HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols

Nicolai Oswald
The University of Edinburgh
nicolai.oswald@ed.ac.uk

Vijay Nagarajan
The University of Edinburgh
vijay.nagarajan@ed.ac.uk

Daniel J. Sorin
Duke University
sorin@ee.duke.edu

Vasilis Gavrielatos
The University of Edinburgh
Vasilis.Gavrielatos@ed.ac.uk

Theo Olausson
The University of Edinburgh
theoxo@mit.edu

Reece Carr
The University of Edinburgh
mail@reececarr.com

***Abstract*—We solve the two challenges architects face when designing heterogeneous processors with cache coherent shared memory. First, we develop an automated tool, called HeteroGen, for composing clusters of cores, each with its own coherence protocol. Second, we show that the output of HeteroGen adheres to a precisely defined memory consistency model that we call a compound consistency model. For a wide variety of protocols—including the MOESI variants, as well as those that are targeted towards Total Store Order and Release Consistency—we show that HeteroGen can correctly fuse them. To validate HeteroGen, we develop the first litmus tests for verifying that heterogeneous protocols satisfy compound consistency models. To understand the possible performance implications of automatic protocol generation, we compared against a publicly available manually-generated heterogeneous protocol. Our results show that performance is comparable.**

I. Introduction

There are two trends in modern processor design that inspire our work: processor core heterogeneity and cache coherent shared memory. Heterogeneity, which was once largely confined to CPU/GPU designs, has expanded to encompass a much wider range of core designs. Because modern processors are power constrained, there is increasing motivation to design special-purpose cores (i.e., accelerators) for important tasks, because these cores can be more power-efficient and performant than general-purpose CPU cores. Current chips from Apple, Qualcomm, and Samsung have many different cores, including CPUs, GPUs, digital signal processors (DSPs), and camera processing cores.

In addition to heterogeneity, our other motivating design trend is the continued reliance on cache coherent shared memory. In the early days of CPU/GPU designs, the CPU cores did not share memory with the GPU cores. Even when shared memory emerged and then became prevalent due to its popular programming model, conventional wisdom suggested that cache coherence was infeasible due to scalability issues. Nevertheless, hardware cache coherence—often with accelerator-specific protocol features —is highly desirable for programmability, and it has become prevalent in heterogeneous processors. Indeed, cache coherent shared memory has become codified in several standardized design frameworks, including HSA [19], CCIX [3], OpenCAPI [5], Gen-Z [4], AMBA CHI [2], and CXL [1].

Architects face two challenges when designing heterogeneous processors with cache coherent shared memory. First, designing heterogeneous coherence protocols is difficult. Designing a coherence protocol for a *homogeneous* processor is already a notoriously challenging task [7], [29], and introducing heterogeneity multiplies the design complexity. This is because communication patterns within a CPU, GPU or an accelerator are very different, often mandating bespoke coherence protocols for each case. Conventional protocols that use writer-initiated invalidations for enforcing the Single-Writer-Multiple-Reader (SWMR) invariant work well for CPUs. But they are ill-suited for GPUs [37], which tend to employ self-invalidating protocols that directly enforce relaxed consistency models instead of SWMR [27]. To compose these very different protocols into a unified heterogeneous whole requires designing a bridge between them. While recent academic [11], [28] and industrial works [2], [4], [5] have developed interfaces or wrappers that facilitate this process, it is still quite challenging to manually compose the protocols.

The second challenge faced by architects is reasoning about the memory consistency model provided by the heterogeneous processor. Recall that a memory consistency model defines the software-visible orderings of loads and stores across all of the threads in a shared memory system [27]. Consider a processor that consists of several clusters of cores, each with its own per-cluster coherence protocol and consistency model. Now assume that we have overcome the first challenge of composing the cluster-level protocols together into a single protocol. What consistency model does this heterogeneous processor provide? How does one ensure that the composed protocol adheres to the intended consistency model?

To overcome these two challenges—design complexity and consistency model—we have developed HeteroGen. HeteroGen is a tool for automatically generating heteroge-

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

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

How does HeteroGen fuse the protocols? The key idea is that when a store from a processor core is made globally visible within one of the clusters, HeteroGen leverages the coherence protocols of the other clusters to make the store visible in the other clusters as well. In doing so, HeteroGen preserves the relative ordering of two stores from a core within a cluster in the other clusters as well – thereby ensuring the compound consistency model.

We have used HeteroGen to generate several heterogeneous protocols, using it to combine a range of protocols including the MOESI variants that enforce SWMR and self-invalidation based protocols that directly enforce the consistency model. We have validated that all of the protocols generated by HeteroGen satisfy their compound consistency models, and that they are deadlock-free. For consistency validation, we create litmus tests [38], [42] that are specific to compound consistency models.

We experimentally evaluate HeteroGen against a manually-designed heterogeneous protocol called HCC [40]; this is a publicly-available protocol that is similar to Spandex [11], running on a heterogeneous multicore system with 60 "tiny" in-order CPU cores and 4 "big" out-of-order CPU cores, with the big cores employing MESI and the tiny cores employing a variant of DeNovo. Our experiments reveal that the performance of our automatically-generated protocol is comparable to the manually-generated HCC.

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

Contributions. We make the following contributions:

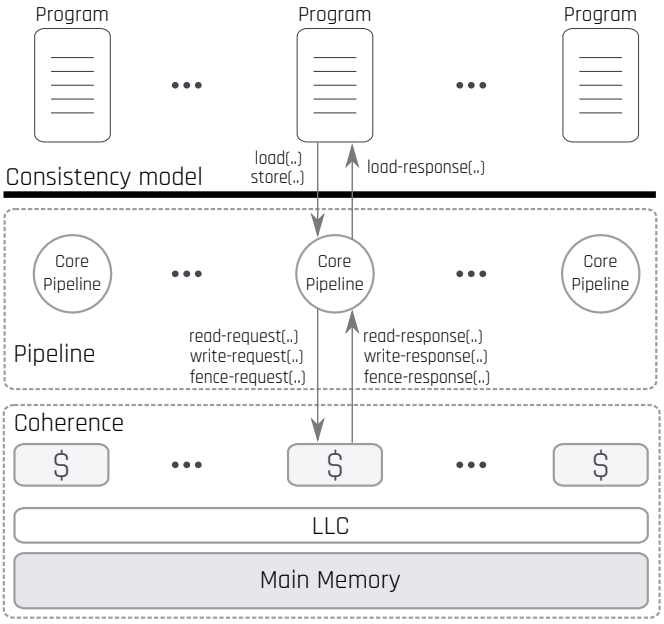

Figure 1. Normally the consistency model enforced by the processor is a function of both the processor pipeline and the coherence protocol combined. We isolate the consistency model enforced by a coherence protocol as the model that is enforced when an in-order pipeline (that issues memory operations one by one) is combined with that coherence protocol.

