THEME ARTICLE: TOP PICKS FROM THE 2022 COMPUTER ARCHITECTURE CONFERENCES

# HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols

Nicolai Oswald, NVIDIA, Zurich, 8004, Switzerland

Vijay Nagarajan, University of Utah, Salt Lake City, UT, 84112, USA

Daniel J. Sorin, Duke University, Durham, NC, 27708, USA

Vasilis Gavrielatos, Huawei, Edinburgh, EH3 8BL, U.K.

Theo X. Olausson, Massachusetts Institute of Technology, Cambridge, MA, 02139, USA

Reece Carr, University of Edinburgh, Edinburgh, EH8 9AB, U.K.

We address the two challenges architects face when designing heterogeneous processors with cache-coherent shared memory. First, we introduce HeteroGen, an automated tool for composing clusters of cores, each with its own coherence protocol. Second, we show that the output of HeteroGen conforms to a precisely defined memory consistency model that we call a compound consistency model. We also demonstrate that HeteroGen can correctly fuse a wide range of coherence protocols. Our experiments indicate that protocols generated by HeteroGen perform comparably to a publicly available manually generated heterogeneous protocol.

Our work draws inspiration from two modern processor design trends: processor core heterogeneity and cache-coherent shared memory. The trend of heterogeneity has grown beyond just CPU/GPU designs, as seen in chips from Apple, Qualcomm, and Samsung that now feature diverse cores such as digital signal processors, neural processors, and camera processors in addition to CPUs and GPUs.

Our other motivating design trend is the continued reliance on cache-coherent shared memory. In the early days, the CPU cores did not share memory with the GPU cores. However, today, cache-coherent shared memory—often with accelerator-specific protocol features—has become prevalent and has been codified in standardized design frameworks such as AMBA CHI and CXL.

Architects face two challenges when designing heterogeneous processors with cache-coherent shared memory. The first challenge is complexity. Designing a coherence protocol for a homogeneous processor is already challenging, and introducing heterogeneity only increases the difficulty. This is because communication patterns within a CPU, a GPU, or an accelerator are very different, often mandating bespoke coherence protocols for each case. To compose these very different protocols into a unified heterogeneous whole is challenging.

The second challenge is reasoning about the memory consistency model of the heterogeneous processor. Consider a processor that consists of several clusters of diverse 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 this model?

To overcome these two challenges—the design complexity and consistency model—we have developed HeteroGen, a tool for automatically generating heterogeneous protocols that adhere to precise consistency models. As shown in Figure 1, HeteroGen takes as the input 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.

0272-1732 © 2023 IEEE
Digital Object Identifier 10.1109/MM.2023.3274993
Date of publication 10 May 2023; date of current version 29 June 2023.

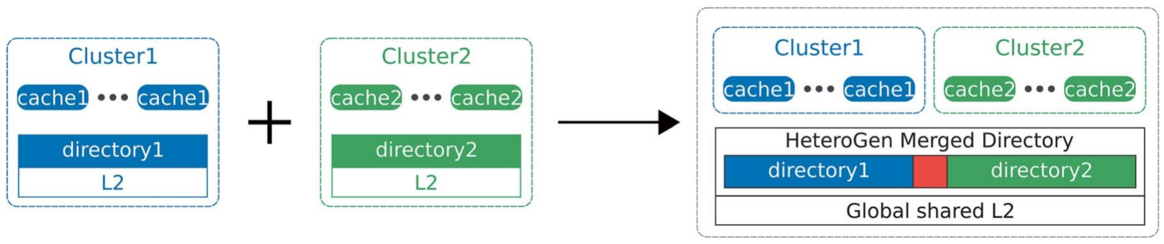

FIGURE 1. HeteroGen takes as its inputs the coherence protocols of the individual clusters (Cluster1 and Cluster2) and automatically merges their directory controllers to produce the merged directory. The target system model consists of multiple clusters of cores with their private L1 caches and global shared L2.

