SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of Heterogeneous Cache Coherence Protocols for CXL

Julian Pritzi

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Formal Verification of Heterogeneous Cache Coherence Protocols for CXL

Formale Verifizierung heterogener Cache-Kohärenz-Protokolle für CXL

Author:	Julian Pritzi
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Dr. Anatole Lefort
Submission Date:	16.10.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 16.10.2024 Julian Pritzi

Acknowledgments

This thesis would not have been possible without the help and encouragement I received from the people around me.

First and foremost I would like to express my sincere gratitude to my supervisor, Pramod Bhatotia, for his valuable support and guidance in my academic life thus far. I sincerely appreciate his dedication to my academic development.

I also want to give special thanks to my advisor, Anatole Lefort, whose feedback and expertise were essential in shaping the final version of this thesis.

My gratitude extends to all the other researchers and staff at the chair for welcoming and supporting me and my work. I am also especially grateful to Denis Hoornaert, for providing me with motivation and different perspectives during our joint research work.

A very big thanks goes to my family, whose unconditional support I can always count on, their ability to get me through difficult times has been invaluable in the past. I also want to thank my friends for being by my side and making my studies more enjoyable, especially Jonas and Clemens who I know I can always count on.

Abstract

Compute Express Link (CXL) is an emerging technology which is designed to provide a unified interconnect for different architectures. It allows connected devices to operate on shared resources as if they were part of their local architecture. However connecting devices which are part of different architectures poses some challenges.

One particular such challenge is how to compose the potential different cache coherence domain, when connecting different architectures under CXL. There is existing work that creates bridges between cache coherence domains, however, their formal verification is lacking. Especially in the context of CXL, where there can be a composition of large heterogeneous subsystems, these verification methods reach their limits.

To this end we propose a compositional model checking approach, that allows to verify the heterogeneous cache coherence protocols of larger heterogeneous systems.

We demonstrate the inadequacy of the model checking employed by previous work, when applied to larger systems, which would be common when integrating CXL. By doing this we find a potential deadlock in previously published related work, as well as a number of inefficiencies in their model checking approach.

As an alternative we present our compositional approach, which allows to decompose the full system at the bridges. Model checking is then performed on the individual clusters. We formally justify this decomposition by showing equivalence for the communication through the bridges in the full system and the decomposed one.

Using this compositional approach, we are able to greatly reduce the required Random-Access Memory (RAM) for performing a liveness check of the full system. We achieve a smaller asymptotic growth, depending on the system complexity, compared to a full system verification. For our target system, which connects two clusters over CXL we are able to reduce the RAM required for the liveness analysis from 268 GB to 7 GB, which is a 97% reduction in resource usage.

Contents

1 Introduction

Modern workloads have increasingly higher memory requirements to achieve scaling. Typical examples are AI training and large scale simulations, which require resources exceeding the capabilities of a single compute node. The classical approach of allowing the workloads to scale by building distributed systems is not suitable for all needs. For one it is not always possible to achieve scaling, especially when working with accelerators, that have great communication overheads. Handling this communication manually and ensuring consistency proves to be challenging on it's own. Even if handled properly, not all workloads can be parallelized to the same degree. Hence it can be beneficial to have a large uniform memory resource. To that end CXL is set to be a promising new technology by providing a unified interface to connect a variety of different devices.

One particular challenge for CXL that is yet unsolved, is how to handle the interoperability of heterogeneous cache coherence protocols when connecting devices from different device architectures. Even though CXL proposes it's own cache coherence protocols, this only amplifies the challenge, by adding more heterogeneity that needs to be addressed in pursuit of a unified interface. We anticipate the development of dedicated hardware with the purpose of translating the cluster specific requests to the CXL domain. Those components will have to bridge between different cache coherence domains. It is crucial to ensure their correctness using formal verification, such that proper operation of the system can be guaranteed.

However when looking at the existing work in bridging cache coherence domains, we believe their verification methods do not scale to CXL enabled systems. Previous work used model checkers for verification, which are tools that explore all possible states of a system, based on a model. Any increase of the system size, generally causes an exponential increase in the potential states of the system. This phenomenon is called state space explosion and brings with it an equally exponential increase in resources required to explore all this states.

The size and complexity of the systems, that previous work is able to analyze, is limited due to these steep resource requirements. Their formal verification is restricted to systems containing a single bridge. Those restricted results are used to argue about the correctness of the larger, full systems. However in this thesis we show that the properties of the restricted models do not transfer to the full system. We are able to show, that one of the bridges, that is proven to be deadlock free by the related work, causes a deadlock, when instantiated multiple times in a full system.

We conclude from this that the full system needs to be taken into account for formal verification. However, this raises the question of how to achieve a feasible system verification, which provides the same level of confidence as a full system analysis, but with less resources?

This thesis proposes a compositional model checking approach to make full system analysis feasible. This is achieved in three essential steps, which are also the contribution of this thesis: (1) the decomposition into manageable subsystems (2) subsystem abstractions, which reduce *internal* states (3) equivalence checks, which can formally justify the substitution of subsystems with abstractions.

For this we decompose the full system of many core-clusters connected over CXL, into smaller subsystems that are individually analyzed. This allows to show the desired properties for each subsystems individually. To ensure that the analysis of these individual subsystems also takes the communication with outer subsystems into account, we use abstractions to represent the interface of the subsystems. These abstractions are chosen to be less resource intensive than the full subsystems they abstract.

We provide four general approaches, that explain how such abstractions can be obtained from full system models, by utilizing different intuitive properties to eliminate state space. To formally justify the substitution of full subsystems with abstractions, we propose an equivalence check, which ensures the subsystem and abstraction behave equivalently. We realize the equivalence check using a custom model for the model checker, which captures the behaviour of one subsystem and verifies whether a different subsystem can reproduce it.

We apply our methods to the liveness verification of a target system, that exceeds the limits of what is feasible to analyze using the model of our related work. We show flaws in the model of the related work, propose fixes and achieve a full system verification, without using any compositional methods, that requires 268 GB. This makes it possible to completely analyze it on our machine, that is equipped with 2 TB of RAM, which were insufficient for the unmodified model of our related work. By using a compositional approach, we are able to greatly reduce the required RAM further from 268 GB to 7 GB. This 97% reduction in resource usage makes it feasible to verify liveness of the system on a ordinary laptop instead of requiring a dedicated server.

In the following chapters we will present our work in detail. Chapter 2 provides background on the fundamental concepts that are necessary to understand the content of this thesis. It also contains an overview of the related work, which similarly looks at bridging different cache coherence domains and servers as a baseline for our work to be compared against.

Chapter 3 gives a more detailed motivation for need for a compositional model checking approach. It explains our target system as well as the limitations and challenges that motivate this work.

Chapter 4 provides an overview of the compositional verification method presented in this thesis, as well as, the overview of it's application to the liveness verification of our target system.

Chapter 5 explains the design and the formal reasoning behind the individual components of this work, including the abstractions and equivalence checks used in this thesis.

Chapter 6 explains how the models and checks are implemented using our model checkers, it also covers how we translate our models so that they are suited for both model checkers. In the evaluation in

Chapter 7 we show the benefits of using compositional verification as well as potential trade-offs. We precise how we compare to the related work as a baseline and what issues we had to fix for the related work to be usable.

We conclude this thesis with Chapter 8, where we summarize the findings of this thesis as well as provide a list of potential areas for future work.

2 Background and Related Work

The first half of this section covers the required background on cache coherence, CXL and model checking. The second half explores the related work on bridging between cache coherence domains.

2.1 Cache Coherence

In modern platforms caches are crucial components, which increase performance by reducing memory access times, by storing copies of the data closer to the processor. In the context of multi-core systems this can lead to the same data being stored in different areas of the architecture. Because caches are transparent to the processor, data has to be synchronized across all sharers. In practice this is achieved through cache coherence protocols. The set of components which utilize a shared protocol to maintain cache coherence is called cache coherence domain.

The Cache Controller (CC) inside a cache coherence domain maintains protocol specific metadata about the cache lines they store, in order to properly operate in accordance with the cache coherence protocol. For many systems metadata consists of a set of stable states that the cache line can be in, when it it not currently part of a transaction, and a set of transient states that are used for the transitions between the stable states. The stable states of modern coherence protocols commonly use a subset of the five states in the MOESI protocol. [12] The five stable states have a one-to-one mapping to the letters of the protocols, which reflect the following associated meaning:

- The **I** state stands for ’invalid’, it marks cache lines that are either missing or outdated. Cache lines in I state can not be read or written. If a read or write needs to be performed, the CC is required to utilize the cache coherence protocol to transition into a state with support for read an writes.
- The **M** state stands for ’modified’, it marks cache lines that are exclusively owned and can be read and written to. In this state the cache has the only valid copy of the data in the whole coherence domain. The cache is therefore required to respond to incoming requests for the cache line.

- The **S** state stands for 'shared', it marks cache lines that contain a read-only copy of data. In this state the cache line is not owned and other caches may have their own read-only copy.
- The optional **E** state stands for 'exclusive', it marks cache lines that are an exclusive read-only copy of the data.
- The optional **O** state stands for 'owned', it marks cache lines that are a non-exclusive read-only copy of the data. Compared to the S state, the caches in O state are required to respond to incoming requests for the cache line similar to the M state.

There exist two common mechanisms to ensuring caches are coherent: (1) Through the use of snooping based protocols (2) Through the use of directory based protocols.

Snooping based protocols monitor the bus transactions and modify their cache line states in accordance to the transactions they observe. This can be faster than directory based protocols for smaller systems, as the messages are broadcasted to all CC, reducing the number of required messages. However this also requires appropriate hardware support to allow for this broadcasting. The result is that snooping based protocols protocols scale poorly and are not in use for larger non-uniform memory access (NUMA) systems, which CXL targets. [13]

The prevalent coherence mechanism in contemporary architectures is the directory based cache coherence protocols. [14] The key to these protocols is the introduction of a directory entity, which maintains a global view of the cache coherence domain. The coherence protocol is based on point to point communication. If a CC wants to perform a request in the coherence domain, it sends the request to the directory. The directory then orchestrates the coherence protocol and sends messages based on the state of the coherence domain to process the request. As an example, the directory might instruct the current owner of a cache line to forward it to the requester.

Messages of the cache coherence protocol, are sent over one of three channels forward (fwd), response (resp) and request (req). The channels have to be served in this order whenever possible.

This thesis contains a number of examples and explanations which for simplicity only use the required MSI subset of the MOESI states. To understand the communication flows that are presented in this paper Table 2.1 lists the relevant subset of messages and their description, which are used in this thesis.

Message	Channel	Recipient	Description
getS/getM	req	directory	requests S/M state for a cache line
putS/putM	req	directory	relinquishes S/M state for a cache line
Inv	fwd	cache	request inavlidation of a cache line, also specifies who to send the Inv_Ack to
Fwd_*x*	fwd	cache	used to forward request *x* from the directory to a cache
*x*_Ack	fwd/resp	cache	acknowledges that *x* was processed, e.g. *x*=putS
*x*_AckD	resp	cache	same as *x*_Ack, with additional data
*x*_AckAD	resp	cache	same as *x*_AckD, but contains a number of Inv_Acks that have to be received before proceeding

Table 2.1: Overview of the MSI messages, which are relevant to this thesis.

2.1.1 CXL

CXL is an emerging technology, which is designed to provide a unified interconnect for different architectures. It allows connected devices to operate on shared resources as if they were part of their local architecture. CXL addresses cache coherency, for the connected devices, to enable the devices to share a coherent memory space. The coherence protocol used by CXL is similar to the MESI family of protocols, which serves as the motivation for the selection of our Second Layer (L2) protocol in the target system.

2.2 Formal Verification using Model Checking

2.2.1 Formal Verification

It is difficult to develop software or hardware that behaves as expected. There are many real world examples demonstrating, that even big established projects and companies with extensive resources will produce bugs in both hardware and software.

This is universally recognized as a problem and many efforts exist to combat it from different directions. Testing, fuzzing, auditing and formal verification are all strategies to increase the confidence in the correctness of an implementation. The right approach varies depending on the criticality of the project and the available resources. For non-critical applications it might be sufficient to do some light testing. For other applications, that have the potential to harm lives, a more extensive evaluation might be desired.

One method which, if applied correctly, can greatly reduce the risk of using a system is formal analysis. Formal analysis attempts to reason about the system from a mathematical background. The goal is to proof, based on logical and mathematical reasoning, a series of properties of the system. These properties can then be used to make an informed decision about the systems risk.

2.2.2 Model Checking

Within the field of formal verification there are numerous frameworks for analyzing software and hardware systems. Model checking involves creating a mathematical model of the system and then automatically verifying its compliance to a specification/set of properties using a model checker.

There are different possibilities how a system could be modeled, with the most common approaches using state machines. The system is expressed as a state or a compound of states and a set of transitions that describe the relation between those states.

There are different characteristics of model checkers working with such models, such as deterministic vs probabilistic or exhaustive vs symbolic. For this thesis we are talking about deterministic exhaustive state space model checkers. These provide a binary result for properties, true if the property holds for the model or false otherwise. In most cases if the result is false, a counter example is also provided, which shows how the model can violate the property. A model checker with exhaustive state space exploration, explores all possible states of the model and records them in a big table. Newly discovered states are added to the table, inserted in a queue for exploration and then explored. The model checker finishes once either all new states are already part of the table or when reaching a state in which one of the desired properties is violated.

If the model is a correct representation of the real system and the verification using the model checker succeeds, then the same properties should apply to the real system. However, creating a model, which correctly represents the real system is a challenge in itself. If there is any discrepancy between the model and the real system, this can impact the analysis significantly and generally invalidates the result.

There are many potential model checkers to chose from, so many, that there is a yearly model checker competition where different model checking tools compete for the best performance [7, 6]. Next we will cover two tools, the first of which (Murφ) is common in the memory consistency and cache coherence field. Rumur, the second tool we cover, is a modern model checker that tries to improve on Murφ, while maintaining some level of compatibility.

2.2.3 Murφ

The Murφ model checker [2] allows to specify an operational model in which a set of conditional ’rules’ describe state transitions that operate on a system state made up of variables. The Murφ compiler parses a description of the model and generates source code for an exhaustive state space exploration binary for that model. This code can then be compiled and executed to perform the exploration, any potential violations of the specification are then reported. If the binary finishes the exploration without finding any errors the formal verification is complete.

