by leveraging the insight that, in a directory based coherence protocol, racing transactions are serialized at the directory. By assigning a unique name to every directory-forwarded request that can arrive at a stable state in a cache, ProtoGen makes it possible for the directory to convey this serialization order to the caches. With the caches and the directory achieving consensus on the order of racing transactions, ProtoGen is able to generate highly-concurrent and non-blocking controller actions that are consistent with this order.

Contributions: The contribution of the ProtoGen publication are presented in chapter 5 - pre-processing the input SSP descriptions - and chapter 6 - explaining the ProtoGen algorithm steps-.

[1] `https://github.com/Errare-humanum-est/ProtoGen.git`
[2] `https://github.com/Errare-humanum-est/HieraGen.git`
[3] `https://github.com/Errare-humanum-est/HeteroGen.git`

In summary, the ProtoGen publication makes the following contributions.

- ProtoGen, an automated tool for taking an SSP specification of a directory coherence protocol and generating a complete directory protocol with those same stable states and all of the transient states needed to maximize protocol concurrency, while preserving safety and preventing deadlocks, is presented.
- ProtoGen is used to generate high-performance MSI, MESI, and MOSI protocols given SSP specifications. The generated protocols are verified for safety and deadlock freedom using the Murφ model checker [26]. The generated protocols are identical to or better than manually generated high-performance protocols [1], at times even discovering opportunities to reduce stalling.
- ProtoGen's versatility is demonstrated by using it to generate a complete TSO-CC protocol [27] given its SSP specification. TSO-CC is an unconventional protocol that is designed specifically to satisfy the TSO memory consistency model.

Since the initial publication, the following additional contributions have been made.

- ProtoGen allows the handling of synchronization operations affecting multiple cache lines related to e.g. acquire or release accesses. This allows ProtoGen to generate complete memory consistency model oriented coherence protocols the like of DeNovo [6], RCCO and RCC [1].
- ProtoGen deduces potential optimizations of the SSP coherence protocol description reducing stalls and model checking state space.
- ProtoGen can generate highly concurrent coherence protocol implementations that allow coherence messages of concurrent transactions to arrive in an arbitrary order while using only a single virtual channel as the generated controllers are completely non-stalling. Compared to state of the art protocols like ARM CHI [5], the ProtoGen generated protocols can be implemented on unordered interconnects without the need of handshakes serializing coherence transactions in most cases.

1.2.2 HieraGen

HieraGen presented in chapter 7 extends the ProtoGen tool generating correct-by-construction hierarchical protocols. The user inputs the SSPs for each cache hierarchy level independently.

Given these inputs, HieraGen produces the design of the complete and concurrent hierarchical protocol.

Accommodating hierarchy introduces a key challenge beyond what ProtoGen addresses: HieraGen must create intermediate directory/cache nodes that were not completely specified by the user; in the example system architecture presented in Figure 1.1, HieraGen must compose the cache from the higher level (in green) using the MSI protocol with the directory from the lower level (in blue) using the MESI protocol to produce the intermediate directory/cache node.

HieraGen addresses this challenge by automatically encapsulating higher-level coherence actions within lower-level coherence transactions (and vice versa), such that coherence invariants are enforced globally.

HieraGen leverages ProtoGen to implement concurrency in a hierarchical system that can have multiple coherence transaction serialization points (e.g., directories). We make the observation that in a hierarchical SSP that correctly enforces coherence globally, any two racing coherence transactions will serialize at exactly one of the directories. This key invariant allows us to leverage ProtoGen for generating concurrency.

Contributions: In summary, the contributions of the HieraGen publication presented in chapter 7 are as follows.

- HieraGen is the first automated tool for taking SSP specifications of coherence protocols of each level of a hierarchical system, and generating the complete and concurrent hierarchical protocol, while preserving safety and preventing deadlocks.
- HieraGen is used to generate high-performance hierarchical protocols with the standard MOESI coherence states
- The generated protocols are verified for safety and deadlock freedom using the Murϕ model checker.

Since the initial publication, the following additional contributions have been made.

- HieraGen is used to generate hybrid high-performance hierarchical protocols using the memory consistency oriented coherence protocols TSO-CC, RCC and RCCO for the lower-level coherence protocol and a MESI protocol for the higher-level.

1.2.3 HeteroGen

HeteroGen presented in chapter 8 extends the ProtoGen tool automatically generating heterogeneous protocols that adhere to precise consistency models. As input, HeteroGen takes

simple, atomic specifications of the per-cluster coherence protocols, each of which satisfies its own per-cluster consistency model. The output is a concurrent, heterogeneous protocol that satisfies a precisely defined consistency model that we refer to as a *compound consistency model*.

The compound consistency model is a compositional amalgamation of each of the per-cluster consistency models where operations from each cluster continue to adhere to that cluster's consistency model. For example, if a CPU cluster employs a MESI coherence protocol that enforces the SWMR invariant and sequential consistency (SC), that cluster will continue to provide SC even when composed with other clusters that have other per-cluster consistency models. One of HeteroGen's key contributions is its guarantee that its output protocol will provide compound consistency. Moreover, we show that compound consistency is a key enabler to supporting language-level consistency models on heterogeneous processors.

Contributions: In summary, the contributions of the HeteroGen publication presented in chapter 8 are as follows:

- Compound memory consistency models are formalized, a compositional approach to specifying consistency models targeted towards heterogeneous systems.
- HeteroGen, the first automated tool for composing heterogeneous coherence protocols is developed and distributed.
- Using heterogeneous litmus testing it is demonstrated that HeteroGen produces protocols that satisfy the intended compound consistency models.
- It is experimentally shown that the performance and network traffic of our automatically-generated protocol is comparable to a manually-generated heterogeneous protocol.

1.3 *Gen Tool Flows and Thesis Outline

In Figure 1.2 the flow of the latest *Gen tool named ProtGen+ is presented. The ProtoGen+ tool is the first *Gen tool to incorporate the HeteroGen, HieraGen and ProtoGen algorithms together. The previously published *Gen Tools (ProtoGen, HieraGen and HeteroGen) do not implement all algorithms of the pipeline. For example, as shown in Table 1.3, the HeteroGen tool does not implement the HieraGen algorithm stage. Therefore, it can only generate concurrent heterogeneous coherence protocols. The HieraGen tool on the other hand does not implement the HeteroGen algorithm stage enabling it to only generate MOESI family hierarchical protocols. The limitations of the different *Gen tools reflect the progress of our

*Gen Tool	Algorithm Stages			
	Pre-Processing	HeteroGen	HieraGen	ProtoGen
ProtoGen	✓	-	-	✓
HieraGen	✓	-	✓	✓
HeteroGen	✓	✓	-	✓
ProtoGen+	✓	✓	✓	✓

Table 1.3: *Gen tool versions and implemented algorithm stages

research and tool development at the time of their development and publication - see section 9.2 for further information.

The *Gen tools are essentially compilers. Therefore the high-level input SSP specifications described in the domain specific PPL described in chapter 4 are first translated into an intermediate representation (IR) transition graph format that is subsequently used by all tool pipeline stages. Throughout the thesis the PPL language and transition graphs are used to explain the function of the *Gen algorithms. The algorithm stages are shown in the logical order in which they must be executed. Depending on the designer's system configuration input, the HeteroGen or HieraGen stages may be skipped.

In the SSP Pre-Processing stage (1) described in chapter 5 dependencies between the separate cache controller and directory controller SSP descriptions are analyzed and extracted. Furthermore, the correctness of the provided SSP is verified. The extracted dependency information is then used by the *Gen stages to correctly construct the new coherence controllers.

After the Pre-Processing stage, the HeteroGen stage (2) described in chapter 8 can be applied that composes the processed SSP input directory controller specifications of multiple input SSPs creating a new heterogeneous SSP directory controller.

The HieraGen algorithm stage (3) described in chapter 7 takes a lower-level directory controller SSP and a higher-level cache controller SSP description as input generating a new hierarchical SSP coherence controller. Depending on the user specification, the lower-level directory controller SSP input of HieraGen can be the atomic heterogeneous directory controller output of the HeteroGen stage.

While the HieraGen and HeteroGen stages generate new atomic SSP descriptions composing the input SSPs to generate hierarchical and heterogeneous SSP controllers, the ProtoGen stage (4) described in chapter 6 generates the transitions and transient states to correctly handle concurrency. ProtoGen generates highly-concurrent versions of the atomic input controllers.

After passing through the *Gen algorithm stages the atomic input SSPs are refined following the user configuration into highly-concurrent, hierarchical and heterogeneous coherence protocol implementations.

The correctness of the generated protocols is verified using Murϕ backend leveraging the Murϕ model checker framework described in chapter 3 converting the IR into a Murϕ output description.

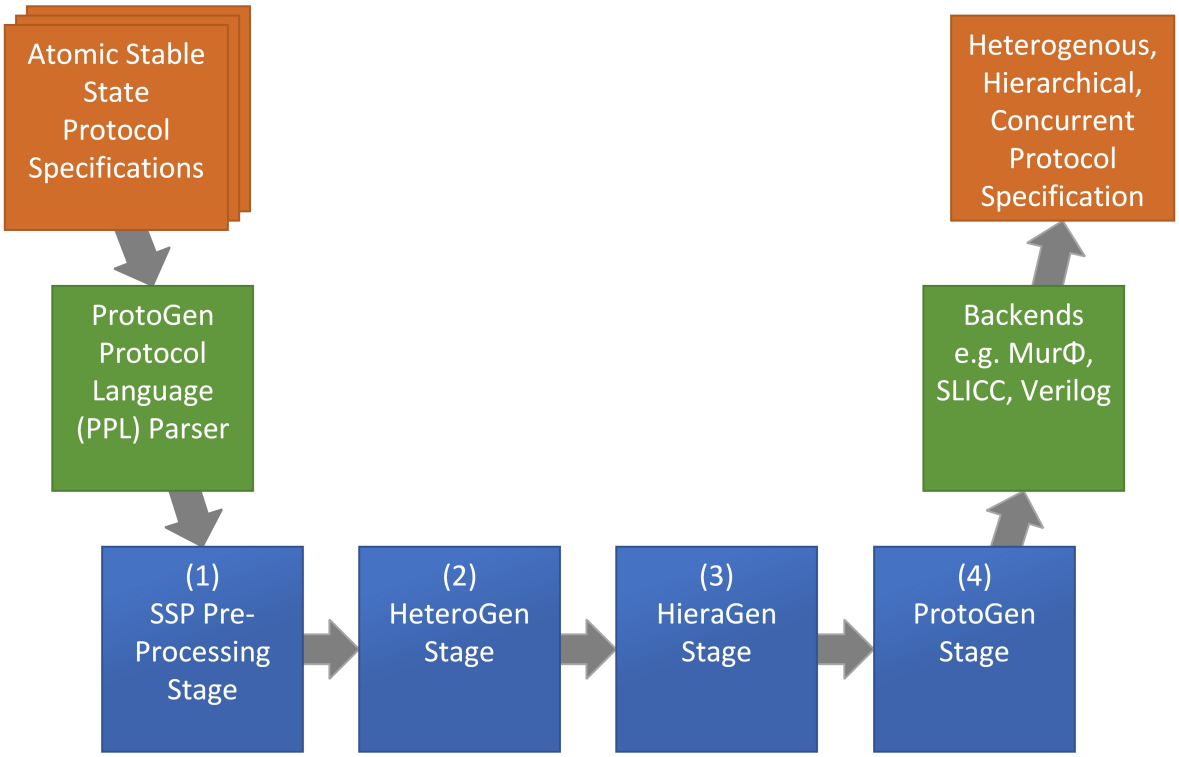


Figure 1.2: ProtoGen+ tool pipeline

Chapter 2

General Background

Cache coherence protocols are often presented using only atomic transactions i.e. SSPs. 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.

Directory based cache coherence protocols scale better in large systems as they require less bandwidth than snooping based protocols [28]. The available bandwidth of modern network-on-chip interconnects only grows logarithmically with an increasing number of participants [29; 30].

2.1 Definition of Coherence

We define cache coherence through two invariants [31]:

- Write-Serialization: All writes to the same memory location are globally serialized, therefore all compute units will observe them in the same order.
- Write-Propagation: A write to a memory location will eventually become visible to all compute units

While these two invariants are satisfied by all coherence protocols, some protocols - e.g. the MOESI protocol variants presented in section 2.2 - also satisfy the single-writer-multiple-reader (SWMR) invariant and Data-Value invariant commonly used to define coherence [1].

The SWMR invariant defines that for any memory location at any given time, there is either a single writer or zero or more readers. When the SWMR invariant is satisfied, the Write-Serialization invariant is satisfied as well due to the behavior of the MOESI protocols

allowing only one compute unit to write to a memory location at any given time, writes are serialized. The SWMR invariant allows us to divide a cache block's lifetime into epochs, during each of which there is either a single writer or zero or more (multiple) readers.

The Data-Value invariant defines that 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. If a coherence protocol satisfies the Data-Value invariant, it also satisfies the Write-Propagation invariant.

Intuitively, these epochs can be in physical time in which the two invariants combine into one single invariant, i.e. in case of SWMR, linearizability for every memory location. However, it is also correct for the epochs to be in logical time, in which the invariants combine into the per-location sequential consistency invariant [32].

A discussion of logical time correctness is beyond the scope of this thesis, but numerous protocols have been developed that rely upon it [33; 34; 18].

The benefit of per-location sequential consistency is that it can enable greater concurrency than linearizability. Consistency oriented coherence protocols therefore relax the per-location consistency guarantees further to allow for more concurrency. Per-location sequential consistency can only be enforced through explicit synchronization operations like for example fences.

2.2 Coherence Protocols

2.2.1 MOESI

Variations of the conventional MOESI state coherence protocols are commonly used in nowadays systems e.g. ARM CHI [5] and TileLink [35]. The MOESI protocols guarantee per-location sequential consistency (SC), by enforcing that before a write to a memory location becomes globally visible that all data copies in other caches are invalidated. This guarantees that a read hitting on a data copy held within a cache will always read the most recent value written to the memory location.

2.2.2 TSO-CC

TSO-CC [27] 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.

2.2.3 RCC

RCC [1] is a block-granular coherence protocol that enforces release consistency (RC). Writes are locally cached and do not become globally visible until the cache writes back the content on a release. On an acquire, the cache writes back all dirty blocks, before self-invalidating the entire cache. This is required, because some bytes in a dirty block may be clean. To distinguish dirty and clean bytes when performing a write back or a data load, all dirty bytes in a cache block are tagged with a dirty bit. Dirty bytes are never overwritten by clean data. Release and acquire raise eponymous events *eAcq* and *eRel* that must be served by all other block. The events must completed by all other blocks, before the coherence controller can continue serving the access. The RCC coherence protocol is shown in Table 2.1 and 2.2.

2.2.4 RCC-O/DeNovo

RCC-O [36; 1] is a block-granular variant of DeNovo [6] that obtains ownership on all writes. It enforces release consistency (RC). Due to the Owner state, writes become immediately globally visible. Therefore the cache does not need to perform write backs on a release or acquire operation. On an acquire, only cache lines for which a cache does not hold ownership must be self-invalidated.

Table 2.1: Specification of Cache in Atomic RCC Protocol

	Access				**Event**		
	Load	Store	Acquire	Release	eAcq	eRel	Eviction
I	send GetV to Dir recv Data load / VC	send GetV to Dir recv Data store / VD	send GetV to Dir recv Data issue eAcq load / VC	issue eRel send GetV to Dir recv Data write send WB to Dir recv WBAck / VC	/ I	/ I	
VC	hit	hit / VD	send GetV to Dir recv Data issue eAcq load	issue eRel store send WB to Dir recv WBAck	/ I	/ VC	/ I
VD	hit	hit	send GetV to Dir recv Data issue eAcq load / VC	issue eRel store send WB to Dir recv WBAck / VC	send WB to Dir recv WBAck / I	send WB to Dir recv WBAck / VC	send WB to Dir recv WBAck / I

Table 2.2: Specification of Directory in Atomic RCC Protocol

	GetV	WB
I	send Data to Requestor / V	send Ack to Requestor / V
V	send Data to Requestor	send Ack to Requestor

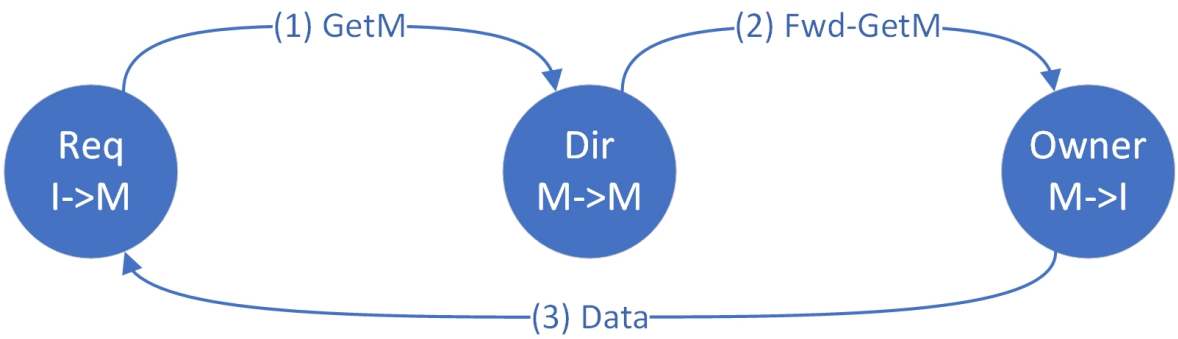


Figure 2.1: Store access cache miss at requestor triggers a GetM request to be sent to the directory. Based on its auxiliary states the directory determines the current owner of the cache block and forwards the request. The remote owner responds to the requestor with data.

2.2.5 GPU

GPU is a simple GPU protocol as specified in Spandex [19] that enforces release consistency (RC). Writes to a cache line are immediately written through to the shared cache becoming globally visible.

2.3 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, synchronization operation e.g. acquire/release or a cache entry 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 2.1 for the example of a MSI coherence protocol 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 request. Auxiliary states (e.g. sharer vectors or owner fields) 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.

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 disallowing 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.

2.4 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.

2.4.1 Racing Transaction Example

To understand the race between concurrent transactions better, Figure 2.2 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

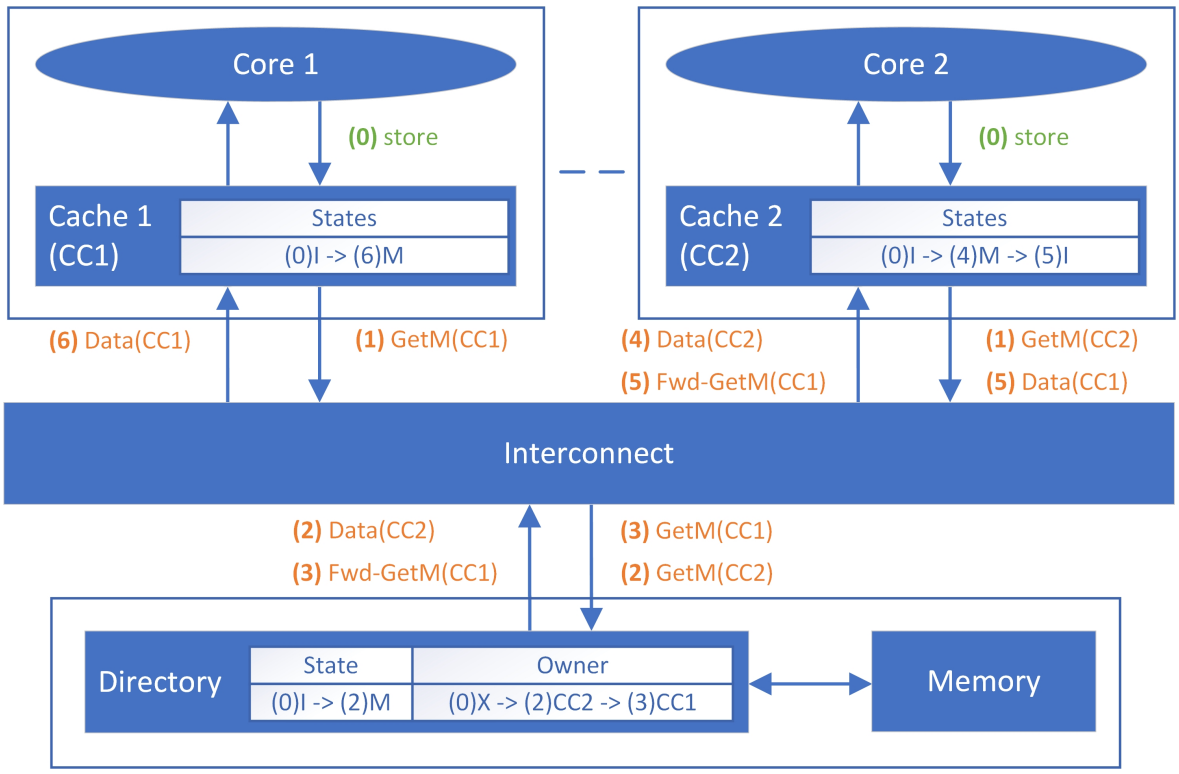


Figure 2.2: Racing Transactions

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 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 6.

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.

2.4.2 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 2.4.1 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 6.6 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 [37].

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.

Chapter 3

Murφ Model Checker Framework

The Murφ model checker framework is essential to verify the correctness of the concurrent, hierarchical and heterogeneous coherence protocols generated by the *Gen tools. We discuss it in detail, because the architect must understand the system models and data types that can be described using the ProtoGen Protocol Language presented in chapter 4 well. If the behaviour of a system is not correctly modelled the generated coherence protocols pass verification, but may exhibit errors after being implemented and tested in the RTL system description. Utilizing the Murφ model checker framework an architect can, using a few configuration parameters, automatically generate model a system model like presented in Figure 1.1. By automating the verification using the developed framework we not only avoid errors that may be introduced by manual implementation, but also ensure that the system model state space is minimal. For example, an architect may omit the modelling of the FIFO input buffers at cache controllers to reduce the model checking state space speeding up the verification runs. However, due to the absence of the FIFO input buffers, deadlocks in the coherence protocol may not be detected. In addition, an architect may fail to minimize the state space of the system model especially when generating hierarchical and heterogeneous systems when implementing the coherence protocol manually by failing to identify opportunities to apply data types like scalarsets that allow the Murφ model checker to leverage symmetry reduction.

3.1 Murφ Model Checker

The Murφ description language [26] was specifically designed to verify the correctness of protocols by supporting non-deterministic and scalable descriptions. A protocol can be described using constant, type, variable and procedure declarations. The model checker exhaustively searches the state space of the protocol description by executing rule definitions.

A rule can be selected to be executed if the condition guarding it becomes true, executing a set of actions like function calls or value assignments altering the system state.

While the Murφ model checker automatically checks for deadlocks in the system description, the designer can define additional invariants that are checked for every newly discovered state. If an invariant is violated, the Murφ model checker can output an error trace that shows the sequence of rules selected causing the invariant to be violated.

The state of the system is composed of all variables defined in the model. Therefore, the number of variables used in a Murφ model should be kept minimal. The state space of a Murφ model grows as a function of the number of rules in the Murφ model. A system architecture should be represented by a minimal number of rules.

Starting the state space exploration from an initial system state, the Murφ model checker records every new system state discovered iterating over all rules defined in the system model. If the guarding condition of a rule is true, the model checker serves the rule. After all actions of a rule have been performed, the model checker checks if it previously observed the final system state. If the final system state was not previously observed, the model checker continues to explore the state space iterating over all rules in order to discover new states. If the model checker has iterated over the rules of all reached states without observing the violation of an invariant it successfully terminates. To explore the state space the Murφ model checker can be configured to use a breadth-first search or depth-first search procedure.

3.2 System Architecture Murφ Model

To verify the correctness of a cache coherence protocol it is necessary to have sufficiently detailed description of the system architecture. The description should be as abstract as possible to minimize the state space and state vector size without omitting potential error sources.

To verify the correctness of the generated protocols, a backend has been developed for the *Gen tools that automatically generates a Murφ model checker output using a framework representing the input system architecture model.

The framework consists of trace processor, coherence controller and configurable network templates.

3.2.1 Trace Processor

The trace processor allows the *Gen tools to execute litmus tests in the model checker verifying whether the coherence protocol violates the expected memory consistency model. Once all trace processors have completed serving their instructions, the values read by every trace processor are evaluated checking if a forbidden outcome is observed. Because the model checker is exhaustive, all possible litmus test instruction completion sequences are checked. To verify the deadlock freedom of a coherence protocol, the trace processors methods are replaced by rules that comprise all possible memory accesses. This guarantees that all possible coherence transaction interleavings are verified for deadlock freedom.

3.2.2 Coherence Controller

The backend automatically generates the coherence controller implementation from the *Gen tools internal intermediate representation. Every coherence controller has a network interface consisting of a set of rules probing the network buffer whether a message has arrived. If a message has arrived, the cache controller checks if it can be served and removed from the network buffer. A message can be served if it is defined as a guard for the current state of the coherence controller and if the network buffers have enough free capacity to store the messages that a coherence controller may need to send in response.

3.2.3 Network

Using ordered and unordered buffers a fully connected network is modelled allowing all coherence controllers to directly communicate with each other. Each virtual channel is modeled as a separate network that provides a specific ordering guarantee. The available ordering guarantees from strongest to weakest are: atomic, ordered, unordered or set. A minimal Murφ model description is provided as a template for every network model. The provided network models suffice to represent state-of-the-art network implementations [30] sufficiently from a coherence protocol perspective.

- **Atomic:** All messages sent by a coherence controller are immediately injected into the ordered receive buffer of the receiver coherence controller. If different coherence controllers send messages to the same receiver, the messages in the ordered receive buffer preserve the order in which the rules at the different coherence controllers were executed. This rules out certain message interleavings with other coherence controller messages to the same destination.

The atomic network allows to omit the buffers representing the network links for every coherence controller in the system model reducing the state space. Figure 3.1b shows the Murφ model of the atomic network.

An example use case for the atomic network model is an atomic bus.

- **Ordered:** The ordered network shown in Figure 3.1c provides less ordering guarantees than the atomic network. Messages sent by a coherence controller are first injected into the ordered buffer representing the network link connecting the sender and receiver coherence controller. The ordered buffer at the receiving coherence controller eventually pops the messages from the network link buffers and stores them locally until the coherence controller serves the message.

  Unlike the atomic network model, the ordered network model enables messages sent by different coherence controllers to race to the same receiving coherence controller. The order in which the messages are pushed into the ordered buffer at the receiving coherence controller is independent from the order in which the different source coherence controller rules were served. Ordering for messages from the same sources is maintained.

  An example use case for the ordered network model is a network-on-chip interconnect using static routing.

- **Unordered:** The unordered network shown in Figure 3.1d relaxes the ordering guarantees compared to the ordered network further. Modelling the network links using unordered buffers, subsequent messages sent by the same sender to the same receiver can be reordered. Because the network links allow the messages to be arbitrarily reordered, all links to a specific destination can be merged into a single unordered buffer. The receive coherence controller buffers remain ordered preventing messages to be reordered once they were popped from the network links.

  An example use case for the ordered network model is a network-on-chip interconnect using dynamic routing.

- **Set:** The set network behavior does not provide any message ordering guarantees. Because the receive buffers of the coherence controllers are implemented as sets, a coherence controller can choose to process any message. Therefore, the network link buffers can be omitted minimizing the model state space. Figure 3.1e shows the Murφ model of the set network.

An example use case for the set network model is a gem5 [38] system architecture using recycling buffers.

Unordered buffers are implemented as sets while ordered buffers are implemented as FIFOs. The state size and state space of FIFO buffers is larger compared to buffers realized as sets especially when symmetry reduction can be applied. Therefore, unordered networks should be chosen whenever possible. To ensure a minimal state space all ordered buffers preceding an unordered buffer in a message path can be omitted in the network model.

An accurate network model is essential to verify the correctness of the coherence protocols as for example deadlocks may not occur in an interconnect with either too strong or too weak ordering guarantees.

3.3 System State Space Optimization

The state space explosion problem when modelling systems with a large number of coherence controllers can make it challenging to verify their correctness.

To reduce the state space the Murφ model checker needs to explore, symmetry reduction can be applied if multiple identical coherence controller exist in a system. From the coherence protocol perspective it is not relevant which coherence controller from a set of identical controllers is e.g. the current owner of a cache line. By defining all coherence controllers to be objects in a scalarset, the Murφ model checker can automatically apply symmetry reduction.

In addition to the symmetry reduction optimization provided by the Murφ model checker, the *Gen tools provide design optimization recommendations to the designer to reduce the state space of the coherence controller models. The guided coherence controller optimization in described in section 5.3.

By using symmetry reduction along with the guided coherence controller model optimizations, the state space of the Murφ model can be reduced.

3.4 System State Size Optimization

In addition to the system state space also the size of every system state can be optimized. The size of the system state of the Murφ model does not only depend on the complexity of the coherence controllers, but also on the network complexity. A large system state size can make it challenging to verify the correctness of systems having a large number of coherence

controllers. Therefore, the number of networks and the size of their exclusive buffers should be kept minimal.

3.4.1 Network Connectivity

The *Gen tools reduce the size of the state vector by instantiating only the network links that are actually used by the coherence protocol in a specific network. For example, if a coherence controller model only receives messages from a specific network, but never uses it to send messages, the outgoing network links can be pruned. As every network link is implemented in the model by a buffer removing unused buffers reduces the system state size.

3.4.2 Network Buffer Sizes

The designer can either manually define the size of the buffers in the network model or use the buffer sizes chosen by the *Gen tool.

If the network model requires buffers to represent the network link, the *Gen tool initially chooses the network link buffers to be twice the size of the maximum number of messages the source coherence controller can send in any coherence transaction. This allows the *Gen tools to verify the deadlock freedom of a coherence protocol for a minimum of two concurrent coherence transactions.

If the network model has network link buffers, the *Gen tool sets the ordered receive buffer size to one. The messages of the coherence transactions are stored in the network links, while the receive FIFO buffer enforces that for every potential interleaving all messages from different sources are serialized to detect deadlocks.

If the network model does not model network link buffers, the *Gen tool sets the receive buffer sizes to two times the maximum number of messages any coherence controller can send as part of a coherence transaction.

Using the Murφ model checker the maximum buffer size for a given system configuration and number of cache lines can be determined.

Whether the buffer sizes were chosen too small can be inferred from a warning printed by Murφ framework. The warning is issued if a message cannot be served at a coherence controller, because the network has not enough capacity to accommodate the response messages. Stalling the message can cause the Murφ model checker to fail to detect a violation of a correctness condition like deadlock freedom.

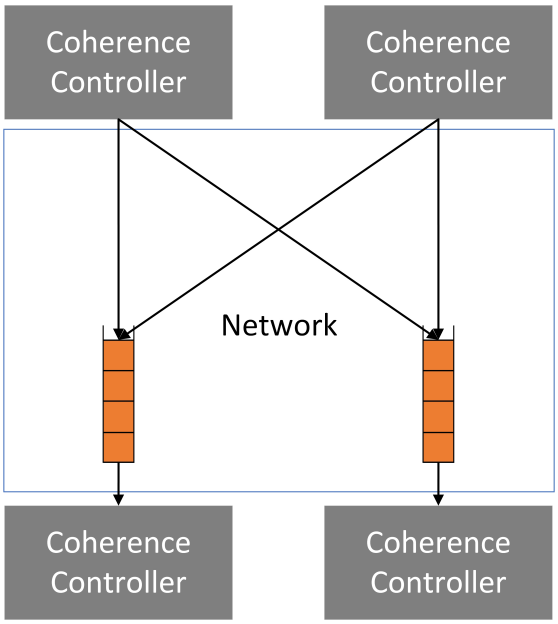
If the network buffer sizes are chosen too large, the Murφ model checker will correctly verify the coherence protocol, but the state vector size is unnecessarily increased. This

increases the memory consumption of the model checker which is a common issue when verifying large systems. When verifying the correctness of the SSP input protocols for a flat hierarchy using the *Gen tools, the maximum buffer sizes are reported by the Murφ model checker. If the number of coherence controllers and addresses is not increased when verifying hierarchical or heterogeneous architectures, the maximum required buffer size remains identical for every level and network. By reducing the buffers size to the required maximum numbers reported, the state vector size can be reduced.

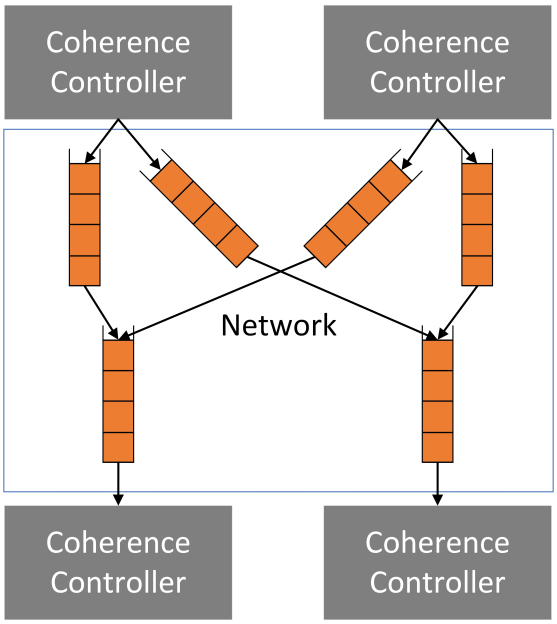
3.5 Summary

In this chapter we have presented the Murφ model checker framework components and briefly discussed the the interaction of the *Gen tools with the framework automatically optimizing the system model to reduce the state space of the model reducing verification time. In the next chapter 4 we introduce the ProtoGen Protocol Language (PPL) showing the relation between the PPL network types and the Murφ model checker framework network templates.

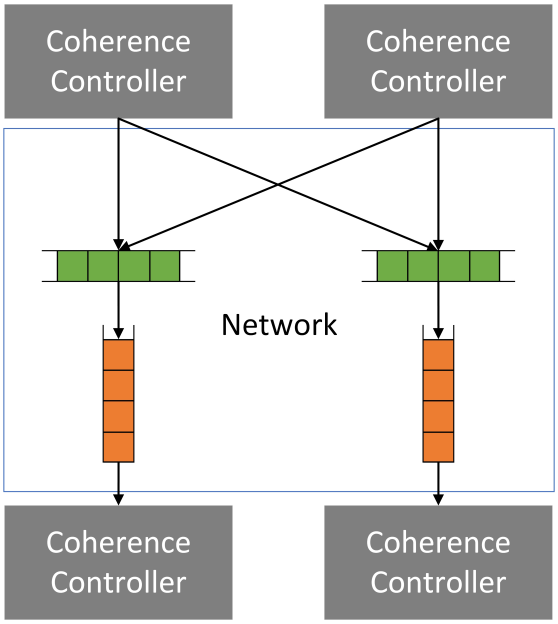
(a) Ordered Buffer and Unordered Buffer Representation



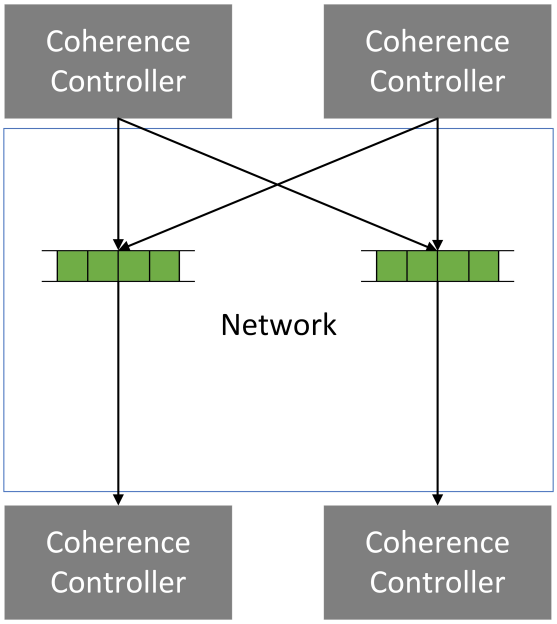
(b) Atomic Network



(c) Ordered Network



(d) Unordered Network



(e) Set Network

Figure 3.1: Murφ Framework Network Types

Chapter 4

ProtoGen Protocol Language

In this chapter the ProtoGen Protocol Language (PPL) and transition graph representation used in subsequent chapters are introduced. In addition, the PPL reference provided will allow architects who want to use the publicly available *Gen tools to understand the provided examples better.