- We formalize compound memory consistency models, a compositional approach to specifying consistency models targeted towards heterogeneous systems.
- We develop and distribute HeteroGen, the first automated tool for composing heterogeneous coherence protocols.
- We demonstrate using heterogeneous litmus testing that HeteroGen produces protocols that satisfy the intended compound consistency models.
- We experimentally show that the performance and network traffic of our automatically-generated protocol is comparable to a manually-generated heterogeneous protocol.

II. Background

A. The Memory Consistency Model

The hardware memory consistency model is part of the instruction set specification and specifies how memory must appear to the (systems) programmer [27]. As such, all major commercial (homogeneous) processors support precisely defined consistency models that are specified as part of the ISA specification. For example, all processors from Intel and AMD support the x86-TSO consistency model [35], whereas ARM [12] and RISC-V [41] processors support more relaxed models.

An important feature of memory models concerns the manner in which stores propagate their values to other processors. In *multi-copy atomic* memory models [27], store values propagate atomically: as soon as the value becomes visible to another processor, no future load (in logical

time) can access an earlier value. It is worth noting that a number of processor vendors and commercial architectures (including x86, ARMv8, and RISC-V) support multi-copy atomic models, and in this work we restrict our attention to such memory models.

Another memory model feature, which pertains to GPUs and existing heterogeneous memory models, is the notion of *scopes* [18]. While scopes are an important feature in today's GPU memory models [23], whether or not future heterogeneous consistency models should involve scopes is still under debate [36]. In this work we limit our attention to memory models without the notion of scopes.

B. The Coherence Interface

The processor cores interact with the coherence protocol through an interface consisting of reads, writes, and fences [27]. A read request takes in a memory location as the parameter and returns a cache block. A write request takes in a memory location and a value (to be written) as parameters and returns an acknowledgment.

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

The hardware memory model enforced is a function of both the processor pipeline and the coherence protocol, as shown in Figure 1. In this work, however, we want to be able to reason about composing together heterogeneous coherence protocols. Therefore, we want to be able to isolate the consistency effects of the coherence interface. We define it as follows.

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

Thus, a conventional writer-initiated SWMR-enforcing protocol is said to enforce sequential consistency (SC). Consequently such a protocol is associated with SC-read-requests and SC-write-requests. Protocols such as TSO-CC [16] and Racer [34], that are designed to target TSO, are said to enforce TSO, and are hence associated with TSO-read-requests and TSO-write-requests. In a similar vein, protocols that target variants of release consistency (RC), such as lazy release consistency [10], are said to enforce RC. Consequently the coherence interface involves two types of writes (release-write-requests and data-write requests) and two types of reads (acquire-read-requests and data-read-requests).

[1]Although reasoning about an in-order pipeline is easiest, we do not require an in-order pipeline, just a pipeline that is compatible with the consistency model [22].

III. Related Work

A. Automatic Protocol Generation

The most related prior work is by Oswald et al. [29], [30], who have developed schemes for automating the generation of flat and hierarchical protocols. The hierarchical protocols could be heterogeneous, but only insofar as the protocols obey the single-writer, multiple-reader (SWMR) invariant; a key feature of HeteroGen is its applicability to non-SWMR protocols (e.g., DeNovo [14], and TSO-CC [16]).

B. Heterogeneous Coherence Protocols

One industrial approach to heterogeneous coherence has been the development of protocol standards such as HSA [19], CAPI [5], CCIX [3], CHI [2], Gen-Z [4], and CXL [1]. Crossing Guard [28] proposes a similar coherence interface between the CPU and accelerators whereas hUVM [21] proposes a unified protocol based on the VIPS [33] line of work. To provide coherence between cores and accelerators that may have very different interfaces to the memory system, Alsop et al. [11] developed the flexible Spandex coherence interface. None of this prior work takes existing protocols and automatically integrates them.

C. Consistency for Heterogeneous Processors

Hower et al. [18] introduce heterogeneous race free (HRF) memory models that accommodate synchronization operations with different scopes, but HRF does not address the composition of different protocols with different constituent consistency models. Extending the definition of compound consistency models to accommodate scopes is future work.

Nagarajan et al. [27] briefly discuss heterogeneous consistency models in their primer (Section 10.2.1). They introduce the concept of a compound consistency model and the associated litmus tests informally but they do not formalize it; nor do they provide a general method for fusing two protocols.

In concurrent work, Iorga et al. [20] formally specify the memory model of heterogeneous CPU/FPGA systems axiomatically as well as operationally, and validate their models. In contrast, in this work we specify more generally how different memory models can be composed together, and how coherence protocols can be automatically fused to match our specification.

Batty [13], in his position paper, argues for a compositional approach towards relaxed memory consistency. Our compound consistency models, and specifically the fact that they preserve compiler mappings within each cluster, are a step in this direction.

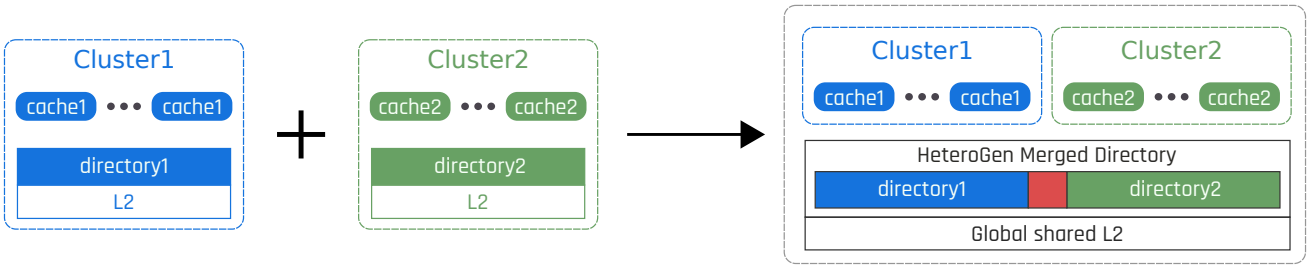

Figure 2. HeteroGen takes as its inputs the coherence protocols of the individual clusters (Cluster1 and Cluster2) and automatically merges their directory controllers to produce the merged directory. The target system model of HeteroGen consists of multiple clusters of cores with their private L1 caches and global shared L2.

