



Memory Consistency Model-Aware Cache Coherence for Heterogeneous Hardware

Rachel Cleaveland 
 Stanford University
 Stanford, CA, USA
 rcleavel@stanford.edu

Caroline Trippel 
 Stanford University
 Stanford, CA, USA
 trippel@stanford.edu

Abstract—Implementing cache-coherent shared memory in heterogeneous systems is challenged by memory consistency model (MCM) mismatches among compute elements: *what* the system-wide MCM should be and *how* it should be enforced are not well-defined. In this paper, we posit that C11—the seminal heterogeneous MCM—is the natural MCM choice for such systems. Based on this philosophy, we design and verify MEMGLUE, an update-based *consistency protocol* (i.e., an MCM-respecting coherence protocol) that enforces a slight strengthening of C11 among a set of interacting heterogeneous compute clusters.

MEMGLUE has three notable features. First, it is *modular*: any cluster equipped with a MEMGLUE translation *shim* can “plug into” any MEMGLUE system. Second, it is *verifiable*: one system-wide proof ensures that MEMGLUE upholds the C11 MCM with respect to MEMGLUE messages exchanged by clusters’ shims, and per-cluster proofs ensure that shims correctly translate relevant cluster coherence protocol messages to MEMGLUE protocol messages. Third, it is *polite*: MEMGLUE is compatible with a wide range of cluster coherence protocols and MCMs and exploits the permissible relaxed ordering behavior of each cluster to a high degree.

I. INTRODUCTION

Modern computer systems are increasingly heterogeneous: outsourcing computation from general-purpose CPUs to special-purpose hardware increases computational throughput while saving power [36], [25], [37]. And, as evidenced by the emergence of several industrial designs and standards (e.g., NVLink-C2C [5], CXL [71], CAPI [73], CHI [12], HSA [31], CCIX [2]), there is growing interest in implementing *cache-coherent shared memory* in such systems. Allowing components to share a coherent address space eases the burden of explicit memory management while reducing intra-system data movement and increasing performance [71], [42], [48], [58].

Unfortunately, implementing cache-coherent shared memory in heterogeneous systems is not straightforward. A core issue is that the heterogeneous processing elements comprising modern Systems-on-Chip (SoCs) [37], multi-chiplet designs [84], [3], and data centers [26], [22], [76], [40] feature disparate *memory consistency models* (MCMs) across their *instruction set architectures* (ISAs). That is, these processing elements assume/enforce differing restrictions on the ordering and visibility of (*all*) shared memory accesses [59]. Traditional coherence invariants order *same-address* memory accesses only [59]. Failure to also coordinate ordering among *different-*

address accesses in heterogeneous shared memory systems can lead to unexpected program outcomes [52], [63], [34], [9].

Most proposals for implementing heterogeneous cache-coherent shared memory today require hardware designers and software developers to collaboratively manage MCM diversity per system [14], [66], [5], [71].

Recent academic work shows that software developer burden can be alleviated by offloading the task of managing MCM heterogeneity to hardware *consistency protocols*. And, hardware designer effort can be reduced by automatically synthesizing a hardware consistency protocol per set of heterogeneous *clusters*, given per-cluster coherence protocol and MCM specifications as input [63]. (In this paper, a cluster denotes a group of *homogeneous* compute elements sharing a memory hierarchy.) Yet, the following key challenges remain.

First, synthesized consistency protocol implementations and the system-wide MCMs they intend to enforce (described as the “amalgamation” of the per-cluster MCMs [63]) are *unique* for distinct inputs to the synthesis procedure. Each protocol-MCM pair requires verification to ensure that synthesis did not unintentionally introduce protocol bugs—a notoriously difficult task [21], [65], [82], [15], [49], [55], [51], [54], [75], [53]. Second, deploying these consistency protocols requires explicitly merging, and thus modifying, cluster coherence protocol implementations (adding new transient states) and cache structures (combining clusters’ directory controllers and last-level caches). This strategy is not readily compatible with systems where some cluster’s memory system cannot be co-designed with the rest (e.g., SoCs/multi-chiplet designs with third-party cores/chiplets, data-centers). Third, clusters with non-multiple-copy-atomic (non-MCA) MCMs [61], [50], [10], [38] and update-based coherence protocols are not supported.