The primary input to the *Gen tools is a high-level specification of a stable state protocol (SSP) described in the PPL, which is a high-level domain specific language (DSL). It enables the designer to define the structure of any machine (e.g. caches and directories) as well as its architectural behavioral specifications. The behavior of a machine is described as concurrent asynchronous processes in a sequential and atomic format. PPL is inspired by other previously proposed DSLs for coherence protocols including Teapot [39] and SLICC [40], but also by the Murϕ Model Checker language [26]. PPL is an imperative, procedural, event-driven programming language [41].

The PPL parser translates the input SSP descriptions into the intermediate representation (IR) transition graph format used by all algorithm stages and various backends.

4.1 PPL in a Nutshell

The PPL was designed to allow the user to describe interactions between machines by passing messages over interconnects. The SSP specification encompasses the behavioral description of the cache controller and the directory controller. The cache controller and the directory controller are defined in PPL as two specific machines using the Cache and Directory keyword. In addition, PPL allows the designer to define networks and custom message type objects.

In the PPL the SSP cache and directory controller transactions like shown in in Tables 1.1 and 1.2 are described as independent processes that communicate with each other by passing messages.

4.1.1 Process

A process describes the behavior of a specific machine e.g. cache controller when receiving a specific message or observing a specific event. Like shown in Listing 4.1, the process definition encompasses a start state, a guard and an optional final state.

A process will be triggered if the current state of the coherence controller matches the process start state and if the guard matches the identifier of a message received on any interconnect or the identifier of an internal event.

If no final state for the process is defined, the start state is considered to be the final state of the machine once the process completes. If the final state depends on the process control flow, it can be declared to be variable using an arbitrary identifier. In the process body, the final state can be assigned to the final state identifier. When the process finishes, the final state is assigned as the current state of the coherence controller.

If the process completes or a subsequent guard is reached, the received message will be automatically cleared from the input buffer or the event will be signalled to have completed, unless it is explicitly stalled using the *stall* keyword.

A process either completes when the end of the process body is reached or if an *exit* statement is executed. All active processes are considered to be running concurrently. Once a process is triggered, it continues to run until it exits.

4.1.1.1 Conditional Statements

In addition to *if-else* statements that operate on machine or message member variables, the PPL language introduces the await-when statement. The *await-when* statement is used if a request message was sent by a process awaiting a response message. The process loops at the *await* statement until a message identifier matching a *when* guard definition is observed. After the instructions guarded by the *when* condition have been served, the process returns to the initial *await* statement waiting for subsequent messages to arrive. To exit an *await* statement the *break* keyword must be used. *Await* statements can be nested, but the *break* keywords only exits the *await* statement in which it is used. In Listing 4.1 an example of the await-when statement is given.

```
1  // A load access from a CPU triggers the process
2  Process(CurrentState == I, load, FinalState){
3      ...
4      await{                          // The process loops awaiting one or multiple message
5          when GetS_Ack:              // Is a message with the GetS_Ack identifier received ?
6              ...
7              FinalState = S;         // Assign the State S as the final state
8              break;                  // Break the await loop to reach end of process
9      }                               // Implicitly assign the new final state S to CurrentState
```

Listing 4.1: Process definition

4.1.2 Types

PPL supports a range of standard data types like *bool, int, arrays* and *sets*, but also introduces new protocol specific data types like *Machine (Cache and Directory), Message, Event, State, Address, Data, ID* and *Timestamp*. The PPL specific data types have certain properties that allow the designer to describe an SSP in a concise format. Member variables of objects can be accessed by using the (.) operator.

4.1.2.1 Network

The network types provided in PPL are equivalent to the network types provided in the Murφ Model Checker Framework 3. The *Gen algorithms do not modify the network type specification provided in the SSP input descriptions. The network type information is used by different backends to automatically generate or configure the networks of the output models or test systems.

```
1  Network { Unordered fwd;
2            Set resp;
3            Ordered req;
4            };
```

Listing 4.2: Message type declaration

4.1.2.2 Message

PPL allows the user to define custom message types. The custom message types inherit from the base PPL *Message* type whose member variables determine the message header format allowing the *Networks* and *Machines* instances to process them. The custom message types allow the designer to declare variables representing the message payloads.

A custom message object can be sent on an arbitrary network irrespective of its payload declarations because it inherits from the base PPL *Message* type shown in Listing 4.3. An example for a custom message type declaration having a variable of type *Data* as member representing the payload is given with its constructor in Listing 4.4. The first arguments passed to the constructor of *Resp* must be the arguments required to construct an object of the parent type *Message*. The subsequent arguments must be given in the order in which the member variables of the *Resp* type are declared.

```
1  Message {
2      Address address;    // Address of the cache line
3      String msg_id;      // Message identifier e.g. 'Get'
4      ID src;             // Unique ID of the source machine sending the message
5      ID dest;            // Unique ID of the destination machine the message is routed to
6  }
```

Listing 4.3: Message type declaration example

```
1  Message Resp{           // Message of type Resp inherits from parent type Message
2      Data cl;               // Payload of type Data
3  };
4
5  ...
6
7   // A message with identifier Fwd_GetM triggers the process
8  Process(CurrentState == M, Fwd_GetM, FState){
9      // Construct a message of type Resp, passing address, msg_id, src, dest, payload
10     msg = Resp(Fwd_GetM.address, Get_Ack_D, ID, directory.ID, cl[Fwd_GetM.address]);
11     req.send(msg);       // Send Resp type message on network named req
12     ...
13 }
```

Listing 4.4: Custom message type declaration and constructor example

4.1.2.3 Accesses

PPL reserves the keywords *none, load, store* as access specifiers. The designer must use these keywords when a process associated with an access operation completes so that the *Gen tools can deduct the access permissions for every state from the PPL SSP description. The process can either be guarded by a message or alternatively directly by an access specifier.

If a process is guarded by an access specifier PPL assumes the associated machine to be directly connected to a compute unit using a standard load/store unit interface. When

the process completes it either yields the loaded cache line data or a store completion acknowledgement to the compute unit.

Furthermore, PPL automatically interprets unknown guards e.g. *acquire* that are not recognized as message identifiers used in a message constructor in any machine - see section 4.1.2.4 - as accesses from a compute unit. However, while PPL automatically assumes the standard load/store interface, the designer must specify whether the new guard exposes a *load* or *store* behavior. This allows the designer to describe any access in PPL including atomic operations.

An example performing a store access is provided in Listing 4.7 and 4.8.

In addition to the access specifiers PPL reserves the keyword *evict*. A coherence controller may issue an *evict* operation for any cache line at any time. If for the current state of the cache line a process guarded by *evict* is defined, the process is executed. Otherwise, the *evict* operation is stalled until the cache line reaches a state that permits the *evict* to be performed.

4.1.2.4 Machine

A machine declaration in PPL consists of an *Machine* type object declaration part and a behavior description part.

- In the *Machine* type object declaration the variables representing the auxiliary states are defined.
- In the behavioral description, processes are listed that define the behavior of a coherence controller for a specific state for one or a sequence of events like an access, an internal controller event or a remote message.

While the *Machine* type can be used to describe coherence controllers it does not leverage domain specific knowledge like *Cache* and *Directory* type coherence controllers presented in section 4.1.2.5 that specifically simplify the description of coherence protocols. Therefore, the *Machine* type can also be used to describe e.g. simple trace processors or global observers monitoring invariants.

4.1.2.5 Cache/Directory

SSPs generally specify the coherence transactions for a single memory block and hence do not explicitly include addresses as indexes in the behavioral description. To cater to this SSP characteristic, PPL introduces the *Cache* and *Directory* types to simplify the SSP description. They are variations of the *Machine* type, but PPL implicitly considers all member variables

defined by the designer to be arrays indexed by an *Address* instead of Singletons. To declare a member variable that is not an array indexed by *Address*, the keyword Singleton must be used. In Listing 4.5 and 4.6 two equivalent machine and cache object declarations are given in PPL. In each declaration the *State* and *Data* member variables are arrays indexed by an *Address* and the *Timestamp* is declared as a Singleton. For each type declaration a constant number of objects specified by the designer is initialized in line 5.

Using the *Cache* and *Directory* types the declarations and behavioral description remain more concise when describing an SSP.

```
1 Machine {
2     State<Address> = I;
3     Data<Address> cl;
4     Timestamp[0..maxTS] timestamp = 0;
5 } set[ConstNumCaches] m_cache;
```

Listing 4.5: Machine type declaration

```
1 Cache {
2     State = I;
3     Data cl;
4     Singleton Timestamp[0..maxTS] timestamp = 0;
5 } set[ConstNumCaches] c_cache;
```

Listing 4.6: Cache type declaration

In addition, every machine has an unique ID that is used by the networks to route messages. The unique ID is stored in a constant member variable of type *ID* when a machine object is instantiated.

Analog to the varying *Machine* and *Cache/Directory* type object declarations, the behavioral descriptions differ. In processes associated with *Cache* or *Directory* type objects it is not required to explicitly index the array variables using addresses or to pass them as parameters to constructors. PPL implicitly uses the address provided by the guard message when accessing elements in arrays that use addresses as index. An element with a different address can be accessed in an array by using the address as the index.

In Listing 4.7 and 4.8 an identical behavioral description is presented for the *Machine* type and for the *Cache* type. While it is uncommon for programming languages to implicitly consider variables to operate on a common index unless stated otherwise, it simplifies the PPL description of coherence protocols for (per-address-) cache blocks leveraging domain specific knowledge. Therefore, the designer can directly transcribe the SSPs from the commonly used table and sequence flow specification formats [5; 35; 1; 19; 27] without being required to manually index array variables like in Teapot [24].

```
Architecture m_cache {                          // m_cache is a Machine type object
    ...
    // Compute unit issues store, check if current State for address is I
    Process(CurrentState[store.address] == I, store, FState){
        // Request exclusive cache line
        msg = Request(store.address, GetM, ID, directory.ID);
        // Send msg on network req
        req.send(msg);

        // Wait for response message to arrive
        await{
            when Data:
                // Check if response address matches the store address
                if (store.address == Data.address){
                    // Assign message cache line to element of cl array at index address
                    cl[Data.address] = Data.cl;
                    FState = M;      // Assign final state
                    store;           // Acknowledge store completion
                    break;           // Exit the await loop
                }
            ...
        }
    }   // End of process
}
```

Listing 4.7: Machine process definition example

```
Architecture c_cache {                          // c_cache is a Cache type object
    ...
    // Compute unit issues store, check if current State is I
    Process(I, store, FState){
        // Request exclusive cache line
        msg = Request(GetM, ID, directory.ID);
        // Send msg on network req
        req.send(msg);

        // Wait for response message to arrive
        await{
            when Data:
            // Assign message cache line to element of cl array at index address
                cl=Data.cl;
                FState = M;      // Assign final state
                store;           // Acknowledge store completion
                break;           // Exit the await loop
            ...
        }
    } // End of process
}
```

Listing 4.8: Cache process definition example

4.1.2.6 Events

The handling of events spanning across multiple cache lines is required to correctly implement consistency oriented coherence protocols. *Events* are closely related to *Messages* and can be interpreted as broadcasts to all cache lines within a machine. The designer can define custom event objects to make information visible to other cache lines. Custom event types inherit from a base *Event* type shown in Listing 4.9.

```
1 Event {
2     Address address;    // Address of the cache line issuing the event
3     String event_id;    // Event identifier e.g. 'ev_release'
4 }
```

Listing 4.9: Event type declaration

The syntax for declaring custom events is analog to the syntax of custom messages declarations. Instead of sending an event on a network, events are pushed into an event queue declared as a member of a specific machine. To complete, the event at the head of an event queue must be performed on all cache lines except the issuing cache line. An event is considered to be served by a cache line once the process triggered by the event has successfully completed. Once the event has completed it is presented to the issuing cache line which must pop it from the event queue. Using *await-when* statements concurrency between the completion of an event and the receiving of messages can be explicitly described in PPL. A machine can have multiple *EventQ* definitions allowing events assigned to different queues to be served concurrently.

Events and messages are equivalent from the perspective of a process. However, while a message may be sent without eventually receiving a response, an event that was issued by a cache line must eventually be popped from the event queue to avoid a deadlock.

4.2 Parser

4.2.1 Intermediate Representation (IR) Format and Notation

The PPL parser converts the input PPL SSP descriptions for each controller into a transition graph denoted as $G_{controller}$. The transition graphs are the intermediate representation (IR) used by all *Gen algorithms and backends. The transition graph's nodes are the stable and transient states, while the edges represent the state transitions and describe any instructions like sending of messages or updating auxiliary states that must be performed.

However, the *Gen algorithms mostly operate on subgraphs or single transitions instead of the transition graph of an entire controller.

From the controller transition graphs $G_{controller}$ subgraphs can be extracted. A subgraph that encompasses all states and transitions that are present in any path between a given start state S_{start} and final state S_{final} is denoted as $G_{S_{start} \to S_{final}}$. If a subgraph has multiple start and/or final states it is denoted as $G_{\{S_{start_i}, S_{start_j}, \ldots\} \to \{S_{final_m}, S_{final_n}, \ldots\}}$.

From the transition graphs single transitions can be extracted. A transition T is an object that encompasses a *Guard* that is an access, event or message; a start state S_{start} and final state S_{final} as well as a sequence of instructions IN.

4.2.2 PPL Parsing

From the input PPL SSP description, the PPL parser generates the transient states and transitions clustering the instructions. If an access initiates a process transitioning from a start state S_i to a final state S_j involving the issuing of a request and awaiting for a single response, the PPL parser would add a single transient state ST_0 that reflects the situation in which the request has been issued but the response has not yet been received. All PPL instructions before the *await* statement describe the behavior of the first transition T_0 from state S_i into ST_0, while all instructions after describe the behavior of the second transition T_1 from ST_0 in the state S_j. In Figure 4.1 the transition graph $G_{S_i \to S_j}$ is shown. If the process involves multiple subsequent responses, the PPL parser would add a transient state for each response.

Figure 4.1: PPL parser generated transition graph $G_{S_i \to S_j}$ for a single *await* statement expecting a single response

By automatically generating the transient states and clustering the PPL instructions into transitions, the PPL parser effectively rewrites the input PPL specification using a unique process for each transition. The transition graph $G_{S_i \to S_j}$ 4.1 can be transpiled into a PPL description like shown in Listing 4.11. While not required, the designer can also choose to describe the SSP directly as a transition graph manually handling the complexity defining all transient states that are not explicitly defined in the input SSP table. In Listing 4.10 the equivalent PPL input is shown using *await* statements.

```
1  Process(S_i, access, S_j){
2          // T_0: Start of transition behavioral description
3          op0;                          // PPL instruction e.g. assignment
4          op1;
5          req.send(Request);
6          ...
7          // T_0: End of transition behavioral description
8          await{                        // PPL parser generates state ST_0
9              when Response:            // Response message triggers transition T_1
10                 // T_1: Start of transition behavioral description
11                 op3;
12                 ...
13                 exit;                 // Exit the process, enter state S_j
14                 // T_1: End of transition behavioral description
15             ...
16         }
```

Listing 4.10: PPL description of example input process to PPL parser with state start state S_i and final state S_j

```
1  Process(S_i, access, ST_0){  // T_0 transition
2          op0;                          // PPL instruction e.g. assignment
3          op1;
4          req.send(Request);
5          ...
6  }
7
8  Process(ST_0, Response, S_j){ // T_1 transition
9          op3;
10         ...
11 }
```

Listing 4.11: PPL description of transitions with PPL parser generated transient state as equivalent processes

In addition to *await* statements the PPL parser also needs to handle conditional statements. The final state of a transition may differ depending on whether a condition protecting an assignment of S_k to the process final state is true or false. If the PPL parser detects a diverging process control flow due to a conditional final state assignment, it generates for both outcomes of the condition a distinct transition and, if the transition end state is not final, also a new transient state. If in ST_0 the transition T_1 would have a conditional statement evaluating an auxiliary variable X resulting into either S_j (true) or S_k (false) being assigned as the process final state, the PPL parser replaces T_1 with the distinct transitions $T_1(X)$ and $T_1(!X)$. Having distinct transitions is an essential prerequisite enabling the HieraGen and HeteroGen algorithms to fuse coherence protocols by concatenating transitions as the final coherence

states may have different properties e.g. access permissions. The resulting transition graph $G_{S_i \to \{S_j, S_k\}}$ is shown in Figure 4.2.

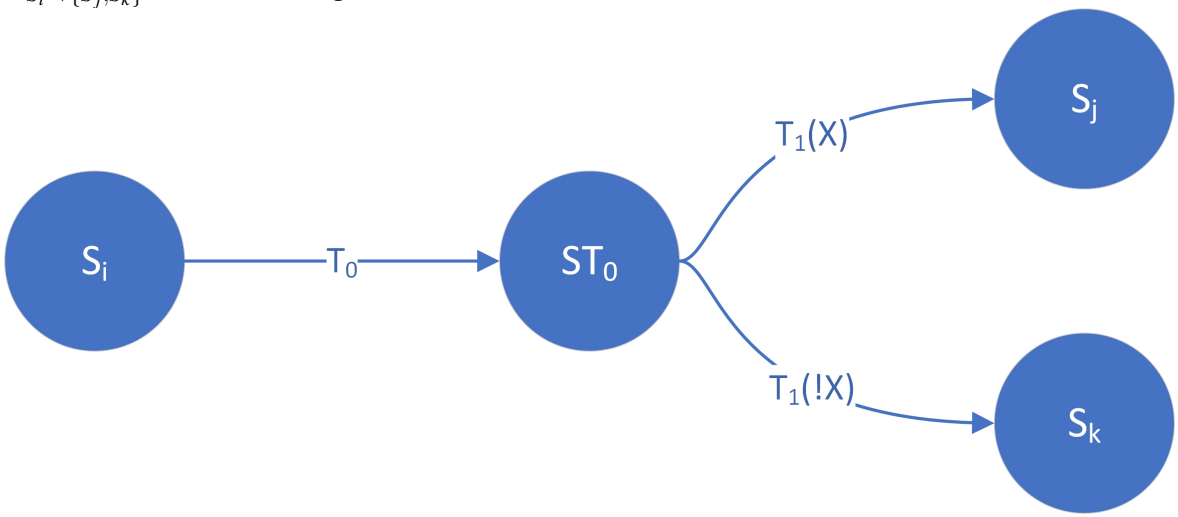


Figure 4.2: PPL parser generated transition graph $G_{S_i \to \{S_j, S_k\}}$ for a single *await* statement expecting a single response with a nested conditional statement. The final state depends on the conditional statement that evaluates the value of the auxiliary variable X.

```
1  Process(Si, access, FState){
2          // T0: Start of transition behavioral description
3          op0;                          // PPL instruction e.g. assignment
4          op1;
5          req.send(Request);
6          ...
7          // T0: End of transition behavioral description
8          await{                        // PPL parser generates state ST0
9              when Response:            // Response message triggers transition T1
10                 // T1(X) & T1(!X): Start of transitions behavioral description
11                 op3;
12                 ...
13                 // Conditional statement dependent evaluates value of variable X
14                 if (X == True){              // T1(X)
15                     op4;                     // ...
16                     FState = Sj;         // T1(X)
17                 } else {                     // T1(!X)
18                     op5;                     // ...
19                     FState = Sk;         // T1(!X)
20                 }
21                 op6;
22                 ...
23                 exit;                 // Exit the process, T1(X) → Sj | T1(!X) → Sk
24                 // T1(X) & T1(!X): End of transitions behavioral description
25             ...
26         }
```

Listing 4.12: PPL description of example input process to PPL parser with state start state S_i and conditional statement dependent final states S_j and S_k

Any instruction prior to the conditional statement are replicated in both transitions followed by the conditional statement, which is simply negated in one transition to accept the previous false outcome. Subsequent to the conditional statements the instructions performed by the different control flows are added to the respective transitions. In Listing 4.12 the PPL description using *await* statements and in Listing 4.13 the equivalent PPL transition process description are given.

```
1  Process(S_i, access, ST_0){  // T_0 transition
2          op0;                          // PPL instruction e.g. assignment
3          op1;
4          req.send(Request);
5          ...
6  }
7
8  Process(ST_0, Response, S_j){  // T_1(X) transition
9          op3;
10         ...
11         if (X == True){
12             op4;
13         }
14         op6;
15         ...
16 }
17
18 Process(ST_0, Response, S_k){  // T_1(!X) transition
19         op3;
20         ...
21         if !(X == True){
22             op5;
23         }
24         op6;
25         ...
26 }
```

Listing 4.13: PPL description of transitions with PPL parser generated transient state as equivalent processes with conditional statements

If multiple processes with identical guards exist like shown in Listing 4.13, all instructions outside of unique conditional statements must be specified by the designer in the same sequence. This is necessary to enable the current *Gen base backend implementation to merge the independent transitions again into a single sequential process description.

4.2.3 MSI Protocol Example

The SSP cache behavior for an MSI coherence protocol is described in Table 1.1. A cache can initiate one of five different coherence transaction types $I \rightarrow S$, $I \rightarrow M$, $S \rightarrow M$, $S \rightarrow I$, and

$M \rightarrow I$. In Listing 4.14 an example of the PPL behavioral description of a cache performing a coherence transaction initiated by a *store* access in state I transitioning to state M is given.

The cache controller can transition from the state I to M by taking two distinct paths encompassing different transitions. The transition path taken by a controller in a coherence transaction depends on the state of the block at the directory: one path with a single data response (if the block is in I or M at the directory) and the second path with multiple responses.

To account for this, the PPL parser creates transient states as shown in the transition graph $G_{I \rightarrow M}$ in Figure 4.3. The initial store access initiates the coherence transaction having the cache sending out a request *GetM* to acquire exclusive ownership of the cache line. To track the completion of the outstanding request a transient state here denoted as $\mathrm{IM}^{\mathrm{AD}}$ (waiting for data as well as potentially acknowledgments) is created due to the *await* statement in line 11 of the PPL input description.

If the response message received is labeled *Data* consisting of only data, the cache block will directly transition to M state as defined in lines 11 to 14. The response message *DataAck* includes both data and an expected count of acknowledgments, causing the cache controller to transition to another transient state called IM^{A} (waiting for acknowledgments) which is again implicitly specified in the PPL description through the *await* statement in line 20. Whether the reception of an invalidation acknowledgment message *InvAck* causes the cache controller to transition to M state depends on the conditional statement shown in line 24 evaluating whether all acknowledgement messages have been received. The PPL parser generates two transitions to represent the conditional statement behavior in the transition graph $G_{I \rightarrow M}$. One transition describes the cache controller behavior if the condition is true and the *Last InvAck* has been received causing the coherence controller to transition into state M as described in lines 24 to 26. The transition describing the cache behavior if the condition is false because the cache controller has not received all acknowledgements yet, causes the cache to remain in state IM^{A}.

4.3 Backend

All backends use a base backend that reverses the operations performed by the parser converting the IR transition graph of every coherence controller again into sequential tokenized process descriptions. The generated sequential descriptions have been successfully transpiled into Murφ model, gem5 SLICC [42], SystemVerilog or Chisel [43].

```
1  ...
2  // Cache behavioural description
3  Architecture cache {
4      ...
5      Process(I, store, FState){
6          msg = Request(GetM, ID, Dir.ID);
7          req.send(msg);
8          acksReceived = 0;
9
10         await{
11             when Data:
12                 cl = Data.cl;
13                 FState = M;
14                 exit;
15
16             when DataAck:
17                 cl = DataAck.cl;
18                 acksExpected = DataAck.acksExpected;
19
20                 await{
21                     when InvAck:
22                         acksReceived = acksReceived + 1;
23
24                         if acksExpected == acksReceived{        // Last Inv-Ack
25                             FState = M;
26                             exit;
27                         }
28                     }
29         }
30     }
31     ...
32 }
```

Listing 4.14: PPL description of the behavior of the MSI cache controller for a store access in state I transitioning to state M

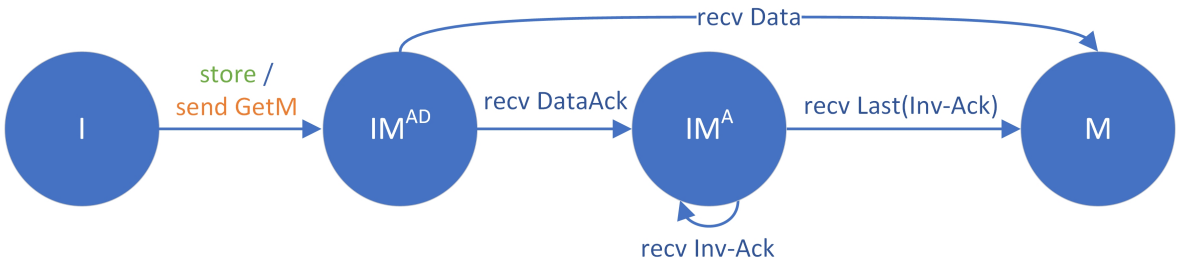


Figure 4.3: Transition graph $G_{I \to M}$ encompassing transitions describing the behavior of the cache controller for a *store* access in state *I* transitioning to state *M*

4.3.1 Summary

In this chapter we have introduced the ProtoGen Protocol Language (PPL) that allows architects to specify cache coherence protocols with a small number of constraints. We describe the PPL syntax and data types and point out - where applicable - their connection to the Murφ model checker framework. Furthermore, we explain the notation of transitions as well as transition graphs used in the subsequent chapters of this thesis.

Chapter 5

SSP Pre-Processing

The *Gen tools process the input SSPs using a sequence of algorithms to generate - depending on the tool input configuration - the concurrent hierarchical and/or heterogeneous coherence controllers as output.

In this chapter we present three SSP pre-processing steps shown in Figure 5.1. Step 1 is essential for the *Gen algorithms - presented in subsequent chapters - extracting information leveraged by these about the interaction between the separately described cache and directory controllers during a coherence transaction. Using the extracted information, for example the identifiers of requests that the directory forwards to the caches are identified. We discuss the details of Step 1 in section 5.1.

The steps 2 and 3 are not essential for the correctness of the subsequent *Gen algorithms, but focus on the optimization and improvement of the input SSP. Step 2 as described in section 5.2, provides hints to the programmer focusing on the reduction of stalls in the SSP due to race conditions among messages in an atomic coherence transaction. In section 5.3 describing step 3, the SSP description is automatically updated resetting the values of stale auxiliary variables to reduce the model checking state space.

5.1 Step 1: State Space Checker (SSC)

The ProtoGen state space checker (SSC) leverages the Murφ model checker [26] to not only verify the correctness of the atomic SSP description with respect to deadlock freedom and user defined invariants, but also to identify the interaction between the coherence controllers - cache and directory - when performing an atomic coherence transaction, see chapter 2.3. The SSC must identify the interaction between the coherence controllers, because the behavior of the state machines of the input cache and directory controller can be described independently

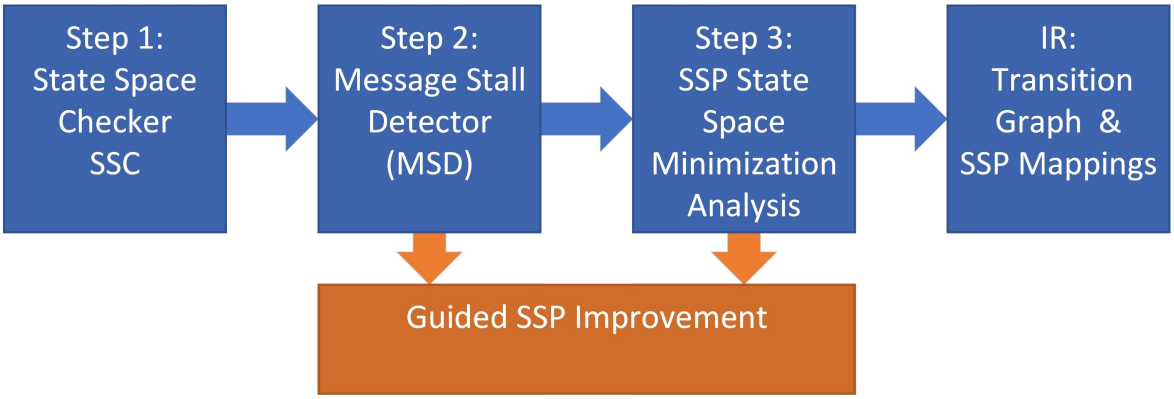


Figure 5.1: SSP input pre-processing stage pipeline

in the PPL. The atomic coherence transactions are analysed to identify accesses, requests, responses, coherence states and their relations. All *Gen algorithm rely on the information extracted by the SSC.

5.1.1 State Space Exploration

Every coherence controller is a finite state automata that can be defined as a quintuple:

$$(S, \Sigma, s_{Init}, \delta, F)$$

- S: Set of coherence controller states with $S = S_{stable} \cup S_{transient}$
- Σ: Input set of all accesses and coherence messages
- s_{Init}: Initial state of coherence controller
- δ: State transition function $\delta : S \times \Sigma \rightarrow S$
- F: Empty set of final states

Coherence controllers do not have final states F as they continuously need to change their coherence state serving a potentially infinite number of accesses and coherence messages.

To correctly enforce coherence, the finite state automata of the coherence controllers are required to interact with each other. To identify the communication between the different controllers and the states traversed when performing a coherence transaction SSC generates a Murφ model checker representation of the input SSP. The number of each type of coherence controller that must be modelled is provided in the PPL SSP description, see section 4.1.2.5.

SSC allows only a single coherence transaction to be in flight at any time. A coherence transaction is initiated by a coherence controller because of a memory access operation. The

coherence transaction is considered to be finished if all messages sent on the interconnect have been consumed by the receivers and no coherence controller is executing a state transition.

Because SSC enforces the coherence transaction to complete atomically all networks can be implemented as sets. This avoids the Murφ model checker to terminate because of a deadlock caused by a message race that the designer failed to consider in the SSP. Two response messages from different coherence controllers to the same receiver can race causing a deadlock unless they can be reordered if required. Potential deadlocks are resolved in Stage 2 of the SSP processing pipeline described in section 5.2 leveraging the output of the SSC.

The SSC records for every coherence controller the transitions performed during a specific coherence transaction that the Murφ model checker reports exploring the state space beginning form the coherence controller initial states.

The algorithm for the Murφ Model checker performing a breath-first search is shown in Algorithm 1.

The start and final states of the coherence controllers in a transaction are the coherence protocol stable states.

Algorithm 1 Murφ model checker state space checker procedure (Breadth-first)

▷ *GSCS: List of all potential global stable coherence states computed by taking the Cartesian product of all coherence controller's possible stable states*

GSCS := $\{S_{stable,0} \times S_{stable,1} \times ... \times S_{stable,x}\}_C \times \{S_{stable,0}\}_D$;

▷ *MT: Union of all cache and directory controller*

$\text{MT}_{Init} = \{c_0, c_1, ..., c_x, d_0\}$;

▷ *GCS function extracts the coherence states from the controllers constructing the global coherence state*

GCS() := $\{S(c_0) \times S(c_1) \times ... \times S(c_x)\} \times \{S(d_0)\}$;

▷ *CT: List (Trace) of coherence transitions performed by the coherence controllers completing an atomic coherence transaction*

CT = [];

▷ *VGSCSG: Records the Valid Reachable Global Stable Coherence State Graph in a dictionary. The key is the tuple ($GSCS_{Start}$, $GSCS_{End}$) of a VGSCSG edge start and final state. The value is a set of transactions represented by the state transitions performed by the coherence controllers involved*

VGSCSG = $\{(\text{GSCS}_{Start}, \text{GSCS}_{End}), \text{CT}\}$;

11: Accesses = [List of accesses defined in SSP]
12: ▷ *The Murφ state is constructed from the union of machines (MT), the last global stable coherence state observed (GCS) and a trace of transitions taken (CT):*
13: S_{Init} = (MT: MT_{Init}, GCS: GCS(MT_{Init}), CT: []);
14: Next = [S_{Init}]; ▷ *Queue of next Murφ states*
15: Served = [S_{Init}]; ▷ *List of previously observed Murφ states*
16: **while** Next **do**
17: S_{Cur} = Next.Pop(); ▷ *Get next Murφ state from Queue*
18: ▷ *Iterate over transitions (Murφ rules) whose guard conditions are true*
19: **for each** T ∈ ReadyTransitions(S_{Cur}) **do**
20: ▷ *To guarantee that transactions are performed atomically a transition with a guard that is an access like load/store/evict must not be performed if the current global coherence state does not only contain stable states*
21: **if** T.Guard ∈ Accesses ∧ S_{Cur}.GCS ∉ GSCS **then**
22: continue;
23: **end if**
24: ▷ *Transition T is performed updating the cache controller state*
25: MT_{Next} = T(S_{Cur}.MT);
26: ▷ *Check if new global coherence state tuple only contains stable states*
27: **if** GCS(MT_{Next}) ∈ GSCS **then**
28: ▷ *GCS contains only stable coherent states indicating that previous transaction has completed. Construct next Murφ state, GCS entry is updated and CT is reset as current transaction has completed*
29: S_{Next} = (MT_{Next}, GCS(MT_{Next}), []);
30: ▷ *Store the transitions constituting the transaction as edge in the VGSCSG graph*
31: VGSCSG[(S_{Cur}.GCS, S_{Next}. GCS)].add(S_{Cur}.CT + [T]);
32: **else**
33: ▷ *Construct next Murφ state, GCS is not updated, append T to CT*
34: S_{Next} = (MT_{Next}, S_{Cur}.GCS, S_{Cur}.CT + [T]);
35: **end if**
36: ▷ *If next Murφ state was not observed before enqueue it for future exploration*
37: **if** S_{Next} ∉ Served **then**
38: Next.push(S_{Next});
39: **end if**
40: **end for each**
41: **end while**

For each transaction, the SSC determines the start and final global coherence system state (GCS). First identical coherence controllers in a transaction are grouped into sets to allow coherence state symmetry reduction. Then the start and final coherence state sets for each controller group are extracted from the transaction. Using the coherence controller state sets the Cartesian product representing the GCS state is computed. If the input to SSC is a flat coherence protocol specification with N cache controllers C and one directory controller D the global state can be described by the Cartesian product:

$$S_{Global} = \{s_0, s_1, ..., s_n\}_C \times \{s_0\}_D$$

5.1.2 Valid Reachable Global Stable Coherence State Graph (VGSCSG)

From the coherence transactions, the valid reachable global stable coherence state graph (VGSCSG) can be constructed. All coherence controller state combinations represented by a GCS in the graph are valid, while all other combinations are forbidden. The forbidden GCS are used as additional invariants when verifying the optimized SSP of the coherence protocol in Stage 2 and Stage 3 of the SSP processing pipeline.