D. Litmus Testing

Litmus testing is a longstanding approach to validating whether a system correctly implements its specified memory consistency model [38], [42]. Litmus tests are small code snippets that are crafted, perhaps with the aid of automation [8], [25], to expose behaviors that distinguish different consistency models [26]. Because no prior work has explored compound consistency models, there has also been no prior work in litmus test development for them.

E. Memory model translation

ArMOR [24] is a framework for specifying and translating between memory consistency models. For example, ArMOR has been used previously to automatically generate translation modules for dynamically translating code compiled for one memory model on hardware that enforces another. In this work, we leverage the ArMOR framework in a novel way: to compose different coherence protocols.

IV. System Model, Assumptions, Limitations, and Problem Statement

Throughout this work, we assume a heterogeneous shared memory computer that consists of multiple clusters of cores, as illustrated in Figure 2. At a high level, HeteroGen takes as input the individual clusters (i.e., cluster1 and cluster2) with their cluster-specific coherence protocols, each enforcing its cluster-specific consistency model, and automatically produces a global protocol that enforces the compound consistency model. (We will define and discuss compound memory models in the next section.) The input protocols are specified using the domain specific language used by ProtoGen [29], and the output finite state machines are in the language of the Murphi model checker, to facilitate using Murphi [15] to validate the protocols.

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

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

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

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

With the system model fleshed out, we are now in a position to precisely describe the problem. Given a set of clusters, each with a distinct directory coherence protocol enforcing a distinct consistency model, how do we automatically merge the individual directory controllers into one global directory controller, as shown in Figure 2?

V. Compound Consistency Models

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

A. Intuition

In this paper, we propose a solution to these questions: a compositional approach to heterogeneous consistency called compound memory consistency models. We define

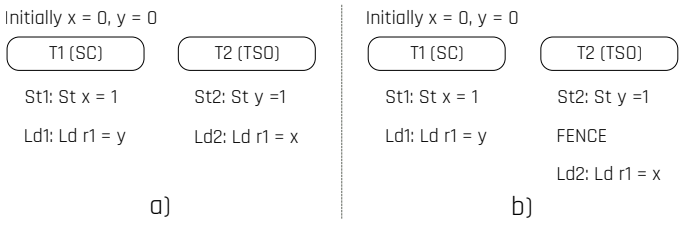

Figure 3. a) Ld1 and Ld2 can both return 0, b) only one of Ld1, Ld2 can return 0

compound consistency as follows. Consider a heterogeneous computer with n clusters, C_1 to C_n, each with its own per-cluster coherence protocol that enforces a per-cluster consistency model M_i. When we combine the clusters into a heterogeneous processor, the compound consistency model guarantees that operations from each cluster C_i continue to adhere to its per-cluster consistency model M_i.

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

Consider the Dekker's litmus test shown in Figure 3(a), which shows thread T1 from the SC cluster and thread T2 from the TSO cluster. For this example, note that it is possible for both Ld1 and Ld2 to read zeroes. This is because the TSO cluster does not enforce the St2 $\rightarrow$ Ld2 ordering, even though the SC cluster enforces the St1 $\rightarrow$ Ld1 ordering.

However, as shown in Figure 3(b), once a FENCE instruction is inserted between St2 and Ld2, the two loads cannot both read zeroes anymore. Note, however, that a FENCE instruction is not required between St1 and Ld1 from T1 because the SC cluster already guarantees this ordering.

B. Formalism

In this section, we formalize the notion of compound consistency models. Starting with the axiomatic framework of Alglave et al. [9], which can capture any multi-copy atomic model [9], we then formally define the compound consistency model enforced when combining a set of given multi-copy atomic models.

Preliminaries. We start by defining some basic relations.

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

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

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

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

$$acyclic(\xrightarrow{po-addr} U \xrightarrow{rf} U \xrightarrow{fr} U \xrightarrow{ws}) \tag{1}$$

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

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

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

$$acyclic(\xrightarrow{ppo} U \xrightarrow{rfe} U \xrightarrow{fr} U \xrightarrow{ws}) \tag{2}$$

For example:

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

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

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

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

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

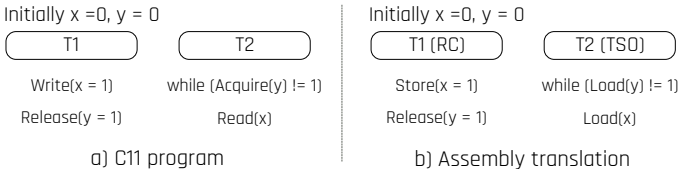

Figure 4. a) The producer-consumer pattern programmed in C11 for a heterogeneous system, b) The C11 program gets compiled for an RC/TSO system; in the RC system the C11 release gets compiled into a release, while in the TSO system the C11 acquire gets compiled into a load.

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

$$acyclic(\xrightarrow{ppo_{com}} U \xrightarrow{rfe} U \xrightarrow{fr} U \xrightarrow{ws}) \quad (3)$$

C. Example

Let us go back to Figure 3(b) which shows the Dekker's litmus test, with thread T1 from the SC cluster and T2 from the TSO cluster. The following sequence of edges:

$$St1 \xrightarrow{ppo} Ld1 \xrightarrow{fr} St2 \xrightarrow{ppo} Ld2 \quad (4)$$

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

D. Programming with Compound Consistency

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

We do not necessarily advocate for programmers to program against the low-level compound consistency model. In fact, we argue that compound consistency makes it easy to support language-level consistency models on the heterogeneous computer. One of the key challenges in supporting a new hardware memory model is to discover correct compiler mappings from language-level atomics to that memory model. Fortunately, with compound consistency models there is no need to discover new mappings. When compiling language-level atomics down to the compound consistency model, depending on where (i.e., which cluster) a thread is mapped to, the existing compiler mappings for that cluster's memory model can be used.