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. One of HeteroGen's key contributions is its guarantee that its output protocol will provide compound consistency. In the following, we first discuss compound consistency before going on to our method for automatically generating heterogeneous protocols. Our IEEE HPCA paper[10] has more details.

## BACKGROUND

### Memory Consistency Models

The memory consistency model is part of the processor instruction set specification and specifies how loads and stores are ordered—in effect, specifying what value a load must return.[8] Sequential consistency (SC), for example, requires that there should appear to be a total order of all loads and stores that respects the program order at every thread. Modern processors do not support SC. Intel and AMD processors support the x86-total store order (TSO) consistency model, whereas ARM and RISC-V processors support variants of release consistency (RC).

The manner in which stores propagate their values to other processors is crucial in memory models. In multicopy atomic (MCA) memory models, 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. This work focuses on MCA memory models, supported by commercial architectures such as x86, ARM, and RISC-V.

### Cache Coherence Protocols

The cache coherence protocol is a critical widget in shared memory multiprocessors that helps enforce the memory model by keeping caches consistent.[8] These protocols can be divided into two categories: those that enforce the single-writer, multiple-reader (SWMR) invariant by invalidating sharers on writes, typically used in CPUs, and those that rely on writebacks and self-invalidations, favored in GPUs.

## COMPOUND CONSISTENCY MODELS

Given that the coherence protocols of the different devices that make a heterogeneous computer could be different and enforce distinct consistency models, what should be the correctness criterion of their composition? We propose a solution: a compositional approach to heterogeneous consistency called compound memory consistency models.

### Intuition

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 Dekker's litmus test, shown in Figure 2(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 loads Ld1 and Ld2 to read zeroes. This is because the TSO cluster does not enforce the store-to-load (St2 ! Ld2) ordering, even though the SC cluster enforces the St1 ! Ld1 ordering.

However, as shown in Figure 2(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

```
Initially x = 0, y = 0
T1 (SC)            T2 (TSO)
St1: St x = 1      St2: St y = 1
Ld1: Ld r1 = y     Ld2: Ld r1 = x

       (a)
```

```
Initially x = 0, y = 0
T1 (SC)            T2 (TSO)
St1: St x = 1      St2: St y = 1
Ld1: Ld r1 = y     FENCE
                   Ld2: Ld r1 = x

       (b)
```

FIGURE 2. (a) Ld1 and Ld2 can both return zero. (b) Only one of Ld1 and Ld2 can return zero.

St1 and Ld1 from T1 because the SC cluster already guarantees this ordering.

## Formalism

Starting with the axiomatic framework of Alglave et al.,[1] which can capture any MCA model, we then formally define the compound consistency model enforced when combining a set of given MCA models.

### Preliminaries

We start by defining some basic relations:

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

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

- $\xrightarrow{ws}$ is the write-serialization relation, which relates two writes of the same address that are serialized in the order specified.
- $\xrightarrow{rf}$ is the read-from relation, which relates a write and read of the same address such that the read returns the value of the write.
- $\xrightarrow{rfe}$ is 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.
- $\xrightarrow{fr}$ is the derived from-read relation, which 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,

$$acyclic(\xrightarrow{poaddr} \cup \xrightarrow{rf} \cup \xrightarrow{fr} \cup \xrightarrow{ws}). \qquad (1)$$

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

### Multicopy Memory Model

An MCA memory model is specified in terms of the preserved program order relation $\xrightarrow{ppo}$, which 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 ($M \xrightarrow{ppo}$) 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,

$$acyclic(\xrightarrow{ppo} \cup \xrightarrow{rfe} \cup \xrightarrow{fr} \cup \xrightarrow{ws}). \qquad (2)$$

For example,

- SC $\xrightarrow{ppo} = \xrightarrow{po}$.
- 86-TSO $\xrightarrow{ppo} = \xrightarrow{po} \setminus st(x) \xrightarrow{po} ld(y), \forall x, y$.