The VGSCSG can also be defined as a sextuple finite state automata:

$$(S_{Global}, \Sigma, \Gamma, s_{Init}, \delta, \omega)$$

- S_{Global}: Set of valid GCS
- Σ: Input set of all accesses that can be performed by distinct coherence controllers
- Γ: Output sets of coherence transactions
- s_{Init}: Initial GCS, product of the initial states of the system coherence controllers
- δ: GCS transition function $\delta : S_{Global} \times \Sigma \rightarrow S_{Global}$
- ω: Output coherence transaction set lookup function $\omega : S_{Global} \times \Sigma \rightarrow \Gamma$

In the VGSCSG for every possible input memory access performed by a specific coherence controller in its current coherence state, the output set contains one or multiple coherence transactions. The output set can have multiple transactions, if for a specific memory access the transitions encompassed in the transaction differ. This is for example the case, if not only the current GCS and the input memory access, but also the auxiliary states determine which coherence transaction must be performed.

Because the *Gen algorithms generally do not consider auxiliary states when optimizing and composing coherence protocols, the resulting protocols can be overly conservative. If for example an auxiliary state in form of a dirty bit is used to track whether a cache line is dirty, the *Gen algorithms consider all writebacks associated with an eviction in a coherence state with write permission to contain dirty data. Therefore, two distinct coherence transactions - one that writes dirty data back and one that does not - can exist in the output set for the same GCS state and for the same input.

While auxiliary states could be included in the GCS representation the implementation complexity for *Gen algorithms being required to infer additional information from the auxiliary states and their types would be increased. Instead the SSC issues a warning to the designer stating the conflicting transactions and the auxiliary state dependency. With the exception of timestamp based coherence protocols, most auxiliary state dependencies can be resolved by modifying the SSP including the related information directly in the coherence state. Furthermore, this also enables the designer to use the VGSCSG to determine which coherence controllers have certain access permissions and dirty data. An example for encoding the dirty bit auxiliary state in the coherence state is provided in section 2.2.3 discussing the RCC protocol.

In Figure 5.2 an example of the MSI VGSCSG for two cache controllers C and a directory controller D is shown. The nodes of the VGSCSG are the stable GCS connected by edges representing the memory access or evict operation that can be performed by a specific type of coherence controller in its current state. For example for the GCS $\{S,I\}_C \times \{S\}_D$, the distinct input store operations performed by the cache controllers C whose coherence state is either S or I will cause the GCS to change to $\{M,I\}_C \times \{M\}_D$. Even though the final state of both stores is identical, the coherence transactions associated with them are different. The final GCS for the distinct input load operations differ. The coherence transaction associated with a load performed by the cache controller C residing in state S does not alter the GCS state. It encompasses only a single transition for the cache initiating the transaction, performing all actions required to serve the load hitting in cache controller local memory. The load performed by a cache controller in state I results into a change of the GCS system to $\{S,S\}_C \times \{S\}_D$. Its associated coherence transactions fetches a copy of the cache line with read permissions for the cache performing the load.

The SSC is not limited to flat hierarchy SSP inputs, but can be used to explore the reachable state space of hierarchical and heterogeneous systems as well.

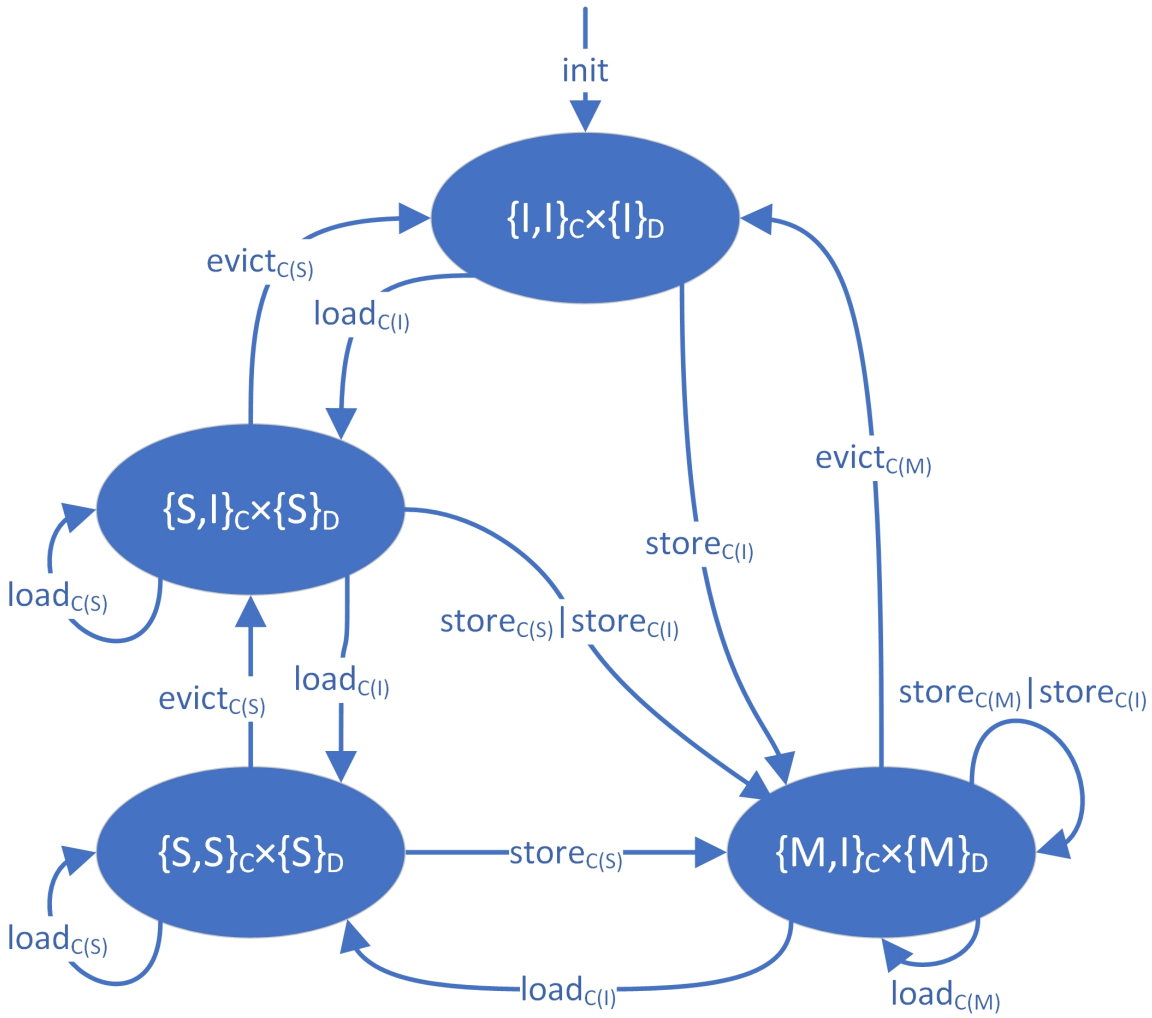


Figure 5.2: Valid Global Stable Coherence State Graph (VGSCSG) showing the reachable stable state space of an MSI coherence protocol for a system with two caches and one directory controller

5.1.3 Basic Sanity Checks

Using the transactions recorded, the SSC also checks for every coherence controller, if all coherence states are reachable and if all coherence controller transitions have been taken once. Transitions that are never taken and unreachable states are reported and can be pruned automatically.

5.1.4 Coherence Transaction Analysis

From the VGSCSG's atomic coherence transactions the relationship between accesses, requests and responses for every state can be extracted. While the tables presented in the

following subsections seem straightforward, they yield essential information for then *Gen algorithms when generating new or composing existing transitions.

5.1.4.1 Directory State to Cache State Relation Table (SSRT)

For every coherence transaction, the final state of the directory controller can be mapped to the final state of the cache controller that initiated the coherence transaction. However, it is possible that a directory controller stable state is mapped not only to one, but multiple cache controller stable states, because a cache may change its stable state due to an access triggering a transaction that does not alter the directory coherence state like a silent eviction or a silent upgrade. Therefore, all cache controller stable states that can be mapped to the same directory controller stable state are belonging to the same state set since the directory cannot distinguish them. Furthermore, for each state in a state set the access permissions of the cache controllers are determined. The state set access permission is the superset of all accesses that can hit in its states. Accesses that can be performed without notifying the directory like silent writes are considered to hit and must be listed.

The directory state to cache state relation table (SSRT) allows the *Gen algorithms to infer the potential cache controller states and access permission from the directory controller state.

Consider e.g. the MESI protocol. A cache controller is initially in stable state *E* maintaining a clean copy of the cache line and the directory controller is in stable state *X* having granted exclusive access permission to the cache controller. When a *store* access modifies the copy of the cache line, the cache controller transitions into the stable state *M* to track that the cache line has become dirty. The coherence transaction initiated by the *store* access does not alter the directory controller state. Therefore, the directory controller stable state *X* is mapped to both the cache controller state *E* and *M*. All states in the state set with directory state *X* are considered to have *read* and *write* access permissions, because the cache controller state *M* belongs to the state set. Since the cache controller states are non distinguishable based on the directory state they are considered to be part of the same state set. An example SSRT for the MESI protocol is given in Table 5.1.

5.1.4.2 Access to Request Relation Table (ART)

In the access to request relation table (ART) for every cache stable state S_{Cache} the possible accesses A_{Own} are mapped to corresponding requests R_{Own} send to the directory. In the Table 5.2 an example access to request mapping is given for an MSI coherence protocol. In case of

$S_{Directory}$	State Set $\{S_{Cache}\}$	Stable State Access Permission
X	$\{E,M\}$	$\{read, write\}$
S	$\{S\}$	$\{read\}$
U	$\{I\}$	$\{\}$

Table 5.1: MESI protocol directory to cache State Relation Table (SSRT)

S_{Cache}	A_{Own}	R_{Own}
M	store	-
M	load	-
M	evict	Put
S	store	Upgrade
S	load	-
S	evict	Put
I	store	GetM
I	load	GetS

Table 5.2: MSI protocol access to request relation table (ART)

S_{Cache}	R_{Fwd}	A_{Other}	R_{Other}
M	Fwd-GetM	store	GetM
	Fwd-GetS	load	GetS
S	Inv	store	Upgrade, GetM
I	-	-	-

Table 5.3: MSI protocol state to forwarded request relation table (FRT)

a store access , the cache issues a GetM request in state I. However, instead of also issuing a GetM request when serving a store access in state S the coherence protocol issues a request denoted as Upgrade. While the requests are both initiated because of a store access, the directory could not identify them as equivalent without the ART table. Using the ART the *Gen algorithms can for every coherence state translate accesses into requests and vice versa.

5.1.4.3 State to Forwarded Request Relation Table (FRT)

In the state to forwarded request relation table the forwarded requests R_{Fwd} that a cache can receive in a specific stable state S_{Cache} are listed. In addition, the accesses by other caches A_{Other} and associated requests R_{Other} that cause R_{Fwd} are listed. We again use our MSI protocol in the example given in Table 5.3 listing the state to forwarded requests relations. When a cache is in M state, it can receive a Fwd-GetM and a Fwd-GetS request from the directory. A cache will receive a Fwd-GetM request in state M if another cache wants to perform a store access sending a GetM request to the directory.

5.2 Step 2: Intra-Transaction Message Stall Detector (MSD)

Leveraging the coherence transaction patterns exposed by the VGSCSG message stalls in the SSP can be identified. When designers describe textbook protocols like given in Table 1.1 and Table 1.2 in PPL focusing on functional correctness, they may fail to identify racing messages taking different paths on the interconnect. If a message arrives in a controller state in which it is not specified as a guard of a transition, it is stalled. The stalled message may lead to a deadlock, because it can depending on the controller input buffer design block subsequent messages arriving on the same virtual channel.

5.2.1 Identify Stalled Messages and Dependent Transitions

To avoid needless message stalls the message stall detector (MSD) identifies message stalls that can be avoided by modifying the SSP. In the first step, for every global coherence transaction, the potential transition execution interleavings of the coherence controllers are analyzed to search for transient states in which a messages is getting stalled.

If transient states in which messages are stalled are found, the MSD identifies the transitions in the subsequently traversed transient states guarded by the stalled messages. The goal of the MSD is to determine whether the instructions performed by these transitions could be performed in e.g. the transient state S_i^T in which the coherence controller stalled the message M_k. A transition T_k that was defined to serve the message M_k in the transient state S_k^T transitioning into S_l^T could be performed in S_i^T, if it is logically independent from any other subsequent transitions preceding S_k^T.

- A transition is partially logical independent if it does not alter any auxiliary variables from which value the execution of subsequent transitions are conditionally dependent.
- A transition is completely logical independent if in addition none of its instructions are data-depend on any other subsequent transition auxiliary variable assignments.

Subsequent transitions are all transitions that are element of any path in the coherence controller transition graph from the current transient state S_i^T to the state S_k^T.

The MSD reports to the designer the stalls detected in the SSP together with their classification. While stalls associated with completely logical independent transitions can be automatically resolved by refining the SSP similar to the ProtoGen algorithm approach presented in chapter 6, partially logical independent transitions may only be resolved by rewriting the SSP. The designer must resolve the conditional dependency of the transition associated

with the stalled message on a subsequent auxiliary variable value assignment reported by the MSD.

5.2.2 MSI Protocol Example

Recall our MSI coherence protocol example given in Figure 4.3 performing an I to M transaction. If one or more remote cache controllers have a local copy of the cache line in shared state S, the directory controller responds to the requestor cache with a message DataAck containing the Data as well as the number of expected acknowledgements. In addition, the directory sends invalidation request messages Inv to the remote caches in shared state S. The remote caches then acknowledge the invalidation to the initial requestor cache by sending Inv-Ack messages.

Like shown in Figure 5.3, the DataAck message and the Inv-Ack messages subsequent to the Inv messages can take different paths in the network. While the DataAck message is directly sent to the requestor cache and the Inv messages must be processed by the remote caches before the Inv-Ack are sent, it can happen in Network-on-Chips due to e.g. a heavily congested link that an Inv-Ack message arrives prior to the DataAck message.

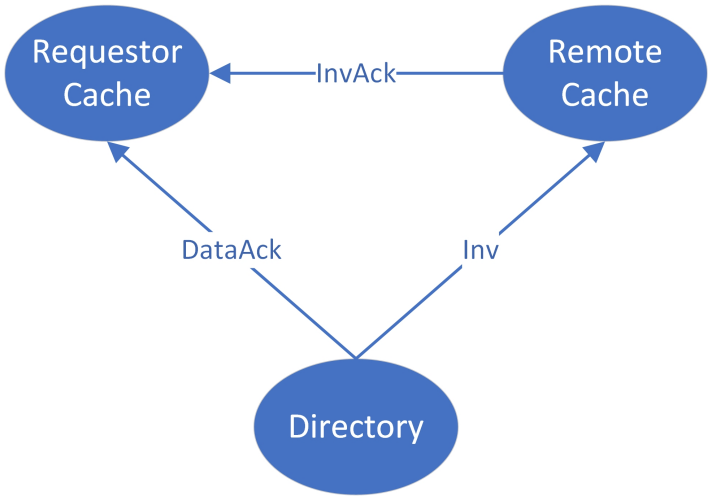


Figure 5.3: Response message race for an MSI coherence protocol. The requestor cache performs an I to M transaction, while a remote cache holds the cache line in state S.

In the past we observed that people working with ProtoGen failed to reason about this race condition when describing the SSP. The Inv-Ack can be stalled potentially blocking subsequent messages on the same virtual channel. This can cause an increase of coherence transaction latencies or even a deadlock if DataAck and Inv-Ack message are assigned to the same virtual channel.

```
1  ...
2  // Cache behavioural description
3  Architecture cache {
4      ...
5
6      Process(I, store, FState){
7          msg = Request(GetM, ID, Dir.ID);
8          req.send(msg);
9          acksReceived = 0;
10
11         await{
12             when Data:
13                 cl = Data.cl;
14                 FState = M;
15                 exit;
16
17             when DataAck:
18                 cl = DataAck.cl;
19                 acksExpected = DataAck.acksExpected;
20
21                 if acksExpected == acksReceived{      // Has Last(InvAck) been already received?
22                     FState = M;
23                     exit;
24                 }
25
26                 await{
27                     when InvAck:
28                         acksReceived = acksReceived + 1;
29
30                         if acksExpected == acksReceived{
31                             FState = M;
32                             exit;
33                         }
34                     }
35
36             when InvAck:  // While waiting for DataAck, when InvAck message is received,
37                 acksReceived = acksReceived + 1;        // increment ackReceived
38         }
39     }
40     ...
41 }
```

Listing 5.1: SSP description resolving message stalls detected by the message stall detector (MSD).

In the SSP transition graph, the transitions guarded by Inv-Ack are partially logically independent from the subsequent DataAck transition. This is the case, because the DataAck transition is not conditionally dependent on any auxiliary state variable altered by the Inv-Ack handling transitions. However, the Inv-Ack transition behavior is conditionally dependent on the number of expected acknowledgements assigned by the DataAck transitions to the

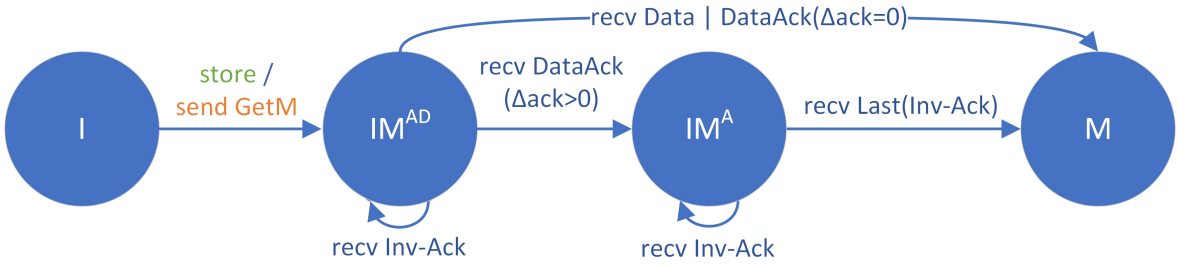


Figure 5.4: Transition graph $G_{I \to M}$ after the stall of Inv-Ack in state IM^{AD} has been resolved.

auxiliary variable acksExpected. Leveraging the information provided by the MSD, the SSP specification is rewritten resolving the stall of the Inv-Ack message in IM^{AD}.

In Listing 5.1 the stall is resolved, by performing the conditional check whether the number of acksExpected is equal to the number of acksReceived when serving the DataAck response message and by incrementing the acksReceived value while waiting for the DataAck response message. The new I to M transition graph is presented in Figure 5.4.

5.3 Step 3: SSP State Space Minimization

Designers focus on the functional correctness when describing SSPs, while keeping the reachable state space including the coherence and auxiliary states for verification purposes minimal is often only a secondary concern.

For example many textbook protocols MESI[1], CHI [5], TSO-CC [27], DeNovo [6] do not explicitly state when an auxiliary variable can be reset to its initial value to reduce the model checking state space as it does not relate to the coherence protocol functional correctness.

Reducing the state space is important due to the state space explosion problem [37] especially when verifying system models with a large number of caches and potentially different coherence protocols like in hierarchical and heterogeneous system.

While past publications [26; 44; 45; 46] have focused on model checker optimizations and methods to mitigate the state space explosion problem, the *Gen tools analyze the SSP focusing on the auxiliary states to identify potential optimizations reducing the state space.

When generating the VGSCSG the Murφ model checker may report the same coherence transaction multiple times to the SSC, if the auxiliary states of the coherence controllers for the same GCS differ. Remember that the Murφ model checker state encompasses all variables in a coherence controller - see chapter 3.1. The auxiliary states for the same GCS may differ if for example in an MSI protocol an acknowledgement counter is not reset after

Protocol	States	Rules Fired	Runtime
MSI	210518	2174123	13.60s
MSI-SSO	2889	31845	0.20s
MESI	341776	3588923	22.97s
MESI-SSO	4266	46568	0.79s

Table 5.4: Comparison of the functional correct and the state space optimized (SSO) concurrent MSI and MESI coherence protocol. The protocols were verified for a Murφ model with four caches and one directory

completing the counting. A coherence controller can have different counter values for the same coherence state, because the state is reachable through different preceding coherence transaction sequences. If the current counter value is not used in a condition or assignment by any other reachable coherence controller transition it is stale. While retaining a stale value may be functionally correct, it causes the model checking state space to grow.

To eliminate stale auxiliary state variable values expanding the model checking state space, a state space minimization algorithm (2) exhaustively searches the coherence controller state machines identifying when these actually can be considered as stale. An auxiliary state variable is stale, if in a coherence state any subsequent reachable state transitions have no conditional statements whose outcome is dependent on its current value, because a new value is assigned to the variable prior to any subsequent conditional statements. When an auxiliary state variable is detected to be stale the designer is notified and an instruction is automatically added to the SSP that resets the auxiliary state variable to its initial value defined in the PPL description. By resetting a stale auxiliary state variable the functional behavior of the coherence protocol is not altered.

In Listing 5.2 an example is given showing two auxiliary variable value reset instructions added to reduce the model checking state space.

By performing the stale auxiliary state variable analysis optimization for the SSP of the MSI protocol specification shown in Tables 1.1 and 1.2 the state space of the Murφ model is significantly reduced. The Table 5.4 lists the number of states explored, number of rules fired and corresponding runtime for the functionally correct as well as the state space optimized (SSO) concurrent coherence protocols. The Murφ model verified encompasses four caches issuing coherence transactions to the same address concurrently and one directory.

The HieraGen and HeteroGen algorithms compose coherence controllers of two flat coherence protocols. Because the state space of the composed coherence protocols is approximately

the Cartesian product of both flat coherence protocols reducing the state space of these can significantly reduce model checking runtime and memory requirements.

```
...
// Cache behavioural description
Architecture cache {
    ...

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

        await{
            when Data:
                cl = Data.cl;
                FState = M;
                exit;

            when DataAck:
                cl = DataAck.cl;
                acksExpected = DataAck.acksExpected;

                if acksExpected == acksReceived{
                    FState = M;
                    acksReceived = 0;       // Resetting auxiliary variable values
                    acksExpected = 0;       // reduces model checking state space
                    exit;
                }

                await{
                    when InvAck:
                        acksReceived = acksReceived + 1;

                        if acksExpected == acksReceived{
                            FState = M;
                            acksReceived = 0;       // Resetting auxiliary variable values
                            acksExpected = 0;       // reduces model checking state space
                            exit;
                        }
                    }

            when InvAck:
                acksReceived = acksReceived + 1;
        }
    }
    ...
}
```

Listing 5.2: State Space Optimized PPL description of the behavior of the MSI cache controller for a store access in state I transitioning to state M

Algorithm 2 SSP State Space Optimization (SSO) Algorithm

▷ GetPaths function returns all paths between a start state and a set of final states in a given coherence controller transition graph. A path is a list of transitions T.
Paths = GetPaths($G_{Controller}$, StartNode: S_{init}, EndNode: S_{init});
for each Aux_Var ∈ Coherence_Controller **do** ▷ *Iterate over all auxiliary variables*
    ▷ *Initially the auxiliary variable is considered to be stale in all states S*
    StateDict = $\{S_x : true \; \forall S_x \in S\}$;
    for each Path ∈ Paths **do** ▷ Iterate over all Paths
        bool stale = false;
        for each $T \in$ Path[-1:0] **do** ▷ Iterate backwards over paths
            ▷ *Iterate backwards over instructions described in the transition T*
            for each Op ∈ $T.IN$[-1:0] **do**
                if Op $\stackrel{\wedge}{=}$ Assignment(Aux_Var) **then**
                    stale = true; ▷ *Value of aux var preceding assignment is stale*
                else if Op $\stackrel{\wedge}{=}$ Conditional(Aux_Var) **then**
                    stale = false; ▷ *Cond outcome depends on preceding aux var value*
                end if
            end for each
            ▷ *A state can be element of multiple paths. If control flow of a single path is dependent on auxiliary variable value in state S_i, the value cannot be stale.*
            if StateDict[$T.S_{Start}$] == true **and** stale == false **then**
                StateDict[$T.S_{Start}$] = false; ▷ *Aux var value not stale in $T.S_{Start}$*
            end if
        end for each
    end for each
    for each $T \in$ Path ∈ Paths **do**
        ▷ *If the Aux_Var is not stale in the start state or a instruction assigns a new value during a transition, but the final state value is stale*
        if (StateDict[$T.S_{Start}$] == false **or** Assignment(Aux_Var) ∈ $T.IN$) **and**
            StateDict[$T.S_{Final}$] == true **then**
            ▷ *If no Aux_Var reset instruction exists yet*
            if Reset(Aux_Var) ∉ $T.IN$ **then**
                $T.IN$.append(Reset(Aux_Var)); ▷ *Append Aux_Var reset instruction*
            end if
        end if
    end for each
end for each

5.3.1 Summary

In this chapter we have presented three SSP pre-processing steps. In Step 1 we explain how the valid reachable global stable coherence state graph (VGSCSG) is constructed from the coherence transactions obtained by using the Murφ model checker. The VGSCSG shows how the global coherence state is altered when a coherence transaction is performed. Furthermore, from the transactions stored as elements of the VGSCSG, essential information like e.g. the transaction serialization points can be determined. This allows us to generate the access to request relation table (ART) and state to forwarded request relation table (FRT). Finally the directory state to cache controller state relations (SSRT) are determined mapping one or multiple cache stable states to a single directory stable state. In the next chapter we see how the ProtoGen algorithm leverages the information obtained from the pre-processing step to generate highly-concurrent cache coherence protocols.

The Steps 2 and 3 focus on improving the quality of the input SSP. In Step 2, we discussed how avoidable message stalls in the SSP can be detected and resolved. In Step 3 we discussed a method to improve the code quality of the input SSP, reducing the model checking state space by resetting stale variables to their initialization values.

Chapter 6

ProtoGen

6.1 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.

ProtoGen overcomes the key challenge in generating (flat) protocols — creating cache and directory controllers that correctly handle incoming coherence messages when transactions are racing — by leveraging the insight that, in a directory-based coherence protocol, racing transactions are serialized at the directory. ProtoGen enables the directory to convey this serialization order to the caches via the forwarded requests and responses it sends to caches; it overcomes possible ambiguities by renaming certain forwarded requests and responses.

With the caches and the directory achieving consensus on the order of racing transactions, ProtoGen can generate highly-concurrent and non-blocking (non-stalling) controller actions that are consistent with this order.

6.2 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 Protocol Language (PPL) - a 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.

6.2.1 From Atomic Specification to Implementation

A number of areas like programming languages (e.g. transactional memory) and hardware synthesis(e.g. Bluespec [47] 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 [48]. 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.

6.2.2 Description Languages

Teapot is a DSL for writing cache coherence protocols [39]. 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 [24]. 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 PPL is conceptually very similar to Teapot. However, in contrast to the PPL, 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.

6.2.3 General Hardware Synthesis

Following the idea of having a custom DSL to describe atomic specifications, some work have used these for hardware synthesis [49; 50]. The Bluespec line of work [50; 47; 51; 52] 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 [47]. 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 [51; 52], but it is not documented whether these approaches are implemented in the Bluespec compiler.

6.2.4 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 [53] 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.

Namulasu and Gopalakrishnan [54] presented a conceptual framework that employs, at a high-level, a similar approach to ProtoGen. The input is a coherence protocol described in the Communicating Sequential Processes (CSP) [55] specification language using atomic transactions. To resolve races of concurrent transactions, the framework introduces transient states,

in which the coherence controllers wait after sending a coherence transaction request to the home node i.e. directory for either an acknowledgement (*ack*) or a negative acknowledgement (*nack*) response message. The reception of an *ack* message indicates that its own coherence transaction was successfully serialized. If the cache controller receives a *nack* message its own coherence transaction was effectively blocked and it returns to its initial stable state before trying to serialize the same or a different coherence transaction again. ProtoGen in contrast generates coherence protocols that are completely non-blocking avoiding to *nack* messages. Furthermore, ProtoGen can also handle coherence protocols that leverage direct communication between cache controllers, while the conceptual framework by Namulasu and Gopalakrishnan is restricted to communication between cache and directory controllers only. Therefore, ProtoGen can uncover a significantly greater potential for concurrency among racing coherence transactions.

6.2.5 Non-Blocking Protocol Synthesis

A case study shows that non-blocking cache coherence protocols can be naturally expressed in Bluespec [56]. 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.

6.2.6 Coherence Protocols via Program Synthesis

TRANSIT [57; 58], which is inspired by program synthesis approaches like sketching [59], 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 [60] 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.

6.2.7 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 [6] and VIPS [61] is to exploit data-race-free cache coherence protocol models to minimize the number of transient states. Another approach is Fractal coherence [14; 62], 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.

6.3 Intuition for ProtoGen

ProtoGen starts with a SSP and creates a complete protocol — 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.

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).

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? For example in an MSI coherence protocol like shown in Table 1.1, 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. What happens when the cache that issued the GetS receives an Invalidation from another cache 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, the SSP is preprocessed ensuring that in the input SSP 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.

6.4 Terminology

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.

6.5 Using ProtoGen

6.5.1 Input

The primary input to ProtoGen is a high-level specification of a stable state protocol (SSP) in the internal intermediate representation transition graph format that is generated by the PPL Parser described in section 4.2.

In addition, ProtoGen also accepts a configuration file that controls 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 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.

Furthermore, ProtoGen can be configured to convert the input SSP that was originally designed for an ordered network into a protocol supporting unordered networks without introducing overhead like handshake messages with few exceptions.

6.5.2 Output

ProtoGen produces fine state machines (FSMs) for the caches and the directory including the transient states (containing information similar to Table 6.6). These FSMs are expressed in the transition graph IR. The backends briefly presented in section 4.3 then translate the IR into the respective output languages like Murφ and SLICC.

6.5.3 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. 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.

6.6 ProtoGen Algorithm

Given an SSP, ProtoGen generates a highly concurrent directory protocol including the transient states. This process is explained step by step in this subsection, 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 subsection, 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).

6.6.1 Step 1: Resolve Race Condition Message Ambiguities

Before generating any transient states, ProtoGen first preprocesses the SSP specification to ensure the invariant that a given forwarded request or response racing the forwarded request can arrive at exactly one SSRT stable state set. If, in an input SSP specification, the same forwarded request can arrive in two stable state sets, ProtoGen creates a new label for one of the forwarded requests. The same principle is also applied to the racing responses.

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.

Table 6.1: MOSI SSP: Before preprocessing

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

Table 6.2: MOSI SSP: After preprocessing

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

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 snippet of the SSP is shown in Table 6.1). In such a case, ProtoGen would rename one of the messages as shown in Table 6.2. 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.

In addition to forwarded requests, ProtoGen also labels responses racing the forwarded request. A response that is part of a transaction subsequent to a previously serialized forwarded

request transaction is considered to be racing if it can be served in the current cache controller transient state and if the network does not guarantee ordering because it is either unordered, the messages are assigned to different virtual channels or because the messages take diverging paths.

If for example a cache C_0 wants to evict a block it holds in M state, it sends a Put request to the directory. However, before the directory observes the Put request, it forwards the GetS issued by C_1 to C_0 as Fwd_GetS. When the directory observes the now stale Put request of C_0, it responds with a Put_Ack message confirming the request completion. The forwarded message Fwd_GetS and the Put_Ack message are racing. If Put_Ack arrives at the cache prior to Fwd_GetS, the cache enters state I and is unable to serve the subsequent Fwd_GetS request. To prevent this deadlock caused by the race, ProtoGen renames the Put_Ack response message into S_Put_Ack enabling the cache to infer the transaction serialization order. The cache controller will only serve the S_Put_Ack after the cache has transitioned into state S by serving the preceding Fwd_GetS.

By labeling forwarded request and racing responses the cache controllers can infer the order in which transactions were serialized at the directory. To identify requests and racing responses that must be labeled the Algorithm 3 leverages the relation tables generated by the SSP-Preprocessing stage described in section 5.1.

6.6.2 Step 2: Assign Transient States to 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.

ProtoGen keeps track of this information using the Stable State Relation Table (SSRT) data structure. The cache controller state sets tracked in the SSRT initially contain only the stable states. For an MSI protocol, the state sets are initially $\{\mathrm{I}\}$, $\{\mathrm{S}\}$, and $\{\mathrm{M}\}$, and we refer to them as the $\boldsymbol{I}$, $\boldsymbol{S}$, and $\boldsymbol{M}$ state sets, respectively. (We use **bold** to distinguish a state set from a stable state.)

Algorithm 3 Forwarded Request and Racing Response Labelling

1: ▷ *Iterate over states in state sets to identify message that must be renamed. Messages that must be renamed are identified by comparing the identifiers of forwarded requests and racing responses a cache may receive in different state sets. We use the information previously extracted and stored in the SSRT and FRT tables.*
2: **for each** State_Set_0, $\text{State\_Set}_1 \in$ SSRT **do**
3: **if** State_Set_0 != State_Set_1 **then**
4: Rename_Msgs = {};
5: ▷ *Identify forwarded message identifiers that are element of both state sets. If responses racing the forwarded requests with identical identifiers like Put-Ack exist in both state sets they must be labelled as well.*
6: **for each** $\text{Cache\_State}_0 \in \text{State\_Set}_0$, $\text{Cache\_State}_1 \in \text{State\_Set}_1$ **do**
7: Rename_Msgs += $\{\text{FRT}[\text{Cache\_State}_0].\text{R}_{Fwd}\} \cap \{\text{FRT}[\text{Cache\_State}_1].\text{R}_{Fwd}\}$;
8: **end for each**
9: **if** Rename_Msgs **then**
10: ▷ *Label the forwarded requests for all cache and directory controller states that are element of the same state set with an unique identifier*
11: Label_Conflict_Msgs(Rename_Msgs, State_Set_1);
12: **end if**
13: **end if**
14: **end for each**

ProtoGen adds the transient states generated by the PPL Parser required to implement logically atomic transaction protocols to one or more state sets as described below. For every coherence transaction in the Valid Global Stable Coherence State Graph (VGSCSG) the transitions performed by the cache controllers involved are processed. The transaction from stable state S_i to stable state S_j performed by a coherence controller can encompass multiple transitions and therefore transient states. Because the cache controller transient states are not visible to the directory, ProtoGen maps the transient states to the SSRT state set of the corresponding stable states in which the cache controller is expected to reside in for the given directory state while actually being in the transient state.

To correctly assign the transient states of the cache controllers to the state sets, ProtoGen leverages the communication pattern between the cache controllers and the directory controller that can be extracted from the coherence transaction. In directory coherence protocols a coherence transaction is globally serialized when it is observed by the directory initiating a

global coherence state (GCS) change. The transient states in which a cache controller can reside prior to the transaction serialization must be assigned to the same state set like its initial stable state S_i[1]. All transient states in which a cache controller can reside after the transaction serialization are assigned to the state set of S_j.