Murφ has it’s own domain specific language for specifying the model. It uses variables to represent the state of the system, which can take the form of various custom defined types. The Murφ type system supports booleans, integer ranges, enums, multisets and scalarsets, furthermore it allows to compose these primitives using records, arrays or unions to create more sophisticated types.

Rules are used to specify the state transitions of the model. They consist of a set of conditions that have to hold for the rule to be able to fire and an instruction block that defines how the state variables are modified when the rule fires. To facilitate expressing the necessary rule conditions and variable modifications, Murφ provides functions and procedures, which enable parameterized access and modification of variables.

2.2.4 Rumur

Rumur [11] is an alternative model checker, that shares most of it’s feature set with Murφ. Rumur uses the same domain specific language as Murφ to allow the specification of an operational model. It also produces code from this description, which can be used to compile a binary that explores the model’s state space.

Rumur has two noticeable advantages over Murφ: (i) it realizes multi-threading to perform more efficient state space exploration (ii) it supports the liveness assertion, which allows to specify a partial system state, which has to be reachable by all states in the system.

However, Rumur does not support the full feature set of Murφ, two features, which are relevant for this thesis in particular, are missing: (i) unions, which are a construct to combine types, by creating an or relationship. E.g. a union of two enums allows the values to be of either of those two enums. (ii) multiset types, which define a multiset collection, which can contain any number of one of the supported types of to a maximum set size.

2.3 Bridging Cache Coherence Domains

This section covers work related to the bridging of cache coherence domains. Including HIERA-Gen, which generates hierarchical protocols, HETERO-Gen, which generates a combined directory of two clusters and the compound memory consistency model, which describes the resulting model of fusing different memory consistency models.

2.3.1 HIERA Gen

HIERA-Gen [9] bridges between cache coherence domains, by automatically generating hierarchical protocols from the stable state protocol (SSP) for each level. To achieve this hiera merges the lower level directory with a higher level cache. The transactions reaching the directory of the lower level are then mapped and encapsulated in the coherence transactions of the higher level and vice-versa. This encapsulation is performed, without altering the SSP of the higher or lower level. To be able to achieve this, HIERA-Gen is limited to protocols enforcing the single writer multiple reader (SWMR) invariant.

To achieve proper encapsulation, the combined directory-cache uses the SSP to determine the access type of transactions. Based on this and the SWMR property it is able to determine, if the transaction needs to be propagated to the other layer. On the higher level this is achieved by making a request of the same access type. To be able to make requests on the lower level, on behalf of a transaction originating from the higher level, HIERA-Gen introduces a proxy cache. This proxy cache is integrated with the directory-cache and is part of the lower level. It generates requests on the lower level to bring it into a state, in which the directory-cache can proceed with the processing of a higher level request.

Murφ is used to verify that the generated protocols never violate coherence and do not deadlock. The system that is modeled for analysis consists of one First Layer (L1) and one L2 cluster connected by the directory-cache bridge. The L2 cluster contains an additional CC, while the L1 cluster contains two CC. The authors note that they are able to verify this system on a server with 256GB of memory, but fail to run a larger system with three L1 CC on the same server. To solve this problem they use a probabilistic approach using state hashing, to achieve result with sufficiently high confidence.

2.3.2 HETERO Gen

HETERO-Gen [10] bridges between cache coherence domains, by automatically generating a combined heterogeneous protocol from the SSP of two clusters. This combined protocol keeps the CC functionality identical and merges the two directories into one. The merged directory has a different interface for the caches underneath it and represents the whole other cluster as if it were a single cache.

Compared to HIERA-Gen, this is not restricted to protocols enforcing the SWMR invariant. Instead the merged directory in HETERO-Gen, reacts to stores which are made globally visible in that cluster, by translating between the consistency models of the clusters using ArMOR [8] to perform an equivalent request in the other cluster. This request is performed in the target cluster using a proxy cache, which is a per-cluster cache, dedicated for requests originating from other clusters.

HETERO-Gen also uses Murφ to verify that the generated protocols do not deadlock and adhere to compound memory consistency. The system that is modeled for analysis consists of two clusters connected by the merged directory, each cluster containing 1 to 3 CC. Deadlock freedom is shown using by modeling the system with two addresses. The authors note that they use a probabilistic approach using state hashing for the models, which have more than one CC per cluster, to tackle the state space explosion.

Compound Memory Consistency

The compound memory consistency model [5] defines the memory model of the combination of heterogeneous machines. The model is based on the POP model [3], which expresses the memory model as the propagation of read and writes between threads. The POP model is extended using the LOST principle, which stands for 'Local Orderings by Stalling on Threads'. In the model the memory subsystem is modelled as threads, and the memory consistency model is expressed as a set of rules that define how the threads operate and propagate read and write accesses. The authors argue that this model is general enough to encompass a variety of different memory consistency models.

3 Motivation

With this thesis we want to provide a suitable framework for the verification of heterogeneous systems integrating CXL. In order to connect these heterogeneous clusters to a CXL layer, it will be necessary to develop and verify bridges, dedicated hardware components capable of inter-operating between two different coherence domains.

To provide more detail about the motivation of this thesis, we start by explaining our target system, which is a more precise specification of the system we want to verify, in Figure 3.1. We will elaborate on the challenges of verifying the correctness of the cache coherence protocol for this system in Section 3.2. We conclude by elaborating why the state of the art of our related work does not address these challenges sufficiently in Section 3.3.

3.1 Target System Model

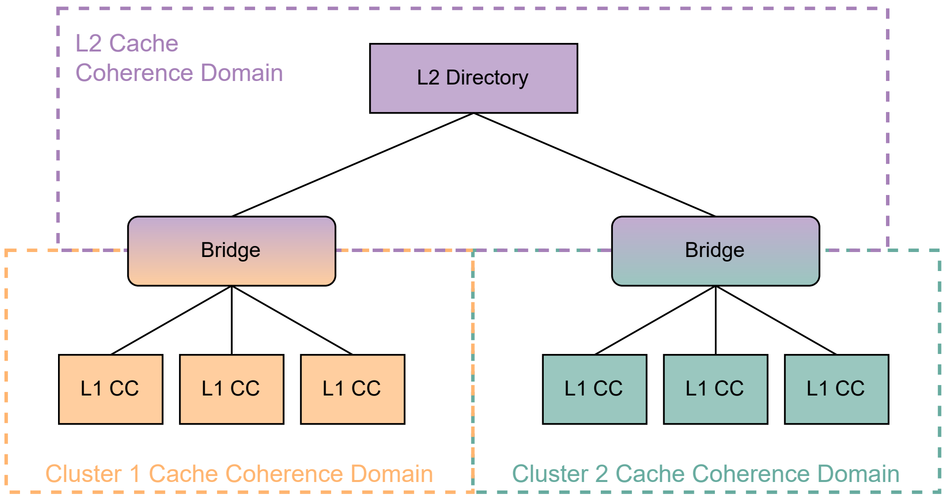


Figure 3.1: An overview of the target system.

In general we target a setup where an arbitrary number of heterogeneous clusters, each with their own cache coherence protocol, are connected using a CXL layer. Such a system can be arbitrarily large and varied in the combinations of protocols, with the CXL layer serving as a common ground.

To restrict the scope and evaluation of this thesis, we specifically target a system with 2 L1 clusters, each with 3 caches. For the evaluation of the cache coherence protocol, the system is sufficiently well defined by the coherence network and their immediately connected components. A simple representation of the target system restricted to those components is depicted in Figure 3.1. The selection of the number of L1 caches is motivated by the operation of the cache coherence protocols, where certain message sequences are only observed when there are at least 3 caches:

An example of this is the scenario depicted in Figure 3.2. One cache has shared (read) access to a cache line. The other two caches are now requesting write access, this results in one of them (i.e. the second cache in the figure), receiving an `AckAD`. This cache now has to await all the invalidation acknowledgements, during this the directory might process the 3rd caches request and sends a message to the second one to forward the cache line to the 3rd once all invalidations are received. Step 3 in Figure 3.2 is crucial, because in the model with a perfect network, the only way to have both orderings (3.1 before 3.2 and vice versa) is only possible with 3 caches. Hence the motivation for our minimal target system, which we analyze for correctness in this thesis.

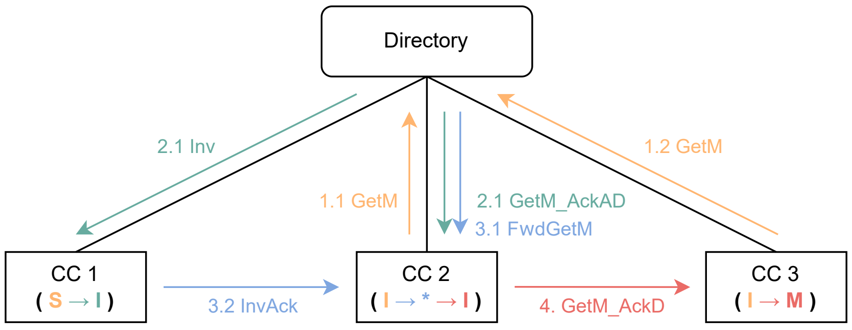


Figure 3.2: An example flow of cache line forwarding, requiring at least 3 caches for all potential orderings of step 3.

3.2 Challenge of Verifying Protocol Correctness

As already discussed in Chapter 2, developing new systems inherently comes with a risk of designing a flawed of malfunctioning system. Determining whether enough steps have been taken to reduce the risk of using a newly designed system/component to an acceptable level is a subjective decision. However we believe it is reasonable to argue that the far reaching implications a flawed cache coherence would have, warrant an in depth analysis of the design to catch potential design errors.

Furthermore we believe that such an analysis needs to be formally rigorous, because the components proof to be too complex to reason about manually. To motivate this Table 3.1 shows the comparison of different components and their internal states and transitions. These values are obtained from the HIERA-Gen generator and show that especially for the bridges, the complexity grows to such a degree that a manual pen and paper proof would become infeasible. The statement that this complexity is not easy to reason about is supported by our finding of design flaws in these bridges (Subsection 7.3.1), which are part of the artifacts of the related work.

Component	States	Transitions
MSI Directory	4	24
MESI Directory	6	45
MOESI Directory	5	66
MSIxMSI Bridge	21	95
MESIxMESI Bridge	40	184
MOESIxMOESI Bridge	59	368

Table 3.1: Comparison of the complexity of different components in terms of states and transitions.

3.3 Limitations of Conventional Model Checking

Applying conventional model checking approaches to our target system, results in two potential limitations: (1) If the full system is modeled and analyzed this will require large amounts of memory, possibly rendering the analysis infeasible. (2) The approaches, used by the related work make assumptions about the composition of the full system that break the verification guarantees.

The **memory limitation** can be observed when building upon the published artifacts of HIERA-Gen [9]. We create a full system model, which contains two clusters, each with 3 cache controllers, that are connected over a shared L2 using two bridges in Subsection 7.3.2. However, it is infeasible to analyze this model as is, due to the large resource requirements. Because of the rapid state space explosion of the paper's model artifacts the required RAM to perform a full liveness verification exceeds 1.9 TB. The analysis in Subsection 7.3.2 shows that this can be greatly reduced when fixing some of the model design flaws, which are present in the version published on [1]. However, after implementing these fixes the full system model still requires 268 GB of RAM. This makes it infeasible to perform the analysis on most ordinary machines and instead necessitates a large server for verification. The required resources to analyze larger models grow exponentially, such that a minimal increase in the model size is likely to also necessitate an upgrade of the platform on which the verification is performed.

The search for ways to mitigate this exponential memory consumption has led to **flawed models**, which no longer capture the behaviour of the system in full. This becomes apparent when considering that the related work, for which we were able to find a deadlock, uses model checking as well.

The HIERA-Gen paper restricted the model checking to a single bridge between an L1 and L2 cluster, in order to circumvent their memory limitation. This implicitly assumes that the bridge will behave the same when it is part of a full system, which contains other bridges. However, this is false, as having a full system model with two bridges reveals that the full system can encounter a deadlock. This deadlock is not detected by the analysis done as part of the HIERA-Gen paper.

This raises the question of how to achieve a feasible system verification, which provides the same level of confidence as a full system analysis, but requires less resources. Motivated by this challenge, this thesis explores an approach using decomposition and formal reasoning about the relation of subsystems to obtain a set of smaller and more feasible checks.

4 Overview

This chapter gives an overview of the work presented in this paper. It contains a general description of the compositional model checking approach proposed by this thesis in Section 4.1. As an application of the compositional approach and for evaluating it's potential benefits, Section 4.2 introduces the use case of compositional liveness verification.

4.1 Compositional Model Checking

Conventional approaches to model checking suffers from state space explosion as the number of caches increases. This is evident when evaluating the resources required to analyze our full baseline model, which is a extension of the model used by HIERA-Gen. Subsection 7.3.2 goes into more detail about the limitations of this unmodified model.

Our **main contribution** is therefore the development of a *new set of models*, which employ a *compositional approach* to analyze the reference system. We identified the bridge as a key component for splitting the system into individual subsystems, as the bridge only operates on one side (lower or higher level) at a time. Under the assumption that the other side is behaving correctly and without stalling (i.e. no deadlock), it can be represented with a simplified atomic system, which responds identically to the same input as the full system. This approach allows us to split the full system model at each bridge, resulting in an individual model for each cluster, with a simplified, but equivalent, interface of the connected clusters.

However, this approach leads to two new challenges: (1) How can a simplified, less resource intensive model be obtained for the external interfaces, which we will address in Subsection 4.1.1. (2) How can we show that using the simplified model is indeed resulting in an equivalent observable behaviour as the full model, which we will elaborate on in Subsection 4.1.2.

4.1.1 Simplified System Models

The crucial insight to the simplification of the external system models is the fact that the bridge merges different execution traces, resulting in observable behaviour that makes it impossible to distinguish between the preceding transitions. Suppose the bridge is waiting for the invalidation acknowledgements of multiple caches in the L2, the order at which they arrive will not be observable from the caches in the L1 as the resulting messages on the lower layer are identical, independent of the ordering or number of acknowledgements the bridge received on the higher level.

It follows, that from the perspective of the lower layer, the system is sufficiently well defined if the upper layer always responds in the same order, with the same number of acknowledgements.

Generalizing this insight we developed a number of independently applicable properties, which can be assigned to components or groups of components to reduce their internal state space, by enforcing a deterministic handling of events:

1. *component ordering*, allows to order components deterministically, and eliminate potential reorderings that would arise from component race conditions.
2. *abstract components*, allows a group of components of the same type to immediately forward internal messages, as if the components functioned as one large component.
3. *component atomicity*, allows a component to process information immediately, effectively eliminating the need for input queues.
4. *cluster atomicity*, is an extension to component atomicity, which guarantees each flow of network messages is completed before starting a new one.