### 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 MCA memory model $M_i \xrightarrow{ppo_i}$.

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 intracluster and intercluster communication relations identically.

Let us partition the threads into n subsets: $T_1, T_2, \ldots, T_n$ 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

$$\xrightarrow{ppo_{com}}(t) \mid t \in T_i := \xrightarrow{ppo_i}.$$

We now specify the compound consistency model as the one that preserves $ppo_{com}$ as defined earlier. 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:

$$acyclic(\xrightarrow{ppo_{com}} \cup \xrightarrow{rfe} \cup \xrightarrow{fr} \cup \xrightarrow{ws}). \qquad (3)$$

### Example

Let us go back to Figure 2(b), which shows Dekker's litmus test with threads T1 from the SC cluster and T2 from the TSO cluster. The following sequence of edges

$$\mathrm{St1} \xrightarrow{ppo} \mathrm{Ld1} \xrightarrow{fr} \mathrm{St2} \xrightarrow{ppo} \mathrm{Ld2} \tag{4}$$

implies that Ld2 will read the value of St1, reading a one. Note that the $\mathrm{Ld1} \xrightarrow{fr} \mathrm{St2}$ edge in (4) relates two operations from different clusters; recall that the compound memory model treats intracluster and intercluster communication relations identically, and, thus, this edge is part of the global memory order.

### 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 instruction set architecture (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.

## HETEROGEN

At a high level, HeteroGen performs the integration illustrated in Figure 1. 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[9] and consistency model translation.[7] We explain how these work together. However, before that, we explore what compound consistency means operationally.

### 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 first-in, first-out store buffer and/or a load buffer is introduced between each processor and the memory, we get TSO. In general, any MCA memory model can be expressed as processors with local buffers connected to atomic memory, with each memory model having its unique buffering logic.[11]

Given the state machine representations of the two memory models as described, 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 3 illustrates the compound SC/RC machine obtained by fusing SC and RC. Because processors 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.)

### 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 just introduced (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

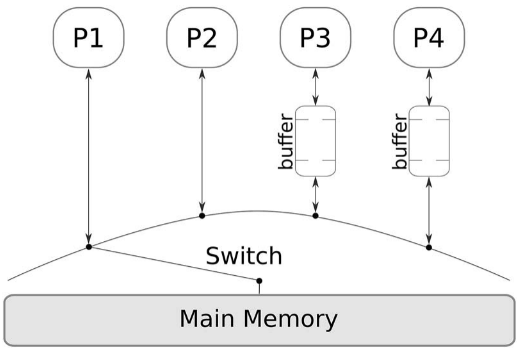

FIGURE 3. Operational intuition of combining sequential consistency (SC) and release consistency (RC). Processors P1 and P2 belong to the SC machine, whereas P3 and P4 belong to the RC machine.

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"?

HeteroGen ensures the 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.

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 for an SC-equivalent store using ArMOR.[7] 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,[2] 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 this 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's) "proxy cache" represents the transient states that bridge the protocol flows from cluster2 to cluster1.

To summarize, as shown in Figure 4, 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 cluster2 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 con-tact directory2 and find that the block is invalid in clus-ter2. 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 direc-tory2, which completes the original read request.

## Using HeteroGen

### 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. 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

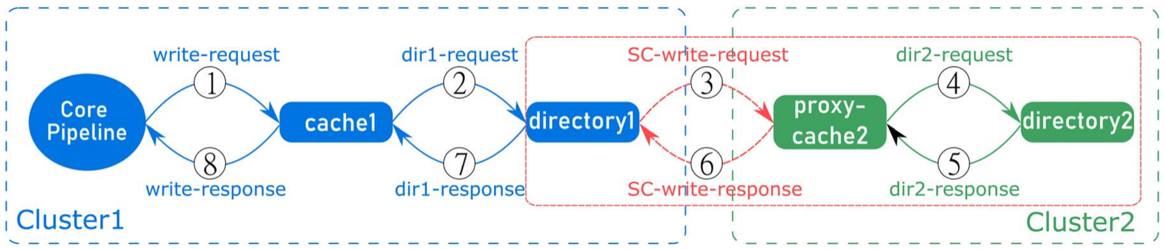

FIGURE 4. HeteroGen protocol flow for a write issued by the pipeline. Note the red box: directory1, proxy-cache2, and directory2 are one merged directory.

cannot fuse an invalidation-based protocol with an update-based protocol because the notion of write permissions is not compatible with update protocols.