While some transient states are only elements of a single state set, there exist transient states that are elements of multiple state sets. This is the case, if a cache coherence controller resides in a transient state waiting for a response, when the coherence transaction is serialized. Therefore, the transient state must be assigned to the state sets of 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).

Recall that the transaction I to M in the MSI coherence protocol leads to the creation of the transient states IM^{AD} and IM^{A}. The state IM^{AD} reflects the situation that the cache has issued the GetM request to the directory and is awaiting a Data or a DataAck and one or more InvAck responses.

When the Data response is received from the directory, the cache will transition from IM^{AD} to M. Because when entering the transient state IM^{AD} the cache controller is still seen by the directory in state I the transient state belongs to the state set $\boldsymbol{I}$. While the cache is in the transient state IM^{AD}, the directory serves the GetM request serializing the coherence transaction and responding with Data. After sending the response the directory expects the cache controller to eventually enter the stable state M. Because the cache controller does not observe the response immediately it remains in IM^{AD}, while already expected by the directory to be in state M. Thus IM^{AD} belongs not only to the state set $\boldsymbol{I}$ but also to the state set $\boldsymbol{M}$.

The transient state IM^{A} is only element of the state set $\boldsymbol{M}$ and not of the state set $\boldsymbol{I}$. The cache controller can only enter this transient state after the coherence transaction has been serialized, because it is guarded by a response messages that presupposes the coherence transaction serialization by the directory.

In Figure 6.1 the state set assignments of the I to M transition graph $G_{I \rightarrow M}$ are shown. At the end of this ProtoGen step, the State Relation Table SSRT is given in Table 6.3 after the Algorithm 4 has been applied on our MSI protocol assigning the transient states to the state sets.

[1] In flat hierarchy level textbook directory coherence protocols, the cache commonly resides only in a single transient state before the coherence transaction is serialized at the directory. However, in hierarchical cache coherence controllers like generated by HieraGen, the cache controller must first enforce coherence with its lower level caches potentially involving multiple transient states before its coherence transaction is requested to be serialized at its associated directory.

Algorithm 4 Transient State State Set Assignment

```
1: ▷ Iterate over all coherence transactions TR in the VGSCSG graph
2: for each TR ∈ VGSCSG do
3:     C_Init = GetInitCache(TR);          ▷ Determine the cache initiating the transaction
4:     ▷ Get the state sets SS containing the start and end stable states of the initiator cache
       from the transaction
5:     SS_Start = SSRT.GetStateSet(TR.S_Start(C_Init));
6:     SS_Final = SSRT.GetStateSet(TR.S_Final(C_Init));
7:     bool serialized = false;
8:     ▷ Iterate sequentially over all transitions that compose the transaction
9:     for each T ∈ TR do
10:        ▷ Determine which controller performed the transition
11:        if C_Init == GetController(T) then
12:            ▷ Add transition final transient state to final stable state set
13:            SS_Final += T.S_Final;
14:            if !serialized then
15:                ▷ Add transition final transient state to start stable state set
16:                SS_Start += T.S_Final;                                  ▷ Duality
17:            end if
18:        else if dir == GetController(T) ∧ C_Init == Request.src then
19:            ▷ When the directory observes a request initiating a state transition from the
                 initiator cache, the transaction it is serialized
20:            serialized = true;
21:        end if
22:    end for each
23: end for each
```

Directory State	Cache state set	Stable State Access Permission
M	$\{M, IM^{AD}, IM^{A}, SM^{AD}, SM^{A}, MI\}$	$\{read,\ write\}$
S	$\{S, IS, SM^{AD}, SI\}$	$\{read\}$
I	$\{I, IS, IM^{AD}, SI, MI\}$	$\{\}$

Table 6.3: MSI protocol directory to cache state relation table (SSRT) including transient states

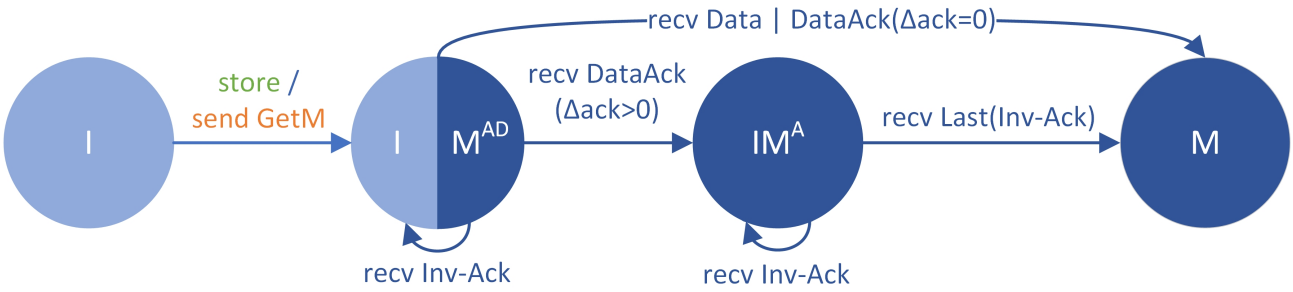


Figure 6.1: Cache controller I to M transition graph $G_{I \to M}$ state set assignments. The shade of each state denotes the state set(s) it belongs to. IM^{AD} is element of state sets $\boldsymbol{I}$ and $\boldsymbol{M}$. The State IM^{A} belongs only to state set $\boldsymbol{M}$, because the transition IM^{AD} to IM^{A} is guarded by the directory response DataAck.

6.6.3 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 assigned to the state sets in Step 2. In this subsection, we explain how this is done for any given transient state. Later in Section 6.6.7, 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 while the controller is 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 while the controller is 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 pre-processing 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 6.3, 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}.

6.6.3.1 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 responds 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 6.6.3.2). 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. If this is not the case, ProtoGen leverages the access to request relation table (ART) to infer the request R'_{own} that C_0 would have issued for the access A_{own} in S_l. A new transition is added to the directory SSP enabling to translate R_{own} into R'_{own}.

Therefore 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. 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 6.6.2).

However, it is possible for such a transient state to not already exist if for S_l the access A_{own} is not defined. C_0 would therefore not attempt to restart its own transaction T_{own} as the access either has been implicitly served as part of T_{other} or has become obsolete. This means that the request R_{own} has become stale. ProtoGen can handle a request that has become stale and whose associated access does not need to be performed anymore by generating a new transient state S_{ijl} and a new transition from S_{ij} to S_{ijl} allowing C_0 to respond to R_{other} immediately. C_0 now awaits in S_{ijl} a response to its own stale request R_{own} confirming that is has reached the directory before transitioning into S_l. ProtoGen creates all transient states and transitions required to serve the response.

In case that an access is not defined for a stable state an optimization can be performed if it is guaranteed that all requests issued by C_0 for the same cache block will be observed by the directory in the order in which they were issued. In this case C_0 can immediately transition from S_{ij} into S_l when responding to R_{other}, because it is guaranteed that a subsequent request will not change the directory state making it impossible to infer that R_{own} is stale.

We provide the complete Algorithm 5 for the case of $T_{other} \rightarrow T_{own}$. We have annotated the algorithm steps in red for the following particular example. In our MSI protocol, consider the transient state SM^{AD} shown in Figure 6.2, 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 another 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

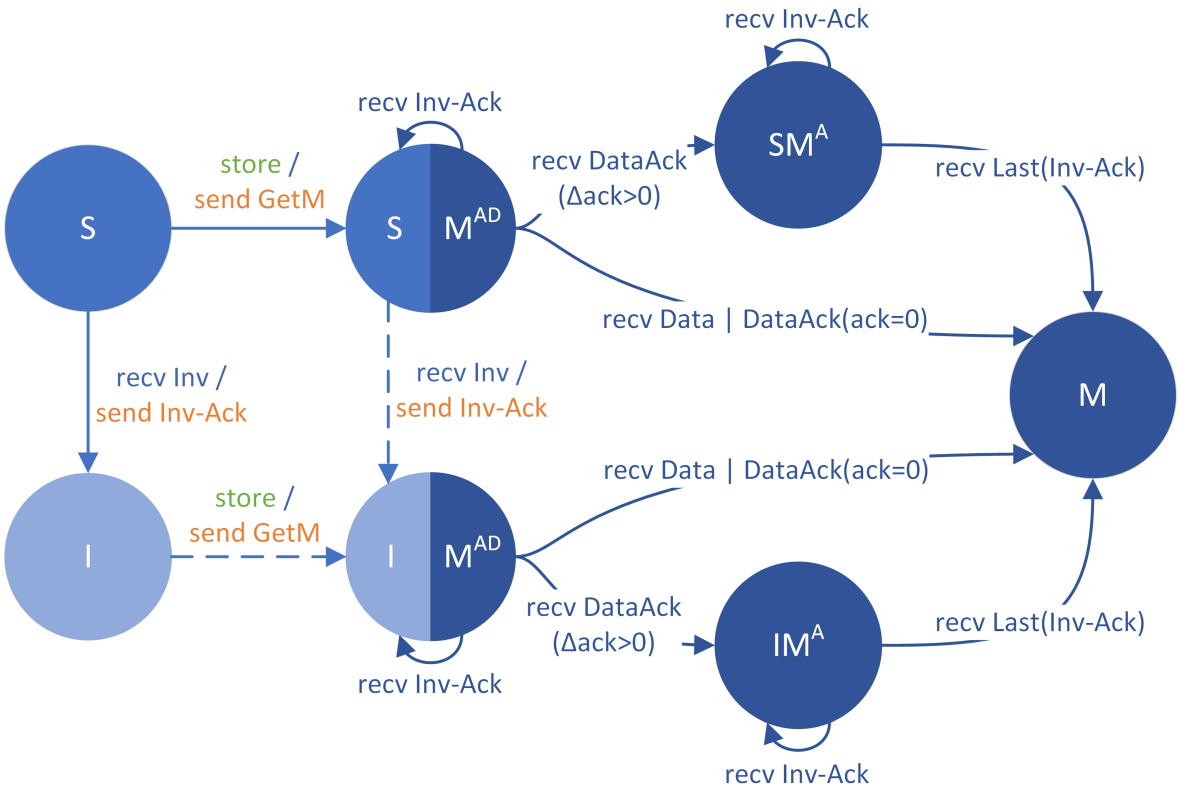


Figure 6.2: Cache I to M and S to M transition graph $G_{\{I,S\}\rightarrow M}$ with $T_{other} \rightarrow T_{own}$. The shade of each state denotes the state set(s) it belongs to. E.g., IM^{AD} is part of I and M state sets

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 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. Leveraging the ART the directory reinterprets the Upgrade as a GetM, the request triggered by the same access (store) in state I.

Algorithm 5 Step 3: Accommodate Concurrency

1: ▷ *In cache controller transition graph identify sub transition graphs that serve accesses and forwarded-request.*
2: ▷ *For every access and forwarded-request extract the corresponding controller sub transition graphs from* $G_{Controller}$ *e.g. 'store:* $G_{I\rightarrow M}$*' shown in 6.1.*
ATG: Access_Transition_Graph ; FRTG: Forwarded_Request_Transition_Graph
3: $SG_{Access} = \{G_{load:I\rightarrow S}, G_{load:S\rightarrow S}, G_{store:I\rightarrow M}, G_{evict:M\rightarrow I}, ...\}$
4: $SG_{Fwd-Request} = \{G_{Inv:S\rightarrow I}, G_{Fwd-GetM:M\rightarrow I}\}$
5: ▷ *List of new transition graphs generated to handle concurrency*
6: $SG_{T_{other}\rightarrow T_{own}} = \{\}$
7: ▷ *In red the values of each step for the example presented in Figure 6.2 are given*
8: **for each** $G_{Access} \in SG_{Access}$ **do** ▷ $G_{store:S\rightarrow M}$
9: SS_{Start} = SSRT.GetStateSet($G_{Access}.S_{Start}$); ▷ SSRT[S]
10: ▷ *Due to control flow dependencies a sub graph can have final states that are element of different state sets e.g. in an MESI protocol* $G_{I\rightarrow\{S,E\}}$
11: **for each** $S_{Final} \in SG_{Access}$ **do**
12: SS_{Final} = SSRT.GetStateSet(S_{Final}); ▷ SSRT[M]
13: ▷ *Identify transient states (TS) from* G_{Access} *that are in* SS_{Start} *and in* SS_{Final}*, i.e. Duality*
14: **for each** TS $\in$ (G_{Access}.TStates $\cap$ SS_{Start} $\cap$ SS_{Final}) **do** ▷ SM^{AD}
15: TOTHERTOWN(TS, $G_{Access}.S_{Start}$, SS_{Final});
16: **end for each**
17: **end for each**
18: **end for each**
19: **procedure** TOTHERTOWN(TS: State, AS_{Start}: State, SS_{Final}: StateSet)
20: ▷ *Iterate over forwarded-request subgraphs*
21: **for each** $G_{Fwd-Request} \in SG_{Fwd-Request}$ **do** ▷ $G_{Inv:S\rightarrow I}$
22: **if** $G_{Fwd-Request}.S_{Start}$!= AS_{Start}) **then** continue;
23: **end if**
24: ▷ *Create a copy of* $G_{Fwd-Request}$ *assigning new unique names to the transient states e.g.* $G_{Inv:S\rightarrow I} \rightarrow G'_{Inv:S\rightarrow I}$
25: $TSG_{Fwd-Request}$ = COPYSUBGRAPH($G_{Fwd-Request}$); ▷ $G'_{Inv:S\rightarrow I}$
26: FINDSUITABLETRANSIENTSTATE($TSG_{Fwd-Request}$, TS, AS_{Start}, SS_{Final});
27: HANDLESTALEREQUEST($TSG_{Fwd-Request}$, $G_{Fwd-Request}$);
28: UPDATEGRAPHSTARTSTATE($TSG_{Fwd-Request}$, AS_{Start},TS) ▷ $G'_{Inv:SM^{AD}\rightarrow IM^{AD}}$
29: **end for each**
30: **end procedure**

```
31: procedure HANDLESTALEREQUEST(TSG_Fwd−Request, G_Fwd−Request)
32:     for each S_Final ∈ TSG_Fwd−Request do
33:         if S_Final ∈ G_Fwd−Request then
34:             ▷ No suitable transient state was found for S_Final. Generate a new transient
                  state and transition to track the completion of the outstanding stale request.
                  See example in Figure 6.3.
35:             TSG_Fwd−Request += AddRequestAckTransition(S_Final);
36:         end if
37:     end for each
38: end procedure

39: procedure FINDSUITABLETRANSIENTSTATES(TSG_Fwd−Request, TS: State,
            AS_Start: State, SS_Final: StateSet)
40:     ▷ A suitable transient state must be reachable from the final state through a sequence
          of transitions that have equivalent guards (leverage ART) as the transitions already
          taken in the start transient state.
41:     TSL_Guard = GETSEQUENCEOFGUARDSINPATH(AS_Start, TS);          ▷ 'GetM'
42:     for each S_Final ∈ TSG_Fwd−Request do
43:         ▷ FTS: Final (state set) Transient State
44:         for each FTS ∈ SS_Final do                                ▷ IM^AD
45:             FTSL_Guard = GETSEQUENCEOFGUARDSINPATH(S_Final, FTS)  ▷ 'GetM'
46:             ▷ Check if FTS is reachable from S_Final by the same guard sequence
47:             if TSL_Guard == FTS_Guard then                        ▷ 'GetM' ≙ 'GetM'
48:                 ▷ Replace S_Final with FTS in TSG_Fwd−Request     ▷ G'_Inv:S→IM^AD
49:                 UPDATEGRAPHFINALSTATE(TSG_Fwd−Request, S_Final, FTS)
50:             end if
51:         end for each
52:     end for each
53: end procedure
```

Let us consider another case for our MSI coherence protocol shown in Figure 6.3. The cache initially in state M issues a Put request due to an eviction entering the transient state MI waiting for an acknowledgement. If a Fwd-GetM request arrives, which is only possible in M state, the cache infers that another transaction was ordered before its own. ProtoGen discovers from the SSP that in this case the cache must transition from state M into state I. However,

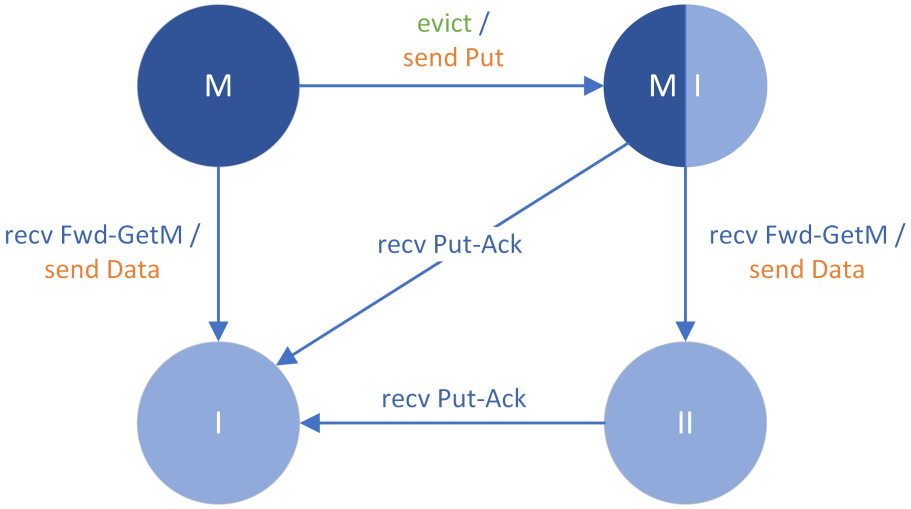


Figure 6.3: Cache M to I transition graph $G_{M \to I}$ with $T_{other} \to T_{own}$. The shade of each state denotes the state set(s) it belongs to. E.g., MI is part of M and I state sets

this time no suitable transient state already exists into which ProtoGen can transition from state MI, because the cache block is considered to be already invalid (implicitly evicted) in state I, so there is no behavior defined for the eviction in the SSP. Therefore, ProtoGen - not attempting to perform an optimization this time - generates a new transient state II into which the cache controller transitions after responding to the Fwd-GetM request. Once the cache receives the confirmation through a Put-Ack from the directory that its own stale Put request has been served, it transitions into state I.

6.6.3.2 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., $T_{own} \to T_{other}$ 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 $T_{own} \to T_{other}$, 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}. Therefore, upon receiving a forwarded request while in S_{new}, C_0 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}. If for example 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. To identify whether C_0 can respond immediately, ProtoGen checks if the response is logically dependent on any subsequent transition like described in section 5.2.

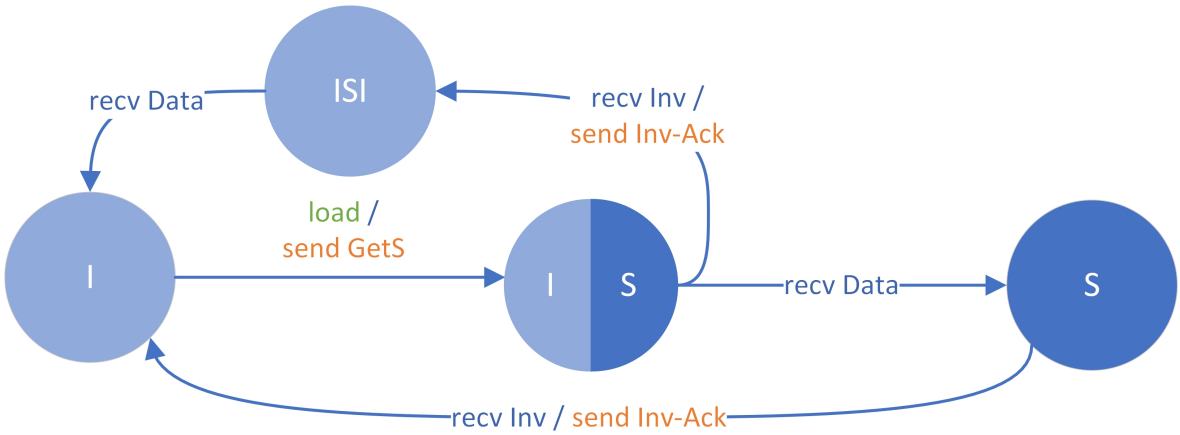


Figure 6.4: The I to S transition graph $G_{I \to \{I,S\}}$ of the MSI protocol. The IS state belongs to the state sets of ***I*** as well as ***S***, which is why it is shown in two shades. The ISI state, on the other hand, belongs to only the state set ***I***.

An Example with Immediate Transitions and Responses. Consider cache C_0 and the I to S transaction 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 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 6.4, 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 $\boldsymbol{I}$, 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.

6.6.4 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.

6.6.5 Step 5: Transient State Events

While accesses like read or write involve only on a single cache line other accesses like acquire or release may require multiple cache lines to perform a certain operation. In this case an event is raised within the cache as part of a coherence transaction. The transaction performed on a cache line that causes the event to be raised can only complete after the event was served by all other cache lines present in the cache.

The SSP defines for every stable state how the event must be served. While it is functionally correct to wait until all other cache lines have completed any inflight transaction at the time the event was raised, it increases the event latency. An event can be served in a transient state, if the behavior of the transaction serving the event is equal in the initial state and in all potential final states. The final state of a cache coherence transaction can depend e.g. on the type of response or forwarded requests observed while being inflight. If an event can be served in a given transient state, ProtoGen generates the required transitions.

6.6.6 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 in *any* state. For example, consider a block in state S at cache C_0 that issues a Put in state S to the directory. Before that Put 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 Put 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, the directory can infer whether a request is stale by leveraging the information provided by the message labeling, its own state and auxiliary state values. If for example, a Put arrives in directory state M, it may be stale being issued from a cache that in a previous epoch held the cache block in state S or M. From the VGSCSG transactions ProtoGen can infer that the transaction granting the modified state M to a cache modifies the owner ID auxiliary state defined in the SSP. To prevent a stale Put request from altering the directory controller state, ProtoGen modifies the directory SSP specification adding a conditional statement to the Put transaction checking whether the cache that sent the

Put request equals the current owner stored in the auxiliary variable. If the cache that issued the Put request is not the current owner, 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.

However, instead of defining a common Put request for all states, an architect may choose to specify different Put requests e.g. PutM, PutS for every coherence state. If in the SSP a request is not defined for a directory state, ProtoGen leverages the ART to identify the cache access associated with the request allowing it to translate the stale request message into the equivalent request message defined in the SSP for the current directory state.

6.6.7 Putting It All Together

In this subsection, we describe how the aforementioned steps are put together to generate cache and directory controllers. ProtoGen first pre-processes the SSP labelling messages to enable the cache to infer the transaction serialization order from the received messages (Step 1). Then, ProtoGen assigns the transient states required by the logically atomic protocols to the state sets (Step 2). Then, for each of these transient states, ProtoGen accommodates concurrency generating new transitions and possibly also new transient states (Step 3). If new transient states are generated, ProtoGen repeats (Step 3) to accomodate concurrency until either no further transient states are generated or we reach the pending transaction limit. After all states are generated, ProtoGen assigns access permissions to every state (Step 4) and determines which events can be served in a given transient state of the protocol (Step 5).

6.7 Evaluation: Protocols Generated with ProtoGen

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.

6.7.1 Stalling Protocols

In our first set of experiments, we used ProtoGen to generate several stalling protocols from Sorin et al.'s primer [1]. 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

Sorin et. al / ProtoGen		
Protocol	States	Transitions
MSI	11 / 11	32 / 32
MESI	13 / 13	40 / 40
MOSI	15 / 15	47 / 47
MOESI	- / 20	- / 56

Table 6.4: Stalling ProtoGen Evaluation

Sorin et al. / ProtoGen			
Protocol	States	Transitions	Stalls
MSI	18 / 20	53 / 73	5 / 0
MESI	- / 25	- / 92	-
MOSI	- / 31	- / 128	-
MOESI	- / 38	- / 155	-

Table 6.5: Non-Stalling ProtoGen Evaluation

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, MESI and MOESI directories.[2] All of the protocols passed Murφ verification of SWMR and deadlock freedom with four caches. The results in this subsection are perhaps unsurprising but they are reassuring. The number of states of the protocols generated by ProtoGen are given in Table 6.4.

6.7.2 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, MOSI and MOESI protocols from the previous subsection. The number of protocol states and transitions are shown in Table 6.5 for a maximal pending transaction limit of three. It is worth noting that the protocols generated were fairly non-trivial due to the large number of states and transitions.

There are no non-stalling MESI, MOSI or MOESI 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 6.6, 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,

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

ProtoGen was able to merge some states that were kept separate in the primer like $IM^AS = SM^AS$, $IM^ASI = SM^ASI$, and $IM^AI = SM^AI$.

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 [63] (as in the protocol generated by ProtoGen). This access *logically* occurs before the block is invalidated.

6.7.3 AMBA CHI 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 the AMBA CHI protocol [5] that does not rely upon point-to-point order [64]. Specifying the SSP for this protocol was not much more difficult than for the MOESI 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.

6.7.4 TSO-CC

TSO-CC [27] 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,

Table 6.6: MSI Non-Stalling Primer vs. ProtoGen

	Load	Store	Evict	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data (Δack=0)	Data (Δack>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 Put 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 Put (+ 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				

Channels:	Request	Response
Messages:	GetS, GetM, Put, Fwd-GetS, Fwd-GetM, Put-Ack	[Data], Inv-Ack

Notation:	[Message Identifier/Label Variant]	Δack = |acksExpected - acksReceived|

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. [65], 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.

6.7.5 DeNovo, RCCO & RCC

The DeNovo [6], RCCO and RCC [1] protocols exploit the data-race free property of memory consistency models. Hence ProtoGen does not need to generated transient states to handle concurrent transactions. For the transient states generated by the PPL Parser 4.2 as elements of the SSP coherence transactions ProtoGen infers whether any given event can be immediately served generating the required transitions, see section 6.6.5.

6.8 Handshake Free Unordered Network Protocol

While we have shown 6.7.3 that it is possible to specify SSPs for unordered interconnects using handshakes to serialize coherence transactions, handshakes create additional coherence traffic on the interconnect compared to an ordered network solution. However, by using the ProtoGen approach, handshake free MESI state based cache coherence protocols for unordered networks can be generated reducing the coherence transaction control traffic overhead. In Table 6.7 a handshake free stalling MESI protocol for an unordered network is given. Its states

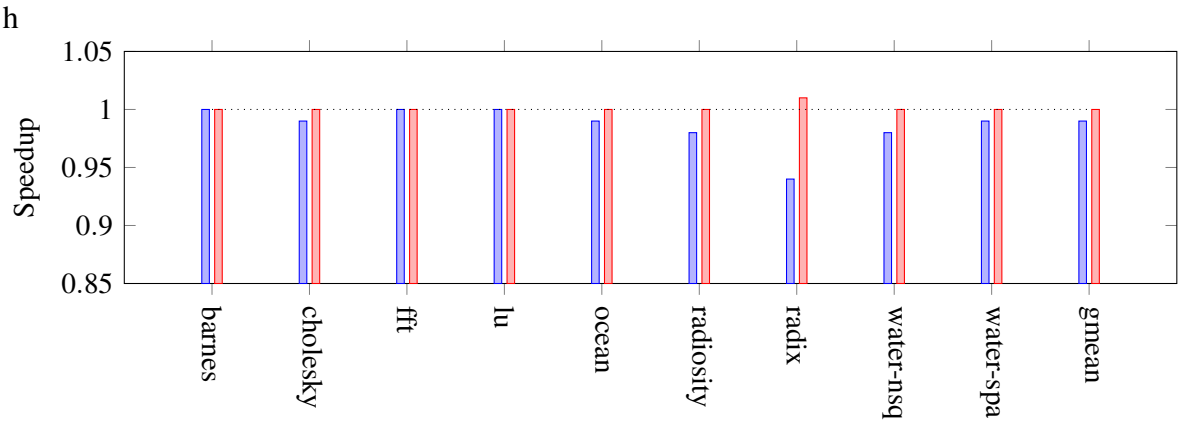


Figure 6.5: Stalling handshake free MSI (blue) and MESI (red) over the baseline CHI protocol)

and transitions are equal to the stalling ordered network protocol implementation. The stalling ordered network and unordered network coherence protocols only differ in the number of labeled messages and virtual channels required. Due to the unordered network more message race conditions exist compared to ordered network implementations, which must be resolved by ProtoGen through message labelling, see section 6.6.1. In addition, to ensure deadlock freedom, the handshake free stalling unordered network MESI protocol given, requires at least three virtual channels, while the ordered network implementation can be realized with only two. To reduce the number of stalls we present an implementation with four virtual channels assigning responses that never stall to a separate virtual channel.

6.8.1 Evaluation

We have verified the correctness of the stalling handshake free MSI and MESI protocol for unordered interconnects using the Murφ model checker. To evaluate its performance we use the gem5 simulator running the Splash 2 benchmarks. As baseline we choose an unordered network AMBA CHI protocol implementation that does not deploy the SD state 6.7.3. The MSI and MESI protocol require four virtual channels, while the CHI protocol only requires three. The reference system configuration is given in Table 6.8. It uses a flat cache hierarchy design and has a tile based architecture with each tile encompassing a CPU, private L1 cache, L1 directory and a memory bank.

While the system shows a similar runtime for the handshake free unordered MESI coherence protocols compared with CHI, the generated traffic is in average 13 percent lower for MESI. The results observed comply with our expectation of a maximum data traffic reduction of 14.3 percent due to the removal of the additional handshake under the condition that all transactions do not span across multiple caches. This is however only the case for

Table 6.7: MESI handshake free unordered network protocol cache controller

	Load	Store	Evict	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data	Inv-Ack	Last Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}								
IS^D	stall	stall	stall			stall		?[Data]{ /S} ?[DataE]{ /E}		
IM^{AD}	stall	stall	stall	stall	stall			?(Δack==0){ /M} ?(Δack>0){ /IM^A}	ack++	
IM^A	stall	stall	stall	stall	stall				ack++	-/M
S	hit	send GetM to Dir/SM^{AD}	send Put to Dir/SI^A			send Inv-Ack to Req/I				
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		?(Δack==0){ /M} ?(Δack>0){ /SM^A}	ack++	
SM^A	hit	stall	stall	stall	stall				ack++	-/M
M	hit	hit	send Put + Data to Dir/MI^A	send Data to Req and Dir/S	send Data to Req/I					
E	hit	hit/M	send Put to Dir/EI^A	send Data to Req and Dir/S	send Data to Req/I					
MI^A	stall	stall	stall	send Data to Req and Dir/SI^A	send Data to Req/II^A		[Put-Ack_EM] /I			
EI^A	stall	stall	stall	send Data to Req and Dir/SI^A	send Data to Req/II^A		[Put-Ack_EM] /I			
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	[Put-Ack_S] I			
II^A	stall	stall	stall				[Put-Ack_I] I			

Channels:	Request	Response	Put-Ack	Inv
Messages:	GetS, GetM, Put, Fwd-GetS, Fwd-GetM	[Data], Inv-Ack	[Put-Ack]	Inv

Notation:	[Message Identifier/Label Variant]	?Conditional{}	Δack = |acksExpected - acksReceived|

Table 6.8: Simulated System Parameters

Tile	
CPU	x86, TimingSimpleCPU, 2 GHz
L1 cache	private, 1-cycle, 4-way
L1 Directory	6-cycle response, 10-cycle recycle latency
Memory	10-cycle latency
Interconnect	Network-on-Chip, 4×4 mesh topology, XY routing, Unordered Buffers, 16B per flit, 1-cycle channel latency, 1-cycle router latency
System	System emulation mode (SE), 16 Tiles (one per mesh node)

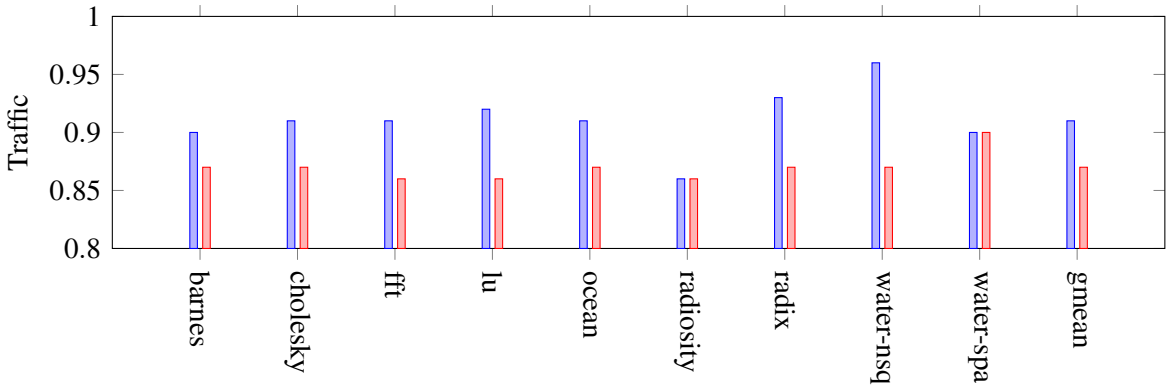


Figure 6.6: Relative reduction of traffic injected by the handshake free MSI (blue) and MESI (red) over the baseline CHI protocol)

read-only shared and private data cache lines. If a transaction spans across multiple caches, the relative overhead of the handshake compared to the remaining traffic is reduced. For example, the water-spa benchmark exhibits a stable producer-consumer relationship having a large number of CPUs consuming the produced data [66]. Therefore, the communicating writes to shared memory result into a large number of remote invalidations resulting into a lower traffic reduction compared to other benchmarks. The additional transaction latency resulting from handshakes does not negatively affect the runtime of the system when using the CHI protocol, because the percentage and frequency of accesses to shared data are across all Splash2 benchmarks significantly smaller than accesses to private data. Furthermore, the network latencies are relatively small minimizing the probability of stalls due to same cache line access conflicts at the directory. The MSI protocol shows a slightly higher runtime compared to the MESI and CHI protocol due to the lack of the exclusive state E, resulting into stalls for writes to private cache lines that were previously read.

Although we do not observe a performance gain and the traffic improvement of the handshake free MESI protocol over the reference CHI protocol are small for the Splash2 benchmarks, the lower amount of traffic generated by the MSI and MESI protocols can allow more compute units to communicate over a given interconnect before reaching the throughput saturation point.

6.8.2 Limitations

ProtoGen cannot generate a handshake free protocol for an unordered network if a cache controller transaction loops in a stable state after receiving a forwarded request like for example in the MOSI protocol O state. If in state O a forwarded request Fwd_GetS arrives, the cache C_0 responds to the request while remaining in state O. If C_0 initiates an eviction it sends a Put request to the directory that sends a Put_Ack_O response. Because the C_0 remains in state O after serving an arbitrary number of Fwd_GetS requests, it cannot infer when receiving the Put_Ack_O response, whether a Fwd_GetS requests of a transaction serialized before its own is still outstanding.

To overcome this challenge two solutions can be applied. The first solution is the introduction of handshakes like used in the CHI coherence protocol [64] to ensure that while the cache is in a state with looping transactions, a subsequent coherence transaction can only be serialized at the directory after the current coherence transaction has completed. The second solution is to introduce two transaction counters. The directory counts the number of forwarded requests sent to a cache and the cache counts the number of forwarded requests received. When a response message can race a forwarded request the directory adds its current forwarded request count to its payload. If the cache receives the response, it compares its receive forwarded request count with the number of forwarded requests sent by the directory. If the cache has served all forwarded requests, it is allowed to serve the response.