4.1.2 Equivalence in Observable Behaviour

The **observable behaviour** of a subsystem is defined on the interface with the remaining system. It precisely includes the input to output mapping of the subsystem, i.e. all possible communication sequences. Additionally, all potential output sequences which can not be matched to a specific input are also part of the observable behaviour. If two components or subsystems have the same observable behaviour it is impossible to distinguish between them from just looking at how the component interfaces with the remainder of the system.

To better illustrate how two different components may have the same observable behaviour, suppose there are two components C_{ctr} and C_{mod}. Both components accept two input messages $msg \in \{inc, read\}$, *inc* increases an internal counter and *read* returns the last bit of the counter. Now suppose C_{ctr} has a full internal counter that stores the number of received *inc* messages, while C_{mod} only stores the last bit and flips it on each *inc* message. Both components have the same observable behaviour but behave differently internally. This is a minimal example but the principle can be generalized to more complex components or whole subsystems.

Checking for **equivalence in observable behaviour**, is challenging in general. Ideally it is possible to compute an exhaustive list of the input sequence to output sequence mappings of both components, such that they can be compared for any mismatches. However this is challenging because input and output sequences are unbounded in length and without knowledge of the internal state of the components, it is impossible to determine how long the sequences are required to be. Additionally, components might initiate internal operations, which causes the input to output sequence to be a one to many mapping. Furthermore, since there may be multiple internal representations of a component that lead to the same partial output trace, we have to explore all of them for each potential continuation of the trace. This reliance on exhaustiveness and exploration of internal states, lends itself to an automated exploration using the model checker.

Therefore we propose an equivalence check that capitalizes on the exhaustiveness of the model checker state space exploration. The check is set up to show the observable behaviour of one model m_{LHS}, is a subset of the observable behaviour of a second model m_{RHS}, i.e. for the same input communication sequence m_{RHS} is able to produce all potential output communication sequences that m_{LHS} can produce. This is expressed as $ob(m_{LHS}) \subseteq ob(m_{RHS})$, where ob is the observable behaviour. By exploring all potential input sequences and comparing the corresponding output sequence for both m_{LHS} and m_{RHS} we can conclude that if they are identical in all cases then $ob(m_{LHS}) \subseteq ob(m_{RHS})$. This is done in both directions, by exchanging m_{LHS} and m_{RHS} to show equivalence in the observable behaviour.

4.2 Compositional Liveness Checks

Showing liveness of a system by decomposing it and looking at individual subsystems is not possible in general. To show this consider a system with two subsystems A and B. Both subsystems operate on two shared locks L_1, L_2, the locks can be exclusively acquired by each subsystem. Suppose subsystem A always acquires L_1 then L_2 before performing any operation and releasing both locks again. If the operation is deadlock free then subsystem A is also deadlock free. The same holds for the full system if subsystem B behaves exactly like subsystem A. However, suppose subsystem B acquires the locks in reverse order, first L_2 then L_1. This would lead to a potential deadlock in the full system when subsystem A acquires lock L_1 and is waiting on L_2, while subsystem B acquires lock L_2 and is waiting on L_1. Even tough the individual subsystems are deadlock free their composition might not be.

This greatly limits the applicability of decomposition to analyze large systems for deadlock freedom. However in this example, it is irrelevant what actual operation is performed by any of the subsystems, as long as it is deadlock free. The crucial part is the order in which a subsystem acquires the locks which are shared between the subsystems. We describe this as the observable behaviour of the subsystem. This implies that the full system can be checked for liveness with two separate checks of a potentially simpler system: (1) Subsystem A is replaced with a subsystem with equivalent observable behaviour but an operation which is easier to analyze. (2) Subsystem B is replaced analogously. A more detailed explanation of this compositional approach and how it can be applied to the target system is contained in Section 5.4.

The cost of model checking generally grows exponentially with an increase in system complexity. When using this compositional approach there is a linear number of checks that have to be performed, one for each subsystem. If the potential benefit is roughly estimated, then the compositional checks should be exponentially less resource intensive, even if the number of checks that have to be performed increased by a linear factor. Asymptotically this is less resource intensive than a full system analysis.

In this thesis we use this approach to verify liveness of our target system. We utilize abstractions and equivalence checks, to reduce the complexity of other clusters that the bridge is interfacing with. This way we decompose the full target system check into one check for each cluster, where the bridge, including the remainder of the full system, is replaced by a equivalent but less resource intensive model. This will leave us with one liveness check per cluster and one equivalence check for each bridge that has to be replaced.

5 Design

This chapter describes the design of the crucial parts of this thesis. Section 5.1 explains the base structure of how a system with multiple components and clusters is modeled. This general model is the basis for all other work in this thesis.

As explained in Section 4.1 there are two parts to our compositional verification approach: (1) Abstractions, which have desired properties, like a decreased internal state space, compared to their corresponding full systems. (2) An equivalence check that proves that the full systems can be replaced by the abstractions, by showing them to have the same observable behaviour. Section 5.2 describes a general approach to creating abstractions. These are generally applicable, due to not requiring any special knowledge about the full system that will be abstracted.

Section 5.3 explains our proposal for how to check whether an abstraction has the same observable behaviour as the full system. This is a necessary step to prove that it is valid to substitute the full system with the abstraction.

The final section in this chapter, Section 5.4, explains how abstractions and equivalence checks can be applied to show liveness of our target system.

5.1 Baseline Model

Our baseline model is similar to the generated models of our related work, HIERA-Gen and HETERO-Gen. It serves as the basis of how the system is represented in Murφ compatible concepts.

Each component in the system is its own state machine in the model. Those state machines react in accordance to messages they receive over the network or by initiating an action, e.g. requesting a cache line upon a read access. Any operation only modifies the state machine of a single component in the system. An exception is the reset functionality that configures the initial state for the whole system.

This allows for a clear mapping of the system components to the state machines in the model. Another special case is the network model. Since the individual components communicate with each other over different network channels, multiple components might affect the state of the network. Thus the network is the crucial part of the model which is shared between the different components.

The network is represented as an ordered receive queue at each component. To send messages over the network a centralized interface is used. This allows to precise how components can interact with the network. For our models the network queues act as a multiple-producer single-consumer (MPSC) queue. The communication between components can therefore be defined as the chronological sequence of their network queue states.

5.2 Abstractions

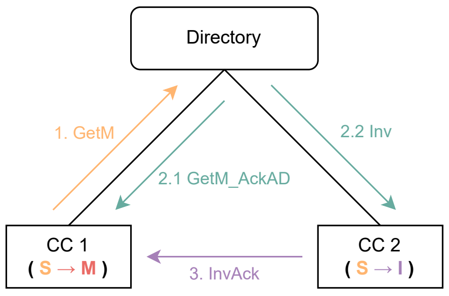


Figure 5.1: An example of the invalidation flow in the full system.

To reduce the overall state space explored by a model, it is possible to use abstractions for subsystems. This section will present our generalized approaches for creating such abstractions. Depending on the use case different behaviours might need to be upheld by the abstractions. In this thesis, with respect to our target system, we are exploring general methods to create an abstraction of a cluster. We know a bridge eliminates complexity of the cluster it connects to. However, determining what is a suitable abstraction can be challenging, this is were these general methods can be useful. A minimal effort proof can just attempt to show equivalence using all these abstractions and then settle on the abstraction where the check succeeds and the observed state space is the smallest.

To this end we first motivate the effect of using these abstractions analytically, by looking at the small example flow depicted in Figure 5.1. The abstractions are then compared in Figure 7.6, to determine their relative state space reduction empirically. If little is known about the semantics of the bridge, the abstraction with the greatest state space reduction, which still succeeds the equivalence check, may be used.

Figure 5.1 depicts an example of a message flow, where CC 1 is requesting to transition into the M state for a cache line, while both CC 1 and CC 2 are currently sharing the cache line (S state).

To understand how the abstractions affect the statespace that is explored, when using a model checker, we first need to specify the explored state space for the full system model, without any abstractions. We use the following notation to talk about the state space:

A system state $s \in S$ is a tuple of the states of the individual components $s = (s_{dir}, s_{cc1}, s_{cc2})$. Where the directory state s_{dir} is a tuple of the cache line state, list of sharers/owners and pending messages $(state, sharers, msgs) \in \{M, S, I, ...\} \times P(\{cc1, cc2\}) \times \{m \in \mathbb{M}\}$ [1]. Where a message $m = (type, sender) \in \{getS, getM, ...\} \times \{dir, cc1, cc2\}$ is a tuple of message type and sender. A cache state s_{cci} is a tuple of cache line state, acks recieved, acks expected and pending messages $(state, aR, aE, msgs) \in \{M, S, I, ...\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{M}$.

Note that this is a simplified view of the system, in reality further information, like the address and cache line value, are part of the model and statespace. This simplification also disregards the queuing system implemented by the components, however it should suffice to explain the relevant mechanisms of the abstractions.

Using this notation the initial state of the system is expressed as:

$$s_0 = ((S, \{cc1, cc2\}, \{\}), (S, 0, 0, \{\}), (S, 0, 0, \{\}))$$

After CC 1 decides to request the M state, the resulting state that follows is:

$$s_1 = ((S, \{cc1, cc2\}, \{(getM, cc1)\}), (S\_getM, 0, 0, \{\}), (S, 0, 0, \{\}))$$

The transition relation $\delta \subseteq S \times S$ specifies which state is reachable from another state, i.e. $(s_0, s_1) \in \delta$. Transitively exploring using δ, results in the explorable state space. If we restrict δ to the behaviour shown in Figure 5.1, we can create a list of all explorable states (Figure 5.2a).

After reaching s_2, the directory processes the *getM* message and sends two messages, one to CC 1 and one to CC 2. The model checker is now required to explore both possibilities of either CC 1 or CC 2 processing it's message first. There are two different outcomes s_3 and s_4 respectively. In s_3 CC 1 processed the Ack from the directory, now knowing it has to wait for 1 Ack, the only next possible state is for CC 2 to process the invalidation to reach s_5. If instead in state s_4, then CC 1 has two pending messages and is able to process either one of them[2], leading to states s_5 and s_6. From both of these states the next and final state is s_7, which is reached once CC 1 processed all the messages and successfully transitioned to cache line state M.

[1] $P(x)$ is the power set of x and is defined as $y \in P(x) \iff y \subseteq x$

[2] This is possible due to the simplified model with no separate message queues. The real world and the full system model used for evaluation is more nuanced.

$$\begin{aligned}
s_0 &= ((S, \{cc1, cc2\}, \{\}), (S, 0, 0, \{\}), (S, 0, 0, \{\}))\\
s_1 &= ((S, \{cc1, cc2\}, \{(getM, cc1)\}), (S\_getM, 0, 0, \{\}), (S, 0, 0, \{\}))\\
s_2 &= ((M, \{cc1\}, \{\}), (S\_getM, 0, 0, \{(getM\_AckAD, dir)\}), (S, 0, 0, \{(Inv, cc1)\}))\\
s_3 &= ((M, \{cc1\}, \{\}), (S\_getM\_AckAD, 0, 1, \{\}), (S, 0, 0, \{(Inv, cc1)\}))\\
s_4 &= ((M, \{cc1\}, \{\}), (S\_getM, 0, 0, \{(getM\_AckAD, dir), (InvAck, cc2)\}), (I, 0, 0, \{\}))\\
s_5 &= ((M, \{cc1\}, \{\}), (S\_getM\_AckAD, 0, 1, \{(InvAck, cc2)\}), (I, 0, 0, \{\}))\\
s_6 &= ((M, \{cc1\}, \{\}), (S\_getM, 1, 0, \{(getM\_AckAD, dir)\}), (I, 0, 0, \{\}))\\
s_7 &= ((M, \{cc1\}, \{\}), (M, 1, 1, \{\}), (I, 0, 0, \{\}))
\end{aligned}$$

(a) List of states in the invalidation example.

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] \arrow[dr] & s_3 \arrow[r] & s_5 \arrow[r] & s_7 \\
 & & & s_4 \arrow[ur] \arrow[r] & s_6 \arrow[ur] &
\end{tikzcd}$$

(b) Graph of state transitions for the invalidation example.

Figure 5.2: States derived from the invalidation example in Figure 5.1.

The following subsections will present the different abstractions presented by this thesis and explain their effect on the aforementioned example message flow in Figure 5.1.

5.2.1 Ordered Cache

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_5 \arrow[r] & s_7
\end{tikzcd}$$

(a) Graph of state transitions with order = (CC 1, CC 2).

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[dr] & & s_5 \arrow[r] & s_7 \\
& & & s_4 \arrow[ur] \arrow[r] & s_6 \arrow[ur] &
\end{tikzcd}$$

(b) Graph of state transitions with order = (CC 2, CC 1).

Figure 5.3: States transition graphs of ordered caches.

The cache or component ordering is an optional abstraction, which enforces a global static ordering of how components are allowed to process pending messages. This particularly targets the elimination of redundant state space exploration, caused by differences in the ordering of network messages. For the concrete example of the invalidation flow, there are two possible scenarios, depicted by Figure 5.3a and Figure 5.3b. If CC 1 is ordered, such that it will always process its messages before CC 2, then this eliminates the transition from s_2 to s_4, and all further transitions that depend on it. The same thing happens for the transition from s_2 to s_4, when using the inverse order. Notably this abstraction does not impact the distance between s_0 and s_7, it only enforces a linearization.

5.2.2 Abstract Cache

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] \arrow[drr] & s_3 \arrow[rr] & & s_7 \\
& & & & s_6 \arrow[ur] &
\end{tikzcd}$$

(a) Graph of state transitions with both abstract CC 1 and CC 2.

Figure 5.4: States transition graph of abstract caches.

The abstracted components (Figure 5.4a) are treated as if one big entity, with atomic internal messaging. In particular for the example in Figure 5.1, this implies that the invalidation acknowledgement, which is sent by CC 2 to CC 1, is atomically processed. This leads to the removal of states s_4 and s_5 from the explorable statespace. This abstraction does reduce the distance between s_0 and s_7, eliminating all the intermediate states, that stem from communication internal to the abstracted. The input/output queues of the abstracted components, when filtering out the internal messages of the abstracted components (for the example this is InvAck), still preserve all of the possible states, i.e. the messages with type $\{\{\}, \{GetM\}, \{GetM\_AckAD, Inv\}, \{GetM\_AckAD\}, \{Inv\}\}$ for this example.