A. This Paper

Towards resolving the challenges above, we propose MEMGLUE, a universal hardware consistency protocol that is *verifiable*, *modular*, and *polite*. We coin the term *consistency protocol* to refer to an MCM-respecting coherence protocol, which provides coherent shared memory for arbitrary sets of heterogeneous clusters while upholding the memory ordering requirements of their respective ISA MCMs.

Insight 1: A universal consistency protocol enables verifiability and modularity: MEMGLUE is a universal consistency

protocol, meaning that its implementation and the system-wide MCM it enforces are the same for any composition of heterogeneous clusters. In particular, MEMGLUE enforces a slight strengthening of the C11/C++11 MCM [39], [46]—henceforth referred to as C11—among clusters. This is a natural design choice, since C11 was explicitly intended to be compatible with the breadth of modern ISA MCMs, and many new ISAs consider C11-compatibility a core requirement [78], [50], [80]. C11 defines a variety of memory and synchronization operations with different ordering *strengths* [4] attached to them, so as to closely match the ordering semantics of a wide range of ISA memory and synchronization instructions. MEMGLUE protocol messages directly adopt the syntax and semantics of these variable-strength C11 operations.

Per-cluster MEMGLUE translation *shims* translate relevant cluster coherence protocol messages to MEMGLUE protocol messages.¹ For correctness, a shim must consider the ordering requirements of the ISA instruction(s) that generate a particular coherence protocol message and output a MEMGLUE message of equal (or greater) ordering strength. However, shim design and its verification can be simplified thanks to MEMGLUE’s C11-centric design and the existence of formally verified compiler mappings from C11 to a variety of ISA MCMs [69]. In particular, a shim can select the strongest C11 operation that could have generated a given coherence protocol message, and output its matching MEMGLUE message.

Any cluster equipped with a MEMGLUE shim can “plug into” any MEMGLUE system and know that its memory ordering behaviors will adhere to its ISA MCM. Moreover, the MEMGLUE protocol, which coordinates C11-style operations among clusters’ shims, need only be designed and verified *once*. Shims are verified once per cluster, but are simpler and can reuse C11 compiler mappings [69]. By decoupling per-cluster translation logic from its system-wide protocol implementation, MEMGLUE adopts a modular design philosophy, which avoids explicitly merging clusters’ memory systems.

Insight 2: Update-based protocols enable politeness:

Ideally, a consistency protocol should be compatible with as many local cluster coherence protocols and MCMs as possible, while retaining performance of intra-cluster shared memory communication (compared to a homogeneous shared memory system), and maximizing performance of inter-cluster communication. We say that such a protocol is *polite*. Our goal of politeness, combined with the producer-consumer access patterns typically seen among clusters in heterogeneous systems [72], [79], motivates us to implement MEMGLUE as an *update-based* protocol [35] (related work adopts an *invalidation-based* approach [62], [9], [71], [63], [34]).

We show that as the fraction of clusters with weak (local) MCMs grows in a MEMGLUE system, so does the number of observable heterogeneous program execution behaviors (§VI-A). This means that MEMGLUE effectively exploits the permissible relaxed ordering behavior of its clusters’ MCMs.

¹A “shim” is “a thin piece of wood, rubber, metal, etc. which is thicker at one end than the other, that you use to fill a space between two things that do not fit well together” [67].

Thread 1	Thread 2
1: Wx = 1	3: Ry = 1
2: Wy = 1	4: Rx = 0

Fig. 1: Message Passing (MP) litmus test. Memory locations contain zero initially. The outcome is permitted by some MCMs (e.g., ARMv8 [11]), but not others (e.g., x86-TSO [8]).

Plus, MEMGLUE accommodates cluster coherence protocols and MCMs that current approaches for designing heterogeneous MCMs [63], [34] and their implementations as consistency protocols [63] do not, e.g., update-based protocols [63], [34], and non-multiple-copy-atomic (non-MCA) MCMs [63].

We summarize our contributions as follows.