While ProtoGen currently applies the first more simple solution it would be possible to automatically alter the SSP to implement the second solution.

6.9 Summary

Although one could argue that many of the protocols 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, RCCO and RCC and uncover additional concurrency in non-stalling protocols.

Furthermore, the ProtoGen algorithm allows us to generate novel handshake free coherence protocols for unordered networks that outperform the state of the art solutions and do not require time stamps or counters.

Chapter 7

HieraGen

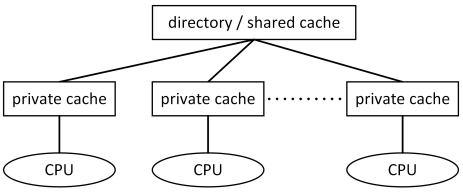
7.1 Introduction

While the ProtoGen algorithm generates concurrent *flat directory protocols*, in which a single directory—perhaps colocated with a shared cache—communicates directly with all of the cores and their private cache hierarchies, as shown in Figure 7.1(a), it is restricted to a very narrow system and protocol model. While we expect directory-like coherence to persist, there are several trends pushing industry away from flat protocols and towards protocols with hierarchy. Hierarchy is a time-tested design strategy for scalable systems [7; 8; 9; 10; 11; 12; 13; 14], and it is an attractive approach to multicore processor design as the number of cores continues to increase. Hierarchy can also enable coherence protocols that are more scalable [15]. Figures 7.1(b) and 7.1(c) show two hierarchical system models with hierarchical directories (and hierarchical shared caches).

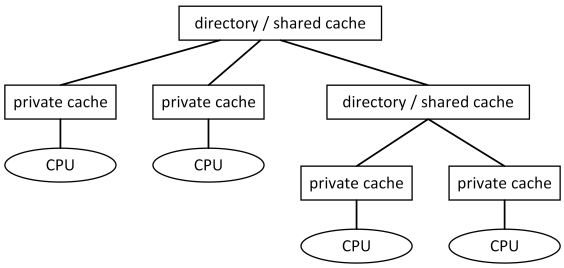
While hierarchy has many desirable features, it also greatly complicates the design of the coherence protocol. There are more states, more transitions, and more possible concurrency. Crucially, communication between levels of the hierarchy must preserve coherence invariants. In addition to the design complexity, verification is also more challenging with hierarchy, due primarily to the much larger state space [16; 13].

To sidestep the challenges in designing (and verifying) hierarchical coherence protocols, HieraGen a design automation tool for generating correct-by-construction hierarchical protocols is introduced.

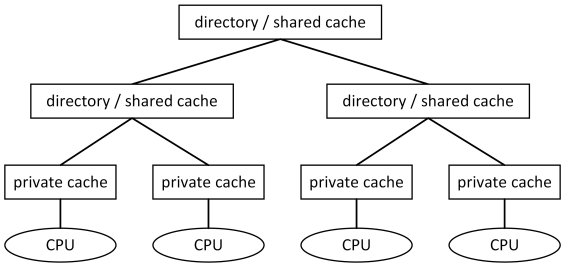
The user inputs the SSPs of each level independently. For example, as shown in Figure 7.1(d), the user would input two SSPs (shown in different colors): one for the protocol of the subtree in the bottom right (oblivious to the higher level) and one for the protocol of the higher level (oblivious to the subtree in the bottom right). The user would also specify the



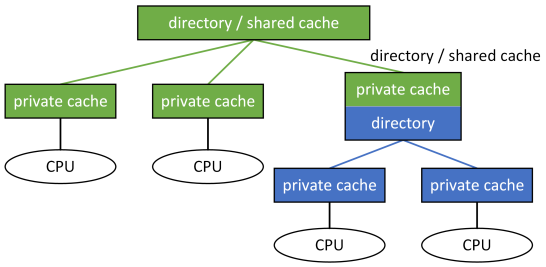
(a) flat directory



(b) hierarchical directory



(c) multi-hierarchical directory



(d) HieraGen composition of two SSPs (in different colors)

Figure 7.1: System Models

point(s) at which the protocols are connected, e.g., that the subtree protocol is a node in the higher level protocol. (For clarity, we refer to a core with its private cache(s) as a core/cache node and a directory with a collocated, optional shared cache as a directory/cache node, and we use standard tree terminology (root, parent, child) to identify nodes in the hierarchy). Thus, the higher level SSP would include only the specifications of the root directory/cache and the core/cache nodes that are its children, as in the specification of a flat protocol; it would *not* include any information about the possibility of a child that is an integrated directory/cache node.

Given these inputs, HieraGen produces, leveraging ProtoGen, the design of the complete and concurrent hierarchical protocol.

Accommodating hierarchy introduces a key challenge beyond what ProtoGen addresses: HieraGen must create intermediate (non-root) directory/cache nodes that were not completely specified by the user; as shown in Figure 7.1d, HieraGen must compose the cache from the higher level (in green) with the directory from the lower level (in blue) to produce the intermediate directory/cache node.

HieraGen addresses this challenge by automatically encapsulating higher-level coherence actions within lower-level coherence transactions (and vice versa), such that coherence invariants are enforced globally.

HieraGen leverages ProtoGen to implement concurrency in a hierarchical system that can have multiple coherence transaction serialization points (e.g., directories). We make the observation that in a hierarchical SSP that correctly enforces coherence globally, any two racing coherence transactions will serialize at exactly one of the directories. This key invariant allows us to leverage ProtoGen for generating concurrency.

Limitations: HieraGen is limited to inclusive cache hierarchies at the point of time. Finally, the protocols generated by HieraGen do not allow for direct communication between nodes of different hierarchy levels.

7.2 Baseline System Model and Terminology

For ease of explanation, we present a baseline system model with certain constraints that we will use when explaining HieraGen. In Section 7.6, we discuss the impact on HieraGen of relaxing some of these constraints.

Our baseline system model encompasses the two designs in Figure 7.1(b)-(c). For purposes of HieraGen, Figure 7.1(b) and Figure 7.1(c) are equivalent. Each directory/cache node tracks the coherence state of blocks held in the private caches of its children as well as blocks held by its collocated shared cache (if any). The root directory/cache is attached to main memory. These are both two-level designs, and we use "higher level" to refer to the level closer to the root and "lower level" to refer to the level farther from the root (e.g., the subtree on the bottom right of Figure 7.1(b)).

For brevity, we refer to the four distinct node types as: root (root directory/cache), cache-H (core/cache node in higher level protocol), cache-L (core/cache node in lower level protocol), and dir/cache (for the intermediate directory/cache node). We refer to the higher level and lower level SSPs as SSP-H and SSP-L, respectively.

We assume the following five constraints for now. In Section 7.6, we relax the first three of these constraints.

- There are only two levels of hierarchy.
- Directories are inclusive[1] and full-map, and evictions of read-only blocks are *not* silent, i.e., each directory has complete knowledge of its children's coherence state.
- Each SSP is a flat directory protocol.

[1]Directory inclusion means that a block may not be in a cache without the directory's knowledge.

- Shared caches are inclusive.
- All communication across hierarchy levels is strictly parent/child, i.e., a node cannot communicate directly with nodes in other levels. Communication within a hierarchy level is general.
- Each SSP enforces the SWMR coherence invariant.

7.3 HieraGen Tool Flow

HieraGen starts with an SSP for each level of the hierarchy, and it produces the finite state machines of all of the distinct core/cache and directory/cache nodes. We illustrate HieraGen's flow from inputs to outputs in Figure 7.2. There are two main steps: (1) from flat SSPs to an atomic hierarchical protocol, and (2) from an atomic hierarchical protocol to a concurrent hierarchical protocol.

The final outputs are the finite state machines (FSMs) for concurrent hierarchical protocols. To enable verification, we produce the FSMs in the Murϕ model checker language. Next, we discuss the two steps in detail in the following sections.

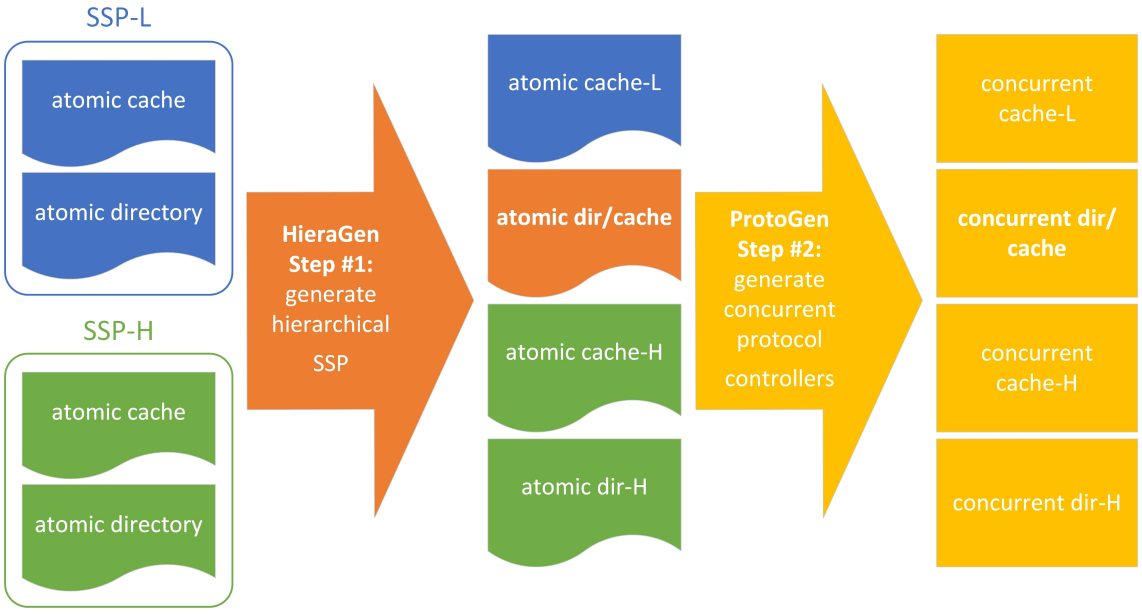


Figure 7.2: HieraGen Tool Flow

7.4 Step 1: Atomic Hierarchical Protocol

All of the complexity in this step involves the generation of the dir/cache (intermediate directory/cache node). As highlighted in Figure 7.2, it is the only node that differs from the input specifications; the other nodes effectively pass through this step unchanged. Specifically, HieraGen must compose the cache-H from the higher level with the dir-L from the lower level into one intermediate dir/cache node. The dir/cache node must integrate the functionality of a directory to its children with the functionality of a child to its parent.

7.4.1 Intuition

HieraGen's philosophy is to perform the protocol composition in the most general way possible without being "aware" of specific protocols or states, i.e., we do not modify the input SSPs in any way. The key idea is to encapsulate the coherence actions of the other level within a coherence transaction, so that SWMR and data-value invariants are enforced globally. Specifically, before completing a read request originating from a particular level, HieraGen first ensures that the other level has no writers (if there is a writer, it is first downgraded). In a similar vein, before completing a write request, HieraGen ensures that the other level does not have any readers or writers (if there are any readers or writers, they are first invalidated).

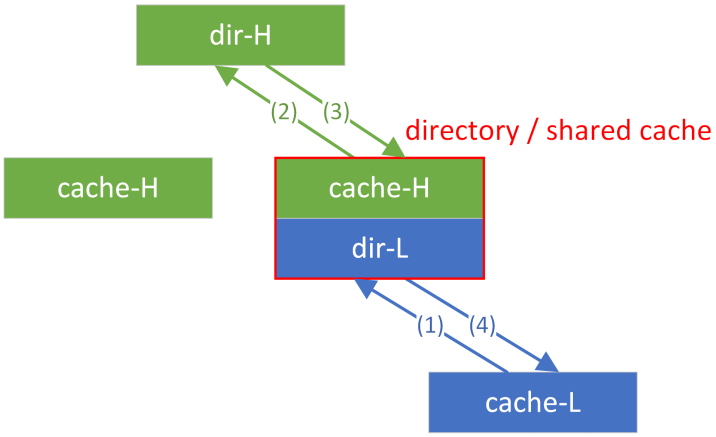


Figure 7.3: Encapsulating higher-level coherence actions within a lower-level coherence transaction

How is this enforced by HieraGen? Consider Figure 7.3 which shows a coherence request originating from cache-L to dir-L (1). Suppose dir-L determines that the request cannot be completely satisfied at the lower level. HieraGen then encapsulates higher-level protocol actions within the lower-level transaction to ensure that the higher level is ready for the access to be performed in the lower level. By processing SSP-L, HieraGen is able to figure out the access type of the lower-level coherence request (whether it is a read or a write). By leveraging

SSP-H, HieraGen makes cache-H generate a higher-level request of the same access type (2). Since SSP-H enforces SWMR in the higher level, when cache-H receives a response from dir-H (3), one can infer that the higher level is ready for the access to be performed in the lower level. Therefore, HieraGen resumes the lower-level coherence transaction, i.e., dir-L responds to cache-L (4), thus completing the lower-level coherence transaction while globally enforcing SWMR.

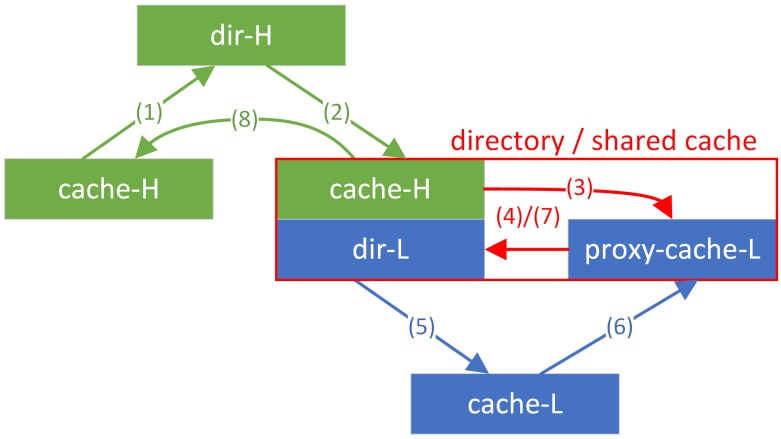


Figure 7.4: Encapsulating lower-level coherence actions within a higher-level coherence transaction via the proxy-cache

Consider Figure 7.4 which shows a coherence request originating from one cache-H (1) and then forwarded by dir-H to another cache-H (2). Suppose the request cannot be completely satisfied at the higher level. HieraGen then encapsulates lower-level protocol actions within the higher-level transaction to ensure that the lower level is ready for the request to be completed in the higher level. Like in the previous scenario, HieraGen can leverage SSP-H to determine the access type of the higher-level request. But what entity can HieraGen leverage to make a lower-level request of that access type? Cache-H cannot be used because it logically belongs to the higher level. Dir-L cannot directly be used because read or write requests do not typically originate from the directory. Therefore, HieraGen clones a cache controller from SSP-L called *proxy-cache* and integrates it into the intermediate dir/cache node as shown in Figure 7.4. HieraGen makes the proxy-cache generate a coherence request to dir-L (4), which then forwards it to one or more caches in the lower level (5). When the proxy-cache receives a response (6), one can infer that the lower level is ready for the access to be performed in the higher level. The proxy-cache then proceeds to evict the block into cache-H (7), allowing cache-H to respond to the requestor (8), thereby enforcing SWMR globally.

7.4.2 HieraGen in Detail via Transaction Flow Examples

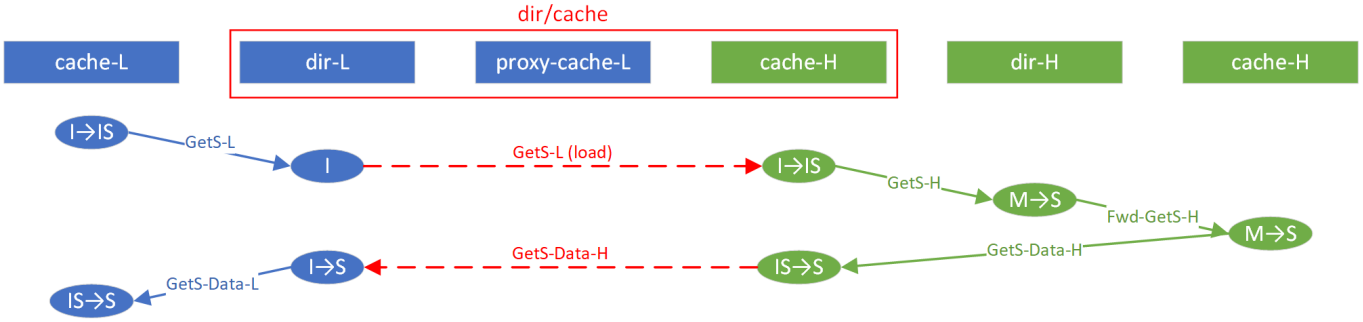


Figure 7.5: Transaction Flow 1: A load from cache-L that involves the higher level. (Read from left to right and back). Dashed arrows denote messages internal to the dir/cache. (The proxy-cache-L is uninvolved in this transaction.)

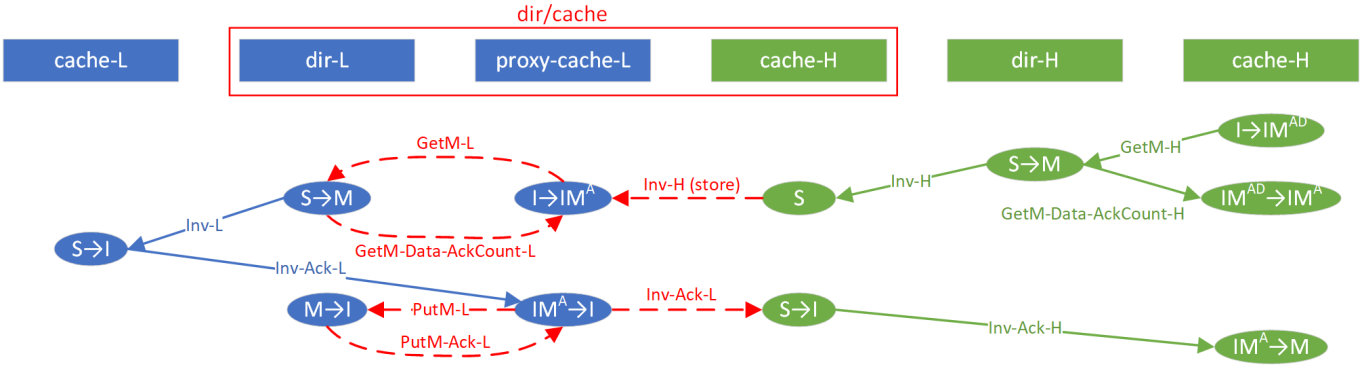


Figure 7.6: Transaction Flow 2: A store from cache-H that involves the lower level. (Read from right to left and back). Dashed arrows denote messages internal to the dir/cache.

We now illustrate how HieraGen works by describing the protocol activity it must generate for every transaction using concrete examples. We assume for now that both levels of the hierarchy use typical MSI protocols.

Protocol transactions that do not require the dir/cache to provide both of its functionalities are effectively unchanged with respect to the input SSPs. A request from a cache-L that can be completely satisfied within SSP-L occurs as expected. For example, a GetShared (GetS-L[2]) request from a cache-L to the dir/cache that can be purely handled by the dir/cache—either by sending the data directly from the dir/cache or by forwarding the request to an owner that is a child of the dir/cache—behaves as in the original SSP-L. Similarly, a request from a cache-H that can be satisfied within SSP-H also occurs as expected; as long as the request does not require the involvement of dir-L from the dir/cache, it behaves as in the original SSP-H.

[2]When otherwise ambiguous, we label coherence messages with "-L" or "-H" to denote which protocol they are in.

There are three types of transactions that require the dir/cache to provide both of its functionalities and thus require HieraGen to generate the corresponding dir/cache logic. We now walk through concrete examples of each type of transaction.

7.4.2.1 SSP-L loads/stores that involve SSP-H

Our first example is illustrated in Figure 7.5. Initially, a block is in state M in one cache-H, and the cache-L issues a GetS-L to the dir/cache. HieraGen can tell from processing SSP-L that this GetS-L request corresponds to a read, by inferring that the final state of the transaction allows only reads and not writes. Similarly, HieraGen can tell from processing SSP-H which coherence request a cache-H would issue if it needed to obtain (at least) read permissions, which is GetS-H. HieraGen matches these two request types; that is, because the cache-L's GetS-L provides read permissions, it has the dir/cache issue the GetS-H request to the dir-H (root) that a cache-H would issue for obtaining read permissions.

The dir-H and the rest of SSP-H behave as usual for that type of coherence request. In this example, the dir-H forwards the GetS-H for readable data to the cache-H that is the owner, and that owner responds with data to the dir/cache. The dir/cache fills its cache-H with the data and responds to the cache-L that made the original GetS-L request. This response is the same response as would be made to the original GetS-L request in a flat SSP-L.

7.4.2.2 SSP-H loads/stores that involve SSP-L

As illustrated in Figure 7.6, our concrete example here is a GetM-H request from a cache-H when one cache-L has the block in state S (and no other caches have the block). The cache-H sends a GetM-H to the dir-H, and the dir-H does two things: it forwards the request (in the form of an Invalidation-H) to the dir/cache, and it sends a message to the requesting cache-H to let it know how many Acknowledgments to expect. HieraGen can tell from processing SSP-H that the Invalidation-H corresponds to a write (and not a read), and it needs the dir/cache to emulate what should happen due to a request for writable data. HieraGen provides this functionality via the proxy-cache introduced in the previous section. (Recall that the proxy-cache is essentially a clone of the cache-L controller that is integrated as part of the dir/cache; it is used to encapsulate coherence transactions, but it does not perform loads or stores.)

HieraGen can tell from processing SSP-L which coherence request a cache-L would issue if it needed to obtain write permissions, which is GetM-L, and it has the proxy-cache-L issue a GetM-L to dir-L. This GetM-L is an internal request since the proxy-cache-L and dir-L are

part of the same controller. The dir-L then sends an Invalidation-L to the cache-L in state S and transitions to state M (because it now views the proxy-cache-L as being in state M). This GetM-L transaction completes once the cache-L in state S has sent an Invalidation-Ack-L to the proxy-cache-L; the transaction ensures that all of the cache-L nodes are in the appropriate coherence state (Invalid) with respect to the cache-H that made the original GetM-H request.

Once the GetM-L transaction completes, the proxy-cache-L immediately evicts the cache block into the dir/cache, causing dir-L's state to change from M to I. The dir/cache then responds to the root with the appropriate response for SSP-H (which is an Invalidation-Ack-H in the example). Note that the response would have included the data if one of the cache-L nodes had been the owner.

7.4.2.3 Dir/cache evictions

To maintain directory inclusion, an eviction from the dir/cache must first evict the block from all cache-L nodes, if any, that have the block. HieraGen again employs the proxy-cache-L for this purpose, and HieraGen exploits its ability to process the SSP-L to discover which SSP-L coherence request invalidates the block from all cache-L nodes. Thus, the proxy-cache-L issues a GetM-L, resulting in the proxy-cache-L holding the only copy of the block in SSP-L. (If a cache-L is the owner when it is invalidated, it sends its data to the proxy-cache-L.) The proxy-cache-L then evicts the block to the dir/cache. Once the only copy is at the dir/cache, the dir/cache issues a PutM-H to the root, the coherence request for evicting an owned block in SSP-H.

7.4.3 Algorithm

We now precisely illustrate how our algorithm composes cache-H, dir-L, and cache-L (proxy cache) to produce the intermediate dir/cache controller. An action can be mapped to a group of ProtoGen Protocol Language (PPL) instructions. Our algorithm essentially takes "instructions" from the input controllers and stitches them together. Therefore, to specify how our algorithm works, we must first provide a notation for the controllers.

In the following discussion, "Accesses" refers to the set of accesses: load, store, and evict. "States", "Requests", and "Fwd-requests" refer to the sets of states, requests, and forwarded requests (resp.) associated with a controller. For example, cache-L.States, refers to the set of stable states associated with the lower level cache controller. Similarly, "send-request", "send-fwd-request-response", "await-response", "update-state", and "send-response" are variables that point to groups of instructions that do what their names indicate.

7.4.3.1 Cache controller

The cache controller component of an SSP specifies, for each stable state, what happens on each access and each incoming forwarded request as well as the final state(s) eventually entered. We specify this as follows. (Note that this abstract specification must be instantiated to make up specific cache controllers as the following example illustrates.)

Algorithm 6 Cache controller transaction actions

∀ access ∈ Accesses, ∀ state ∈ cache.States

cache.send-request(access,state);

cache.await-response(access,state);

cache.update-state(access,state);

∀ fwd-request ∈ cache.Fwd-requests, ∀ state ∈ cache.States

cache.update-state(fwd-request,state);

cache.send-response(fwd-request,state);

Consider a lower level MSI cache controller: cache-L.send-request(store,I) points to instructions that sends a GetM to dir-L; cache-L.await-response(store,I) points to instructions that wait for Data; and cache.update-state(store,I) points to instructions that change state from I to M. Some of these pointers may point to empty actions—for example, a store to a block in state M needs no messages to be sent nor any state update.

7.4.3.2 Directory controller

The directory controller component of an SSP specifies, for each stable state, what happens on an incoming request. We specify this as follows. (As before, the abstract specification has to be instantiated to make up specific directory controllers.)

Algorithm 7 Directory controller transaction actions

∀ request ∈ dir.Requests, ∀ state ∈ dir.States

dir.send-fwd-request-response(request,state);

dir.await-response(request,state);

dir.update-state(request,state);

For instance, for a higher level directory controller employing a conventional MSI protocol: dir-H.send-fwd-request-response(GetS, M) points to instructions that either sends a Fwd-GetS

to the owner or responds to the request with Data; dir-H.await-response(GetS, M) points to instructions that await for Data to come from the owner; and dir.update-state(GetS, M) points to instructions that change directory state from M to S (and update sharer vector).

7.4.3.3 Generating dir/cache controller

We can now specify how dir-L, cache-H, and cache-L are composed to form the intermediate dir/cache controller. This compound controller, being a directory as well as a cache, will have to specify for the cross-product of dir-L/cache-H states, what happens on: (a) an incoming request from a cache-L (Figure 7.3); and (b) an incoming forwarded request from dir-H (Figure 7.4).

Algorithm 8 SSP-L request dir/cache controller SSP composition

```
1:  ∀ request ∈ dir-L.Requests
2:  ∀ (dir-state, cache-state) ∈ dir-L.States × cache-H.States
3:      ▷ Compute access that generated request at lower level
4:          access = compute-access(request, SSP-L);
5:      ▷ Consume the request and make a temporary copy to avoid head of line blocking
6:          tmp-request = consume-request(request);
7:      ▷ Issue request to the higher level of same access type
8:          cache-H.send-request(access, cache-state);
9:          cache-H.await-response(access, cache-state);
10:         cache-H.update-state(access, cache-state);
11:     ▷ Now respond to the initial request in the lower level
12:         dir-L.send-fwd-request-response(tmp-request, dir-state);
13:         dir-L.await-response(tmp-request, dir-state);
14:         dir-L.update-state(tmp-request, dir-state);
```

We consider the former first, as shown in Algorithm 8. By parsing SSP-L, we determine the access that leads to the request. We then perform the tmp-store-request action that allows the controller to consume the request, removing it from the input buffer and storing the data required by subsequent actions like e.g. the source of the request in temporary variables. Introducing this action does not violate HieraGen's philosophy, because the functional behavior of the SSP controllers is not changed, but it allows us to prevent potential head-of-line blocking in the input buffers.

We then make the controller send a request to the higher level of the same access type (if necessary). Finally, we have the controller respond to the original request.

Next, we deal with incoming forwarded requests. As shown in Algorithm 9, the idea is to first compute the access that led to the forwarded request. (It is worth reiterating that this computation, and indeed all of the following steps, happen at design time.) Then, we must

Algorithm 9 SSP-H fwd-request dir/cache controller SSP composition

```
1:  ∀ fwd-request ∈ cache-H.Fwd-requests
2:  ∀ (dir-state, cache-state) ∈ dir-L.States × cache-H.States
3:      ▷ Compute access that generated fwd-request at higher level
4:          access = compute-access(fwd-request, SSP-H);
5:      ▷ Consume the request and make a temporary copy to avoid head of line blocking
6:          tmp-fwd-request = consume-request(fwd-request);
7:      ▷ The proxy cache logically issues a request of the same access to dir-L. But this
            request, being internal, needn't actually be sent. We simply compute the request
            it would generate (from Invalid state) so that dir-L can respond to this virtual
            request
8:          request = compute-request(access, Invalid, SSP-L);
9:          dir-L.send-fwd-request-response(request, dir-state);
10:         dir-L.await-response(request, dir-state);
11:         dir-L.update-state(request, dir-state);
12:     ▷ The virtual proxy cache waits for response
13:         cache-L.await-response(access, Invalid);
14:     ▷ Then the proxy cache updates its state
15:         final-state = cache-L.update-state(access, Invalid);
16:     ▷ The proxy cache must now evict the block, but this is again an internal request
            and needn't be sent. We simply compute the request it would generate so that
            dir-L can respond to this virtual eviction request
17:         evict-request = compute_request(evict, final-state, SSP-L);
18:         dir-L.update-state(evict-request, dir-state);
19:     ▷ Now respond to the forwarded request
20:         cache-H.update-state(tmp-fwd-request, cache-state);
21:         cache-H.send-response(tmp-fwd-request, cache-state);
```

make the controller "perform" the access in the lower level. The "instructions" for how to do this are available in the input SSP-L, and the source of this transaction is the cache-L. That is why we leverage a proxy cache to initiate this transaction. In reality, the proxy cache is a single temporary cache line (and state) which is physically integrated within the dir/cache controller. Therefore the request from the proxy cache to dir-L need not be physically sent. Instead, we compute (at design time) what request it would have sent and simply make dir-L react to this request. We then make the controller await the response that the proxy cache would have normally waited for. Once the response is received, we make the controller update its state and make it evict the block into dir-L. Again this eviction is virtual and so no message is actually sent. Instead the request corresponding to the evict access is computed, and dir-L is made to react to this request. Finally, cache-H is made to respond to the incoming forwarded request from the higher level.

7.4.3.4 Identifying instructions Constituting an Action

From processing the SSP we know the identifiers of the request, forwarded-request and responses. By performing a static code analysis groups of instructions that constitute an action can be identified in the SSPs. If for example in a transition that is part of the transaction instructions construct and send a request message we group them into a send-request action. Any instructions awaiting and processing a response message are grouped into a await-response action.

The consume-request action stores all member variables of the request object that must be preserved for subsequent instructions when consuming the initial request. The variables that must be preserved can be determined, by statically searching the subsequent instructions for references. If an occurrence is found, we update the instruction accessing the request member variable to use our temporary copy instead.

7.4.3.5 MSI Dir/Cache Controller Transaction Flow Examples Continued

Applying the algorithms to the transaction flow examples discussed section 7.4.2 we generate the dir/cache controller by composing the actions.

In Listing 7.1 the instruction sequence generated for the dir/cache controller performing the transaction flow from Figure 7.5 is presented in PPL. In the PPL description HieraGen replaces the singleton state of the flat protocols with a custom state object tracking the state of each of the composed controllers separately: State(SProxy, SDir_L, SCache_H). All states

combined represent the dir/cache state. While no instructions update the state of SProxy when composing the SSP, HieraGen internally updates the state when generating transient states.

At completion, HieraGen converts all instruction sequences generated when composing the controllers into the transition graph intermediate representation IR forwarding them to the ProtoGen algorithm tool stage. The transition graph representation of the instruction sequence presented in Listing 7.1 is given in Figure 7.7.

Figure 7.7: Dir/cache transition graph $G_{(I_P,I_D,I_C)\rightarrow(I_P,S_D,S_C)}$ for a cache-L load access with cache-H initially in State I.

In Listing 7.2 the instruction sequence for the transaction flow shown in Figure 7.6 is given. Because the proxy-cache-L is part of the dir/cache controller and the interaction between it and the dir-L is immediate some instructions can be optimized from the composed dir/cache description. In the given example for instance, the owner field is not required to track the ownership of the proxy cache, because it will perform an immediate logical eviction after receiving the data precluding potential races caused by other transactions. Likewise it is not required to maintain a conditional statement checking whether the proxy cache is the owner when it performs the logical eviction.

Figure 7.8: Dir/cache transition graph $G_{(I_P,S_D,S_C)\rightarrow(I_P,I_D,I_C)}$ for a cache-L load access with cache-H initially in State I.

```
1  // Cache behavioural description
2  Architecture dir_cache {
3
4      // HieraGen replaces the singleton state with a custom state object tracking the state
           of the composed controllers separately: State(SProxy, SDir_L, SCache_H)
5      ...
6      // The initial composed state is (SProxy: I, SDir_L: I, SCache_H: I)
7      Process((I,I,I), GetS_L, FState){
8          // consume-request
9          tmp_req_src = GetS_L.src;
10
11         // cache-H.send-request
12         msg = Request(GetS_H, ID, Dir_H.ID);
13         req.send(msg);
14
15         // cache-H.await-response
16         await{
17             when Data_H:
18                 cl = Data_H.cl;
19
20                 // cache-H.update-state
21                 FState.SCache_H = S;
22
23                 // dir-L.send-fwd-request-response
24                 msg = Response(Data_L, ID, tmp_req_src, cl);
25                 resp.send(msg);
26
27                 // dir-L.await-response
28                 FState.SDir_L = S;
29
30                 // Transaction complete
31                 // The final composed state is (SProxy: I, SDir_L: S, SCache_H: S)
32                 exit;
33
34         }
35     }
36     ...
37 }
```

Listing 7.1: PPL description of the behavior of the dir/cache controller for a cache-L load access with cache-H initially in State I.

```
1  // Cache behavioural description
2  Architecture dir_cache {
3
4      // HieraGen replaces the singleton state with a custom state object tracking the state
           of the composed controllers separately: State(SProxy, SDir_L, SCache_H)
5      ...
6      // The initial composed state is (SProxy: I, SDir_L: I, SCache_H: I)
7      Process((I,S,S), Inv-H, FState){
8
9          // consume-request
10         tmp_fwd_req_src = Inv-H.src;
11
12         // dir-L.send-fwd-request-response
13         msg = Request(Inv-L, ID, sharers);     // Multicast Inv-L to all sharers
14         req.send(msg);
15         // Instead of sending a message from the dir-L to proxy-L directly assign variable
               values
16         acksExpected = sharers.count();
17         sharers.clear();
18
19         // dir-L.update-state
20         // Optimization example: A new value is subsequently assigned to  variable
21         // FState.SDir_L without the current assignment being ever used
22         // FState.SDir_L = M;
23
24         // cache-L.await-response
25         await{
26             when InvAck-L:
27                 acksReceived = acksReceived + 1;
28
29                 if acksExpected == acksReceived{
30                     acksExpected = 0;
31                     acksReceived = 0;
32
33                     // dir-L.update-state
34                     undefine owner;
35                     FState.SDir_L = I;
36
37                     // cache-H.update-state
38                     FState.SCache_H = I;
39                     // cache-H.send-response
40                     msg = Response(InvAck-H, ID, tmp_req_src.ID);
41                     resp.send(msg);
42
43                     // Transaction complete
44                     // The final composed state is (SProxy: I, SDir_L: I, SCache_H: I)
45                     exit;
46                 }
47         }
48     }
49     ...
50 }
```

Listing 7.2: PPL description of the behavior of the dir/cache controller for a cache-L load access with cache-H initially in State I.