5.2.3 Atomic Cache

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] & s_3 \arrow[r] & s_7
\end{tikzcd}$$

(a) Graph of state transitions with CC 1 atomic.

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[dr] & & s_5 \arrow[r] & s_7 \\
& & s_4 \arrow[ur] \arrow[r] & s_6 \arrow[ur] &
\end{tikzcd}$$

(b) Graph of state transitions with CC 2 atomic.

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r] & s_7
\end{tikzcd}$$

(c) Graph of state transitions with both CC atomic.

Figure 5.5: States transition graphs of atomic caches.

The atomic components immediately and atomically process anything in their input queues, this in general results in a single transition in the state transition graph, whenever atomic components are involved. Atomicity is general enough such that it can be applied to single components or groups. Figure 5.5 shows the transition graphs of different groups of atomic components, it shows that for this example atomic components have a have the potential for a greater state space reduction than the previous abstractions. However this comes at the cost of losing out on the additional properties that the previous abstractions provide.

$$\begin{tikzcd}s_0 \arrow[r] & s_7\end{tikzcd}$$

(a) Graph of state transitions with atomic network.

Figure 5.6: States transition graph of atomic network.

5.2.4 Atomic Network

The atomic Network is the generalization of atomicity to a whole network. In a flat architecture, such as the one depicted in Figure 5.1, this essentially equates to making all components atomic. The resulting transition graph for the example (Figure 5.6) is a single transition. However this can not generally be applied to hierachical systems, because bridges are part of two networks and have to be handled separately to ensure only the desired networks behave atomically.

5.3 Equivalence Checks

The equivalence check is a set of custom models with the purpose of verifying that the observable behaviour of two different components is the same. More precisely the equivalence check verifies equivalence in two steps using at least two models, each one showing a subset relation on the observable behaviour.

Suppose we have a full system S which integrates a component M_1. Further suppose M_2 is a different component which should have the same observable behaviour as M_1. What we want to show as the end result of the equivalence checks is $ob(M_1) = ob(M_2)$, we do this in two symmetric steps by showing $ob(M_1) \subseteq ob(M_2)$ and $ob(M_2) \subseteq ob(M_1)$ separately using different models. Next we will explore the design of a general method to prove the subsystem relation of observable behaviour. With such a method, we can simply perform it twice for the two directions that are required, by switching M_1 and M_2.

Because the same model framework can be used for both directions, hence we generalize both the left hand side (LHS) and right hand side (RHS). The property the model verifies is then $ob(M_{LHS}) \subseteq ob(M_{RHS})$ for general models M_{LHS} and M_{RHS}. In other words the model will verify that with respect to all possible input communication-sequences of M_{LHS} in S, the set of all potential output communication-sequences is a subset of the potential output communication-sequences of M_{RHS} for the same input communication-sequences. An input sequence can be of length 0, meaning the component initiates the communication. This empty input sequence is therefore special case, as it's corresponding output communication-sequences can be compared independent of the remainder of the system S. All other sequences are dependent on the remainder of the system, as the output is dependent on the input, which could potentially be produced by a component other than M_{LHS} or M_{RHS}. The consequence of this is that equivalence in observable behaviour can be tied to a specific system. To demonstrate this consider the following example:

Suppose a system S_0 contains M_{echo} which does not initiate communication and always replies to input communication with the same messages it received, i.e. it just returns the input. Further, suppose S_0 will only ever send msg_0 to M_{echo}. It is possible that within that system M_{echo} has equivalent observable behaviour to a component which always replies msg_0 to any incoming communication. However the observable behaviour is generally not equivalent: Consider for example a system S_1 which exclusively sends msg_1 to M_{echo}. Therefore when we show equivalence using our equivalence check, rather than being able to exhaustively explore all possible input sequences in all possible systems, we only show equivalence tied to a specific system.

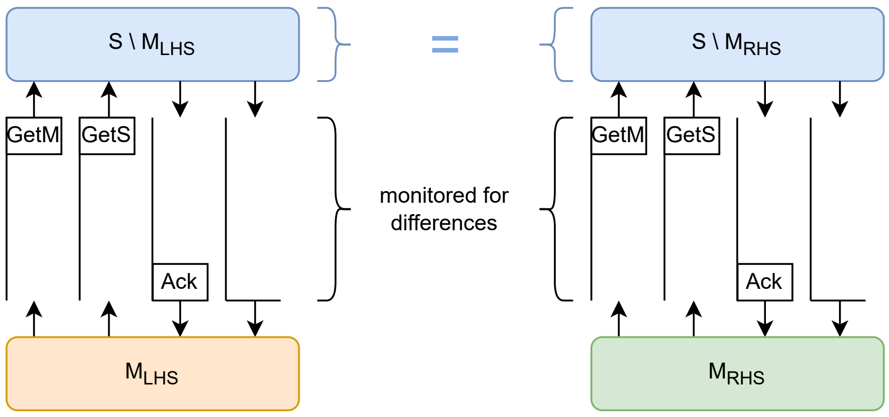


Figure 5.7: High level diagram of the equivalence check setup.

For comparing the observable behaviour of M_{LHS} or M_{RHS} suppose S_{LHS} and S_{RHS} are the system S using M_{LHS} and M_{RHS} respectively. The relation of the components in the system is visualized in Figure 5.7. The $ob(M_{LHS}) \subseteq ob(M_{RHS})$ check is achieved on a high level through these rules:

(i) instantiate both S_{LHS} and S_{RHS} at the same time.

(ii) perform an exhaustive state space exploration of S_{LHS}.

(iii) monitor the communication between M_{LHS} and $S_{LHS} \setminus M_{LHS}$ and compare it with the communication between M_{RHS} and $S_{RHS} \setminus M_{RHS}$.

(iv) when a mismatch in the communication on the LHS and RHS is detected halt the state space exploration of S_{LHS}, create a backup of the current state of S_{RHS} and switch exploration to the RHS.

(v) when exploring the RHS, determine whether the mismatch in the observed communication was caused by M_{LHS} or $S_{LHS} \setminus M_{LHS}$. Depending on the result perform a state space exploration of M_{RHS} or $S_{RHS} \setminus M_{RHS}$ respectively.

(vi) when exploring the RHS, allow S_{RHS} to reset back to the state, which was saved when execution switched to the RHS.

(vii) when exploring the RHS and the communication on the LHS and RHS matches allow switching the exploration back from the RHS to S_{LHS}.

(viii) check whether an arbitrary length path to switch back from RHS to S_{LHS} can always be found.

To demonstrate that this shows $ob(M_{LHS}) \subseteq ob(M_{RHS})$ within the system, we will provide the following reasoning:

Suppose the equivalence check succeeds, but $ob(M_{LHS}) \not\subseteq ob(M_{RHS})$. This implies there is some sequence of communication steps $seq \in ob(M_{LHS}) \setminus ob(M_{RHS})$. Suppose without loss of generality (WLOG) that seq_n are the last communication steps, up until all steps are also part of $ob(M_{RHS})$.[3] Because the equivalence check succeeded, the exhaustive state space exploration on the LHS will reach the point where the communication sequence seq is observed on the LHS. If this is equivalent to the observed sequence on the RHS, then M_{RHS} was able to reproduce the observable behaviour. Else if the communication sequence is not equivalent, then the check switches to the RHS. Since the equivalence check succeeds, the RHS is able to switch back to the LHS, which implies it is able to reproduce the same communication steps. In both cases is it possible for M_{RHS} to reproduce the observable behaviour.

[3]This is possible WLOG, since otherwise a subsequence of seq, having the same starting point, can be found where this holds.

5.4 Compositional Liveness Check

To show liveness of the target system using our compositional approach, we are required to perform a series of checks which together allow to deduct liveness for the whole system. Our main goal for the compositional approach is the reduction of the RAM required to show liveness for the target system. Therefore we will optimize towards a smaller state space, as this has a linear effect on the memory footprint (see Chapter 3). A side effect of a smaller state space is the reduction in time that is required to explore it.

To keep the state space small we ideally want to split the target system into three clusters, which we analyze individually. Therefore we decompose the target system at the bridge interfaces. This is motivated by the bridge being active in exactly one of the two connected clusters at a time. Therefore from the perspective of either cluster, the requests to the other cluster appear atomically. From this property about the construction of the bridges, which we consider, we can make the following assumption:

Assumption: Because the bridge is active either in the lower or the upper level at any point in time, it follows that if the lower layer is stalling, the bridge is awaiting a message on the upper layer. We assume that the bridges we consider for our compositional liveness checks, only stall the L1 when they also sent a request to the upper layer. In other words, it is impossible for the bridge to stall requests on the lower layer and not having any outstanding request in the L2.

Breaking this assumption, would lead to a system with a trivial deadlock, where all bridges stall their local clusters, but make no requests on the L2. Due to all bridges awaiting a message from the L2, but none sending any messages, the system trivially can not make any progress.

With the assumption and the decomposition of the system decided, the next step is to find suitable abstractions. These abstractions need to produce the same observable behaviour from the perspective of the subsystem (i.e. cluster) where they replace the full system.

5.4.1 L2 cluster

For the L2 cluster a lot of the complexity of of the two L1 clusters is eliminated by the bridge interface. Therefore a valid approach would be to apply the abstractions in Section 5.2 to each L1 cluster and perform a equivalence check with each of them. This way the least resource intensive abstraction, which still succeeds the equivalence check can be used. However in our case we decided to use an abstraction, which is more specific to our target system: We want to show that both L1 clusters can be abstracted with a ordinary MESI CC, in the L2. Note that we will use MESI as the L2 cluster, due to our target L2 being CXL. The following argumentation is generally applicable to any fixed L2 protocol.

This requires four equivalence checks:

(i) $ob_{L2}(CC_{MESI}) \subseteq ob_{L2}(Bridge_{MESIxMESI})$

(ii) $ob_{L2}(Bridge_{MESIxMESI}) \subseteq ob_{L2}(CC_{MESI})$

(iii) $ob_{L2}(CC_{MESI}) \subseteq ob_{L2}(Bridge_{MOESIxMESI})$

(iv) $ob_{L2}(Bridge_{MOESIxMESI}) \subseteq ob_{L2}(CC_{MESI})$

Remember from Section 5.3, that the equivalence check shows both: (1) the two components initiate communication in the same way, which is independent of the other components in the system and (2) the two components respond to incoming communication identically.

We can therefore use the abstraction for the MOESI L1 Cluster in (i) and (ii), and the abstraction for the MESI L1 Cluster in (iii) and (iv). If all of the four checks (i-iv) succeed, this substitution is valid.

Next we will prove this statement, using contradiction:

Suppose that substituting $Bridge_X$ with CC_{MESI} is affecting the result of the equivalence check for $ob_{L2}(Y) \subseteq ob_{L2}(Z)$. Note that $Y \neq Bridge_X \neq Z$ are chosen such that the statement refers to one of the equivalence checks (i-iv), as otherwise the statement would be irrelevant.

Since the equivalence check succeeds with CC_{MESI}, this implies it will fail with $Bridge_X$. A failing equivalence check implies there is some sequence of communication steps $seq \in ob_{L2}(Y) \setminus ob_{L2}(Z)$, which is is part of the observable behaviour of $ob_{L2}(Y)$ but not $ob_{L2}(Z)$. Let $l(seq)$ be the length of seq. We are now showing that such a seq can not exist using induction over $l(seq)$.

Induction base:

$l(seq) = 0$, therefore seq is the first change of observable behaviour. According to (1), the equivalence checks ensure initiated communication is equivalent regardless of the rest of the system. Therefore exchanging $Bridge_X$ and CC_{MESI} does not affect the result of the equivalence check. This implies that seq does also exist in the equivalence check using $CC_{MESI})$, therefore that equivalence check has to fail. This is a contradiction with the premise, thus there is no seq with $l(seq) = 0$

Induction step:

Suppose there is no *seq* with $l(seq) < i$ and $i > 0$, this implies for any substitution of Y with Z there is no mismatch of observable behaviour caused by a *seq* of length $l(seq) < i$. This is true because Y and Z are general, so the same partial proof can be made for $ob_{L2}(Z) \subseteq ob_{L2}(Y)$ up until $l(seq) < i$.

We want to show that this implies there is no *seq* with $l(seq) = i$ by using contradiction. Suppose there is a *seq* with $l(seq) = i$. There are two cases: (a) Y or (b) the remaining system, including $Bridge_X$, obtained *seq* by adding communication steps to a predecessor sequence seq' of *seq*.

Case (a): If Y obtained *seq* after seq', then $len(seq') < i$. Because of the induction step it follows that for any substitution of $Y = Bridge_X$ with $Z = CC_{MESI}$ there is no mismatch of observable behaviour caused by a seq' of length $l(seq') < i$. Therefore seq' can be obtained both by using $Y = Bridge_X$ or substituting it with $Z = CC_{MESI}$. Therefore Y is also able to obtain *seq* after seq' in the equivalence check with CC_{MESI}, therefore that equivalence check has to fail. This is a contradiction with the premise, thus there is no *seq* with $l(seq) = i$ where Y added the violating communication steps.

Case (b): If the remaining system obtained *seq* then $Bridge_X$ has to have caused *seq*, since the rest is identical to the equivalence check using CC_{MESI}. However this implies *seq* with $l(seq) = i$ is part of the equivalence check where $Y = Bridge_X, Z = CC_{MESI}$ in which $Y = Bridge_X$ added the violating communication steps. This is a contradiction with the result of case (a).

Therefore both cases lead to a contradiction, which implies there is no *seq* with $l(seq) = i$.

Thus it is shown by induction that there is no $seq \in ob_{L2}(Y) \setminus ob_{L2}(Z)$. Therefore the equivalence check with $Bridge_X$ succeeds and it follows that the substitution does not affect the result of the equivalence check. The proof due to its generality can similarly be adapted to more L1 clusters.

After all equivalence checks succeed, it is therefore possible to model the system as a flat MESI protocol. Both the directory and the CC in this system are ordinary MESI components. Since MESI is a well known protocol family, we skip it's liveness verification in this thesis. If one were to challenge this assumption, it is trivial to include an additional liveness check for the flat MESI protocol with 2 CC.

Using this approach we can deduce liveness for the L2 layer. Thus the only way for the full system to deadlock is if one of the L1 clusters goes into a deadlock in response to an L2 message, which we will check for next.