### 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. Although we could ask the user to specify these consistency models, HeteroGen uses extensive litmus testing of each input protocol to infer its consistency model.

## CASE STUDIES AND VALIDATION

To explore HeteroGen's ability to create performant, correct protocols, we used it to generate heterogeneous protocols from a wide range of homogeneous protocols.

### Case Studies

We consider seven homogeneous protocols: two SWMR-enforcing protocols (MESI and MSI) and five protocols that are designed for weaker consistency models (RCC, RCC-O, GPU, TSO-CC, and PLO-CC).

RCC[8] 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[2,8] is a block-granular variant of DeNovo[4] that obtains ownership on all writes. GPU is a simple GPU protocol as specified in Spandex,[3] where stores write through to the shared cache. GPU, RCC-O, and RCC enforce RC. PLO-CC is a variant of RCC-O without a release, and it enforces a memory model called partial load order[7] that enforces the W ! W and the R ! W orderings but not the other two. TSO-CC[6] 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.

Table 1 lists the pairs of protocols that we composed with HeteroGen. For each fused protocol, we show the number of states and transitions.

### Validation

We validated all of our generated protocols with litmus tests. Litmus testing is a time-tested technique for detecting whether a system is capable of behaving in ways that violate the desired consistency model. Suites of litmus tests exist for validating that a homogeneous protocol satisfies a (noncompound) memory consistency model.

To extend litmus testing to heterogeneous protocols, we generated heterogeneous litmus tests that can reveal whether a system violates a compound consistency model. Our process automatically transforms existing (homogeneous) litmus tests for existing (noncompound) consistency models. In a system with two protocol clusters, we take the litmus test for the weaker of the two consistency models and use consistency model translation[7] to remove any synchronization operations (e.g., fences) that are not required for the stronger consistency model. This approach scales to an arbitrary number of clusters.

Using the herd7 tool,[1] we generated 111 litmus tests, and, for each test, we explored all possible mappings of threads to cores. We used the Murphi model checker[5] to perform the validation of each litmus test. For each

TABLE 1. Case studies with their respective HeteroGen directory states and transitions.

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

litmus test, Murphi exhaustively searches the reachable state space of possible executions and determines whether the system is capable of violating the consistency model. All of our HeteroGen-generated protocols were validated; none ever permitted behaviors that violated the compound consistency models.

Finally, we also used Murphi to validate that every protocol—both the constituent protocols and the generated protocols—is deadlock free.

## PROTOCOL PERFORMANCE

Although we had no reason to believe that the generated protocols would perform worse than manually developed protocols, we performed experiments to study this issue. Our baseline protocol is HCC,[12] a publicly available protocol that is similar to Spandex[3]; we focus on the heterogeneous protocol obtained by manually combining the DeNovo protocol with MESI. We compare this baseline to the HeteroGen-produced RCC-O/MESI protocol.

We simulated the protocols with gem5, using the same simulation parameters as those used in HCC. We simulated a 64-core system with two clusters: 60 “tiny” cores and four “big cores.” The big cluster uses the MESI protocol, whereas the tiny cluster uses DeNovo. We used the same 13 applications with fine-grained synchronization as in HCC. Experimental results show that HCC and HeteroGen perform comparably, with HeteroGen performing similarly to the manually generated HCC on average.