7.4.4 Compatibility Between Protocol Levels

HieraGen can compose SSPs together into a hierarchical protocol, but not all SSPs are immediatly compatible. Specifically, there is one protocol feature—silent upgrading of coherence permissions—that can cause incompatibility if not handled appropriately. In "typical" protocols that use a subset of the MOESI stable coherence states, the culprit is the (E)xclusive state. In the E state, which is read-only, a cache can silently upgrade to the M state, which is read-write.

The issue of protocol compatibility is best explained through an example of incompatibility. Consider the case in which SSP-L is MESI and SSP-H is MSI. Assume initially that the block is Invalid in all caches. One cache-L performs a load, misses, and issues a GetS-L to the dir/cache. The dir/cache issues a GetS-H to the root, and the root responds to the dir/cache with Shared (read-only) permission and data. The root records that its dir/cache child is in state S. Because SSP-L is MESI and there were no sharers at the time of the GetS-L from the requesting cache-L, the dir/cache responds to the cache-L with data and Exclusive permissions. Now the cache-L can silently transition from E to M and write the block. Meanwhile, any cache-H can issue a GetS-H to the root and obtain Shared access. In this situation, the hierarchical protocol violates the SWMR coherence invariant.

Fortunately, HieraGen can automatically detect incompatibility when processing the SSPs, because it can detect when an SSP permits silent permission upgrades. In our example above, since the state E and M are element of the same state set and

In our example above, since the cache-L can silently transition from E to M and write the block, HieraGen can infer that E state is writable. There are two solutions to this problem. The first one is more intuitive, but the second one offers better performance.

The intuitive solution is for HieraGen to have the dir/cache conservatively issue a GetM-H request that corresponds to the greatest permissions that the cache-L could receive (in this case, read-write access), rather than the GetS-H request that corresponds to the original request (for read-only access). While this solution ensures safety, it has negative performance implications due to needless SSP-H invalidations if the originating cache-L does not write to the block.

The higher performance solution avoids these needless invalidations. The dir/cache issues a GetS-H, and the root responds to the dir/cache with read-only access. HieraGen, when generating the dir/cache, adds logic to detect mismatches between the permission its cache-H received from SSP-H (S=read-only) and the permission its dir-L would otherwise grant to the cache-L requestor (E=silently upgradeable to read-write). In this case, it has the proxy-

cache-L issue a request for the *received* permission, i.e., the proxy-cache-L issues a GetS-L. In this way, the proxy-cache-L mimics the possible behavior of an external cache-H. The proxy-cache-L's GetS-L, which is serialized before the cache-L's GetS-L, puts the dir-L momentarily in state E, from which it will now grant S (not E) permissions to the requesting cache-L. Once the dir/cache has responded to the cache-L, the dir-L changes to state S, and the proxy-cache-L evicts its block.

With this more optimized solution, there is one last issue to resolve. Let us assume that both the SSP-H and SSP-L have protocols with E states. Initially let us assume that all blocks are in I state. Consider the situation in which cache-L issues a GetS-L to request a block for reading; the dir/cache first obtains the block in state E (due to SSP-H) and provides the block in E state for the requesting cache-L (due to SSP-L). Now the cache-L can silently upgrade to M and modify the block, without notifying the dir/cache, as per SSP-L. We now have another mismatch to resolve. We want the dir/cache's cache-H to change to state M so that it is compatible with the earlier store that was performed at cache-L. Furthermore, we want to do this without modifying SSP-H, which we recall is HieraGen's philosophy.

When cache-L evicts the block to the dir/cache, the type of eviction message (PutE-L or PutM-L) reveals the access that was performed in the cache-L. In our case, PutM-L reveals that a write occurred. Therefore, there must now be a write of cache-H to update it with the evicted data. But because cache-H is already in state E, it does not need to issue a request to dir-H; it can silently go to state M as per SSP-H.

7.4.5 Optimizing Protocol Finite State Machines

As explained thus far, HieraGen will create somewhat unoptimized finite state machines for the root and dir/cache. Consider a state machine to be a 2D matrix, in which rows are coherence states, columns are events (incoming coherence messages), and entries specify what happens for that state/event pair. Naively, a specification of a directory (either the root or the directory part of the dir/cache) or a cache controller would have an entry for every possible state/event pair, i.e., NumRows*NumCols entries. However, many of those entries are not actually reachable, because certain events cannot occur in certain states.

Without optimization, the finite state machines for directories will needlessly include logic to handle unreachable state/event pairs. This logic is not harmful, but it is unnecessary; it may also complicate debugging in that it may be useful to know that a believed-to-be unreachable state/event has been reached. We have implemented a custom model checker that explores the reachable state space of the protocols so as to eliminate these unreachable state/event pairs.

7.5 Step 2: Concurrent Hierarchical Protocol

At the end of Step 1, HieraGen has produced an atomic hierarchical protocol in the form of finite state machines for the cache-L, dir/cache, cache-H, and root. In Step 2, we add concurrency to this protocol.

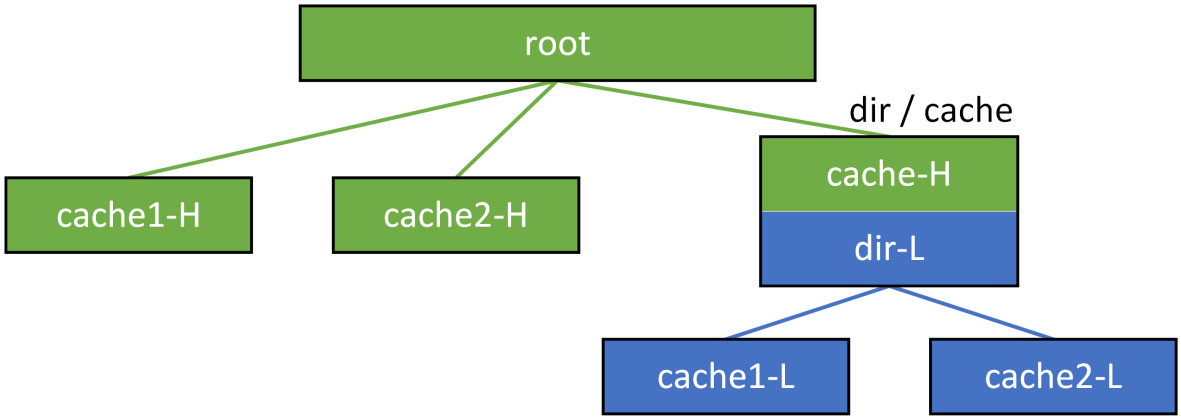


Figure 7.9: There is a unique serialization point for any two racing transactions.

For flat protocols, ProtoGen injected concurrency as explained in Section 6. ProtoGen could leverage the fact that every coherence transaction was serialized at the directory. (There is just one directory in a flat protocol). HieraGen can still leverage transaction serialization, but in a hierarchical protocol, there can be multiple serialization points, depending on the protocol level of the block's current owner. Consider the system model in Figure 7.9. A request from a cache-L that is satisfied entirely within SSP-L is serialized at the dir/cache. However, a request from a cache-H or a request from a cache-L that cannot be satisfied within SSP-L is serialized at the root.

Despite the two serialization points (or more, if more levels of hierarchy), the tree structure of the hierarchy provides the invariant that any two racing coherence transactions are serialized at exactly one of these serialization points. ProtoGen can thus be leveraged for extracting concurrency.

Consider an example in which cache1-H holds a block in writable state. Assume two racing transactions: cache2-H and cache1-L both want to write to the block. Accordingly, both of their requests will attempt to obtain ownership of the block; the request that reaches the root first will win the race. In other words, the root is the serialization point.

For the same initial state—i.e., cache1-H initially holding the block in writable state—let us now consider two racing transactions coming from the lower level. Specifically, assume both cache1-L and cache2-L want to obtain ownership of the block. In this situation, although there are potentially two serialization points in play, the hierarchical nature means that the

first transaction to reach the dir/cache wins the race; the transaction to reach the dir/cache second will be able to infer that it has lost the race and will not proceed to the root. In other words, the dir/cache is the serialization point.

Let us now generalize. Because a HieraGen-generated atomic protocol enforces SWMR globally, there can be exactly one owner for any block. (It is the cache that holds the block in writable state; if there is no such cache, the owner is the root). Any two racing transactions, therefore, will both attempt to reach this unique owner by traversing a path consisting of one or more directory nodes. The tree structure guarantees there will be exactly one directory node in common across the two paths. This is because the transaction that reaches this common node second can infer that it has lost and will not proceed any further towards the original owner. In other words, this common directory will serve as the unique serialization point, allowing us to use ProtoGen for performing Step 2.

7.6 Other System Models

In Section 7.2, we presented our baseline system model, and we listed six design constraints that we imposed at the time. We now explore the effects of relaxing each of them.

7.6.1 Deeper Hierarchies

As systems continue to scale, there is likely to be incentive to use more levels of hierarchy. We consider whether deeper hierarchies affect how HieraGen composes SSPs into a hierarchy (Step 1) and how HieraGen introduces concurrency (Step 2).

7.6.1.1 Step-1

Composition is unaffected by the depth of the hierarchy, for two reasons. First, at each point of SSP composition, there is a structured interface consisting of a single dir/cache node that provides the cache functionality to its parent and the directory functionality to its children. This structured interface is used by HieraGen to ensure that before a coherence request from one level completes, each of the other levels are in a state that allows for this coherence request to complete without violating SWMR globally. Second, HieraGen does not permit any communication across levels, without this structured interface. Thus, the reasons for why HieraGen works for composing two SSPs into a hierarchy also apply for composing additional SSPs.

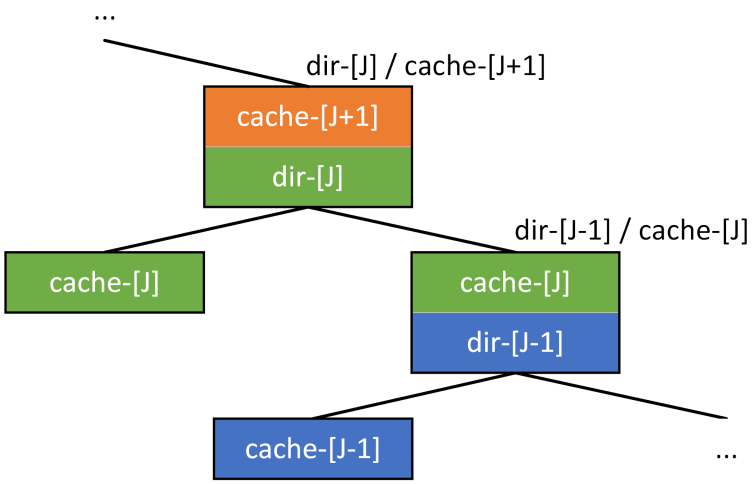


Figure 7.10: How HieraGen works with deeper hierarchies. A write from cache-J leads to a write in the higher level (via dir–[J]/cache–[J+1]) and write in the lower level (via dir–[j-1]/cache–[J])

Figure 7.10 illustrates how HieraGen's tree structure ensures SWMR globally. Consider an n-level hierarchy in which cache-J (a cache from the j^{th}) level performs a write. Assume that there are one or more sharers in level(s) higher than j as well as level(s) lower. Before cache-J's write request is completed, HieraGen issues write requests to the higher levels (via dir–J/cache–[J+1]) as well as lower levels (via dir–[j-1]/cache–J), thereby ensuring SWMR globally.

7.6.1.2 Step-2

Similarly, deeper hierarchies have no impact on how HieraGen introduces concurrency into the hierarchical protocol. This is because, irrespective of the depth of the hierarchy, any two racing transactions serialize at exactly one node. This enables the use of ProtoGen to uncover concurrency.

7.6.2 Incomplete Directory Knowledge

There are three ways in which a directory can have incomplete or stale knowledge of its children's coherence states: the directory uses an incomplete data structure (e.g., coarse sharing vector), its child caches are permitted to perform silent evictions of read-only blocks, or the directory is not inclusive. Fortunately, this design issue does not affect HieraGen because it is handled in the input SSPs.

For example, assume that SSP-L uses a non-inclusive directory. Recall that in a non-inclusive directory, a directory miss does not mean that the block is uncached in any of its children. Therefore, to ensure SWMR, SSP-L would already have had to revert to a broadcast

on a write to a block missing in the directory. This feature of SSP-L is what HieraGen leverages to ensure SWMR at the level, and hence globally.

7.6.3 Other SSP Protocol Types

We have assumed that each SSP is a flat directory protocol, but there is some flexibility here. The key is that the protocol must have a single structure that can serve as an interface between hierarchy levels. Directory protocols naturally have that structure: the directory.

However, a snooping protocol can also be viewed as a directory protocol with a "null directory," sometimes denoted Dir_1B. In such a snooping protocol, every coherence request is sent to the "directory," and the stateless directory simply broadcasts the request to all of its children. As long as the interconnection network provides point-to-point ordering between the "directory" and each of its children, this protocol will provide broadcast snooping.

7.6.4 Non-Inclusive Shared Caches

We have thus far assumed that shared caches are inclusive, but there are systems that provide either exclusion or non-inclusion (i.e., neither strictly inclusive nor exclusive).

In its current form, HieraGen is limited to inclusive shared caches. The underlying reason for this limitation is our philosophy of (a) allowing the user to specify the SSPs completely independently, and (b) not modifying the SSPs when composing them. Because we have drawn a sharp line between the SSPs, they have no knowledge of each other.

To illustrate the problem with non-inclusion, consider the following scenario. One cache-L is in M, its dir-L knows that the cache-L is in M, and the block is in M in the shared cache (i.e., the cache part of the dir/cache). In a non-inclusive model, the shared cache would consult with the dir-L to decide how to proceed. If any of its cache-L children still has the block in state M, the shared cache can evict silently; else, the shared cache has the only up-to-date owned copy, and it must not drop it silently. However, with our separation between SSPs, the shared cache cannot consult the dir-L in this fashion, and thus it cannot know how to proceed.

Future work will explore the possibility of relaxing the sharp break between the SSPs, in order to enable non-inclusive shared caches.

7.6.5 Communication Across Levels

We have assumed a tree-structured hierarchy in which communication is strictly hierarchical. A node can communicate only with its parent, children, or siblings. Thus, for example, a

cache-L cannot directly communicate with the root or a cache-H. This assumption enables us to clearly reason about the composition of SSPs, and it is critical to the current design and implementation of HieraGen. We can imagine a future tool that overcomes this limitation, but we leave this project to future work.

7.6.6 Non-SWMR Protocols

We have assumed that each SSP enforces the SWMR coherence invariant. In HieraGen, we have leveraged the fact that the hierarchical composition of SWMR protocols continues to provide SWMR. Thus HieraGen has no impact on subtle issues of memory consistency.

However, there exist protocols that relax SWMR and instead enforce invariants that suffice for a specific memory consistency model. Leveraging the compound memory consistency model developed for HeteroGen presented in chapter 8, enables HieraGen to hierarchically compose coherence protocols with different memory consistency models. However, while being functionally correct some coherence protocol combinations may yield a poor performance. This is the case, because the directory must reason about the memory accesses performed by its associated caches either based on the messages observed or its own coherence state. In e.g. the RCC coherence protocol presented in Table 2.1 and 2.2, the directory is unable to determine, whether a data read request is associated with an acquire or a read access. Therefore, when generating the dir/cache node HieraGen must interpret the data read request to be issued due to the access with the strongest access ordering guarantees, which in this case would be the acquire. The strongest ordering guarantees of the access must be a super set of all ordering guarantees assumed by the other possible accesses. However, in case of a hierarchical RCC protocol translating every read access into a higher-level acquire will result into the dir/cache node being required to perform a self-invalidation for every read effectively eliminating its caching functionality.

One solution to this problem is to use instead e.g. a MOESI coherence protocol for the higher-level supporting a strong memory consistency model like SC by design. A second solution is to label the lower-level cache requests with an identifier that allows the directory to infer which access type caused the data request to be sent. E.g. in case of the RCC coherence protocol, both the read and acquire accesses cause a data read request *GetV* to be sent to the directory. The requests are renamed to *GetV_R* in case of a read and *GetV_A* in case of an acquire. This enables the directory to infer the access performed by the lower-level cache translating it into the correct higher-level cache access adhering to the compound consistency model.

7.7 Experimental Evaluation

The goal of this evaluation is to determine the effectiveness of HieraGen in producing concurrent, hierarchical protocols. To illustrate the design automation benefits of HieraGen, we first compare the complexity of the input SSPs to the complexity of the hierarchical SSP and the concurrent hierarchical protocol. We then discuss the verification of the generated protocols.

We view hierarchy as useful—as do the architects of existing hierarchical protocols—and we do not perform experiments to quantitatively confirm its benefits.

7.7.1 Benchmarks

Our "benchmarks" are flat input SSPs along with the descriptions of how the hierarchy is structured (e.g., that SSP-L is attached to SSP-H at a specified point). These SSPs include typical MSI, MESI, MOSI, and MOESI protocols, like those found in Nagarajan et al. [1] but without the concurrency.

In Table 7.1, we present the complexity of the flat input SSPs. Complexity is difficult to quantify precisely but, for a *flat* coherence protocol, the numbers of states and reachable state/event pairs (i.e., transitions) are reasonable proxies.

Protocol	Cache	Directory
MI	2/9	2/4
MSI	3/26	3/16
MESI	4/33	4/25
MOSI	4/38	4/24
MOESI	5/45	5/33
RCCO	3/28	2/6
RCC	3/37	1/3

Table 7.1: Flat atomic protocols. Each entry is the number of stable states/transitions.

7.7.2 Design Complexity

The goal of HieraGen is to overcome the design complexity of manually designing hierarchical protocols. Thus we provide HieraGen with pairs of input SSPs, one SSP-L and one SSP-H, and study the concurrent hierarchical protocols it produces.

SSP-L/SSP-H	dir-L	cache-H	dir/cache
MSI/MI	4/16	5/9	10/42
MI/MSI	2/4	10/26	12/37
MSI/MSI	4/16	10/26	21/94
MESI/MSI	6/25	10/26	26/119
MESI/MESI	6/25	12/33	40/184
MOSI/MSI	4/24	10/26	28/149
MOSI/MOSI	4/24	14/38	42/227
MOESI/MOESI	5/33	16/45	59/368
RCCO/MOESI	2/6	16/45	25/85
RCC/MOESI	1/3	16/45	16/55

Table 7.2: Complexity of atomic hierarchical protocols produced by HieraGen. Each entry is the number of states(stable+transient)/transitions.

For additional insight, we first study the results of Step 1, i.e., the atomic hierarchical protocols. In Table 7.2, we show the quantifiable complexity results for seven different hierarchical protocols. Each row of the table compares, for a given hierarchical protocol, the complexity of the dir-L and cache-H of the input SSPs to the automatically-generated dir/cache. We note first that the dir-L and cache-H appear to have greater complexity than they did in Table 7.1; this discrepancy is because we are now considering the dir-L and cache-H after HieraGen has expanded them to include transient states (but still no concurrency).[3] Potential races introduced due to these transient states will have to be considered in Step 2, and hence we report these transient states.

Looking at these results, we observe that the complexity introduced in Step 1 can be considerable, and it varies considerably across the protocols. At the low end, the dir/cache in the MSI/MI hierarchy, when compared to the sum of its constituent parts, has only one more state and roughly double the number of reachable transitions. At the high end, the MOESI/MOESI dir/cache has 59 states and 368 transitions, far more than the sum of its parts (21 states and 78 transitions). We also remind the reader that not all states and transitions are equally easy to reason about, and we find it particularly challenging to reason about the dir/cache and how it must bridge the two SSPs.

We now examine the additional complexity introduced during Step 2, when HieraGen adds concurrency to the protocols. We run HieraGen twice for each protocol, once with the

[3]Even atomic protocols have transient states. Atomicity simply means that messages from other transactions cannot intervene in transient states.

	Atomic Hierarchical				Concurrent Hierarchical Stalling				Concurrent Hierarchical Non-stalling			
SSP-L/SSP-H	cache-L	dir/cache	cache-H	root	cache-L	dir/cache	cache-H	root	cache-L	dir/cache	cache-H	root
MSI/MI	10/26	10/42	5/9	2/4	9/31	10/43	4/9	2/6	12/42	10/43	4/9	2/6
MI/MSI	5/9	12/37	10/26	4/16	4/9	12/41	9/31	4/24	4/9	16/53	12/42	4/24
MSI/MSI	10/26	21/94	10/26	4/16	9/31	21/102	9/31	4/24	12/42	28/126	12/42	4/24
MESI/MSI	12/33	26/119	10/26	4/16	10/37	26/126	9/31	4/24	13/48	31/145	12/42	4/24
MESI/MESI	12/33	40/184	12/33	6/25	10/37	39/191	10/37	6/45	13/48	44/210	13/48	6/45
MOSI/MSI	14/38	28/149	10/26	4/16	10/40	31/170	9/31	4/24	13/51	39/206	12/42	4/24
MOSI/MOSI	14/38	42/227	14/38	4/24	10/40	47/273	10/40	4/37	13/51	64/353	13/51	4/37
MOESI/MOESI	16/45	59/368	16/45	5/33	11/46	64/415	11/46	5/66	14/57	81/495	14/57	5/66
RCCO/MOESI	13/28	38/179	16/45	5/33	14/30	39/195	16/57	5/66	14/30	44/214	19/66	5/66
RCC/MOESI	20/37	22/68	16/45	5/33	20/37	22/76	16/57	5/66	20/37	25/85	19/66	5/66

Table 7.3: Concurrent hierarchical protocols. This table shows the complexities of the HieraGen-generated concurrent cache-L, dir/cache, cache-H, and root nodes compared with their atomic counterparts. We present both stalling and non-stalling (maximum pending transaction limit (L) chosen as 1, see section 6.6.3.2) protocol variants. Each entry is the number of states(stable+transient)/transitions.

input flag set to produce a stalling protocol and once with the flag unset. Table 7.3 shows the quantifiable complexity for HieraGen's final output; the table reproduces the results from Table 7.2 to facilitate visual comparisons.

One perhaps curious result is that the nodes in the concurrent protocols often have approximately the same—or even fewer—states than the corresponding atomic protocols. This phenomena is because, even though the protocol complexity has increased, HieraGen can often discover how to merge states that are equivalent. For example, a cache in an MSI protocol may have states MI and SI, which denote that the cache has evicted a block in state M or state S, respectively. Because all messages that arrive in these states are distinct, they can be merged. Manual protocol designers, including the authors, tend not to merge states in this way. In fact, for clarity when reading and debugging a protocol, designers are likely to want to distinguish states like these, but HieraGen does not need to care as much about readability or debuggability (since the generated protocols are correct by construction).

Ultimately, what is more important than the quantifiable complexity metrics is HieraGen's ability to automatically and nearly instantaneously produce protocols that are correct by construction. HieraGen took less than 10 seconds to correctly generate each of the protocols in this section.

7.7.3 Verification of Correctness

To confirm HieraGen produces correct protocols—protocols that never violate coherence and never deadlock—we perform three verifications for every protocol.

First, we use the Murϕ model checker [26] to formally and completely verify the atomic hierarchical protocol that is produced by Step 1 of HieraGen for a configuration depicted in Figure 7.1b and Figure 7.1d. The configuration consists of a single root directory, two cache-H nodes, the generated dir/cache (including the proxy-cache-L), and two cache-L nodes.

Second, we verify the concurrent hierarchical protocols generated by HieraGen for the same configuration. The verification is performed on a server with 256GB of memory.

Third, to gain more confidence we add one additional cache-L node, resulting in a configuration consisting of: a single root directory, two cache-H nodes, the generated dir/cache (including the proxy-cache-L), and three cache-L nodes. However, Murϕ runs out of memory for this configuration. To extend the verification to this configuration, we used the hash compaction capability provided by Murϕ [67]. Hash compaction compresses the state descriptors stored within the state table to reduce the memory footprint of the model checker during verification. Due to the compression, there exists a small but non-zero probability that system states are omitted during verification; after each verification run, Murϕ model reports this omission probability. Because Murϕ randomly picks independent compression functions for the state descriptors, the omission probabilities of different runs can be multiplied. For each coherence protocol, we performed multiple verification runs, until the probability of an undetected bug fell below a threshold of 0.001%.

7.8 Related Work

There are three primary areas of related work: frameworks for structured hierarchical protocols, design automation for coherence protocols, and hierarchical protocols that are designed for verifiability.

MCP [7] is a design framework that seeks to minimize design complexity by cleanly separating the two functionalities of the dir/cache (our term) into the manager (directory) and client (cache). Unlike HieraGen, MCP does not provide any design automation. Cook [68] automates by providing a protocol communication template. The template has many nice features, including hierarchy, but several critical constraints, including: blocking directories, no sibling-sibling communication, and caches that block during writebacks.

The second area of related work is in hierarchical protocol design that facilitates verification. Verifiability can ensure that the bugs that are likely to be introduced during manual protocol design are caught, but verifiability does not necessarily simplify the design process. Verifiable hierarchical protocols include HCC [10], Fractal Coherence [14], and protocols that conform to the Neo framework Neo [13].

A recent publication citing the HieraGen publication [69] that covers all three primary areas although focusing on verifiability is the Hemiola DSL and Verification Tool [70]. Like HieraGen, Hemiola provides a DSL that allows the user to describe coherence protocols by using only atomic transactions. However, the atomic transactions must be described using rule templates provided by Hemiola ruling out direct cache-to-cache communication. For any transaction spanning across hierarchy levels Hemiola and HieraGen enforce that all response messages are observed by the directories. However, while Hemiola uses locks rather than transient states to identify whether a subsequent request must be stalled, HieraGen can use transient states to generate non-stalling protocol implementations. Furthermore, HieraGen allows caches within a hierarchy level to directly communicate with each other.

HieraGen can construct cache hierarchies from a wide range of coherence protocols that assume different memory consistency models. The Hemiola framework was only applied to MSI and MESI coherence protocols that enforce SWMR, but it can also generate non-inclusive cache hierarchy protocol versions of these due to its framework approach.

The Hemiola framework uses the Coq proof assistant to formally verify the correctness of the generated coherence protocols. Performing a formal verification of the generated hierarchical protocols is advantageous compared to using a exhaustive model checker like Murϕ if the computation resource availability is limited.

7.9 Summary

In this chapter we have presented and discussed the HieraGen algorithm. The HieraGen algorithm composes SSPs to generate controllers facilitating the implementation of hierarchical cache coherence protocols. HieraGen can automatically generate correct-by-construction hierarchical protocols with dozens of states and hundreds of transitions.

Chapter 8

HeteroGen

8.1 Introduction

There are two trends in modern processor design that inspire our work: processor core heterogeneity and cache coherent shared memory. Heterogeneity, which was once largely confined to CPU/GPU designs, has expanded to encompass a much wider range of core designs. Because modern processors are power constrained, there is increasing motivation to design special-purpose cores (i.e., accelerators) for important tasks, because these cores can be more power-efficient and performant than general-purpose CPU cores. In addition to heterogeneity, our other motivating design trend is the continued reliance on cache coherent shared memory. In the early days of CPU/GPU designs, the CPU cores did not share memory with the GPU cores. Even when shared memory emerged and then became prevalent due to its popular programming model, conventional wisdom suggested that cache coherence was infeasible due to scalability issues. Nevertheless, hardware cache coherence—often with accelerator-specific protocol features —is highly desirable for programmability, and it has become prevalent in heterogeneous processors. Indeed, cache coherent shared memory has become codified in several standardized design frameworks, including HSA [71], CCIX [72], OpenCAPI [21], Gen-Z [22], AMBA CHI [5], and CXL [73].

To overcome the challenges created by design complexity and consistency model we have developed HeteroGen. The key idea of HeteroGen allowing it to fuse protocols is that when a store from a processor core is made globally visible within one of the clusters, HeteroGen leverages the coherence protocols of the other clusters to make the store visible in the other clusters as well. In doing so, HeteroGen preserves the relative ordering of two stores from a core within a cluster in the other clusters as well – thereby ensuring the compound consistency model.

We have used HeteroGen to generate several heterogeneous protocols, using it to combine a range of protocols including the MOESI variants that enforce SWMR and self-invalidation based protocols that directly enforce the consistency model. We have validated that all of the protocols generated by HeteroGen satisfy their compound consistency models, and that they are deadlock-free. For consistency validation, we create litmus tests [74; 75] that are specific to compound consistency models.

We experimentally evaluate HeteroGen against a manually-designed heterogeneous protocol called HCC [76]; this is a publicly-available protocol that is similar to Spandex[19], running on a heterogeneous multicore system with 60 "tiny" in-order CPU cores and 4 "big" out-of-order CPU cores, with the big cores employing MESI and the tiny cores employing a variant of DeNovo. Our experiments reveal that the performance of our automatically-generated protocol is comparable to the manually-generated HCC.

Limitations. Although HeteroGen can handle a wide variety of protocols it cannot yet handle update-based protocols, or protocols that use leases. Furthermore, because our compound consistency formalism is limited to non-scoped multi-copy-atomic memory models, HeteroGen cannot offer any guarantees when fusing protocols that enforced scoped consistency models or non-multi-copy-atomic memory models.

8.2 Background

8.2.1 The Memory Consistency Model

The hardware memory consistency model is part of the instruction set specification and specifies how memory must appear to the (systems) programmer [1]. As such, all major commercial (homogeneous) processors support precisely defined consistency models that are specified as part of the ISA specification. For example, all processors from Intel and AMD support the x86-TSO consistency model [77], whereas ARM [78] and RISC-V [79] processors support more relaxed models.

An important feature of memory models concerns the manner in which stores propagate their values to other processors. In *multi-copy atomic* memory models [1], store values propagate atomically: as soon as the value becomes visible to another processor, no future load (in logical time) can access an earlier value. It is worth noting that a number of processor vendors and commercial architectures (including x86, ARMv8, and RISC-V) support multi-copy atomic models, and in this work we restrict our attention to such memory models.

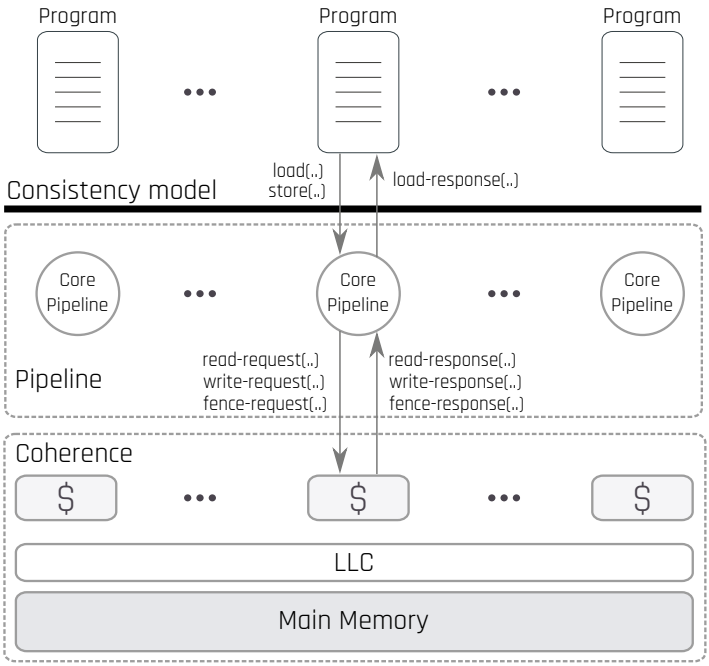


Figure 8.1: Normally the consistency model enforced by the processor is a function of both the processor pipeline and the coherence protocol combined. We isolate the consistency model enforced by a coherence protocol as the model that is enforced when an in-order pipeline (that issues memory operations one by one) is combined with that coherence protocol.

Another memory model feature, which pertains to GPUs and existing heterogeneous memory models, is the notion of *scopes* [80]. While scopes are an important feature in today's GPU memory models [81], whether or not future heterogeneous consistency models should involve scopes is still under debate [82]. In this work we limit our attention to memory models without the notion of scopes.

8.2.2 The Coherence Interface

The processor cores interact with the coherence protocol through an interface consisting of reads, writes, and fences [1]. A read request takes in a memory location as the parameter and returns a cache block. A write request takes in a memory location and a value (to be written) as parameters and returns an acknowledgment.

There are many coherence protocols that have appeared in the literature and been employed in real processors. Some of these protocols—especially the ones targeted towards the CPUs—enforce the Single-Writer-Multiple-Reader (SWMR) invariant by invalidating sharers on a write. Not all protocols enforce SWMR, however. Some protocols eschew writer-initiated invalidations, and instead rely on writebacks and self-invalidations.

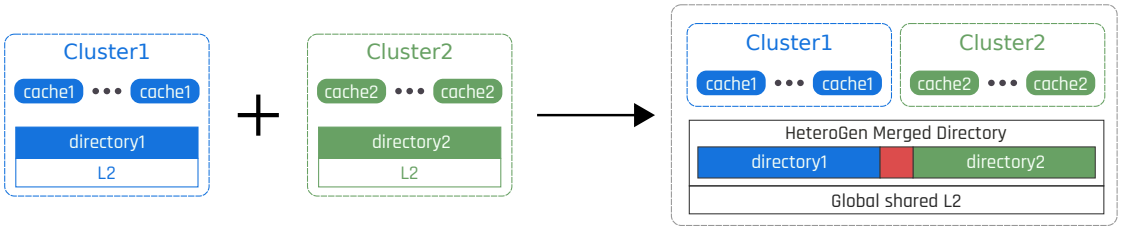


Figure 8.2: HeteroGen takes as its inputs the coherence protocols of the individual clusters (Cluster1 and Cluster2) and automatically merges their directory controllers to produce the merged directory. The target system model of HeteroGen consists of multiple clusters of cores with their private L1 caches and global shared L2.

The hardware memory model enforced is a function of both the processor pipeline and the coherence protocol, as shown in Figure 8.1. In this work, however, we want to be able to reason about composing together heterogeneous coherence protocols. Therefore, we want to be able to isolate the consistency effects of the coherence interface. We define it as follows.

Consider for now a simple "in-order" pipeline that simply makes calls to the coherence protocol interface one by one, waiting for the previous to return before the next request. We call the ensuing consistency model the consistency model enforced by the coherence protocol. This consistency model allows us to abstract coherence protocol heterogeneity by associating consistency labels for the read-requests and write-requests of the coherence interface.[1]

Thus, a conventional writer-initiated SWMR-enforcing protocol is said to enforce sequential consistency (SC). Consequently such a protocol is associated with SC-read-requests and SC-write-requests. Protocols such as TSO-CC [27] and Racer [84], that are designed to target TSO, are said to enforce TSO, and are hence associated with TSO-read-requests and TSO-write-requests. In a similar vein, protocols that target variants of release consistency (RC), such as lazy release consistency [36], are said to enforce RC. Consequently the coherence interface involves two types of writes (release-write-requests and data-write requests) and two types of reads (acquire-read-requests and data-read-requests).