5.4.2 L1 clusters

For the liveness verification for the L1 clusters we use a simplified model of the full system, in which the other L1 cluster are abstracted away by a MESI CC on the L2. We know from the previous section that this is a valid substitution, if all the listed equivalence checks succeed and our assumption about the operation of the bridge holds.

It is also possible to create a further abstraction for the whole L2 layer to further decompose the system. However, because we split the target system at the bridges and we require an equivalence check to verify the correctness of abstractions, we are always required to have at least one two-cluster system in the equation. Since the LHS of the equation effectively performs an exhaustive state space exploration it will be similar in resource requirements to a ordinary liveness check for this two-cluster system.

For this thesis we still decided to perform further decomposition at the MESIxMESI bridge, to serve as a proof of concept and show the same methods can be applied in both directions (i.e. abstracting L1 in L2 and vice versa). We again use a target system specific abstraction by using a slightly modified directory.

The intuition behind this check is that not only do we want to show deadlock freedom, we also want to show some safety properties by proving how the bridge relates to a directory. This can help developers better understand the semantics of the system they are working with. This is done by defining the behaviour that a single CC can expected from the remainder of the system.

However a bridge can not have equivalent behaviour to a simple directory. Directories only behave reactively, by responding to request of a cache, a bridge however might be required to send messages in response to requests from the other cluster. Therefore we will use an abstraction of a directory and additional CC, one for each other cluster on the L2. Effectively this abstraction "pulls" the caches of the higher level down under a common directory.[4] These caches are treated and behave slightly differently, compared to normal caches. By iteratively modifying the abstraction, based on the result of the equivalence checks, we found that the following properties suffice for our target system:

(i) If a cache makes a request that would cause the directory to send an invalidation request to any of the caches in the abstraction, then instead the abstraction behaves as if the caches in the abstraction evicted their cache lines before the directory processed the request. (ii) If a chain of cache line forwarding crosses the boundary between ordinary and abstract caches, then the directory stalls request to avoid double crossing of the boundary.

[4]This description is similar to what HeteroGen directories try to achieve. In future work we want to explore whether a HeteroGen directory could be used as an abstraction. For the scope of this thesis we settled on iteratively finding the required changes to an ordinary directory instead.

While (ii) only affects the timing and liveness behaviour, (i) has further semantic implications that make sense in the context of a bridge. It is possible to interpret (i) as: "If a ordinary CC makes request for a cache line where it will potentially have to await invalidation acknowledgements, then the abstract caches will not be in possession of this cache line." This implies that if the CC implements a counter for counting the received invalidation acknowledgement, then it's value is always less than the number of ordinary caches. This is also trivial when looking at the full system using clusters, as invalidation acknowledgements are only counted inside a cluster.

For the purpose of showing liveness, the semantics of the abstraction are irrelevant as the equivalence check guarantees it has the same observable behaviour as the full bridge. Therefore the bridge can be substituted by the abstraction for the liveness verification of the L1 cluster.

6 Implementation

6.1 Model Generator

This section details the major modifications to the generator of the models used by the model checkers. First, the structure of the baseline model is explained in Subsection 6.1.1. Subsection 6.1.2 discusses our changes to obtain a more customizable and efficient network, while Subsection 6.1.3 describes how we map the Murφ models to Rumur ones.

6.1.1 Baseline Model

This subsection explains the structure of the baseline model, which is also the basis for our compositional models. The model file is structured in the following sections, which are partly motivated by the restrictions of the muphi language:

Constants are declared first, in particular this includes the fixed number of addresses, values, caches and the maximum length of the network queues.

What follows are the **type** declarations, which include the enum of all possible message types that are supported by the model, this combines the message types of the different clusters. The component state enums are also declared here, with one enum per type of component. The object set declarations, define custom types for each cluster and component type, which assign an identifiable id to each component in the system. These object set declarations also make use of scalarsets to exploit the symmetry of the models. A *machine* union is created of all object sets to be used for source and destination fields in network messages, which are declared next (see Figure 6.1). The message type is then used to declare the network types which are further discussed in Subsection 6.1.2. The last of the type declarations are the records that constitute an entry for a specific cache line, these contain fields to store the cache line state as well as the received and expected acknowledgments.

The next section of **variable** declarations contains the component instance as well as network queue declarations. What follows are a number of function and procedure declarations, the first of which operate on the network by modelling the sending and receiving of messages on a specific queue. As well as handling multi and broadcast

```
type
    Message: record
        adr: Address;
        mtype: MessageType;
        src: Machines;
        dst: Machines;
        acksExpected: 0..NrCaches;
        cl: ClValue;
    end;
```

Figure 6.1: Message Type declaration.

functionality and message construction. There are also helper functions for managing the multiset entries of directories.

The heart of the model are the component **functions**, which model the operation of the component state machine, in particular how a component reacts to an input message. Those functions are expressed as a finite state machine with multiple switch-case statements which determine the correct behaviour, derived from the component specification. The last of the helper functions specify how a component behaves on a load, store or evict operation.

Towards the end of the model specification are two sets of **rules**, the first ones model load, store or evict accesses to the components that support them. The second set of rules models the handling of network messages, which calls the appropriate component state machine functions. Lastly the startstate section specifies how the model should be initialized.

6.1.2 Ideal Network Design

The baseline model does model the network queues as a large two dimensional array, over all potential machines. This corresponds to the declaration in Figure 6.2, there is one type for the whole network and also one maximum queue size. This type is then used for each of the three queues: *fordward (fwd)*, *response (resp)* and *request (req)*. This approach allows for some nice network helper function that are generalized to work independent of the source and destination components. However network queues can make up a potentially large part of the model state. The use of a two dimensional array for modeling the network is wasteful, because it does not allow to adjust queue sizes according to the component's needs.

This is why we propose a different way to model the network, by splitting the type and variable declarations according to component type, as shown in Figure 6.3. This

```
type
    OBJ_Net: array[Machines] of array[0..NET_MAX-1] of Message;
    OBJ_Netcnt: array[Machines] of 0..NET_MAX;
var
    fwd: OBJ_Net;
    cnt_fwd: OBJ_Netcnt;
    ...
```

Figure 6.2: Combined network declaration.

requires additional logic to properly manage network messages, by ensuring they are sent and read from the appropriate queues. However the customization allows more flexibility in choosing network queue lengths, such that e.g. an atomic component can have a message queue of size one. Furthermore it allows to omit unnecessary queues, for the protocols that were analyzed by this thesis and the related work, the directory never accessed its *forward (fwd)* channel. Similarly the caches never used their *request (req)* channels, this can be detected on model generation and result in a model which omits these unnecessary channels. The separate network model is also no longer reliant on a union of all machine types, which helps in making the model compatible with Rumur, as will be discussed in the next subsection.

```
type
    NET_Cache: array[Cache] of array[0..CACHE_NET_MAX-1] of Message;
    NET_Cache_cnt: array[Cache] of 0..CACHE_NET_MAX;
    NET_Dir: array[Directory] of array[0..DIR_NET_MAX-1] of Message;
    NET_Dir_cnt: array[Directory] of 0..DIR_NET_MAX;
var
    fwd_Cache: NET_Cache;
    cnt_fwd_Cache: NET_Cache_cnt;
    ...
```

Figure 6.3: Separate network declaration.

6.1.3 Mapping CMurphi to Rumur

Rumur does not have support for unions and multisets, which are key components of the Murφ models. Therefore an equivalent representation has to be used to represent the same semantics: **Unions** in the Murφ models are exclusively used for enums and scalarsets, which allows the Rumur models to use a larger enum containing a one to one mapping of each individual entry in the unionized enums and scalarsets. This together with a mapping function to map between individual and unionized enum is sufficient for our use case. However, since the scalarset entries are not individually accessible, the mapping of scalarsets has to be initialized at startup using an array to store the next free enum entry for each scalar value.

Figure 6.4a shows a subset of the changes required to map a scalarset union from Murφ to Rumur. This maps from a Murφ version, which consists only of the two base type definitions (lines 1-2) and a union instruction over the two types `Caches` and `Directories`. To achieve the same thing in Rumur, while still maintaining the same benefits that come with scalarsets, the definitions of the base types (lines 1-2) need to be kept unaltered. The union is replaced with a combined enum, and all code instances that access the type are required to use mapping functions that map between `Machines` and the base types `Caches` and `Directories` based on the their requirements. However the scalarset entries can not be addressed individually in code, thus it is not possible to explicitly include them in the mapping functions.

Therefore we are required to add the new variable (line 8) which is an array mapping `Caches` to `Machines` entries (line 5). This causes an overhead in both state size and run-time execution. The array is initialized during the reset as part of the start state, where we iterate over all entries in the scalar set and assign them the next free unmapped entry in the unionized enum (lines 32-34). The function to determine the next free unmapped entry in the unionized enum, uses a helper function to determine if a specific unionized enum entry is already mapped, by iterating over all entries (lines 12-17). It is then possible to construct an explicit if-else chain to check for all the possible unionized enum entries, which can be mapped to this scalar type, in a predefined order (lines 22-28). Compared to the plain Murφ version, this is more overhead also in terms of code lines, especially when considering the required mappings between the unionized enums and the base types.

Similarly **multisets** are also not supported by Rumur, however since all potential values inserted in the multisets of the Murφ models are known, it is possible to replace the multiset with a boolean array at the expense of some additional space required per system state. The functions used to manipulate the multiset are updated accordingly.

```
type
    Caches: scalarset(2);
    Directories: enum { directory };
    Machines: enum { Caches_1, Caches_2, Directories_1 }
    SCALARMAP_Caches: array[Caches] of Machines;

var
    i_map_caches: SCALARMAP_Caches;

function mapped_Caches(v: SCALARMAP_Caches, m: Machines) : boolean
begin
  for c : Caches do
    if !isundefined(v[c]) & v[c] = m then
        return true;
    endif;
  endfor;
  return false;
end;

function next_scalar_Caches(v: SCALARMAP_Caches) : Machines
begin
    if mapped_Caches(v, Caches_1) then
        return Caches_1;
    elsif mapped_Caches(v, Caches_2) then
        return Caches_2;
    else
        error "next␣scalar␣can␣not␣be␣determined"
    endif;
end;

startstate
    for c : Caches do
        i_scalar_map_caches[c] := next_scalar_Caches(i_map_caches);
    endfor;
endstartstate;
```

(a) Scalar union in Rumur.

Figure 6.4: Example mapping of Murφ scalar union to Rumur.

6.2 Abstractions

The implementation of the abstractions presented in Section 5.2 is designed to be agnostic to the state machines and actual semantics of the cache coherence protocols. With the exception of the 'abstract components', all abstractions only rely on the network rules and queues for their functionality. The different abstractions are also generally independent, which allows for combining them to achieve a greater state reduction.

The ordered network can be realized by merging the component specific rules for accepting and processing network messages. Instead of a general ruleset over the ordered machine instances a single rule is used to handle the next message accept and process. This is realized by an if-else chain of all ordered components, that checks if the next component is able to accept a message, else it continues with the next component in order.

The 'abstract components' abstraction is the only one that is dependant on the individual finite state machines of the components. It checks each time the component sends a network message if the destination and source are both part of the same abstraction. If they are and the message would be the next that is handled by the destination, the message is applied to the destination's state machine directly. This might not always be possible, in which case the message is added to the end of the queue as usual.

Both of the atomic abstractions have an Implementation consisting of two parts: (i) a detection function and (ii) rule restrictions. The detection function evaluates the network queues of the atomic components or networks and determines whether any messages need to be handled atomically. All of the model rules are restricted such that if the detection function determines an atomic handling is required, then only atomic components can process. In the case of an atomic network, the access rules of atomic components which cause new network messages are also restricted, as soon as a message is in transit in the network. Due to the directory potentially sending multiple responses, as in the invalidation example in Figure 5.1, there are still potentially multiple messages in transit on the network. Therefore a combination with the ordered components abstraction can still be beneficial.

6.3 Equivalence Checks

The equivalence check implementation, realizes the steps outlined in Section 5.3. In particular it is required to contain both the LHS and RHS models and instrument communication, such that the check switches correctly between the two.

Instantiating both systems is possible by merging the two models, to avoid name clashes entities and function are suffixed with LHS or RHS depending on which system model they belong to. The suffix is emitted if a function is general and works for both LHS and RHS.

These models are already designed for extensive state space exploration, however additional restrictions on the rules are created, such that they can only fire when the respective LHS or RHS is currently explored. To achieve this a new global variable stores the current system state which is explored, e.g. `system_LHS` for the LHS.

With respect to the existing system models, the communication behaviour can be monitored by reading and filtering the network queues of the components. The logic is split into a mapping function which maps messages from LHS to RHS or from RHS to LHS as well as function that walks over the network queues to compare them, using the mapping function. A custom filter is used to determine whether a message on the network queues should be ignored, if e.g. it is a message from M_{LHS} to M_{LHS}.

Helper functions are declared which determine whether there is a mismatch in the queues. The output of these helper functions is used to restrict the rules of the LHS, ensuring that exploration of S_{LHS} is halted when a mismatch is detected. A new set of rules is added to the model, which handle the transitions between the LHS and RHS. This includes the creation of a backup of M_{RHS}, for which there is a copy of all variables comprising the state of M_{RHS}. The RHS is also extended with a rule that restores the values in this backup, for it's search of a potential way to replicate the LHS behaviour. Without this the RHS could reach a state with a mismatch in the queues, e.g. LHS has a *getS* from Cache1 while the RHS executed a rule that produces a *getS* from Cache2. Reaching this mismatched state is ok, if there is also a way to reach the correct state from the same starting state, i.e. executing the rule that produces a *getS* from Cache1.

Lastly, the crucial part of the equivalence check, that is only possible to express using a Rumur model is the liveness statement, which ensures that the currently explored system variable can always reach `system_LHS`. This instructs Rumur to make sure that there is always an arbitrary length path from each state, to a state where this statement is true.

However, as is the model is very resource intensive and can explore even more state space than the full system model. A large part of this state is redundant or irrelevant exploration on the RHS. This is due to the model checker performing an exhaustive state search, while on the RHS a simpler search would suffice. Therefore we require the following two optimizations to the equivalence check to reduce the explored state space:

The first optimization, eliminates a lot of unnecessary exploration on the RHS by using the fact that the network queues are directional and append only. Because the system is split according to the direction of the queues, each network queue is either a potential input or output queue of M_{RHS}. The consequence is, that M_{RHS} will only ever add to the end of output queues and remove messages from the front of input queues. Therefore checking for a mismatch in observable, can determine if the current network queue state can lead to an equivalent one in the future by running M_{RHS}. If this is not possible, e.g. because the input queues are already shorter or the output queues are already longer, then further exploration can be stopped by forcing the only possible rule execution to be a reset of the RHS. To realize this in the model, all the rules on the RHS, except the reset rule are restricted by a function, which determines if a reset of the RHS is necessary.