- **MEMGLUE Approach.** We propose MEMGLUE, a universal consistency protocol for heterogeneous shared-memory systems. To our knowledge, MEMGLUE represents the only attempt to go beyond an operational model and implement C11 directly as a hardware protocol.
- **MEMGLUE Design.** We design MEMGLUE as an update-based protocol to accommodate clusters with a variety of local coherence protocols and MCMs and to optimize for inter-cluster producer-consumer communication patterns.
- **MEMGLUE Mur φ Model.** We implement MEMGLUE in the Mur φ model checker and prove that it *closely* upholds C11 for a suite of 6,738 litmus test programs.
- **MEMGLUE Correctness Proof.** We manually prove that MEMGLUE upholds C11 for all programs.

II. BACKGROUND AND MOTIVATION

Memory consistency models (MCMs) govern the ordering and visibility of shared memory accesses in parallel programs [59]. They define what program *executions*, and thus *outcomes* (mappings of a program’s shared memory loads to the values they return), are permitted/forbidden. For the same program, one MCM may permit an outcome that another forbids. Such distinctions are often captured using small parallel programs, called *litmus tests* (Fig. 1).

MCMs span the hardware-software stack: from high-level languages (HLLs) [19], [56], [18] to intermediate representations (IRs) [83] and ISAs [50], [64], [78], [11], [38]. Yet, for HLL programs, the HLL MCM ultimately dictates which of its executions are permitted, and compilation to IRs and/or ISAs must avoid creating more permitted execution possibilities.

A. Memory Consistency Model Overview

MCMs are often specified axiomatically [8], [18], [46], [50], [68] by defining a “happens-before” relation (\rightarrow_{hb}) between instructions that restricts which executions are permitted. Permitted executions are those *not* containing happens-before cycles. In Fig. 1, for example, a *strong* MCM may instantiate: $1 \rightarrow_{\text{hb}} 2 \rightarrow_{\text{hb}} 3 \rightarrow_{\text{hb}} 4 \rightarrow_{\text{hb}} 1$ (i.e., 1 happens-before 2, etc.). This cycle implies a contradiction—instruction 1 happened-before itself—indicating that the MCM disallows this execution. A *weaker* MCM may only instantiate $2 \rightarrow_{\text{hb}} 3$ and $4 \rightarrow_{\text{hb}} 1$, resulting in an acyclic execution that is permitted.

```

rs = [W] ; (sb&loc)? ; [W& ~ NA] ; (rf;rmw)*
sw = [REL | ACQREL | SC] ; ([F];sb)? ; rs ; rf ;
    [R& ~ NA] ; (sb;[F])? ; [ACQ | ACQREL | SC]
hb = (sb | sw)+
eco = rf | mo | fr | mo;rf | fr;rf
scb = hb | mo | fr
psc_base = ([SC] | [F&SC] ; hb?) ; scb ; ([SC] | hb? ; [F&SC])
psc_f = [F&SC] ; (hb | hb;eco;hb) ; [F&SC]
psc = psc_base | psc_f

```

Fig. 2: A subset of the C11 derived relations [46]. The $|$, $;$, $?$, $+$, and $*$ relational operators represent union, sequencing, union with the identity relation, transitive closure, and reflexive transitive closure. W, R, and F represent write, read, and fence instructions. ACQ and REL denote instructions with the C11 acquire and release memory orders, and so on.

B. The C11 Memory Consistency Model

MEMGLUE targets C11—the seminal heterogeneous MCM [19]—and more specifically, the RC11 variant [46]. From this point on, we use “C11” to refer to RC11, unless otherwise stated. C11 programs are intended to be compiled to and executed on virtually any hardware, despite the fact that each target ISA has its own MCM.