## THE FUTURE

### Coherence Protocol Design Automation

Design automation has long been a transformative technology in processor design. Early processors were designed entirely manually, transistor by transistor. The scale and complexity of such processors were limited not just by transistor budgets but by the ability of architects to design and verify them. Major advances in design automation—including gate-level design tools, hardware design languages (e.g., Verilog), and high-level synthesis—have each led to major advances in the processors that could be designed and verified. Modern processors are far more complicated than could be reasonably designed at the transistor or gate level.

As we enter an era of processor heterogeneity—with a multitude of new cores and accelerators that are often designed independently—the challenges of design and verification are only increasing. Creating a single homogeneous multicore processor is already a multiyear task for large corporations like Intel, AMD, and Nvidia. Now, we face the added complexity of heterogeneity. In the same way that prior design automation techniques enabled the creation of complicated processor cores, computer architects now need a new tool to facilitate the creation of heterogeneous multicore processors.

AS WE ENTER AN ERA OF PROCESSOR HETEROGENEITY—WITH A MULTITUDE OF NEW CORES AND ACCELERATORS THAT ARE OFTEN DESIGNED INDEPENDENTLY—THE CHALLENGES OF DESIGN AND VERIFICATION ARE ONLY INCREASING.

Furthermore, the community wants to “democratize” the development of processors, such that they can be produced by smaller companies and academic researchers. The open source hardware movement (e.g., OpenPiton) could be a boon to the community, but only if design automation tools exist to enable rapid design and verification.

The impact of HeteroGen’s design automation will be on the industrial and academic architects who design protocols. If adopted, HeteroGen would greatly simplify and shorten the design and verification processes for the heterogeneous coherence protocols that will predominate in a market of heterogeneous processors [also known as systems on chip (SoCs)]. Given the extraordinary amount of effort—hours and money—that goes into protocol design and verification today, this benefit would be significant.

HeteroGen is starting to see industrial impact. Protocols generated by HeteroGen are being evaluated by a major company designing mobile SoCs. A team from NVIDIA Research is building upon HeteroGen for ongoing research and development. Furthermore, to help with the goal of the democratization of processor design, we have publicly released HeteroGen.[10]

### Precise Heterogeneous Consistency Specification

The introduction of clear, precise specifications of system correctness have often had transformative effects on the field of computer architecture. As one example, the introduction of precise ISAs was revolutionary for many reasons. An ISA enabled architects to clearly define the functional behavior of their processor, in an implementation-independent way, and create families of microarchitectures that all conform to it. Equally important, an ISA provided a clear specification for verification (validation). Verifying any system requires a precise definition of correctness.

Another notable example was the introduction of memory consistency models. Prior to that, there was no implementation-independent definition of memory system behavior. Today, it seems inconceivable that architects designed memory systems without a consistency model—almost all ISAs have precisely defined memory models—and that verification teams had to reason about correctness without a clear, hardware-independent specification of it.

This is the era of heterogeneity in both mobile SoCs and the server space. Industry is standardizing cache coherence frameworks for gluing different processors together and providing a shared memory interface. Meanwhile, the programming languages community is developing heterogeneous programming frameworks, often with well-defined shared-memory consistency models. Until this article, however, the heterogeneous processors themselves have been lacking precise definitions of their consistency models.

THIS IS THE ERA OF HETEROGENEITY IN BOTH MOBILE SoCs AND THE SERVER SPACE.

We expect our introduction of compound consistency to change the way that consistency is specified in heterogeneous processors, which would have a significant impact on architects and verification teams. It will simplify design by making it clear what invariants must be maintained when gluing the heterogeneous processor together, and it will facilitate verification by providing a correctness specification against which to compare the behavior of the glued heterogeneous processor. Finally, last but not least, it will also provide a viable path to get to language-level memory consistency models such as C and OpenCL. It is worth noting that we are working with NVIDIA Research on the problem of specifying a compound GPU/CPU consistency model, integrating a GPU memory consistency model with scopes (e.g., PTX) with a CPU memory consistency model (e.g., ARM). It is possible that, years from now, architects will look back and wonder how heterogeneous shared memory processors were ever created before compound consistency models, much like we look back today to the time before memory consistency models.