We illustrate this with an example for a compound consistency model consisting of two models: Release Consistency (RC) and TSO. Let us consider a producer-consumer pattern expressed in a language-level consistency model such as C, as shown in Figure 4. Note that there are two language-level atomics here: the release on the producer side and an acquire on the consumer side. Further, let us assume that the producer thread is mapped to the RC cluster and the consumer is mapped to the TSO cluster. The producer thread uses the compiler mapping for a C11 release on RC (which is a release store) while the consumer thread uses the compiler mapping for a C11 acquire on TSO (which is a normal TSO load).

VI. HeteroGen

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

A. What does HeteroGen do?

At a high level, HeteroGen performs the integration illustrated in Figure 2. Given two distinct directory coherence protocols, each of which enforces a potentially distinct consistency model, HeteroGen produces a single heterogeneous protocol.

HeteroGen does this by merging the two directories into one single merged directory, while leaving the cache controllers unchanged. The merged directory presents a directory1-like interface to the caches of type cache1 and a directory2-like interface to the caches of type cache2. From the point of view of cluster1 (i.e., directory1 and its caches), cluster2 behaves as if it were a single cache1. Similarly, cluster2 views cluster1 as if it were a single cache2.

Within the merged directory there is bridging logic, such that a request from cache1 has the appropriate impact on caches of type cache2 (and vice versa). There are two logical aspects to bridging between the protocols: proxy caches [30] and consistency model translation [24]. We will explain in Section VI-C how these work together. But before that we will explore what compound consistency means operationally.

B. Operational Intuition

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

One way to specify memory models is via abstract state machines that exhibit the memory model's behaviors. For example, SC can be expressed as a bunch of in-order processors connected via a switch to an atomic memory. If a FIFO store buffer [35] and/or a load buffer [6] is introduced between each processor and the memory, we get TSO. In general, any multi-copy atomic memory model can be expressed as processors with local buffers connected to atomic memory, with each memory model having its unique buffering logic [31].

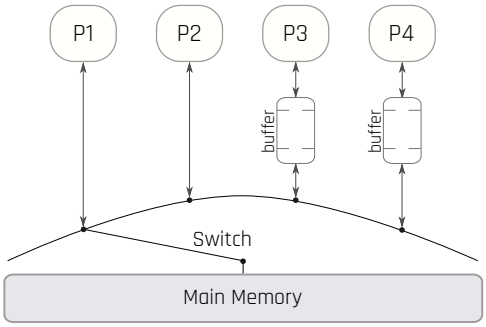

Figure 5. Operational intuition of combining SC and RC. P1 and P2 belong to the SC machine whereas P3 and P4 belong to the RC machine.

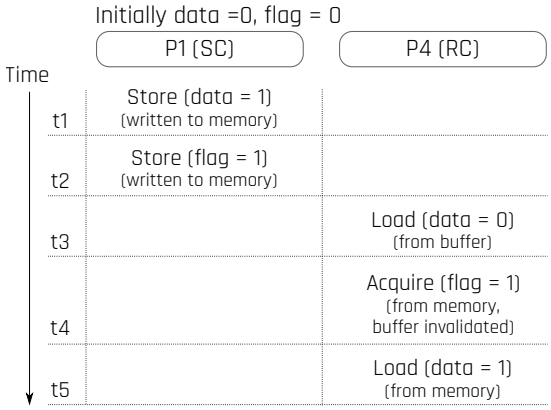

Figure 6. A legal execution on the compound SC/RC machine that adheres to the SC/RC compound memory consistency model. At time t4, an acquire from the RC machine sees flag, at t5 it correctly reads the up-to-date value of 1 from memory.

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

Example. Figure 5 illustrates the compound SC/RC machine obtained by fusing SC and RC. Because P1 and P2 are part of the original SC machine, they do not have any local buffers. Because P3 and P4 are part of the RC machine, they have local store buffers and load buffers. (Stores write to the local store buffer, which is flushed on a release. Loads are allowed to read potentially stale values from the local load buffer, which is invalidated upon an acquire.)

To understand how the SC/RC machine enforces compound consistency, consider the execution shown in Figure 6. Assume that initially P4 has a copy of data with a value of 0 in its local load buffer; flag and data have initial values of 0 in memory. At times t1 and t2, P1 writes 1 to data and flag, respectively. At t3, P4 reads the locally buffered value of 0. Note that a stale read of 0 does not violate the compound SC/RC consistency model, because P4 has not performed an acquire yet. At time t4, an acquire for flag at P4 reads 1 from memory and invalidates the local buffer, as mandated by RC. At t5, a load to data from P4 gets the up-to-date value of 1 from memory.

C. Refining the Intuition

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

Whereas each input in the concrete problem is still a state machine that enforces a memory model, the state machine is more detailed, with caches and a directory coming into the mix. Each input of the concrete problem is thus a refinement of an input of the abstract problem. Naturally, we must ensure that the concrete problem's output, too, is a refinement of the abstract problem's output. In other words, we must merge the directories such that the merging has the same operational effect as merging the memory components into one (but leaving the buffering components untouched).

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

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

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

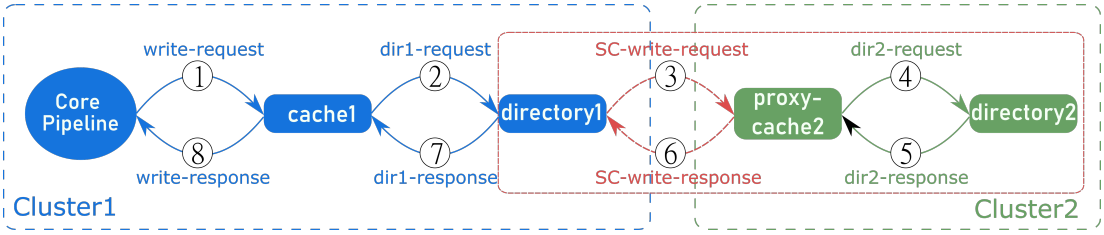

Figure 7. HeteroGen protocol flow for a write issued by the pipeline. Note the red box: directory1, proxy-cache2 and directory2 are one merged directory.

protocol enforces its consistency model correctly, it must intrinsically distinguish between these lines anyway.