The second important optimization to the equivalence check is based on the realization that $S_{LHS}\ M_{LHS} = S_{RHS}\ M_{RHS}$. This implies that for the same observable input behaviour there will be the same observable output behaviour of $S_{LHS}\ M_{LHS} = S_{RHS}\ M_{RHS}$. Therefore it is possible to eliminate switching to $S_{RHS}\ M_{RHS}$ on the RHS, and instead only retaining it's queues. On the LHS when switching to $S_{LHS}\ M_{LHS}$, the state of the network queues of LHS and RHS is equal. Therefore after processing on the LHS, a replication function can reproduce the result of running $S_{LHS}\ M_{LHS}$ on the RHS. The replication function takes into account the direction of the network queues and their sizes, if the queue on the RHS is too large and the queue is an input queue to $S_{RHS}\ M_{RHS}$, then messages are removed until the length matches. Similarly, if the queue is an output queue of $S_{RHS}\ M_{RHS}$ and the length of the filtered queue is shorter then the equivalent queue on the LHS, then the missing messages are pushed to it, using the same mapping function that is also used for the queue comparison.

6.4 Compositional Liveness Check

The compositional liveness checks are implemented in accordance with the design in Section 5.4. We use the equivalence checks presented in the previous section to first show the equivalences before running the individual cluster liveness checks. Table 6.1, contains the checks we perform and what property they are showing.

Check	Tool	Property
`comp_2sH_`X	Rumur	$ob_{L2}(CC_{MESI}) \subseteq ob_{L2}(Bridge_{XxMESI})$
`comp_Hs2_`X	Rumur	$ob_{L2}(Bridge_{XxMESI}) \subseteq ob_{L2}(CC_{MESI})$
`comp_Hsd_MESI`	Rumur	$ob_{L1}(Dir'_{MESI}) \subseteq ob_{L1}(Bridge_{MESIxMESI})$
`comp_dsH_MESI`	Rumur	$ob_{L1}(Bridge_{MESIxMESI}) \subseteq ob_{L1}(Dir'_{MESI})$
`comp_3B1_MOESI`	Murφ	liveness of bridge with 3 L1 and 1 L2 CC
`comp_flat4_MESI`	Murφ	liveness of a flat protocol with 4 MESI CC

Table 6.1: List of compositional checks and the properties they show.
With $X \in \{MESI, MOESI\}$.

7 Evaluation

The main part of this evaluation is the the validation of the compositional verification approach in Section 7.2. We first establish our methodology and the baseline we compare against in Section 7.1. This is also the section where we give an overview of the necessary changes to the baseline. A more detailed experimental analysis of those changes follows in Section 7.3. Lastly we also we also perform some micro benchmarks that compare different abstractions in section Section 7.4.

7.1 Methodology

To evaluate the resource requirements for different models, we run the full verification flow on our server. The server has two Intel Xeon Gold 6438Y+ CPUs, each with 32 cores that can reach up to 4 GHz. The cores support hyper-threading, therefore the server has 128 threads in total. The platform is equipped with with 2 TB of system memory, composed of 32x 64GB RAM operating at 4800 MHz. The server has a layered cache structure with 2560 KB of L1 cache, 64 MB of L2 cache, and 60 MB of L3 cache.

A list of the tools and the versions used for the evaluation is contained in Table 7.1. These tools are used for different steps of the verification flow, which are:

1. The model description has to be converted to source code. When using the Murφ model checker this is achieved using '`mu -b Model.m`'. This enables bit compaction for deterministic results and prioritization of small memory footprint over analysis time. For Rumur we use '`rumur Model.m -symmetry-reduction exhaustive -output checker.c`' to use similar symmetry reduction behaviour to Murφ.

2. The source code has to be compiled to an executable. For the Murφ generated code we use '`g++ -O3 *.cpp -I`*`ipath`*` -o checker`', where *`ipath`* points to the Murφ include files. For the Murφ generated code we use '`gcc -std=c11 -march=native -O1 checker.c -lpthread -mcx16 -o checker`'. The flags are recommeded by rumur, however due to long compile times we use '-O1' instead of '-O3', as suggested by gcc for machine generated code [4].

3. The executable has to perform the state space exploration. For Rumur this is done by just invoking the binary '`./checker`'. Murφ allocates the memory statically, therefore we use '`./checker -m 1945600`' to specify the number of MB that can be allocated for the model exploration.

We measure multiple things in this evaluation: Both the *explored States* and the individual *Statesize* are directly reported by both Murφ and Rumur upon completion of the exploration. Because Murφ statically allocates all the available memory ahead of the model exploration, it is challanging to accurately measure the memory consumption. Therefore we will estimate all the RAM requirements by computing *explored States* times *Statesize*. For our evaluation we found this to be a suitable estimate for the required RAM, as both Murφ and Rumur fail their exploration when less RAM is provided. We also collecting timing measurements for model checker and compilation, by using GNU Time 1.9.

Tool	Version
rumur	github [11] @tag v2024.07.14
mu (Murφ)	version included in HIERA-Gen [2]
gcc	gcc (GCC) 13.3.0
g++	g++ (GCC) 13.3.0

Table 7.1: Versions of the tools used for evaluation.

7.1.1 Baseline

For the baseline, which we will compare our results against, we use the publicly available artifacts for HIERA-Gen [9]. Their generator includes a similar hierarchical bridge that allows to connect different cache coherence domains. The generator also has a Murφ backend which generates deadlock tests for a setup with two clusters, connected by a bridge. The structure of the generated models is explained in Section 5.1.

The baseline, as is, does not consider the same target system as this thesis, but rather a simplified version. This was likely a decision to reduce the resource utilization of the baseline liveness check. However we do not want to blindly make the same assumptions as the related work. Particularly, we want to refrain from any assumptions on how the setup would behave when multiple bridges are introduced. So we are required to modify the generated Murφ model output of HIERA-Gen to fit our use case.

For this we replace the sole L2 cache with a second bridge and L1 cluster. Furthermore we increase the number of caches in each L1 cluster from 2 to 3, to match our target system. However when analyzing this model using the model checker we discovered two issues with this baseline:

1. the baseline bridge without any modifications is not guaranteed to be deadlock free and causes deadlocks when used in our target system.
2. the baseline model handles internal states inefficiently, causing an unnecessary state space explosion. This makes it impossible to analyze even a reduced target system with hardware supporting 1.9TB of RAM.

To make the baseline usable for our evaluation, we consequently had to modify it slightly. First we fixed the deadlock behaviour, resulting in a functional but sub-optimal bridge. The deadlocking situation, our modifications and the limitations of the modified bridge are discussed in Subsection 7.3.1.

Since the baseline is unusable without any optimizations, we performed an analysis in Subsection 7.3.2. We discovered a set of optimizations which are both minimal and resulting in great performance improvements. These optimizations still model the same system, they are just better suited for the model checker. We settled on a set of optimizations which we applied to the model to obtain the *optimized baseline*. Unless stated otherwise, we will use the *optimized baseline* to compare our work against.

7.2 Target System Evaluation

To evaluate the effectiveness of our compositional model checking approach, we compare the resource usage for verifying liveness of our target system model. The liveness check is performed both with the optimized baseline model as well as the set of checks, which make up the compositional approach and are explained in Section 6.4.

With this we want to evaluate whether our proposal for a compositional model checking approach can reduce the overall memory and time required to perform the verification.

RAM requirements

The optimized baseline requires 267.8 GB of RAM and takes 8 hours and 54 minutes to analyze on our platform. This is a resource requirement which at the time of writing this thesis can only be met by a dedicated server, as most personal use computers are not equipped with enough RAM to perform this analysis.

In comparison the list of checks performed to verify liveness compositionally, have much lower memory footprints. The most expensive individual compositional check is

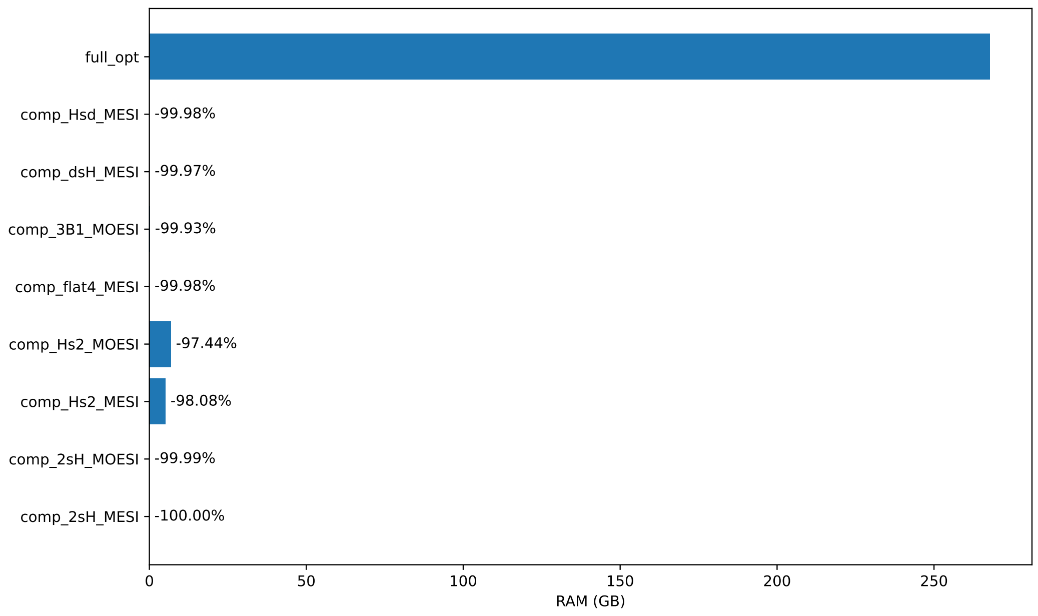


Figure 7.1: RAM usage for the full optimized liveness check (full_opt), as well as the checks making up the full compositional liveness check (comp_*) of the target system.

comp_Hs2_MOESI, requiring 6.9 GB of RAM. This check verifies that a bridge with a MOESI L1 cluster has an observable behaviour in the L2, which can be reproduced by a L2 MESI cache. Similarly the next most expensive individual compositional check is *comp_Hs2_MESI*, which performs the same check but for a MESI L1 cluster. The large memory footprint of these two checks stems from the large number of states that the L1 cluster can be in which are indistinguishable from the L2. The model checker needs to exhaustively explore all those states and ensure they all result in observable behaviour on the L2 which matches the behaviour of a cache.

Notably the checks for the inverse direction, i.e. ensuring that the observable behaviours of an L2 cache can be reproduced by a bridge with an L1 cluster, are much less memory intensive. This is because it is possible to restrict the L1 cluster exploration, as one L1 cache is sufficient to generate requests that allow the bridge to reproduce the observable behaviour.

The *comp_Hsd_MESI* and *comp_dsH_MESI* checks, which verify the equivalence in observable behaviour of the bridge and a modified directory from the perspective of the L1 are also not memory intensive. The likely cause is that the upper layer has much less states that equate the same observable behaviour of the bridge on the L1 cluster, since there is only one other L2 cache.

The remainder of the compositional checks *comp_flat4_MESI* and *comp_3B1_MOESI* are small models compared to the full compositional model. Even tough the check *comp_3B1_MOESI* shares more than 50% of the system that is modeled with the full compositional model, the required RAM and statespace is reduced by much more than 50%. This is also an implication of the exponential state space growth, where eliminating a few individual states in a compound state can achieve large total state space reduction.

This result shows that our compositional verification methodology can greatly reduce the memory requirements. Running the checks consecutively allows to show liveness of the target system with only 6.9 GB compared to the 267.8 GB of the baseline. This makes it feasible to perform the analysis on a typical personal computer instead of requiring a server.

Time requirements

Figure 7.2a shows a breakdown of the time required to perform a liveness verification for both the full optimized baseline model, as well as the set of checks for the compositional verification. The timings are split into the different stages of the model verification process according to the steps explained in Section 7.1.

This first figure shows, that the optimized baseline requires 8 hours and 54 minutes to analyze the target system on our platform. This high number is likely the result of the large state space which has to be explored, as well as the Murφ model checker only supporting single threaded exploration. Notably the Murφ model checker is able to produce source code from the model description, almost instantly, taking less than a second. The compilation of the Murφ generated code also completes comparatively fast, taking no more than 18 seconds on our platform. This trend can also be observed for the compositional checks which use Murφ. In particular these are the *comp_flat4_MESI* and *comp_3B1_MOESI* checks, which complete in less than 20 seconds total.

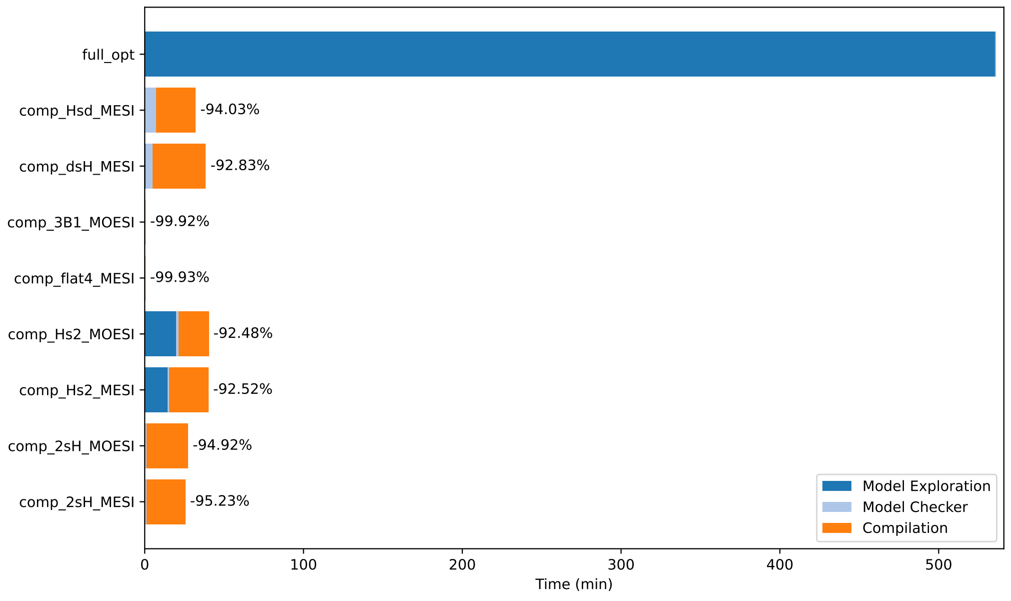
The checks which rely on Rumur, take considerably more time to both generate the code for the model as well as comping the generated code. The longest time spent on generating code from the model description is the 6 minutes 57 seconds of the *comp_Hsd_MESI* check. Compile times of the Rumur generated code on our platform, were even larger reaching up to 33 minutes and 19 seconds for the *comp_dsH_MESI* check.