## ACKNOWLEDGMENTS

We thank Susmit Sarkar, Caroline Trippel, and the anonymous reviewers for their constructive comments on earlier versions. This work is supported by Google, Huawei through their PhD Scholarship program, EPSRC grants EP/V028154/1 and EP/V038699/1 to the University of Edinburgh, and the National Science Foundation under grant CCF-200-2737. This work was carried out while Nicolai Oswald, Vijay Nagarajan, Vasilis Gavrielatos, Theo X. Olausson, and Reece Carr were at the University of Edinburgh, U.K.

## REFERENCES

1. J. Alglave et al., "Herding cats: Modelling, simulation, testing, and data mining for weak memory," ACM Trans. Program. Lang. Syst., vol. 36, no. 2, pp. 1–74, Jul. 2014, doi: 10.1145/2627752.
2. J. Alsop et al., "Lazy release consistency for GPUs," in Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO), 2016, pp. 1–14, doi: 10.1109/MICRO.2016.7783729.
3. J. Alsop et al., "Spandex: A flexible interface for efficient heterogeneous coherence," in Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA), 2018, pp. 261–274, doi: 10.1109/ISCA.2018.00031.
4. B. Choi et al., "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in Proc. Int. Conf. Parallel Archit. Compilation Techn., 2011, pp. 155–166, doi: 10.1109/PACT.2011.21.
5. D. L. Dill, "The Murphi verification system," in Proc. 8th Int. Conf. Comput. Aided Verification, 1996, pp. 390–393.
6. M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for TSO," in Proc. 20th IEEE Int. Symp. High Perform. Comput. Archit. (HPCA), 2014, pp. 165–176, doi: 10.1109/HPCA.2014.6835927.
7. D. Lustig et al., "ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures," in Proc. ACM/IEEE Annu. Int. Symp. Comput. Archit. (ISCA), 2015, pp. 388–400, doi: 10.1145/2749469.2750378.
8. V. Nagarajan et al., A Primer on Memory Consistency and Cache Coherence, 2nd ed. San Rafael, CA, USA: Morgan & Claypool, 2020.
9. N. Oswald et al., "HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols," in Proc. ACM/IEEE Annu. Int. Symp. Comput. Archit.

(ISCA), 2020, pp. 888–899, doi: 10.1109/ISCA45697.2020.00077.

10. N. Oswald et al., "HeteroGen: Automatic synthesis of heterogeneous cache coherence protocols," in Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA), 2022, pp. 756–771, doi: 10.1109/HPCA53966.2022.00061.
11. C. Pulte et al., "Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8," Proc. ACM Program. Lang., vol. 2, no. POPL, Jan. 2018, Art. no. 19, doi: 10.1145/3158107.
12. M. Wang et al., "Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems," in Proc. ACM/IEEE Annu. Int. Symp. Comput. Archit. (ISCA), 2020, pp. 173–186, doi: 10.1109/ISCA45697.2020.00025.

NICOLAI OSWALD is a research scientist at NVIDIA, Zurich, 8004, Switzerland. Contact him at noswald@nvidia.com.

VIJAY NAGARAJAN is a professor at the Kahlert School of Computing, University of Utah, Salt Lake City, UT, 84112, USA. He is a Member of IEEE. Contact him at nvijayananad@gmail.com.

DANIEL J. SORIN is a professor at Duke University, Durham, NC, 27708, USA. He is a Senior Member of IEEE. Contact him at sorin@duke.edu.

VASILIS GAVRIELATOS is a senior researcher at Huawei Research UK, Edinburgh, EH3 8BL, U.K. Contact him at vasigavr1@gmail.com.

THEO X. OLAUSSON is a Ph.D. student at the Massachusetts Institute of Technology, Cambridge, MA, 02139, USA. Contact him at theoxo@mit.edu.

REECE CARR is a graduate software engineer at KAL. Contact him at mail@reececarr.com.