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

First, HeteroGen identifies the access sequence in cluster2's consistency model[2] for an SC-equivalent store using ArMOR [24]. For example, the equivalent of an SC store in RC would be a release. Why an SC-equivalent store? Because that is guaranteed to trigger a write request that propagates globally before the write's completion.

Second, HeteroGen consults cluster2's cache specification and identifies the sequence of coherence requests that would be triggered for the SC-equivalent access sequence. For example, in the lazy RC coherence protocol [10], a release would trigger an ownership request for that cache line, and HeteroGen introduces a proxy cache to issue that request to the directory. Logically, the proxy cache is a clone of a cluster2 cache controller that HeteroGen leverages for issuing the above request transparently. (Logically, there is one proxy cache per cluster.) In reality proxy caches are part of the merged directory that HeteroGen generates, and a cluster's (say cluster1) "proxy cache" represents the transient states that bridge the protocol flows from cluster2 to cluster1.

To summarize, as shown in Figure 7, when a write is made globally visible in cluster1—i.e., when directory1 receives a write permissions request or a writeback request—HeteroGen propagates that write by translating it into an appropriate request (with the help of ArMOR) and then issuing that request in cluster-2 via its proxy cache. Once the request has completed, the proxy cache evicts the line, marking the location as invalid in cluster2. Then, directory1 resumes by completing the original write request within cluster1.

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

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

Example. To understand how the HeteroGen-fused directories enforce compound consistency, let us consider the execution shown in Figure 8 on a heterogeneous machine consisting of an RC and an SC cluster. Processors P1 and P2 belong to the SC cluster, which runs a conventional writer-initiated MSI protocol. Processors P3 and P4 belong to the RC cluster. The RC cluster runs a simple RC protocol that buffers writes in the local cache, writes back data upon a release, and self-invalidates the local cache on an acquire. Initially P1 and P4 have local copies of flag and data with a value of 0.

At time t1, P4 performs a store to data and its value is locally updated to 1. At time t2, P4 performs a release to flag. The release initiates a writeback of dirty lines in the cache, causing data to be written back at time t3. The dirty write back of data is propagated to the SC cluster; HeteroGen discovers that this is a request corresponding to a store and translates it into its SC-equivalent in the MSI world – which happens to be a store. By looking at the cache controller of the MSI protocol, HeteroGen discovers that a store will lead to a request for write permissions in the MSI world, and has the proxy cache issue the same to the SC-directory; the SC-directory sends invalidations to sharers and ends up invalidating P1's local copy of data. We show the state transitions at the combined directory controller in Figure 9.) At time t4, a similar sequence of events leads to the local copy of flag being invalidated at P1.

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

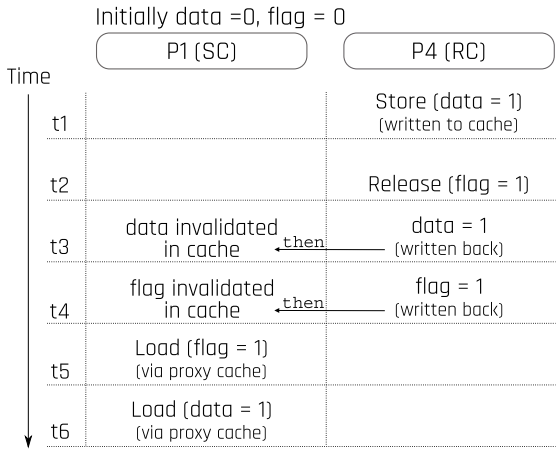

Figure 8. A legal execution on the merged MSI(SC)/RC protocol that adheres to the SC/RC compound memory consistency model. At time t5 a load of flag at P1 reads 1 and at time t6 it correctly reads 1 for data.

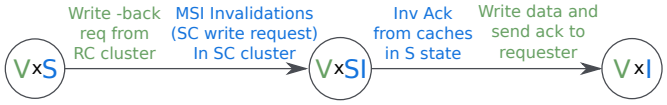

Figure 9. State transitions at the combined directory controller. At time t2, at the combined directory controller, data is in V(alid) state in the RC cluster and in S(hared) state in the SC cluster. This is represented as state VxS at the combined directory controller. At time t3, when the write-back request of data reaches the directory, the proxy cache (which is actually part of the combined directory) propagates the write in the SC cluster, forwarding invalidations to all caches in the SC cluster that are caching data in state S, and enters a transient state denoted as VxSI. Once the proxy cache receives invalidation acknowledgments, the original write-back request is handled: data is written, and an acknowledgment is sent back to the cache that initiated the write-back of data. In the end, data is in V state in the RC cluster and I(nvalid) state in the SC cluster, denoted VxI.

D. Implementation details

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

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

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

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

The first condition is self-explanatory. The second condition checks whether the value written can potentially become globally visible. For example, consider an RC protocol with write-back caches that buffers writes locally. On a store, if the cache line is invalid, the protocol requests the line from the lower level and goes to valid. Although this is really a read request, the final state (valid) allows for writes because RC allows for writes before a release to be be buffered. This illustrates why the second condition is required: the fact that there cannot be any forwarded requests for a cache line in valid state ensures HeteroGen does not mis-classify the original request as a write.

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

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

Consider two clusters: cluster1 enforcing RC, and cluster2 enforcing SC. Let's say that processor P1 from cluster1 performs a release to address X that reaches directory1, and needs to be propagated to the SC cluster. Accordingly, cluster2's proxy cache issues an appropriate request (say reqA) to directory2 and is waiting for a response. Meanwhile, let's say there is another release from P1 to address Y that reaches directory1, and also needs to be propagated to the SC cluster, with this second release coming after the first in P1's program order. Question: can the proxy cache issue reqB (the second request) concurrently with reqA?

To answer this question, let us first ask ourselves why the two releases were issued by P1 concurrently in the first place. Typically, the processor pipeline orders stores (including releases) by issuing them in order, waiting for completion of the first before issuing the next. (Aggressive implementations [17] allow for reordering but the onus is on the pipeline to achieve the same effect.) Under this pipeline ordering assumption, the only way in which the releases could reach directory1 (and hence proxy cache) concurrently is if the caches responded to the pipeline even before the release became globally visible. Consider an RC protocol that simply writes through every store to the lower level. Supposing the interconnect were totally ordered, the caches can simply stream releases without waiting for acknowledgments from the directory, relying on

the interconnect to enforce the release → release ordering.