Model exploration for the compositional model checks appears to be a significant factor only for the models which have a presumably large state space. For the liveness verification of the target system these are *comp_Hs2_MESI* and *comp_Hs2_MOESI*.

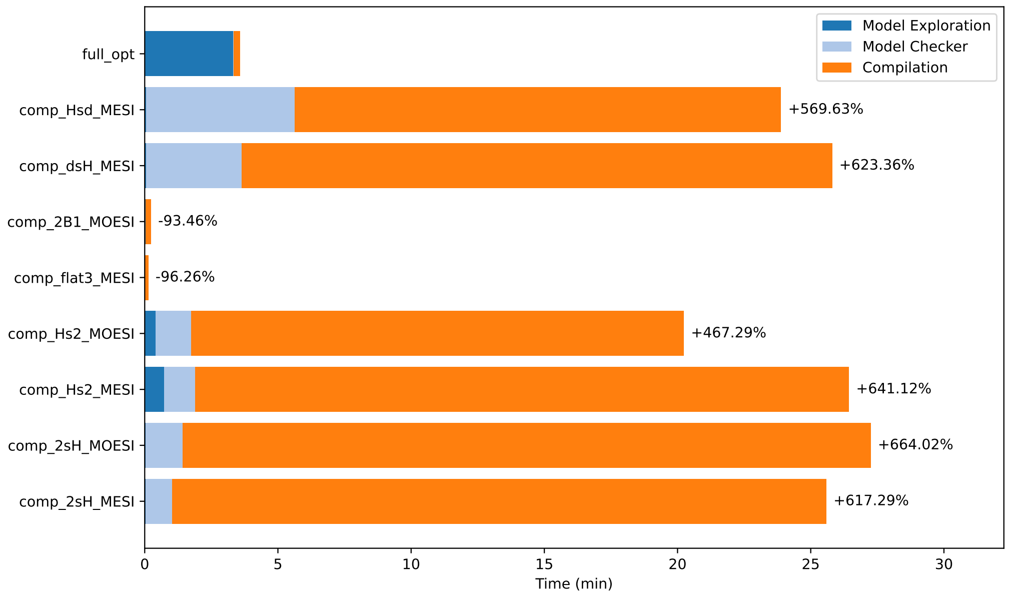
For the target system the overhead of the model checker and the compilation for the compositional checks still results in an overall gain compared to the full system evaluation. This indicates that the compositional verification of our target system takes less time, especially if the stages can be performed in parallel.

The model checker and the code compilation make up significant portion of the compositional verification Time. Unlike the model exploration, the effect of the model size on the model checker and compilation time is not as obvious. Therefore we evaluated a second *reduced target system*, which only has 2 CC in each L1 cluster. The results are shown in Figure 7.2b and clearly show that the model checker and compilation times of the compositional checks, which use Rumur, clearly overshadow the time for the complete verification of the full system. Even if all checks could be run in parallel, the compositional checks using Rumur take more than 6 times as long as the full system verification baseline.

We conclude from this experiment, that the compositional verification does not always have lower total verification times compared to a full system. Even tough further analysis is required, this small sample motivates the theory that compositional verification might only be beneficial to larger, resource intensive systems. For smaller systems the overhead introduced by model checker and code compilation might become the bottleneck. Although out of scope for this paper, future work can explore potential improvements to those areas as will be discussed in Section 8.1.



(a) Full Target System, with 3 CC in each L1 cluster.



(b) Reduced Target System, with 2 CC in each L1 cluster.

Figure 7.2: Time required for the full optimized liveness check (full_opt), as well as the checks making up the full compositional liveness check (comp_*).

7.3 Baseline Modifications

The unmodified baseline, consisting of the publicly available artifacts of HIERA-Gen [9], has two major issues which prevent it from being an adequate point of comparison. In particular the generated bridge has a deadlock, which becomes apparent when including it in our target system. Further, even after fixing this deadlock, the model is designed sub optimally for a model checker performing exhaustive state space exploration. In this part of the evaluation we first explain the deadlock of the baseline, argue why it was not caught before and propose a minimal fix for the bridge in Subsection 7.3.1. Subsection 7.3.2, explores the second issue, by exploring different potential optimizations for the model. These optimizations are evaluated for a set of smaller systems, to motivate which optimizations are used to generate the *optimized baseline*.

7.3.1 Deadlock in the Baseline Model

We discovered that the generated bridge models of HIERA-Gen, result in a deadlock when used in a setup like out target system. This occurred both in the full system as well as the compositional checks, so we decided to investigate and found a design flaw in the bridges as well as the Murφ model generator.

The problem arises when the bridge is in the IxS state. This state signals that no cache in the lower level has a valid entry for the cache line, i.e. they have no copies, while the bridge itself still has a shared access copy of the cache line. In this state the bridge refuses to handle any *getS* request from the lower level. This is likely an oversight in the generator as responding to the *getS* should be trivial, since the bridge already is in possession of the cache line.

This by itself is only a design flaw, which does not necessarily impact the liveness of the system as long as there is some way for the bridge to exit the IxS state such that it can handle the pending *getS* request. However the bridge only has three potential ways to leave the IxS state: (1) when it has to evict the cache line (2) when it has to store the cache line (3) when it receives an invalidation message on the upper layer.

However the generated model of the bridge did not allow for the bridge to decide it has to evict or store a cache line. The model only considers these actions in response the processing of a L1 message. However since the head of the request queue is blocked by a *getS* request, no other L1 message are processed. The only remaining way for the bridge to leave the IxS state is to receive an invalidation message on the L2. In a system with only bridges in the L2, it can happen that all of these bridges enter the IxS state, awaiting an invalidation message. However since invalidation messages are sent in response to a L2 request, there is no way for them to be sent in this state. Therefore such a system is in a deadlock.

Why is this not caught by the HIERA-Gen liveness verification?

The HIERA-Gen liveness verification only models a system with a second CC in the L2 instead of a full system. They implicitly assume that a their bridge has the same observable behaviour as a cache. However this assumption is not true, as the bridge can not exit the IxS state on it's own, whereas a cache can evict a cache line in S state.

How can the deadlock be avoided?

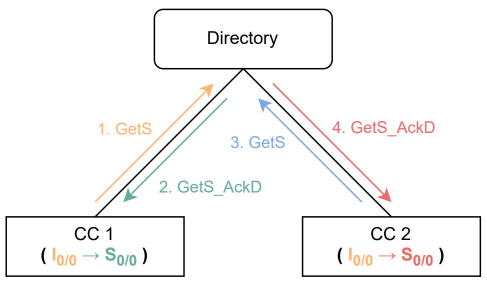
The proper way to resolve the deadlock is to modify the bridge such that in can reply to a *getS* request while it is in the IxS state. However the design of the bridge and it's transition tables is out of scope for this thesis, so instead we modified the model to allow the bridge to autonomously decide on evicting a cache line when in the IxS state.

This is results in a suboptimal bridge as it could happen, that the bridge receives a *getS* while in IxS and not yet having evicted the cache line. Instead of responding the *getS* directly the bridge now first evicts the cache line, before requesting it again from the directory to process the *getS* request. We use this bridge version for all evaluation in this chapter.

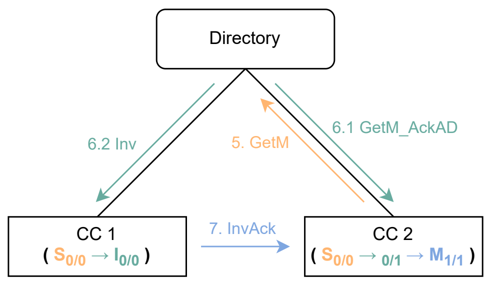
7.3.2 Inefficient Baseline Model

The unaltered baseline model implements the system's components in such a way, that internal states are reset when necessary, instead of when possible. A concrete example of this is the acknowledgement counting, as depicted in Figure 7.3: When a cache makes a request which forces other caches to invalidate the cache line, they will send invalidation acknowledgements to the requesting cache. The requesting cache waits until all acknowledgements are received before proceeding. The caches keep an internal counter to count the received acknowledgements. The baseline model does not reset this counter once the cache received all acknowledgements. Instead it is reset when the cache is starting to wait for new invalidation acknowledgements again.

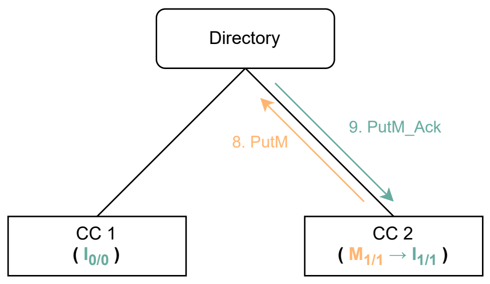
This is a valid implementation, as the counter value is unused and effectively in a do-not-care state in-between acknowledgement counting operations. However, the model checker considers the whole system state, when performing a state space exploration. This included all internal states of the components. This implies that the model checker takes these different counter values into account; leading to a larger state space. A concrete example of this is shown in Figure 7.3, the system state at the end of the sequence is the same as the state at the start of the sequence, only differing in the internal counter of the cache. This leads to a redundant exploration of states.


Directory
1. GetS
2. GetS_AckD
3. GetS
4. GetS_AckD
CC 1
($I_{0/0} \rightarrow S_{0/0}$)
CC 2
($I_{0/0} \rightarrow S_{0/0}$)

(a) Both caches request and receive shared access to a cache line.


Directory
6.2 Inv
5. GetM
6.1 GetM_AckAD
7. InvAck
CC 1
($S_{0/0} \rightarrow I_{0/0}$)
CC 2
($S_{0/0} \rightarrow {}_{0/1} \rightarrow M_{1/1}$)

(b) The second cache upgrades to write access, invalidating the other cache.



(c) After evicting the cache line, the internal state of the second cache still retains the acknowledgement information.

Figure 7.3: Example flow that highlights how the unoptimized baseline handles the acknowledgement information.

Based on these findings, we designed a set of potential optimizations to the model. Because we had to restrict the scope of this thesis we refrained from performing extensive optimization of the model generator itself. Future work could benefit from such a modification. Because all evaluation in this chapter uses the modified versions of the baseline model, both the baseline as well as the evaluated models in this chapter will benefit from the optimizations.

The potential optimizations we explored are:

- **AckReset**: This optimization attempts to eliminate the inefficiency that was previously explained and is shown in Figure 7.3. This is achieved by modifying the finite state machine of the component, such that both `acksRecieved` and `acksExpected` are set to 0 if they are equal.
- **AckDiff**: This optimization improves on AckReset, by using the property that the

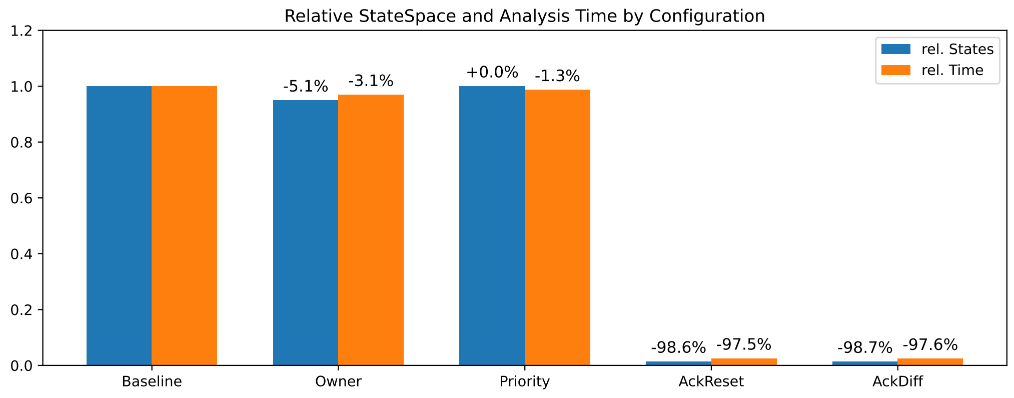
state machines only take the difference between `acksRecieved` and `acksExpected` into account. Therefore only the difference is stored. However, since it is technically possible, according to the state machines of the components, to receive an acknowledgement before knowing how many acknowledgements are expected, a separate variable is used to track a "negative difference".

- **Owner**: We noticed that the inefficient clearing of states also occured for directories. In particular we observed that a cache line in the directory I state still had a reference in it's (previous) owner. This optimization removes this reference whenever transitioning into the I state.
- **Priority**: We observed that the rules used to model the processing of the network queues, technically allowed them to be processed in any order, as long as the component state machine accepts the message. This on it's own is not problematic, as long as the state machines do not accept messages such that the prioritization of the network queues would be violated. Nonetheless, we created an optimization for the model which enforces the queue prioritization at the level of the network rules.

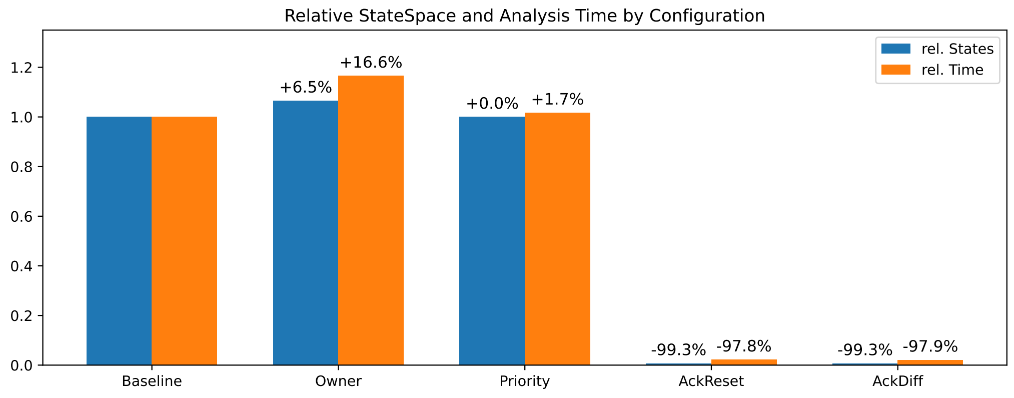
To evaluate the effect of these optimizations we first perform a set of micro benchmarks to determine their potential before applying them to the target system for evaluation. The first set of micro benchmarks are performed for the liveness check of a flat MESI protocol with 3 CC. The results of which are shown in Figure 7.4a. The second benchmarks depicted in Figure 7.4b are obtained by applying the optimization to a simple hierarchical setup. This hierarchical setup contains a MESIxMESI bridge, two MESI CC in the L1 and one MESI CC in the L2.