8.3 Related Work

8.3.1 Heterogeneous Coherence Protocols

One industrial approach to heterogeneous coherence has been the development of protocol standards such as HSA [71], CAPI [21], CCIX [72], CHI [5], Gen-Z [22], and CXL [73]. Crossing Guard [20] proposes a similar coherence interface between the CPU and accelerators whereas hUVM [85] proposes a unified protocol based on the VIPS [86] line of work. To provide coherence between cores and accelerators that may have very different interfaces to the memory system, Alsop et al. [19] developed the flexible Spandex coherence interface. None of this prior work takes existing protocols and automatically integrates them.

8.3.2 Consistency for Heterogeneous Processors

Hower et al. [80] introduce heterogeneous race free (HRF) memory models that accommodate synchronization operations with different scopes, but HRF does not address the composition of different protocols with different constituent consistency models. Extending the definition of compound consistency models to accommodate scopes is future work.

Nagarajan et al. [1] briefly discuss heterogeneous consistency models in their primer (Section 10.2.1). They introduce the concept of a compound consistency model and the associated litmus tests informally but they do not formalize it; nor do they provide a general method for fusing two protocols.

In concurrent work, Iorga et al. [87] formally specify the memory model of heterogeneous CPU/FPGA systems axiomatically as well as operationally, and validate their models. In contrast, in this work we specify more generally how different memory models can be composed together, and how coherence protocols can be automatically fused to match our specification.

Batty [88], in his position paper, argues for a compositional approach towards relaxed memory consistency. Our compound consistency models, and specifically the fact that they preserve compiler mappings within each cluster, are a step in this direction.

8.3.3 Litmus Testing

Litmus testing is a longstanding approach to validating whether a system correctly implements its specified memory consistency model [74; 75]. Litmus tests are small code snippets that

[1] Although reasoning about an in-order pipeline is easiest, we do not require an in-order pipeline, just a pipeline that is compatible with the consistency model [83].

are crafted, perhaps with the aid of automation [89; 90], to expose behaviors that distinguish different consistency models [91]. Because no prior work has explored compound consistency models, there has also been no prior work in litmus test development for them.

8.3.4 Memory model translation

ArMOR [92] is a framework for specifying and translating between memory consistency models. For example, ArMOR has been used previously to automatically generate translation modules for dynamically translating code compiled for one memory model on hardware that enforces another. In this work, we leverage the ArMOR framework in a novel way: to compose different coherence protocols.

8.4 System Model, Assumptions, Limitations, and Problem Statement

Throughout this work, we assume a heterogeneous shared memory computer that consists of multiple clusters of cores, as illustrated in Figure 8.2. At a high level, HeteroGen takes as input the individual clusters (i.e., cluster1 and cluster2) with their cluster-specific coherence protocols, each enforcing its cluster-specific consistency model, and automatically produces a global protocol that enforces the compound consistency model. (We will define and discuss compound memory models in the next section.)

Without loss of generality, we assume each cluster contains a set of cores with local L1 caches and a shared L2; the L1s are kept coherent by a cluster-specific directory coherence protocol. By directory protocol, we mean a protocol that makes writes visible to other processors by sending a request to the directory. We support many different flavors of directory protocols. The protocol can be a conventional writer-initiated invalidation based protocol that enforces SWMR, as exemplified by the MOESI family of protocols commonly employed in CPUs. Or it can be a protocol that eschews SWMR and instead employs self-invalidations and write-backs to directly enforce the consistently model, as is commonly employed in GPUs. (Note that we do not support update-based protocols or protocols based on the notion of leases.)

We abstract this protocol heterogeneity by specifying the consistency model of each cluster's coherence interface through labels associated with read and write requests. (Recall that in Section 8.2.2 we defined the consistency model enforced by the coherence interface as the one that is enforced when the pipeline presents coherence requests in program order.)

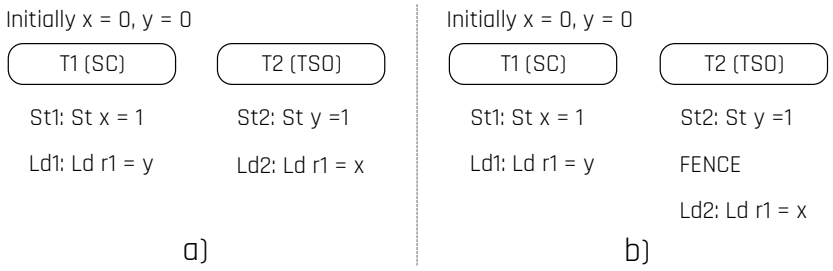


Figure 8.3: a) Ld1 and Ld2 can both return 0, b) only one of Ld1, Ld2 can return 0

For this work, we restrict ourselves to coherence protocols that enforce multi-copy atomic memory models; furthermore, we restrict ourselves to non-scoped memory models. These two restrictions are mainly due to the limitations of our compound consistency formalism rather than any limitation of HeteroGen per se.

We make no assumptions regarding the on-chip network, other than that each cluster connects to it. We focus here on single-chip implementations, but conceptually HeteroGen is not restricted to single chips.

With the system model fleshed out, we are now in a position to precisely describe the problem. Given a set of clusters, each with a distinct directory coherence protocol enforcing a distinct consistency model, how do we automatically merge the individual directory controllers into one global directory controller, as shown in Figure 8.2?

8.5 Compound Consistency Models

Implicit in the above problem statement is the question of correctness. Given that the coherence protocols of the different clusters could be different, and given that they could enforce distinct consistency models, what should be the correctness criterion of their composition? How does one program the resulting heterogeneous shared memory computer?

8.5.1 Intuition

In this paper, we propose a solution to these questions: a compositional approach to heterogeneous consistency called compound memory consistency models. We define compound consistency as follows. Consider a heterogeneous computer with n clusters, C_1 to C_n, each with its own per-cluster coherence protocol that enforces a per-cluster consistency model M_i. When we combine the clusters into a heterogeneous processor, the compound consistency model guarantees that operations from each cluster C_i continue to adhere to its per-cluster consistency model M_i.

To understand compound consistency better, assume that cluster C_1 supports SC and cluster C_2 supports TSO. Compound consistency mandates that operations from threads belonging to C_1 adhere to SC, while operations from threads belonging to C_2 adhere to TSO.

Consider the Dekker's litmus test shown in Figure 8.3(a), which shows thread T1 from the SC cluster and thread T2 from the TSO cluster. For this example, note that it is possible for both Ld1 and Ld2 to read zeroes. This is because the TSO cluster does not enforce the St2 $\rightarrow$ Ld2 ordering, even though the SC cluster enforces the St1 $\rightarrow$ Ld1 ordering.

However, as shown in Figure 8.3(b), once a FENCE instruction is inserted between St2 and Ld2, the two loads cannot both read zeroes anymore. Note, however, that a FENCE instruction is not required between St1 and Ld1 from T1 because the SC cluster already guarantees this ordering.

8.5.2 Formalism

In this section, we formalize the notion of compound consistency models. Starting with the axiomatic framework of Alglave et al. [93], which can capture any multi-copy atomic model [93], we then formally define the compound consistency model enforced when combining a set of given multi-copy atomic models.

Preliminaries. We start by defining some basic relations.

- $\xrightarrow{po}$ the program order relation, the per-thread total order that specifies the order in which memory operations appear in each thread.
- $\xrightarrow{po-addr}$ the program order relation on a per-address basis.

Consider an execution of a multi-threaded program on a shared-memory computer. Such an execution can be captured by the following communication relations:

- $\xrightarrow{ws}$ the write-serialization relation that relates two writes of the same address that are serialized in the order specified.
- $\xrightarrow{rf}$ the read-from relation which relates a write and read of the same address such that the read returns the value of the write.
- $\xrightarrow{rfe}$ the read-from-external relation which relates a write and read of the same address from two different threads, such that the read returns the value of the write.
- $\xrightarrow{fr}$ the derived from-read relation that relates a read r and a write w such that the read returns a value of some write that was serialized before w (in $\xrightarrow{ws}$).

An execution is said to be legal, if SC is satisfied on a per-address basis. That is:

$$acyclic(\xrightarrow{po-addr} U \xrightarrow{rf} U \xrightarrow{fr} U \xrightarrow{ws}) \tag{8.1}$$

Legality of execution is the axiom that ensures, among other things, that a read always reads the most recent write before it in program order. In the following, we consider only legal executions.

Multi-copy memory model. A multi-copy atomic memory model is specified in terms of the preserved-program order relation $\xrightarrow{ppo}$, that relates pairs of operations from any thread whose ordering is preserved in any execution.

Specifically, an execution is said to conform to a given memory model ($M \equiv \xrightarrow{ppo}$), if there exists a global memory order implied by the execution that is consistent with the preserved program order promised by the memory model. That is:

$$acyclic(\xrightarrow{ppo} U \xrightarrow{rfe} U \xrightarrow{fr} U \xrightarrow{ws}) \tag{8.2}$$

For example:

- SC $\xrightarrow{ppo} \triangleq \xrightarrow{po}$
- x86-TSO $\xrightarrow{ppo} \triangleq \xrightarrow{po} \backslash st(x) \xrightarrow{po} ld(y), \forall x, y$

Compound memory model. We axiomatically define the compound consistency model enforced by a heterogeneous computer with n clusters, C_1 to C_n, where each cluster adheres to its per-cluster multi-copy atomic memory model $M_i \equiv \xrightarrow{ppo_i}$.

Consider a multithreaded execution on this heterogeneous computer consisting of a set of threads T. We again characterize the execution using the communication relations we defined earlier ($\xrightarrow{ws}$, $\xrightarrow{rfe}$ and $\xrightarrow{fr}$). Note that we treat intra-cluster and inter-cluster communication relations identically.

Let us partition the threads into n subsets: $T_1, T_2 ... T_n$, such that all of the threads belonging to the set T_i are mapped to the processor cores belonging to cluster C_i. Let us define a new relation called $\xrightarrow{ppo_{com}}$ dubbed "preserved program order compound" which specifies the program order preserved for a given thread in the heterogeneous computer. Specifically, the preserved program order of a thread t is the same as the $\xrightarrow{ppo}$ of the memory model of the cluster in which the thread is mapped to:

$$\xrightarrow{ppo_{com}(t)} | \; t \in T_i \equiv \xrightarrow{ppo_i}$$

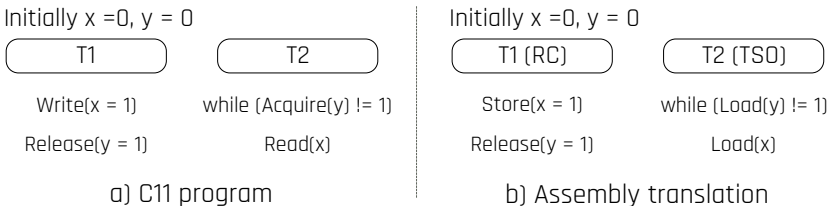


Figure 8.4: a) The producer-consumer pattern programmed in C11 for a heterogeneous system, b) The C11 program gets compiled for an RC/TSO system; in the RC system the C11 release gets compiled into a release, while in the TSO system the C11 acquire gets compiled into a load.

We now specify the compound consistency model as the one that preserves ppo_{com} as defined above. In other words, an execution is said to conform to the compound memory model if the global memory order implied by the execution is consistent with the preserved program orders of the threads belonging to each of the clusters.

$$acyclic(\xrightarrow{ppo_{com}} U \xrightarrow{rfe} U \xrightarrow{fr} U \xrightarrow{ws}) \tag{8.3}$$

8.5.3 Example

Let us go back to Figure 8.3(b) which shows the Dekker's litmus test, with thread T1 from the SC cluster and T2 from the TSO cluster. The following sequence of edges:

$$St1 \xrightarrow{ppo} Ld1 \xrightarrow{fr} St2 \xrightarrow{ppo} Ld2 \tag{8.4}$$

implies that Ld2 will read the value of St1, reading a 1. Note that the Ld1$\xrightarrow{fr}$St2 edge above relates two operations from different clusters; recall that the compound memory model treats intra-cluster and inter-cluster communication relations identically, and thus this edge is part of the global memory order.

8.5.4 Programming with Compound Consistency

How does one program with compound consistency models? Because the compound consistency model honors the memory orderings of the original model of each of the clusters, programmers/compilers need only be aware of the cluster to which a thread is mapped; when a thread is mapped to C_i the programmer can program that thread assuming that the memory model is M_i, that cluster's memory model. Note that if each of the clusters supports a distinct ISA, the programmer/compiler must already know which cluster each thread is mapped to for code generation.

We do not necessarily advocate for programmers to program against the low-level compound consistency model. In fact, we argue that compound consistency makes it easy to

support language-level consistency models on the heterogeneous computer. One of the key challenges in supporting a new hardware memory model is to discover correct compiler mappings from language-level atomics to that memory model. Fortunately, with compound consistency models there is no need to discover new mappings. When compiling language-level atomics down to the compound consistency model, depending on where (i.e., which cluster) a thread is mapped to, the existing compiler mappings for that cluster's memory model can be used.

We illustrate this with an example for a compound consistency model consisting of two models: Release Consistency (RC) and TSO. Let us consider a producer-consumer pattern expressed in a language-level consistency model such as C, as shown in Figure 8.4. Note that there are two language-level atomics here: the release on the producer side and an acquire on the consumer side. Further, let us assume that the producer thread is mapped to the RC cluster and the consumer is mapped to the TSO cluster. The producer thread uses the compiler mapping for a C11 release on RC (which is a release store) while the consumer thread uses the compiler mapping for a C11 acquire on TSO (which is a normal TSO load).

8.6 HeteroGen

In this section, we present HeteroGen, our scheme for automatically synthesizing heterogeneous protocols that satisfy compound consistency models.

8.6.1 HeteroGen Tool Flow

At a high level, HeteroGen performs the integration illustrated in Figure 8.2. Given two distinct directory coherence protocols, each of which enforces a potentially distinct consistency model, HeteroGen first produces a single atomic heterogeneous protocol as shown in Figure 8.5. In a second step the HeteroGen tool leverages ProtoGen to generate the concurrent heterogeneous protocol.

8.6.2 What does HeteroGen do?

To create the coherence protocol for the heterogeneous system model on the right side of Figure 8.2, HeteroGen merges the two directories into one single merged directory, while leaving the cache controllers unchanged. The merged heterogeneous directory presents a directory1-like interface to the caches of type cache1 and a directory2-like interface to the caches of type cache2. From the point of view of cluster1 (i.e., directory1 and its caches),

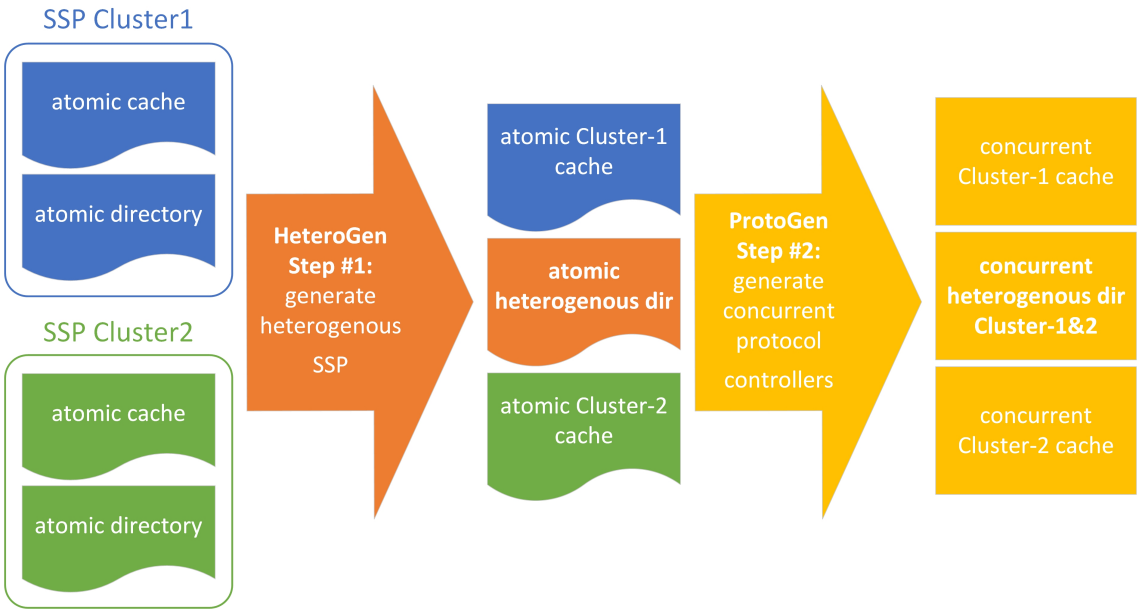


Figure 8.5: HeteroGen Tool Flow

cluster2 behaves as if it were a single cache1. Similarly, cluster2 views cluster1 as if it were a single cache2.

Within the merged directory there is bridging logic, such that a request from cache1 has the appropriate impact on caches of type cache2 (and vice versa). There are two logical aspects to bridging between the protocols: proxy caches [69] and consistency model translation [92]. We will explain in Section 8.6.4 how these work together. But before that we will explore what compound consistency means operationally.

8.6.3 Operational Intuition

HeteroGen is informed by the operational intuition behind compound consistency models.

One way to specify memory models is via abstract state machines that exhibit the memory model's behaviors. For example, SC can be expressed as a bunch of in-order processors connected via a switch to an atomic memory. If a FIFO store buffer [77] and/or a load buffer [94] is introduced between each processor and the memory, we get TSO. In general, any multi-copy atomic memory model can be expressed as processors with local buffers connected to atomic memory, with each memory model having its unique buffering logic [95].

Given the state machine representations of two memory models as described above, the compound model can be realized by merging the memory components into one, leaving the buffering logic untouched. This is the high-level insight that drives HeteroGen.

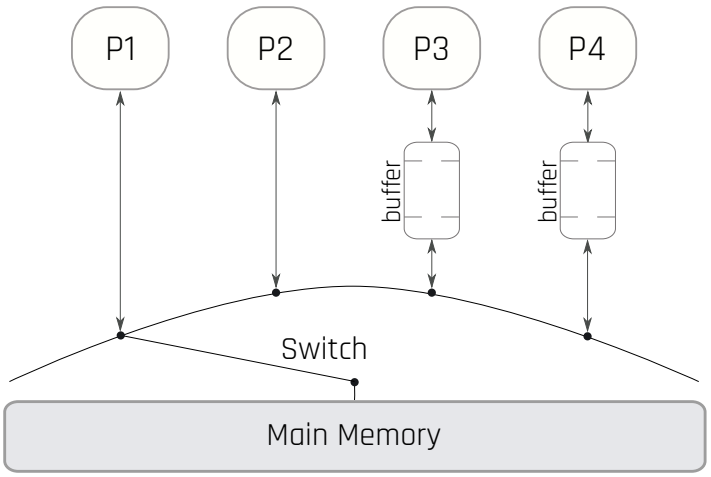


Figure 8.6: Operational intuition of combining SC and RC. P1 and P2 belong to the SC machine whereas P3 and P4 belong to the RC machine.

Example. Figure 8.6 illustrates the compound SC/RC machine obtained by fusing SC and RC. Because P1 and P2 are part of the original SC machine, they do not have any local buffers. Because P3 and P4 are part of the RC machine, they have local store buffers and load buffers. (Stores write to the local store buffer, which is flushed on a release. Loads are allowed to read potentially stale values from the local load buffer, which is invalidated upon an acquire.)

To understand how the SC/RC machine enforces compound consistency, consider the execution shown in Figure 8.7. Assume that initially P4 has a copy of data with a value of 0 in its local load buffer; flag and data have initial values of 0 in memory. At times t1 and t2, P1 writes 1 to data and flag, respectively. At t3, P4 reads the locally buffered value of 0. Note that a stale read of 0 does not violate the compound SC/RC consistency model, because P4 has not performed an acquire yet. At time t4, an acquire for flag at P4 reads 1 from memory and invalidates the local buffer, as mandated by RC. At t5, a load to data from P4 gets the up-to-date value of 1 from memory.

8.6.4 Refining the Intuition

Now we return to the original problem of merging two different coherence protocols (the "concrete problem"). Compare this problem against the more abstract version we introduced in Section 8.6.3 (dubbed post-hoc as the "abstract problem").

Whereas each input in the concrete problem is still a state machine that enforces a memory model, the state machine is more detailed, with caches and a directory coming into the mix. Each input of the concrete problem is thus a refinement of an input of the abstract problem. Naturally, we must ensure that the concrete problem's output, too, is a refinement of the

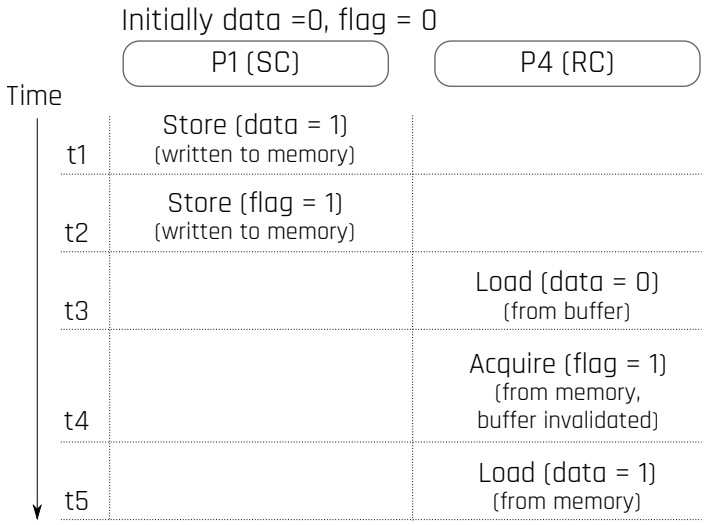


Figure 8.7: A legal execution on the compound SC/RC machine that adheres to the SC/RC compound memory consistency model. At time t4, an acquire from the RC machine sees flag, at t5 it correctly reads the up-to-date value of 1 from memory.

abstract problem's output. In other words, we must merge the directories such that the merging has the same operational effect as merging the memory components into one (but leaving the buffering components untouched).

In contrast to the abstract problem, where merging the memory components is conceptually simple, merging directories is not. This is because the directory is not just an interface to memory; each directory, in conjunction with the caches, implements a (distinct) coherence protocol. Fundamentally, a coherence protocol allows for cache lines to be obtained with read and/or write permissions. When a cache line obtains read permissions, it is essentially spawning a local replica of the global memory location. When a cache line obtains write permissions, it is essentially obtaining ownership of the global memory location. Thus, for every memory location, there are potentially multiple replicas of the location across both clusters. In fusing the directories, we must ensure that all of the memory replicas behave like there is just one copy. How can we ensure this "compound consistency invariant"?

Ensuring the Single-Writer-Multiple reader invariant—across all cached copies of a location, across both clusters—serves the purpose but is overkill. This is because not all cached copies of a location are globally visible. Some of them can be held without read/write permissions. (Recall that relaxed memory models allow for reads and writes to be buffered locally.) Thus, the compound consistency invariant need only apply to cache lines that are globally visible. Given a cached copy, how can we determine whether it is globally visible or whether it represents a buffered copy?

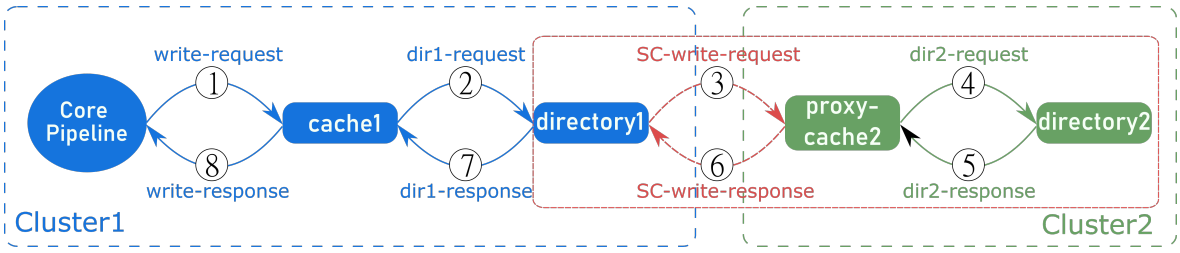


Figure 8.8: HeteroGen protocol flow for a write issued by the pipeline. Note the red box: directory1, proxy-cache2 and directory2 are one merged directory.

HeteroGen ensures the compound consistency invariant as follows. Whenever a write is made globally visible in one of the clusters (say cluster1), HeteroGen makes the write globally visible in cluster2 as well. Crucially, HeteroGen does this with the help of cluster2's coherence protocol, and in doing so, offloads the problem of distinguishing between buffered versus globally visible cache lines to the coherence protocol itself. Because cluster2's coherence protocol enforces its consistency model correctly, it must intrinsically distinguish between these lines anyway.

In order to propagate writes between the two clusters, HeteroGen must automatically synthesize the bridging logic. Specifically, when a write is made globally visible in cluster1, HeteroGen must automatically identify and trigger the exact request in directory2's specification for making that write globally visible within cluster2. HeteroGen does this with two mechanisms: consistency model translation and proxy caches.

First, HeteroGen identifies the access sequence in cluster2's consistency model[2] for an SC-equivalent store using ArMOR [92]. For example, the equivalent of an SC store in RC would be a release. Why an SC-equivalent store? Because that is guaranteed to trigger a write request that propagates globally before the write's completion.

Second, HeteroGen consults cluster2's cache specification and identifies the sequence of coherence requests that would be triggered for the SC-equivalent access sequence. For example, in the lazy RC coherence protocol [36], a release would trigger an ownership request for that cache line, and HeteroGen introduces a proxy cache to issue that request to the directory. Logically, the proxy cache is a clone of a cluster2 cache controller that HeteroGen leverages for issuing the above request transparently. (Logically, there is one proxy cache per cluster.) In reality proxy caches are part of the merged directory that HeteroGen generates, and a cluster's (say cluster1) "proxy cache" represents the transient states that bridge the protocol flows from cluster2 to cluster1.

[2]The consistency model (Section 8.2.2) enforced by cluster-2's coherence interface.

To summarize, as shown in Figure 8.8, when a write is made globally visible in cluster1—i.e., when directory1 receives a write permissions request or a writeback request—HeteroGen propagates that write by translating it into an appropriate request (with the help of ArMOR) and then issuing that request in cluster-2 via its proxy cache. Once the request has completed, the proxy cache evicts the line, marking the location as invalid in cluster2. Then, directory1 resumes by completing the original write request within cluster1.

A future load to that location from cluster2 will contact directory2 and find that the block is invalid in cluster2. At this point, HeteroGen has cluster1's proxy cache take over and trigger an SC-equivalent read from directory1. Once the value comes back, the proxy cache evicts the line and relinquishes control to directory2, which completes the original read request.

Example. To understand how the HeteroGen-fused directories enforce compound consistency, let us consider the execution shown in Figure 8.9 on a heterogeneous machine consisting of an RC and an SC cluster. Processors P1 and P2 belong to the SC cluster, which runs a conventional writer-initiated MSI protocol. Processors P3 and P4 belong to the RC cluster. The RC cluster runs a simple RC protocol that buffers writes in the local cache, writes back data upon a release, and self-invalidates the local cache on an acquire. Initially P1 and P4 have local copies of flag and data with a value of 0.

At time t1, P4 performs a store to data and its value is locally updated to 1. At time t2, P4 performs a release to flag. The release initiates a writeback of dirty lines in the cache, causing data to be written back at time t3. The dirty write back of data is propagated to the SC cluster; HeteroGen discovers that this is a request corresponding to a store and translates it into its SC-equivalent in the MSI world – which happens to be a store. By looking at the cache controller of the MSI protocol, HeteroGen discovers that a store will lead to a request for write permissions in the MSI world, and has the proxy cache issue the same to the SC-directory; the SC-directory sends invalidations to sharers and ends up invalidating P1's local copy of data. We show the state transitions at the combined directory controller in Figure 8.10.) At time t4, a similar sequence of events leads to the local copy of flag being invalidated at P1.

At time t5, P1 loads flag; because flag is not available in the SC cluster, HeteroGen translates the load into an SC-equivalent load in the RC world – which is an acquire. By looking at the cache controller of the RC protocol, HeteroGen discovers that an acquire leads to a read request in the RC world and has the proxy cache issue the same to the RC-directory; the RC-directory returns the up-to-date value of 1, which is evicted to the LLC by the proxy cache; at this point the SC-directory resumes the request and reads 1 from the LLC to complete the load to flag. At time t6, a similar sequence of events leads data = 1 to be read.

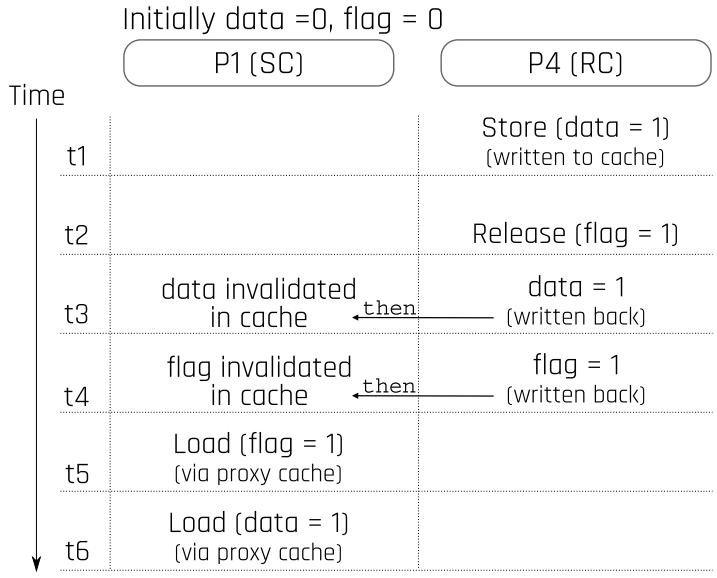


Figure 8.9: A legal execution on the merged MSI(SC)/RC protocol that adheres to the SC/RC compound memory consistency model. At time t5 a load of flag at P1 reads 1 and at time t6 it correctly reads 1 for data.

8.6.5 Implementation details

8.6.5.1 Identifying globally visible writes

Recall that the merged directory synthesized by HeteroGen ensures that, when a cluster makes a write globally visible, the write is propagated to the other cluster as well. But how does HeteroGen identify when a cluster is making a write globally visible?

One observation is that a globally visible write has to necessarily inform the directory – either for obtaining write permissions, or for performing a write back or a write through.

Write back and write through requests are easy to identify: such requests are the only ones that are sent with values that are written to the shared cache.

So the challenge lies in identifying write permission requests. HeteroGen identifies such requests by statically analyzing the cache controller. Specifically, for each request from the cache to the directory, HeteroGen inspects the final state (s1) of the cache line after the final response. If both of the following conditions are satisfied, the original request to the directory is classified as a globally visible write: (a) state s1 allows for stores to hit without external communication (possibly transitioning to a new state s2); and (b) either s1 or s2 accepts forwarded requests that lead to a data value response from the cache.

The first condition is self-explanatory. The second condition checks whether the value written can potentially become globally visible. For example, consider an RC protocol with write-back caches that buffers writes locally. On a store, if the cache line is invalid, the

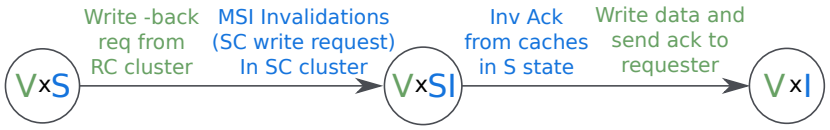


Figure 8.10: State transitions at the combined directory controller. At time t2, at the combined directory controller, data is in V(alid) state in the RC cluster and in S(hared) state in the SC cluster. This is represented as state VxS at the combined directory controller. At time t3, when the write-back request of data reaches the directory, the proxy cache (which is actually part of the combined directory) propagates the write in the SC cluster, forwarding invalidations to all caches in the SC cluster that are caching data in state S, and enters a transient state denoted as VxSI. Once the proxy cache receives invalidation acknowledgments, the original write-back request is handled: data is written, and an acknowledgment is sent back to the cache that initiated the write-back of data. In the end, data is in V state in the RC cluster and I(nvalid) state in the SC cluster, denoted VxI.

protocol requests the line from the lower level and goes to valid. Although this is really a read request, the final state (valid) allows for writes because RC allows for writes before a release to be be buffered. This illustrates why the second condition is required: the fact that there cannot be any forwarded requests for a cache line in valid state ensures HeteroGen does not mis-classify the original request as a write.

For another example, consider the exclusive (E) state of the classic MESI protocol. Although the exclusive state does not allow for any forwarded request that results in a value response, it can silently transition to the modified state, which both permits stores to hit and accepts forwarded requests that lead to a data response.

8.6.5.2 Proxy Cache Concurrency

How concurrent can the proxy cache be? In this section, we present two options for proxy cache design: (a) a conservative, but more general, processor-centric design, where requests from cluster1 are serialized at cluster2's proxy cache; (b) an aggressive, but limited, memory-centric design that permits requests to different locations to be overlapped. But first we motivate the trade-off with an example.

Consider two clusters: cluster1 enforcing RC, and cluster2 enforcing SC. Let's say that processor P1 from cluster1 performs a release to address X that reaches directory1, and needs to be propagated to the SC cluster. Accordingly, cluster2's proxy cache issues an appropriate request (say reqA) to directory2 and is waiting for a response. Meanwhile, let's say there is another release from P1 to address Y that reaches directory1, and also needs to be

propagated to the SC cluster, with this second release coming after the first in P1's program order. Question: can the proxy cache issue reqB (the second request) concurrently with reqA?

To answer this question, let us first ask ourselves why the two releases were issued by P1 concurrently in the first place. Typically, the processor pipeline orders stores (including releases) by issuing them in order, waiting for completion of the first before issuing the next. (Aggressive implementations [96] allow for reordering but the onus is on the pipeline to achieve the same effect.) Under this pipeline ordering assumption, the only way in which the releases could reach directory1 (and hence proxy cache) concurrently is if the caches responded to the pipeline even before the release became globally visible. Consider an RC protocol that simply writes through every store to the lower level. Supposing the interconnect were totally ordered, the caches can simply stream releases without waiting for acknowledgments from the directory, relying on the interconnect to enforce the release $\rightarrow$ release ordering.

Conservative processor-centric design. Under this more general assumption (that permits caches to provide early responses as above), the proxy cache must be serializing. This serialization is not as bad as it sounds, though, for two reasons. First, the proxy cache can still allow requests from different processors to overlap. Second, we can leverage the serialization of the proxy cache, such that we can ask ArMOR to avoid generating SC-equivalent accesses; instead, we ask it to generate one or more accesses in cluster2 that match the ordering guarantees of the original access, and use this to uncover concurrency in the proxy cache.

Aggressive memory-centric design. Assuming that caches do not provide early completion responses (i.e., assuming each globally visible write needs to be acknowledged by the directory), it is safe for the proxy cache to allow requests to different addresses to overlap. (It needs to only order requests to the same address.) Under this assumption, the proxy cache functions akin to a conventional coherence controller, allowing for inter-address concurrency.

We have implemented both of these designs. HeteroGen analyzes the cache state machine: if it allows for early write acknowledgments, we use the conservative design. Else, we use the aggressive design.