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

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

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

3) Handling arbitrary number of clusters: Thus far, we have assumed that HeteroGen takes two protocols as inputs, and this is purely to simplify the discussion. HeteroGen can naturally handle an arbitrary number of protocols. The only change is that, upon a globally visible write to one cluster, the write has to be propagated to all other clusters, which entails issuing requests to each of the other clusters via a proxy cache. On a read that cannot be served locally within a cluster, a read request is issued to the cluster that wrote to that location most recently.

4) Pipeline-Coherence interaction: Thus far we have assumed that a pipeline interacts with the coherence protocol using a simple interface, where the pipeline issues reads and writes to the coherence subsystem in accordance with that cluster's memory model. In high-performance implementations, however, the pipeline can interact with the coherence protocol via a richer interface: e.g., in speculative load replay [17] the pipeline might issue reads out of order, relying on the coherence protocol to flag memory ordering violations.

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

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

- **Analyze input protocols.** HeteroGen first analyzes each of the input directory controllers to identify those requests that are globally visible write requests (as explained in Section VI-D1). Further, it also analyzes each of the input cache controllers to identify whether any write request to the cache is acknowledged early; if even one of the writes in one of the input protocols is acknowledged early, HeteroGen uses a conservative processor-centric approach (explained in the following under "concurrency").
- **Writes.** Consider each globally-performing write request to any of the input directory controllers. Before handling the request within that cluster, HeteroGen first propagates that write within other clusters. This is accomplished as follows. First, the ArMOR framework [24] is employed to identify the corresponding write request in each of the other clusters. Then, these requests are initiated in parallel, and once all of these are performed, the original write request is handled. Once this is done, the cluster that performed the write is set as the owner of the block address.
- **Reads.** For each read request that cannot be serviced within a cluster, HeteroGen handles that read request as if it was initiated by the current owner cluster of that cache block.
- **Concurrency.** While handling a read or a write request, requests to that address from any processor are blocked. Additionally, in case of conservative processor-centric approach, requests from the processor that initiated the original read or write are also blocked.

E. Using HeteroGen

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

For example, HeteroGen cannot fuse an invalidation based protocol with an update based protocol because the

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

For another example, HeteroGen cannot fuse Tardis [43] (or Relativistic Coherence [32], G-TSC [39]) with a conventional invalidation based protocol. This is because the notion of read permissions is not directly compatible with leases. Read permissions allow for a block to be held potentially indefinitely, whereas leases expire.

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

2) Consistency Models of Input Protocols: For HeteroGen to select the appropriate ArMOR translations at the merged directory, it must know the consistency models of the input protocols. In theory, we could ask the user to precisely specify these consistency models, but architects do not often reason about protocols in that way; instead, they tend to reason about consistency as a function of both the protocol and core pipeline. To avoid relying on the user, HeteroGen uses extensive litmus testing of each input protocol to infer its consistency model.

VII. Case Studies and Validation

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

A. Case Studies

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

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

RCC [27] is a simple protocol that enforces RC by: buffering writes in the cache; writing back the cache contents on a release; and self-invalidating the cache on an acquire. RCC-O [10], [27] is a block-granular variant of DeNovo [14] that obtains ownership on all writes. GPU is a simple GPU protocol as specified in Spandex [11], where stores write through to the shared cache. GPU, RCC-O, and RCC enforce RC. PLO-CC is a variant of RCC-O without a release, and it enforce a memory model called partial-load-order [24] that enforces the W→W and the R→W orderings but not the other two. TSO-CC [16] is a protocol tailored to enforce TSO; we model the basic version of the protocol without timestamps. These protocols represent a wide range of protocols, highlighting the generality of HeteroGen.

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

	Protocol
SC	MSI, MESI
TSO	TSO-CC [16]
RC	RCC-O [10], [27] , RCC [27], GPU [11]
PLO	PLO-CC

Table I
Protocols used in the case studies

B. Heterogeneous Litmus Testing

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

To validate that HeteroGen's generated protocols satisfy their compound consistency models, we generated heterogeneous litmus tests. Starting with the version of the litmus test for the weaker of the two consistency models, we use consistency model translation [24] to remove any synchronization operations (e.g., Fences) that are not needed for the stronger consistency model.

We used the herd7 tool [9] to generate 111 litmus tests, including commonly used tests like MP, S, IRIW, 2+2W, CoRR, LB, R, RWC, SB, WRC, WRW+WR, WRW+2W, and WWC [31]. For each litmus test, we consider all possible allocations of threads to processor cores.

To perform the validation of every heterogeneous protocol generated by HeteroGen, we used the Murphi model checker [15]. In Murphi, we preload the caches with the initial values in the litmus test, and we ensure that loads and stores are executed based on the litmus test, while permitting evictions at any time. Murphi then exhaustively explores every possible ordering of the events in the litmus test, and it reveals whether the protocol permits an outcome prohibited by the consistency model. In all of our litmus tests, the HeteroGen-generated protocols were successful.

C. Validating Deadlock Freedom

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

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

Table II
Case Studies with their respective Heterogen Directory States and Transitions

VIII. Protocol Performance

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

The most relevant comparison is Spandex, which recall is an interface for manually integrating multiple different protocols. For our baseline we consider HCC [40], which is a publicly available protocol that is similar to Spandex [11]; we focus on the heterogeneous protocol obtained by manually combining the DeNovo protocol with MESI. For comparison we use the automatically-generated RCC-O/MESI protocol generated by HeteroGen. (Recall that RCC-O is a block-granular variant of DeNovo.)

We simulated the protocols on gem5, and our simulation parameters are identical to those used in HCC [40]. We simulated a 64-core system with 2 clusters: 60 "tiny" cores and 4 "big cores". The big cluster employs the MESI protocol whereas the tiny cluster employs DeNovo. The two clusters are manually fused using HCC, whereas in our setup MESI and RCC-O are automatically fused using HeteroGen. The detailed simulation parameters are shown in Table III. We used the same 13 applications with fine-grained synchronization as in HCC.