Figure 7.4 shows that the relative improvements in terms of explored states and verification time are most pronounced for the AckReset and AckDiff optimizations. These achieve state space reductions of more than 98% for the flat model and more than 99% for the hierarchical model. The verification time for models using those optimizations is also reduced by more than 97%, likely due to the smaller state space that needs to be explored.

The effects of the other optimizations for the flat model are negligible, with the queue prioritization not causing any change in the explored state space. This is an indicator that the state machines of the components properly realize the prioritization of the queues, such that it is not required to be enforced as network rules. The same is true for the Priority optimization applied to the hierarchical model. The Owner optimization has a negative impact on both state space and verification time when applied to the hierarchical model. This is likely because resetting the owner only when entering the I state might be too conservative. We suspect that the owner state is only cleared in a



(a) Effect of optimizations on flat model with three MESI CC.



(b) Effect of optimizations on hierarchical model with two L1 MESI CC, a MESIxMSI Hiera bridge and one L2 MSI CC.

Figure 7.4: The effect of different optimizations on a set of models.

subsection of cases where it could be cleared, by using this optimizations. This could lead to partially cleared states, which have the potential to increase state space as they effectively are newly introduced states.

Based on these results we conclude that the optimizations to the acknowledgement management are the most promising, hence we apply them to all models to obtain the *optimized baseline*. To verify how the optimizations scale with bigger systems we compare both the unmodified baseline and the optimized baseline in terms of required RAM in Figure 7.5. The figure shows the RAM usage of verifying liveness for a flat MESI protocol, with the number of CC on the x-axis. It shows how the optimized baseline grows exponentially in memory footprint with more CC. The unmodified baseline's memory footprint grows more drastically, exploding when introducing a fourth CC. It appears the optimized baseline is not just a constant factor improvement, but actually appears to have greater benefit for larger systems. This aligns with the observation, that the number of distinct values of the acknowledgement counter directly depends on the number of CC in the cluster. So for n caches this corresponds to n values ranging from 0 to $n-1$.

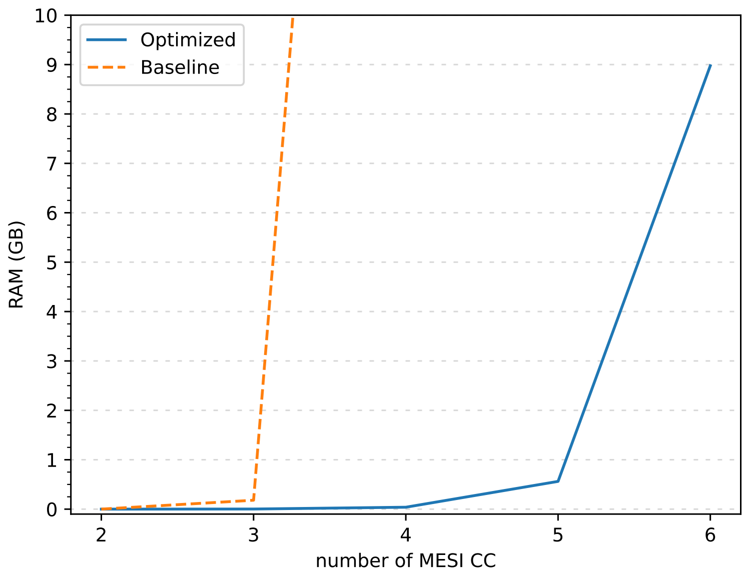


Figure 7.5: RAM usage for a flat MESI setup using the baseline models.

Finally, when the optimizations are applied to the target system we were able to analyze them on our evaluation server. The full target system check with 2 CC in each L1 cluster previously failed after 42 hours due exceeding the allotted 1.9 TB of RAM.

Using the Optimizations the same model can run using only 2.6 GB of RAM. Similarly the target system with 3 CC in each L1 cluster, which was previously failing as well, now completes using 267.8 GB of RAM.

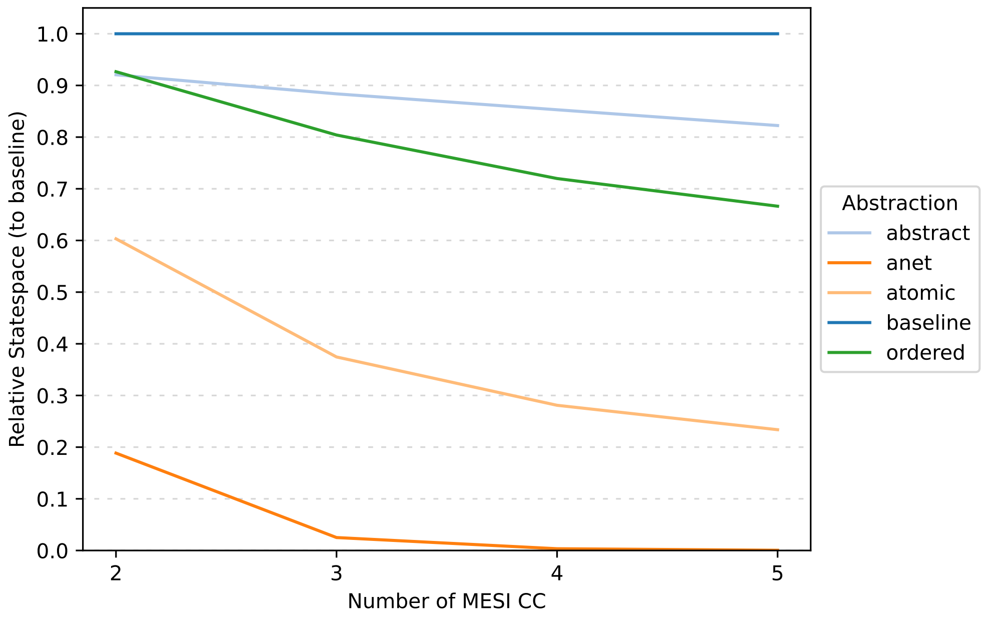
7.4 Abstractions

Section 5.2 introduces a set of potential methods to find abstractions for performing compositional verification. This allows to use the equivalence check to determine which abstraction is equivalent to the full system and then choosing the one with the least state space for maximum reduction in the compositional checks. Performing an equivalence check can be expensive especially if not many abstractions are applied. This is due to equivalence check performing an exhaustive state space exploration of the LHS. We therefore want to answer which of the abstractions from Section 5.2 provide the largest state space reduction and should thus be prioritized for trying to show equivalence.

Figure 7.6 shows the results of a set of micro benchmarks, where different abstractions are applied individually to a flat MESI protocol with the number of CC on the x-axis. The y-axis shows the explored state space of the abstractions compared to the baseline, i.e. the full system.

The figure shows that the atomic network (anet) can achieve the greatest reduction in state space. The relative reduction increases with larger systems indicating that the state space of models using the abstract network grows slower in those cases than the baseline. If the atomic network is checked early for equivalent observable behaviour to the full system it can significantly reduce the cost of further checks. Using an set of atomic components (atomic) is the next most beneficial abstraction to reduce the state space, it might be useful when a full atomic network creates an oversimplification of the system. Both abstracting components (abstract) and introducing an ordering (ordered) only have small effect on the state space reduction, both reducing the state space of two caches by less than 10%, as the system grows ordering appears to achieve a greater reduction than abstracting components.

Overall this shows what order of magnitude in reduction can be expected from these presented abstractions.



(a) Comparison of abstractions introduced to optimized baseline.

Figure 7.6: Comparison of reachable state space for different abstractions in a flat setup with one directory and a varying number of CC, which all employ the abstraction.

8 Conclusion

This thesis presents a compositional model checking approach, that allows to verify the bridging between heterogeneous cache coherence protocols.

We show the limitations of the model checking employed by previous work, both in terms of resource utilization and correctness. We propose concrete fixed to the deadlock, as well as a number of optimizations for their model checking approach.

As an alternative we present our compositional approach, that does not suffer from the same limitations. We achieve this by decomposing the full system at the bridges, using abstractions and arguing how they can be used as a substitution for subsystems. We show how to analyze these subsystems individually and deduct properties for the full system, with the example of a liveness verification.

Using our compositional approach, we are able to greatly reduce the required resources for performing a liveness check. For our target system, we are able to reduce the RAM required for the liveness analysis from 268 GB to 7 GB. With this 97% reduction in resource usage, we believe it to be feasible to verify liveness of the target system on a ordinary laptop instead, which was not the case before.

8.1 Future Work

The work presented in this thesis is only a partial result. There are still many directions that need to be further explored and evaluated to properly understand and quantify the benefits of compositional model checking for heterogeneous cache coherence protocols for CXL.

Some of the areas we believe could be future directions are:

- Optimizing the compilation times of the Rumur models. The evaluation in Section 7.2 shows that the compilation time of Rumur is largely contributing to the total execution time of the compositional checks. This is due to Rumur generating a single large machine generated code file containing all the information about the model. A simple first optimization step could explore splitting this file up to parallelize the compilation. More in depth analysis could determine what features in the model contribute to the increased compilation times.

- Further evaluation is required to create a better comparison between Rumur and Murφ. In particular evaluation exploring both model checkers in terms of memory consumption and exploration time. Further, this evaluation could also measure the impact of our mappings on the resource utilization.
- The baseline optimizations, which are performed in this thesis, are minimal and do not make any assumptions about the generator of the models. A more detailed approach, which takes the internal representation of the generator into account might be able to evaluate information like the end-of-life of variables more clearly. This allows to create optimizations that reset the system state at all correct occurrences.
- Eliminating the assumption of an atomic bridge. The reasoning in Section 5.4, relies on the assumption of the atomic behaviour of the bridge. However, this limits the current approach to a subsection of models, which realize this behaviour. Finding a way to relax this assumption, e.g., by expanding the notion of observable behaviour, could allow for a more general applicability of the presented compositional approach.
- The flaw found in HIERA-Gen is currently only fixed using evictions, the underlying state transition table is still flawed as it does not handle *getS* requests properly. Future work, might consider exploring the reason for this missing operation in the state transition table of HIERA-Gen and thus fix the generator.
- The compositional verification has not been applied to larger systems yet, it is not yet clear from the experiments in this thesis how the compositional approach scales for larger systems. Future work might include evaluating this larger systems, potentially consisting of more clusters to derive a trend for the resources required. An interesting direction is also the exploration of ways to potentially parameterize the number of cache controllers in a cluster for the compositional checks. This would open up the possibility to proof properties for an arbitrary number of caches and achieving better horizontal scaling of the verification method.

Abbreviations

L1 First Layer

L2 Second Layer

CC Cache Controller

CXL Compute Express Link

RAM Random-Access Memory

WLOG without loss of generality

LHS left hand side

RHS right hand side

MPSC multiple-producer single-consumer

NUMA non-uniform memory access

SWMR single writer multiple reader

SSP stable state protocol

List of Figures

List of Tables

Bibliography

[1] Errare-humanum-est. *Hieragen, https://github.com/Errare-humanum-est/HieraGen*.

[2] Errare-humanum-est. *Hieragen/Murphi · Errare-Humanum-Est/Hieragen, https://github.com/Errare-humanum-est/HieraGen/tree/master/Murphi*.

[3] S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. "Modelling the ARMv8 architecture, operationally: Concurrency and ISA." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 608–621.

[4] GNU. *Optimize options (using the GNU compiler collection (GCC)), https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html*.

[5] A. Goens, S. Chakraborty, S. Sarkar, S. Agarwal, N. Oswald, and V. Nagarajan. "Compound memory models." In: *Proceedings of the ACM on Programming Languages* 7.PLDI (2023), pp. 1145–1168.

[6] F. Kordon, P. Bouvier, H. Garavel, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, D. Donatelli, S. Dal Zilio, P. Jensen, L. Jezequel, C. He, S. Li, E. Paviot-Adet, J. Srba, and Y. Thierry-Mieg. *Complete Results for the 2022 Edition of the Model Checking Contest*. http://mcc.lip6.fr/2022/results.php. June 2022.

[7] F. Kordon, P. Bouvier, H. Garavel, F. Hulin-Hubard, N. Amat., E. Amparore, B. Berthomieu, D. Donatelli, S. Dal Zilio, P. Jensen, L. Jezequel, E. Paviot-Adet, J. Srba, and Y. Thierry-Mieg. *Complete Results for the 2023 Edition of the Model Checking Contest*. https://mcc.lip6.fr/2023/results.php. Apr. 2023.

[8] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi. "ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures." In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 388–400.

[9] N. Oswald, V. Nagarajan, and D. J. Sorin. "Hieragen: Automated generation of concurrent, hierarchical cache coherence protocols." In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (May 2020), pp. 888–899. DOI: `10.1109/isca45697.2020.00077`.

[10] N. Oswald, V. Nagarajan, D. J. Sorin, V. Gavrielatos, T. Olausson, and R. Carr. "Heterogen: Automatic synthesis of heterogeneous cache coherence protocols." In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* 1102 (Apr. 2022), pp. 756–771. DOI: `10.1109/hpca53966.2022.00061`.

[11] Smattr. *SMATTR/Rumur: Yet another model Checker, https://github.com/Smattr/rumur/tree/v2024.07.14*.

[12] D. Sorin, M. Hill, and D. Wood. "6.4.1.1 Stable States." In: *A Primer on memory consistency and cache coherence Daniel Sorin, Mark Hill, David Wood*. Morgan & Claypool Publishers, 2012, pp. 89–91.

[13] D. Sorin, M. Hill, and D. Wood. "7. Snooping Coherence Protocols." In: *A Primer on memory consistency and cache coherence Daniel Sorin, Mark Hill, David Wood*. Morgan & Claypool Publishers, 2012, pp. 99–138.

[14] D. Sorin, M. Hill, and D. Wood. "8. Directory Coherence Protocols." In: *A Primer on memory consistency and cache coherence Daniel Sorin, Mark Hill, David Wood*. Morgan & Claypool Publishers, 2012, pp. 139–175.