8.6.5.3 Handling arbitrary number of clusters

Thus far, we have assumed that HeteroGen takes two protocols as inputs, and this is purely to simplify the discussion. HeteroGen can naturally handle an arbitrary number of protocols. The only change is that, upon a globally visible write to one cluster, the write has to be propagated to all other clusters, which entails issuing requests to each of the other clusters

via a proxy cache. On a read that cannot be served locally within a cluster, a read request is issued to the cluster that wrote to that location most recently.

8.6.5.4 Handling word-granularity writes

Thus far, we have assumed that each of the input protocols operates at cache line granularity. However, it makes for accelerators such as GPUs to support byte-granular writes [6]. HeteroGen can handle such protocols in a general way by leveraging that these protocols would have bitmasks that mask which values are written to memory. If a bit masked write is performed it must be communicated to the remote clusters, in case the cache is not the owner of the masked cache block section. If the remote clusters support bit masked writes of the same granularity a direct translation is possible, otherwise, the write is issued for the entire cache line.

For example, consider fusing an MSI cache-line-granular protocol with an RC protocol that performs word-granular writes. Assume that a cache in the RC cluster has a dirty word. Also assume that in the MSI side, a cache has the same cache-line in M state. When the word is written back from the RC cache, the writeback is translated into a write by the proxy cache in the MSI side. This write causes the MSI cache to forward its dirty copy to the proxy cache, which then immediately writes-back to the global LLC. Finally, the proxy cache communicates to the RC-directory that the writeback of the word can take place in the global LLC.

8.6.5.5 Pipeline-Coherence interaction

Thus far we have assumed that a pipeline interacts with the coherence protocol using a simple interface, where the pipeline issues reads and writes to the coherence subsystem in accordance with that cluster's memory model. In high-performance implementations, however, the pipeline can interact with the coherence protocol via a richer interface: e.g., in speculative load replay [96] the pipeline might issue reads out of order, relying on the coherence protocol to flag memory ordering violations.

HeteroGen continues to enforce the compound consistency model correctly in the presence of such non-trivial pipeline-coherence interactions. Specifically, when HeteroGen fuses two clusters, the second cluster (with its pipeline and coherence protocol) will not affect the first cluster's pipeline-coherence interaction. This is because HeteroGen ensures that from the perspective of one cluster, the other cluster—including its coherence protocol and pipeline—appears like one of its children (and vice versa). In other words, clusters interact with one another in a structured fashion in which each cluster's pipeline and coherence protocol are

accessed atomically. For example, when a write is propagated from one cluster to another, the other cluster's coherence protocol (and, optionally, pipeline, in case the coherence protocol forwards invalidates to the pipeline) is accessed. This structured interaction ensures that each cluster's pipeline-coherence interaction remains unaffected.

8.6.5.6 Summary

We now summarize the steps involved in combining multiple directory controllers (of each cluster) into one heterogeneous directory controller. The combined directory controller maintains auxiliary states (metadata) for each block address: the owner field, which maintains the identity of the last writer (cluster) to that address.

- **Analyze input protocols.** HeteroGen first analyzes each of the input directory controllers to identify those requests that are globally visible write requests (as explained in Section 8.6.5.1). Further, it also analyzes each of the input cache controllers to identify whether any write request to the cache is acknowledged early; if even one of the writes in one of the input protocols is acknowledged early, HeteroGen uses a conservative processor-centric approach (explained in the following under "concurrency").

- **Writes.** Consider each globally-performing write request to any of the input directory controllers. Before handling the request within that cluster, HeteroGen first propagates that write within other clusters. This is accomplished as follows. First, the ArMOR framework [92] is employed to identify the corresponding write request in each of the other clusters. Then, these requests are initiated in parallel, and once all of these are performed, the original write request is handled. Once this is done, the cluster that performed the write is set as the owner of the block address.

- **Reads.** For each read request that cannot be serviced within a cluster, HeteroGen handles that read request as if it was initiated by the current owner cluster of that cache block.

- **Concurrency.** While handling a read or a write request, requests to that address from any processor are blocked. Additionally, in case of conservative processor-centric approach, requests from the processor that initiated the original read or write are also blocked.

Algorithm 10 HeteroGen controller composition

```
1: ∀ request ∈ dir-CI.Requests, ∀ dir-CI-state ∈ dir-CI.States
2:     ▷ Compute access that generated request
3:     access = compute-access(request, SSP-cache-CI);
4:     ▷ Determine if access is globally visible
5:     if !check-access-global(access) then
6:         ▷ Local accesses do not need to be performed by proxy caches in remote clusters
7:         continue;
8:     end if
9:     ▷ Consume the request and make a temporary copy to avoid head of line blocking
10:        tmp-request = consume-request(request);
11:    if access == write then
12:        RemoteProxyBlocks = [];
13:        ∀ CR ∈ Cluster \ CI, ∀ dir-CR-state ∈ dir-CR.States
14:            ▷ Convert the instruction sequence into graphs
15:            RemoteProxyBlocks += REMOTEPROXYDIR(access, CI, CR, dir-CR-state))
16:            ▷ Generate all possible permutations for the remote proxy actions blocks to
                 facilitate parallel writes
17:            PARALLELWRITEPERMUTATION(RemoteProxyBlocks)
18:            ▷ Update Owner Field
19:            owner = CI;
20:    else
21:        ▷ If cluster CI is not owner, proxy cache in owner cluster must perform load
22:        if CI != owner then ∀ dir-CR-state ∈ dir-owner.States
23:                ▷ Read from owner
24:                REMOTEPROXYDIR(access, CI, owner, dir-CR-state)
25:        end if
26:    end if
27:    ▷ Now respond to the initial request in the initiator cache cluster-CI
28:        dir-CI.send-fwd-request-response(tmp-request, dir-CI-state);
29:        dir-CI.await-response(tmp-request, dir-CI-state);
30:        dir-CI.update-state(tmp-request, dir-CI-state);
```

```
1:  procedure REMOTEPROXYDIR(access-CI, CI: Initiator Cluster, CR: Remote Cluster,
            dir-CR-state: Remote cluster directory state)
2:      ▷ Leverage ArMOR to translate access from CI → CR
3:      if isMemoryCentric(SSP-CI) then
4:          Accesses = compute-access-ARMOR(access-CI, CI → CR, SC-equivalent = true)
5:      else
6:          Accesses = compute-access-ARMOR(access-CI, CI → CR, SC-equivalent = false)
7:      end if
8:      ▷ Determine initial proxy state of SSP-CR
9:      proxy-CR-state = compute-invalid-cache-state(SSP-CR);
10:     ▷ Proxy-Cache performs sequence of accesses determined by ArMOR
11:     ∀ access ∈ Accesses
12:         ▷ The proxy-CR-state is update with every access performed
13:         request = compute-request(access, proxy-CR-state, SSP-CR);
14:         dir-CR.send-fwd-request-response(request, dir-CR-state);
15:         dir-CR.await-response(request, dir-CR-state);
16:         dir-CR.update-state(request, dir-CR-state);
17:         ▷ The virtual proxy cache waits for response
18:         proxy-cache-CR.await-response(access, proxy-CR-state);
19:         ▷ Then the proxy cache updates its state
20:         proxy-CR-state = cache-CR.update-state(access, proxy-CR-state);
21:     ▷ The proxy cache must now evict the block, but this is again an internal request
            and needn't be sent. We simply compute the request it would generate so that
            dir-CR can respond to this virtual eviction request
22:         evict-request = compute_request(evict, proxy-CR-state, SSP-CR);
23:         dir-CR.update-state(evict-request, dir-CR-state);
24: end procedure
```

Analog to the HieraGen algorithm presented in 7.4.3.3, we present the HeteroGen algorithm 10 following the same notation. HeteroGen is essentially a variation of the HieraGen method exploiting proxy caches to communicate between different controllers. HeteroGen translates accesses associated with certain requests between two directories, while HieraGen translates between a higher-level cache and a lower-level directory. By adopting the translation

modules generated by ArMOR, the HieraGen algorithms can be used to generate hierarchical heterogeneous protocol implementations as briefly discussed in section 7.6.6.

8.6.5.7 Handling Read-Modify-Writes

Read-modify-writes (RMW) can be handled for coherence protocols that take ownership of a cache block as well as for coherence protocols that use write backs to make writes visible to memory. For both variants it is only relevant, that the write to memory like in case of a store access is made globally visible in all clusters to enforce coherence and that the data has not been changed after the read.

8.6.6 Using HeteroGen

8.6.6.1 HeteroGen-compatible Protocols

We have confirmed that HeteroGen works for a wide variety of protocols, encompassing protocols that satisfy SWMR as well as those that are targeted to relaxed consistency models (see Section 8.7.1 and Table 8.1). HeteroGen cannot fuse *any* two protocols, however. Some protocols are incompatible in important ways that make it hard to compose them automatically and efficiently.

For example, HeteroGen cannot fuse an invalidation based protocol with an update based protocol because the notion of write permissions is not compatible with update protocols. (A cache block with write permissions can safely write without communicating with the directory in the former, but an update based protocol is based on all writes being propagated.)

For another example, HeteroGen cannot fuse Tardis [97] (or Relativistic Coherence [98], G-TSC [99]) with a conventional invalidation based protocol. This is because the notion of read permissions is not directly compatible with leases. Read permissions allow for a block to be held potentially indefinitely, whereas leases expire.

Given two protocols, can we tell whether the two are compatible? We believe this is a challenging problem that is beyond the scope of this work. However, the fact that HeteroGen-generated protocols are automatically validated mitigates the risk that a HeteroGen user employs it for incompatible protocols.

8.6.6.2 Consistency Models of Input Protocols

For HeteroGen to select the appropriate ArMOR translations at the merged directory, it must know the consistency models of the input protocols. In theory, we could ask the user to

precisely specify these consistency models, but architects do not often reason about protocols in that way; instead, they tend to reason about consistency as a function of both the protocol and core pipeline. To avoid relying on the user, HeteroGen uses extensive litmus testing of each input protocol to infer its consistency model.

8.7 Case Studies and Validation

To explore HeteroGen and the protocols it creates, we used it to generate a wide range of heterogeneous protocols, and we validated them.

8.7.1 Case Studies

We took a set of homogeneous protocols, and we used HeteroGen to generate heterogenous protocols from various combinations of these constituent protocols.

In Table 8.1, we list the seven homogeneous protocols that we consider. These protocols include two MOESI variants that support SC, and five protocols that are designed for weaker consistency models.

RCC [1] is a simple protocol that enforces RC by: buffering writes in the cache; writing back the cache contents on a release; and self-invalidating the cache on an acquire. RCC-O [36; 1] is a block-granular variant of DeNovo [6] that obtains ownership on all writes. GPU is a simple GPU protocol as specified in Spandex [19], where stores write through to the shared cache. GPU, RCC-O, and RCC enforce RC. PLO-CC is a variant of RCC-O without a release, and it enforces a memory model called partial-load-order [92] that enforces the W$\rightarrow$W and the R$\rightarrow$W orderings but not the other two. TSO-CC [27] is a protocol tailored to enforce TSO; we model the basic version of the protocol without timestamps. These protocols represent a wide range of protocols, highlighting the generality of HeteroGen.

In Table 8.2, we show the pairs of protocols that we composed with HeteroGen. As we explain later, we validated that all of these generated protocols satisfy their compound consistency models and are deadlock-free.

8.7.2 Heterogeneous Litmus Testing

One way of validating a coherence protocol against a consistency model is via litmus testing. Each litmus test is designed to expose a behavior that, if observed, reveals a violation of the consistency model. Existing litmus tests validate that a single homogeneous protocol obeys a (non-compound) consistency model.

	Protocol
SC	MSI, MESI
TSO	TSO-CC [27]
RC	RCC-O [36; 1] , RCC [1], GPU [19]
PLO	PLO-CC

Table 8.1: Protocols used in the case studies

Case-study				States/ Transitions
1	MSI	&	MESI	25/171
2	MESI	&	TSO-CC	17/88
3	MESI	&	PLO-CC	17/88
4	MESI	&	RCC-O	27/117
5	MESI	&	RCC	23/109
6	MESI	&	GPU	23/101
7	RCC-O	&	RCC	12/43
8	RCC	&	RCC	3/16

Table 8.2: Case Studies with their respective Heterogen Directory States and Transitions

To validate that HeteroGen's generated protocols satisfy their compound consistency models, we generated heterogeneous litmus tests. Starting with the version of the litmus test for the weaker of the two consistency models, we use consistency model translation [92] to remove any synchronization operations (e.g., Fences) that are not needed for the stronger consistency model.

We used the herd7 tool [93] to generate 111 litmus tests, including commonly used tests like MP, S, IRIW, 2+2W, CoRR, LB, R, RWC, SB, WRC, WRW+WR, WRW+2W, and WWC [95]. For each litmus test, we consider all possible allocations of threads to processor cores.

To perform the validation of every heterogeneous protocol generated by HeteroGen, we used the Murϕ model checker [26]. In Murϕ, we preload the caches with the initial values in the litmus test, and we ensure that loads and stores are executed based on the litmus test, while permitting evictions at any time. Murϕ then exhaustively explores every possible ordering of the events in the litmus test, and it reveals whether the protocol permits an outcome prohibited

by the consistency model. In all of our litmus tests, the HeteroGen-generated protocols were successful.

8.7.3 Validating Deadlock Freedom

We also used Murϕ to validate that every protocol—both the constituent protocols and the generated protocols—are deadlock-free. This validation is an exhaustive search of the reachable state space for a system with two addresses, and systems with 1–3 caches per cluster. To avoid the state space explosion problem for systems with more than one cache per cluster, we use state space hashing and run the model checker until the probability of omitting a state is less than 0.05%.

8.8 Protocol Performance

We have already demonstrated that HeteroGen achieves its primary purpose of automatically generating protocols. Although there is no fundamental reason why the generated protocols cannot achieve comparable performance, one might worry that the particulars of HeteroGen could hurt performance.

The most relevant comparison is Spandex, which recall is an interface for manually integrating multiple different protocols. For our baseline we consider HCC [76], which is a publicly available protocol that is similar to Spandex [19]; we focus on the heterogeneous protocol obtained by manually combining the DeNovo protocol with MESI. For comparison we use the automatically-generated RCC-O/MESI protocol generated by HeteroGen. (Recall that RCC-O is a block-granular variant of DeNovo.)

We simulated the protocols on gem5, and our simulation parameters are identical to those used in HCC [76]. We simulated a 64-core system with 2 clusters: 60 “tiny” cores and 4 “big cores”. The big cluster employs the MESI protocol whereas the tiny cluster employs DeNovo. The two clusters are manually fused using HCC, whereas in our setup MESI and RCC-O are automatically fused using HeteroGen. The detailed simulation parameters are shown in Table 8.3. We used the same 13 applications with fine-grained synchronization as in HCC.

Our results are summarized in Figure 8.11, and they reveal that HCC and HeteroGen have comparable runtimes, with HeteroGen performing similarly to the manually-generated HCC on average. The important point of difference in the two protocols is the use of conservative handshaking messages in the manually-generating protocol, whereas HeteroGen eschews these redundant handshakes. The effect of the reduced handshaking messages translates into faster reads: specifically, when a core reads the value written by another core it incurs

significantly reduced latencies. The effect of faster communicating reads translates into performance for two benchmark programs (nq and lu) which spend a significant time on such reads.

Most writes are, as perhaps expected, slower with handshaking. However, in the presence of a burst of writes and false sharing, both of which occur in these benchmarks, handshaking slows down the transfer of a block between cores. Indeed, with handshaking, a core can perform multiple writes before losing the block to another core, which turns out to be more efficient. Thus, the absence of handshaking is a reason why some of the benchmarks, such as bf and bfsbc, see a small performance hit. To confirm our hypothesis we experimented with a variant that actually performs handshaking on the writes but not the reads. (Note that HeteroGen can generate variants with handshaking.) This variant of HeteroGen consistently outperforms the baseline, and by 2% on average, vindicating our hypothesis.

We also measured the network traffic incurred by both variants of HeteroGen in comparison with HCC, and our results indicate that traffic incurred is within 5% of HCC on average. Our takeaway is that the HeteroGen-generated protocols appear to have similar performance and network traffic to a manually-generated heterogeneous protocol.

Table 8.3: Simulated System Parameters [76]

Big Cores Cluster 1	RISC-V ISA (RV64GC), 4-way out-of-order, 16-entry LSQ, 128 Physical Reg. 128-entry ROB. L1 cache: 1-cycle, 2-way, 64KB L1I and 64KB L1D, hardware-based coherence
Tiny Cores Cluster 2	RISC-V ISA (RV64GC), single-issue, in-order, single-cycle execute for non-memory inst. L1 cache: 1-cycle, 2-way, 4KB L1I and 4KB L1D, software-centric coherence
L2 Cache	Shared, 8-way, 8 banks, 512KB per bank, one bank per mesh column, support heterogeneous cache coherence
Interconnect	Network-on-Chip, 8×8 mesh topology, XY routing, 16B per flit, 1-cycle channel latency, 1-cycle router latency, buffer size 8 flit
Main Memory	8 DRAM controllers per chip, one per mesh column. 16GB/s total bandwidth

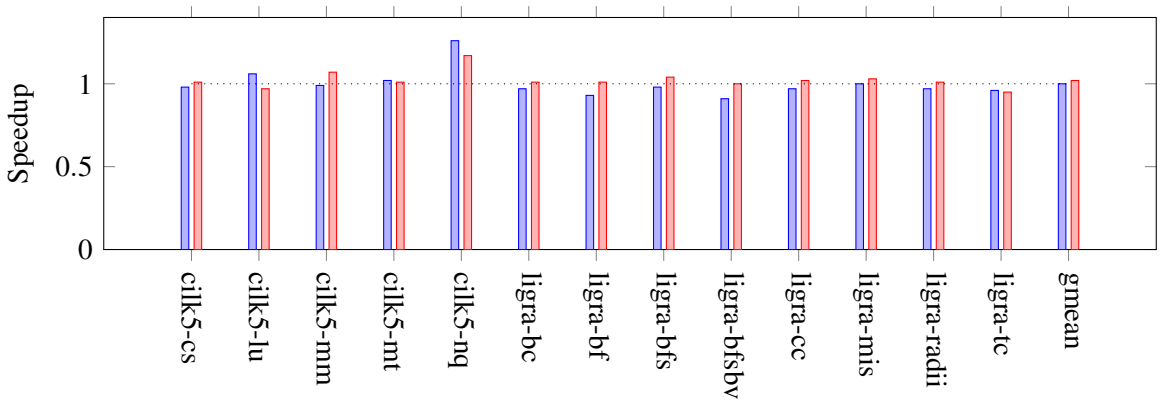


Figure 8.11: Speedup of HeteroGen over HCC [76]: without handshaking (blue) and with handshaking (red)

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This thesis explored the automatic generation of highly-concurrent hierarchical and heterogeneous cache coherence protocols from atomic input specifications. The algorithms and methods presented cover a wide variety of different directory cache coherence protocols and different memory consistency models. The tools developed allow architects to design correct coherence protocols for arbitrary cache hierarchies and network topologies with a small number of constraints. The feedback from academia [100] and industry alike show that the *Gen tools are already getting adopted.

9.1.1 ProtoGen

We have developed ProtoGen to help architects design directory cache coherence protocols. ProtoGen simplifies the design process by requiring the architect to specify only the stable state protocol with atomic transactions. At its heart, ProtoGen is a method for refining an atomic specification into a non-atomic implementation. In contrast with other general techniques, ProtoGen exploits domain knowledge about how directory protocols work—most importantly, the fact that coherence transactions serialize at the directory—to enable it to generate highly-concurrent non-blocking protocols for ordered and unordered networks alike.

We have shown, for a variety of protocols, that ProtoGen successfully generates the finite state machines for cache and directory controllers. Furthermore, ProtoGen has generated protocols with at least as much concurrency as those found in existing protocols, suggesting that generated protocols need not sacrifice performance compared to manually designed protocols.

9.1.2 HieraGen

We have presented HieraGen, a new tool for automatic generation of hierarchical cache coherence protocols. We have demonstrated that HieraGen can successfully compose atomic, flat SSPs into a highly concurrent hierarchical protocol. The generated protocols are verifiably correct and avoid the substantial design and verification effort—cache and directory controllers with dozens of states and hundreds of transitions—required by manual design. We believe that design automation of hierarchical protocols is both practical and preferable to manual design.

9.1.3 HeteroGen

HeteroGen can automatically compose multiple coherence protocols, including MOESI variants and other protocols that are targeted towards specific consistency models. The resulting heterogeneous protocol satisfies the precisely defined compound consistency model that can be inferred from the consistency models enforced by the constituent protocols. We validated HeteroGen with newly developed litmus tests.

As the computer architecture community—both in academia and industry—continues the trend towards heterogeneity, we hope that HeteroGen and clear compound consistency models can greatly reduce design time and increase confidence in the design.

9.2 Critical Analysis

In hindsight we review the decisions taken during the course of this study. While the *Gen algorithms nicely build on each other according to the sequence of their development, this is unfortunately not the case from a software engineering perspective. Developing the *Gen tools was challenging because at the start of each project we only had an idea about the functionality that we would be required to implement, but not a clear specification.

When we started to work on the development of algorithms for the automatic generation of coherence protocols we only thought about ProtoGen and mainly ‘MOESI’ coherence protocols. Therefore we chose the Python programming language to develop a - at that time seemingly small and simple - tool that would allow us to share our algorithm with the community and make it usable to other researchers.

The ProtoGen tool had a clear code structure using object orientation and inheritance, but this changed when we had to adapt ProtoGen to support HieraGen. The atomic hierarchical coherence controllers were - apart from being both cache and directory controllers at once -

significantly more complex than the atomic coherence controllers we previously provided to ProtoGen as an input. While the ProtoGen algorithm itself remains correct our previous ProtoGen tool implementation was not directly usable. While we were able to modify the existing ProtoGen implementation for the HieraGen tool we published, the code quality significantly decreased making it difficult to maintain and extend the HieraGen tool.

Due to the experience we gained from HieraGen we decided to start the development of the HeteroGen tool largely from scratch. In addition to implementing the HeteroGen algorithm, we introduced a common data structure for the intermediate representation of the coherence protocols that is used throughout the entire tool. We re-implemented the Parser, Backend and ProtoGen algorithm for the HeteroGen tool now considering composed controllers and adding support for events required by consistency-oriented protocols. However, the HeteroGen tool does not implement the HieraGen algorithm.

The HieraGen and HeteroGen tool are publicly available as two stand alone tools with limited compatibility. Automatically generating a heterogeneous and hierarchical cache coherence protocol from given input SSPs is not possible with these tools without manual intervention.

Therefore, we are currently working on the ProtoGen+ tool. The ProtoGen+ tool encompasses the latest implementations of all *Gen algorithms developed. In addition, it integrates a variety of Backends like Rumur and SLICC that have been developed using different - often beta - *Gen tool versions. Like the previously *Gen tool we plan to make the ProtoGen+ tool publicly available. To bring all algorithms developed together

Finally we want to give an impression of the complexity of *Gen tool development. Developing the *Gen tools is challenging requiring a large number of unit tests, because they are effectively compilers allowing the user describe arbitrary coherence protocol specifications with only few constraints. In Table 9.1 the number of python code lines for the different *Gen tool versions are given. Although the number of code lines is not a good metric for the complexity of code it still provides a rough idea about the challenges faced when developing and maintaining the *Gen tools.

9.3 Future Work

The work on the *Gen tools is ongoing post the submission of this thesis and a short summary of each of the future work topics is presented below.

*Gen Tool	Lines of Python Code	ProtoGen Protocol Language Version	Algorithm Implementation Version		
			ProtoGen	HieraGen	HeteroGen
ProtoGen	8849	v1	v1	-	-
HieraGen	15771	v1	v1.1	v1	-
HeteroGen	30413	v2	v2	-	v1
ProtoGen+	45000+	v2.1	v2.1	v2	v1.1

Table 9.1: *Gen tool versions lines of Python code

9.3.1 ProtoGen: Automatic Virtual Channel Assignment

While the ProtoGen tool can automatically generate highly-concurrent coherence protocols from atomic specifications, ProtoGen does not automatically assign messages to virtual networks yet for ensuring deadlock freedom. We are currently working on adding this feature.

9.3.2 HieraGen: Non-Inclusive Shared Caches

The automatic generation of non-inclusive cache hierarchies is not supported by HieraGen yet. This is the case because HieraGen does not modify the SSPs when composing them. Since non-inclusive cache hierarchies are commonly deployed [100; 101], we are currently working on overcoming this restriction by developing new rules that will allow HieraGen to modify given input SSPs to generate the transactions required by a non-inclusive cache hierarchy protocol.

9.3.3 HeteroGen: Scoped and Non-Multi-Copy-Atomic Models

We have developed a generalized operational memory model called the compound memory model that allows us to formalize the composition of wide variety of memory models including scoped and non-multi-copy-atomic memory models [102]. Leveraging the compound memory model we plan to extend the range of coherence protocols that can be generated by HeteroGen.

9.3.4 Snooping Protocols

The *Gen tools do not support snooping protocols yet. However, we believe the algorithms and methods presented can be transferred to handle this outstanding challenge. One of key insights of ProtoGen is that the directory is the point of serialization for transactions. This insight can be extended to the bus in the snooping protocol acting as the point of serialization

for all transactions. A cache can infer whether its own transaction has been serialized, by observing the request initiating this transaction on the bus.

9.3.5 Timestamp Protocols

We have not looked exhaustively into timestamp protocols yet. However, we have successfully used ProtoGen in the past to automatically generate the flat timestamp Linearization protocol [103] from an atomic specification. For generating hierarchical and heterogeneous timestamp protocol cache hierarchies, we expect the methods presented in this thesis to remain applicable if updates to timestamps visible to all caches within a cluster also become visible to all remote caches in different clusters that have a copy of the data. However, we must perform an exhaustive analysis of existing time stamp protocols to better understand potential limitations of the *Gen methods.

9.3.6 Replication Protocols

We have used the *Gen tools in the past to generate replication protocols automatically from input SSP specifications [104]. The replication mechanism however, was part of the input SSP specification. Future work may explore whether rules can be defined that allow to automatically convert a non-replication SSP specification into a replication protocol.

9.3.7 Formal Proof

While the *Gen algorithms generate correct coherence protocols, we rely on model checkers to confirm their correctness. To avoid the necessity of exhaustive model checking, future work may focus on formally proving the correctness of the *Gen algorithms.

Bibliography

[1] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[3] Linley Gwennap. Raptor Lake Adds Little Cores. In *Microprocessor Report*, 2022.

[4] Alan Smith and Norman James. AMD Instinct™ MI200 Series Accelerator and Node Architectures. HOT CHIPS, 2022.

[5] The AMBA CHI Specification. `https://developer.arm.com/architectures/system-architectures/amba/amba-5`. Accessed: 15th July 2019.

[6] 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. *PACT*, 2011.

[7] Jesse G. Beu, Michael C. Rosier, and Thomas M. Conte. Manager-client Pairing: A Framework for Implementing Coherence Hierarchies. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–236, 2011.

[8] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, 2000.

[9] Erik Hagersten and Michael Koster. Wildfire: A Scalable Path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, 1999.

[10] Edya Ladan-Mozes and Charles E. Leiserson. A Consistency Architecture for Hierarchical Shared Caches. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, pages 11–22, 2008.

[11] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[12] Michael R. Marty and Mark D. Hill. Virtual Hierarchies to Support Server Consolidation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 46–56, 2007.

[13] Opeoluwa Matthews and Daniel J. Sorin. Architecting Hierarchical Coherence Protocols for Push-button Parametric Verification. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 477–489, 2017.

[14] 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.

[15] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[16] Opeoluwa Matthews, Jesse Bingham, and Daniel J. Sorin. Verifiable Hierarchical Protocols with Network Invariants on Parametric Systems. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, pages 101–108, 2016.

[17] Thomas N. Theis and H.-S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

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

[19] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 261–274, 2018.

[20] Lena E. Olson, Mark D. Hill, and David A. Wood. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 163–176, 2017.

[21] The OpenCAPI Consortium. `https://opencapi.org/`. Accessed: 21st January 2019.

[22] The GenZ Consortium. `https://genzconsortium.org/`. Accessed: 21st January 2019.

[23] Craig Williams, Paul F. Reynolds Jr., and Bronis R. de Supinski. Delta Coherence Protocols. *IEEE Concurrency 2000*, 2016.

[24] 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*, DSL'97, page 5, USA, 1997. USENIX Association.

[25] Coherency was broken and manually disabled in Galaxy S4. `https://www.anandtech.com/show/7164/samsung-exynos-5-octa-5420-switches-back-to-arm-gpu`. Accessed: 2022-05-05.

[26] David L. Dill. The Murφ Verification System. In *International Conference on Computer Aided Verification (CAV)*, pages 390–393, 1996.

[27] 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.

[28] M.M.K. Martin, D.J. Sorin, M.D. Hill, and D.A. Wood. Bandwidth adaptive snooping. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 251–262, 2002.

[29] 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.

[30] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[31] Vijay Nagarajan. Parallel Architectures. `https://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture04-multi.pdf`, 2017.

[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] SiFive TileLink Specification. `https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf`. Accessed: 01st January 2022.

[36] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. Lazy Release Consistency for GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[37] 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.

[38] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo

Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+. *CoRR*, abs/2007.03152, 2020.

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

[40] 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), 2005.

[41] The principal programming paradigms. `https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng101.pdf`. Accessed: 01st January 2022.

[42] Theo Olausson. Generating Gem5 Cache Coherence Controllers with ProtoGen. *4th Year Project Report*, 2021.

[43] James Arnold. Automatically Mapping Coherence Protocols to the TileLink Network. *Honours Project*, 2022.

[44] L. Ivanov and R. Nunna. Modeling and verification of cache coherence protocols. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 5, pages 129–132 vol. 5, 2001.

[45] Fong Pong and Michel Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Comput. Surv.*, 29(1):82–126, mar 1997.

[46] Xiaofang Chen, Steven M. German, and Ganesh Gopalakrishnan. Transaction Based Modeling and Verification of Hardware Protocols. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 53–61, 2007.

[47] BluespecTM SystemVerilog Reference Guide. 2014.

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

[49] Jørgen Staunstrup and Mark R. Greenstreet. From High-Level Descriptions to VLSI Circuits. *BIT*, 28(3), 1988.

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

[51] 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.

[52] 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.

[53] 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.

[54] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving Efficient Cache Coherence Protocols Through Refinement. *Form. Methods Syst. Des.*, 20(1):107–125, jan 2002.

[55] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[56] 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.

[57] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 287–296, New York, NY, USA, 2013. Association for Computing Machinery.

[58] 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)*, 2015.

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

[60] Marco Elver, Christopher J. Banks, Paul Jackson, and Vijay Nagarajan. VerC3: A library for explicit state synthesis of concurrent systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1381–1386, 2018.

[61] Alberto Ros and Stefanos Kaxiras. Complexity-effective Multicore Coherence. In *PACT*, New York, NY, USA, 2012. ACM.

[62] 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.

[63] 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.

[64] Reece Carr. Developing a Library of Verified Cache Coherence Protocols. *4th Year Project Report*, 2021.

[65] 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.

[66] Nick Barrow-Williams, Christian Fensch, and Simon W. Moore. A communication characterisation of Splash-2 and Parsec. In *Proceedings of the 2009 IEEE International*

Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA, pages 86–97. IEEE Computer Society, 2009.

[67] Ulrich Stern and David L Dill. Improved Probabilistic Verification by Hash Compaction. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224, 1995.

[68] Henry Cook. *Productive Design of Extensible On-Chip Memory Hierarchies*. PhD thesis, University of California, Berkeley, 2016.

[69] Nicolai Oswald, Vijay Nagarajan, and Daniel J. Sorin. HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols. In *Procedings of the 47th Annual International Symposium on Computer Architecture*, 2020.

[70] Joonwon Choi, Adam Chlipala, and Arvind. Hemiola: A DSL and Verification Tools to Guide Design and Proof of Hierarchical Cache-Coherence Protocols. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification*, pages 317–339, Cham, 2022. Springer International Publishing.

[71] HSA Foundation. Heterogeneous System Architecture: A Technical Review, 2012.

[72] The CCIX Consortium. `https://www.ccixconsortium.com/`. Accessed: 21st January 2019.

[73] Compute Express Link. `https://www.computeexpresslink.org/`. Accessed: 18th June 2021.

[74] Richard L. Sites. *Alpha Architecture Reference Manual*. Prentical Hall, 1992.

[75] David L. Weaver and Tom Germond. *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994.

[76] Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pages 173–186. IEEE, 2020.

[77] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[78] ARM Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, 10 2018. Initial v8.4 EAC release.

[79] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The RISC-V Instruction Set Manual, 2014.

[80] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free Memory Models. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[81] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 257–270. ACM, 2019.

[82] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 647–659, New York, NY, USA, 2015. Association for Computing Machinery.

[83] D. Lustig, M. Pellauer, and M. Martonosi. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[84] Alberto Ros and Stefanos Kaxiras. Racer: TSO consistency via race detection. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 33:1–33:13. IEEE Computer Society, 2016.

[85] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. Building Heterogeneous Unified Virtual Memories (UVMs) without the Overhead. *ACM Trans. Archit. Code Optim.*, 13(1):1:1–1:22, 2016.

[86] Alberto Ros, Mahdad Davari, and Stefanos Kaxiras. Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies. In *21st IEEE International Symposium on High Performance Computer Architecture,*

HPCA 2015, Burlingame, CA, USA, February 7-11, 2015, pages 186–197. IEEE Computer Society, 2015.

[87] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. The semantics of shared memory in Intel CPU/FPGA systems. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021.

[88] Mark Batty. Compositional relaxed concurrency. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104), September 2017.

[89] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in Weak Memory Models. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, 2010.

[90] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated Synthesis of Comprehensive Memory Model Litmust Test Suites. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[91] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Generating Litmus Tests for Contrasting Memory Consistency Models. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, 2010.

[92] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the International Symposium on Computer Architecture*, 2015.

[93] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats. *ACM TOPLAS*, 36(2), jul 2014.

[94] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. The Benefits of Duality in Verifying Concurrent Programs under TSO. *27th International Conference on Concurrency Theory (CONCUR 2016)*, 59(5), 2016.

[95] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018.

[96] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume I: Architecture/Hardware*, pages 355–364. CRC Press, 1991.

[97] X. Yu and S. Devadas. Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation*, 2015.

[98] X. Ren and M. Lis. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 625–636, 2017.

[99] Abdulaziz Tabbakh, Xuehai Qian, and Murali Annavaram. G-TSC: Timestamp Based Coherence for GPUs. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 403–415. IEEE Computer Society, 2018.

[100] Jennifer Brana, Brian C Schwedock, Yatin A Manerkar, and Nathan Beckmann. Kobold: Simplified Cache Coherence for Cache-Attached Accelerators.

[101] Cheng-Chieh Huang, Rakesh Kumar, Marco Elver, Boris Grot, and Vijay Nagarajan. C3d: Mitigating the numa bottleneck via coherent dram caches. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[102] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. Compound Memory Models. In *44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2023.

[103] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. Scale-out CcNUMA: Exploiting Skew with Strongly Consistent Caching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[104] Adarsh Patil, Vijay Nagarajan, Rajeev Balasubramonian, and Nicolai Oswald. Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 526–539, 2021.