Our results are summarized in Figure 10, and they reveal that HCC and HeteroGen have comparable runtimes, with HeteroGen performing similarly to the manually-generated HCC on average. The important point of difference in the two protocols is the use of conservative handshaking messages in the manually-generating protocol, whereas HeteroGen eschews these redundant handshakes. The effect of the reduced handshaking messages translates into faster reads: specifically, when a core reads the value written by another core it incurs significantly reduced latencies. The effect of faster communicating reads translates into performance for two benchmark programs (nq and lu) which spend a significant time on such reads.

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

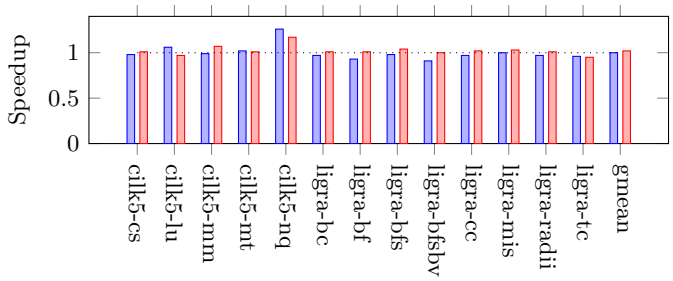

Figure 10. Speedup of HeteroGen over HCC [40]: without handshaking (blue) and with handshaking (red)

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

Table III
Simulated System Parameters [40]

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

IX. Conclusions

HeteroGen can automatically compose multiple coherence protocols, including MOESI variants and other protocols that are targeted towards specific consistency models. The resulting heterogeneous protocol satisfies the precisely

defined compound consistency model that can be inferred from the consistency models enforced by the constituent protocols. We validated HeteroGen with newly developed litmus tests.

As the computer architecture community—both in academia and industry—continues the trend towards heterogeneity, we hope that HeteroGen and clear compound consistency models can greatly reduce design time and increase confidence in the design. To that end, we are publicly releasing HeteroGen.[3]

X. Acknowledgments

We thank Susmit Sarkar, Caroline Trippel, our shepherd, and the anonymous reviewers for their constructive comments and feedback. This work is supported by Huawei, Google through their PhD Scholarship program, EPSRC grant EP/V028154/1 to the University of Edinburgh, and the National Science Foundation under grant CCF-200-2737.

References

[1] "Compute Express Link," https://www.computeexpresslink.org/, accessed: 18th June 2021.

[2] "The AMBA CHI Specification," https://developer.arm.com/architectures/system-architectures/amba/amba-5, accessed: 15th July 2019.

[3] "The CCIX Consortium," https://www.ccixconsortium.com/, accessed: 21st January 2019.

[4] "The GenZ Consortium," https://genzconsortium.org/, accessed: 21st January 2019.

[5] "The OpenCAPI Consortium," https://opencapi.org/, accessed: 21st January 2019.

[6] P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo, "The benefits of duality in verifying concurrent programs under TSO," in *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, ser. LIPIcs, J. Desharnais and R. Jagadeesan, Eds., vol. 59. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 5:1–5:15. [Online]. Available: https://doi.org/10.4230/LIPIcs.CONCUR.2016.5

[7] D. Abts, S. Scott, and D. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[8] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in Weak Memory Models," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, 2010.

[9] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats," *ACM TOPLAS*, vol. 36, no. 2, pp. 1–74, jul 2014.

[10] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy Release Consistency for GPUs," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[11] J. Alsop, M. D. Sinclair, and S. V. Adve, "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," in *Proceedings of the 45th International Symposium on Computer Architecture*, 2018.

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

[13] M. Batty, "Compositional relaxed concurrency," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, no. 2104, September 2017. [Online]. Available: https://kar.kent.ac.uk/64300/

[14] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[15] D. L. Dill, "The Murphi Verification System," in *CAV*, vol. 1102, 1996.

[16] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in *HPCA*, 2014.

[17] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume I: Architecture/Hardware*. CRC Press, 1991, pp. 355–364.

[18] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free Memory Models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

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

[20] D. Iorga, A. F. Donaldson, T. Sorensen, and J. Wickerson, "The semantics of shared memory in intel CPU/FPGA systems," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–28, 2021. [Online]. Available: https://doi.org/10.1145/3485497

[21] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building heterogeneous unified virtual memories (uvms) without the overhead," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 1:1–1:22, 2016. [Online]. Available: https://doi.org/10.1145/2889488

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

[23] D. Lustig, S. Sahasrabuddhe, and O. Giroux, "A formal analysis of the NVIDIA PTX memory consistency model," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 257–270. [Online]. Available: https://doi.org/10.1145/3297858.3304043

[24] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures," in *Proceedings of the International Symposium on Computer Architecture*, 2015.

[25] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux, "Automated Synthesis of Comprehensive Memory Model Litmus Test Suites," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[26] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Generating Litmus Tests for Contrasting Memory Consistency Models," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, 2010.

[27] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 2nd ed., ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[28] L. E. Olson, M. D. Hill, and D. A. Wood, "Crossing guard: Mediating host-accelerator coherence interactions," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 163–176.

[29] N. Oswald, V. Nagarajan, and D. J. Sorin, "ProtoGen: Automatically Generating Directory Cache Coherence Protocols from Atomic Specifications," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018, pp. 247–260.

[30] N. Oswald, V. Nagarajan, and D. J. Sorin, "HieraGen: Automated Generation of Concurrent, Hierarchical Cache Coherence Protocols," in *Procedings of the 47th Annual International Symposium on Computer Architecture*, 2020.

[3] https://doi.org/10.5281/zenodo.5826339

[31] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for armv8," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 19:1–19:29, 2018. [Online]. Available: https://doi.org/10.1145/3158107

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

[33] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: The key to simple and efficient coherence for clustered cache hierarchies," in *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*. IEEE Computer Society, 2015, pp. 186–197. [Online]. Available: https://doi.org/10.1109/HPCA.2015.7056032

[34] A. Ros and S. Kaxiras, "Racer: TSO consistency via race detection," in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 2016, pp. 33:1–33:13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783736

[35] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, p. 89–97, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[36] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 647–659. [Online]. Available: https://doi.org/10.1145/2830772.2830821

[37] I. Singh, A. Shriraman, W. Fung, M. O'Connor, and T. Aamodt, "Cache Coherence for GPU Architectures," in *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, 2013.

[38] R. L. Sites, *Alpha Architecture Reference Manual*. Prentical Hall, 1992.

[39] A. Tabbakh, X. Qian, and M. Annavaram, "G-TSC: timestamp based coherence for gpus," in *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, pp. 403–415. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00042

[40] M. Wang, T. Ta, L. Cheng, and C. Batten, "Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 2020, pp. 173–186. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00025

[41] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," 2014.

[42] D. L. Weaver and T. Germond, *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994.

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

A. Artifact Appendix

A.1 Abstract

Our artifact provides the code for HeteroGen in Python and the input protocols expressed in the PCC language. It outputs the complete concurrent heterogeneous protocol in Murphi. Our artifact leverages the CMurphi infrastructure to verify the generated protocols. We also provide scripts for automatically generating the protocols and validating them.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Automatic generation of heterogeneous coherence protocols
- **Program:** HeteroGen
- **Run-time environment:** Linux (Ubuntu 20.04), Python 3.8, CMurphi 5.4.9.1, Docker
- **Hardware:** System with at least 16GB of RAM and Intel Skylake or AMD Ryzen
- **Output:** HeteroGen generates heterogeneous cache coherence protocols in Murphi
- **Experiments:** Litmus tests provided
- **Publicly available:** Yes
- **Code licenses:** MIT

A.3 Description

A.3.1 How to access

Code available on Zenodo: https://doi.org/10.5281/zenodo.5826339

A.3.2 Hardware dependencies

Any PC with at least 16GB of RAM suffices to run most tests and these will complete in a matter of minutes on an Intel Skylake or comparable CPU. However, a few deadlocks tests can require up to 1TB of RAM and hours of computation time.

A.3.3 Software dependencies

- Linux distribution (e.g. Ubuntu 20.04)
- Graphviz 2.43.0
- Python 3.8 or higher
  - antlr3 3.5 (Source code provided in Zenodo)
  - colorama 0.4.3
  - graphviz 0.16
  - networkx 2.5.1
  - psutil 5.8.0
  - tabulate 0.8.9
- CMurphi 5.4.9.1 (Source code provided in Zenodo)
- Efficiently Supporting Dynamic Task-Parallelism on Heterogeneous Cache-Coherent Systems by Wang et al.

  https://zenodo.org/record/3910803

A.3.4 Antlr3 setup

To install antlr3 first open to the antrl3 python3 directory.

cd antlr3-master/runtime/Python3

Then run the install script provided in the directory.

sudo python3 setup.py install

A.3.5 CMurphi setup

To install CMuprhi run from the parent directory:

cd src && make

A.3.6 Datasets

Stable state protocols, used as inputs by HeteroGen to generate the heterogeneous cache coherence protocol, are provided in the Protocols/MOESI_Directory/ord_net directory. A set of litmus tests to verify the correctness of the protocols is provided in the MurphiLitmusTests directory.

A.4 Experiment workflow

In the top level directory run:

python3 HeteroGen.py

This will generate the heterogeneous coherence protocol state machines and litmus tests of protocols presented in the paper, which will be verified using the Murphi model checker.

When HeteroGen is running warnings are displayed. These warnings can be ignored when using the provided protocols, but can help to debug problems when using new atomic protocols as inputs to HeteroGen.

The generated files can be found in the directory:
Protocols/MOESI_Directory/ord_net/HeteroGen

To compile the generated litmus tests update the variable 'murphi_compiler_path' in the file ParallelCompiler.py to your local Murphi path. Now compile the model checker files:

python3 ParallelCompiler.py

A.5 Evaluation and expected results

A.5.1 Correctness of automatically generated protocols

To verify the correctness of the generated heterogeneous cache coherence protocols, the Murphi model checker is used.

To start the verification, run:

python3 ParallelChecker.py

The runtime of all litmus tests depends on the amount of RAM and number of CPUs available. The ParallelChecker.py will automatically run all litmus tests and generate a report file 'Test_Result.txt' in the TestScript directory.

If the ParallelChecker reports that no Murphi test files were found, change the access permission of the executables to '+x'.

In the report file 'Test_Result.txt' the litmus tests failing are listed. None of the litmus tests should fail under normal operation. The types of failure that are listed can be as follows:

- 'Not served yet': Due to some error the litmus test was not served
- 'Out of memory': The verification test ran out of available RAM.
- 'File not found': No executable was found.
- 'Fail': The litmus test failed because of an unknown error (e.g. executable could not be run)
- 'Litmus test fail': A litmus test has failed
- 'Deadlock': A deadlock in the protocol was found
- 'Invariant': An invariant specified was violated

A.5.2 Generated protocols performance evaluation

To evaluate the performance of the automatically generated MESI-RCCO HeteroGen protocol it is compared against the HCC-Denovo protocol.

To compare the performance of the protocols, please follow the instructions provided by the authors of the Efficiently Supporting Dynamic Task-Parallelism on Heterogeneous Cache-Coherent Systems publication to setup the reference system.

Once the reference system has been setup, copy the provided *gem5_HeteroGen_protocols* directory into the docker container and run:

setup.sh

The setup script copies and modifies all the required files into the alloy-gem5 directory.

Change directory to:

cd alloy-gem5

After running the setup script, run the benchmark execution scripts in the alloy-gem5 directory to produce the simulation results.

HCC-DeNovo protocol:

run_DeNovo.sh

HeteroGen MESI-RCCO protocol without any handshakes:

run_RCCO_GEN_NO_HS.sh

HeteroGen MESI-RCCO protocol with write handshakes:

run_RCCO_GEN_WR_HS.sh

The simulation results are provided in the sim_res directory.

/alloy-gem5/sim_res

Each result folder is labeled by the type of protocol (e.g. RCCO_GEN_NO_HS) that has been run followed by the name of the executed benchmark (e.g. BC).

A.6 Experiment customization

The scripts included do not have any customization options. (Note, however, that using the PCC language, new atomic cache coherence protocols can be specified. HeteroGen can then use the newly defined protocols to generated a new heterogeneous protocol.)