To leverage weak ISA MCM offerings, C11 provides several *memory orders* [4] (or “strengths”) for each memory and fence operation: *relaxed* (RLX), *acquire* (ACQ), *release* (REL), *acquire-release* (ACQREL), and *sequentially consistent* (SC).² Programmers may label writes as RLX, REL, or SC; reads as RLX, ACQ, or SC; and fences as ACQ, REL, ACQREL, or SC. The read and write components of atomic “read-modify-write” (RMW) operations can take on read and write labels, respectively, yielding RLX, ACQ, REL, ACQREL, or SC RMWs. The strength of each memory order subsumes that of all weaker orders, per the partial order $RLX \prec REL/ACQ \prec ACQREL \prec SC$. So, any ordering restrictions on RLX instructions will apply to REL instructions, and so on. Note that this partial order does not relate REL to ACQ, but both are stronger than RLX and weaker than ACQREL. Compilers translate these “labeled” C11 operations into sequences of load, store, and fence instructions in the target ISA, such that the C11 ordering guarantees will be upheld when compiled programs run on hardware [69].

As illustrated in Figs. 4, 5 and 6, C11 defines a variety of relations between operations, which it uses to specify legal program outcomes. *Base relations* include:

- **sb** (sequenced-before): describes program order.
- **rf** (reads-from): relates writes to same-address reads that read from them.
- **mo** (modification order): relates same-address writes in the order that they commit to memory.
- **fr** (from-reads): relates a read to a “newer” same-address write that happened mo-after the write that it read from.

²RC11 does not support C11’s *consume* memory order, as it is not used by major compilers [46].

```

Coherence = irreflexive (hb ; eco)
SC = acyclic (psc)
Atomicity = rmw  $\cap$  (fr;mo) =  $\emptyset$ 
No-Thin-Air = acyclic (sb | rf)

```

Fig. 3: C11 axioms [46].

- **rmw** (read-modify-write): relates the read component of an RMW to the write component.

Note that sb, rf, mo, and fr are execution-specific, with sb encoding a program’s dynamic control-flow and the others encoding its data-flow. The remaining C11 relations (Fig. 2) are derived from these base relations.

Two notable *derived relations* are sw (synchronizes-with) and hb (happens-before): hb is the union of sw and sb, and sw relates *release operations* to *acquire operations*.³ For example, as shown in Fig. 4a, when a release write (or any write that is sb-after a same-address release write) is related to an acquire read by rf, the release write is *also* related to the acquire read by sw. The sw relation can also involve fences. A release fence that is sb-before a write may feature an outgoing sw edge, and an acquire fence that is sb-after a read may feature an incoming sw edge. As shown in Fig. 4b, an rf edge between such a write and such a read instantiates an sw edge between their corresponding release and acquire fences. Note that release fences (release writes) can also be related to acquire reads (acquire fences) by sw.

The different C11 memory orders induce different relations between program operations, and thus different constraints on permitted program executions. RLX operations are subject to few restrictions, only guaranteeing *atomicity* (i.e., partially-performed writes cannot be observed) and *coherence* (i.e., all threads can agree on a total order in which same-address memory operations take place, or *SC-per-location* [47]). REL and ACQ operations further restrict legal program outcomes by requiring that all operations visible to a release must be visible to any acquire that is related to the release by sw. This requirement renders the outcome in Fig. 5a forbidden. Finally, all threads must agree on a total order in which SC operations take place. That is, SC reads on different threads may not disagree on the order of SC writes.

Overall, C11 defines legal program executions using four axioms (Fig. 3): Coherence, SC, Atomicity, No-Thin-Air.

The Coherence axiom states that pairs of operations related by hb in one direction may not be related by eco (Fig. 2) in the opposite direction. It enforces (among other things) SC-per-location. Fig. 5 shows how the outcome in Fig. 1 can be forbidden (5a) or permitted (5b) by Coherence depending on the strengths of the program’s C11 operations and how they instantiate hb.

The SC axiom asserts that SC operations must be totally ordered. Fig. 6 shows the Independent Readers, Independent

³“Release operations” (“acquire operations”) denote memory and fence operations whose strengths are at least as strong as REL (ACQ) per §II-B.

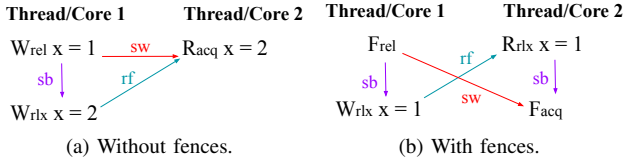


Fig. 4: Example instantiations of the sw relation.

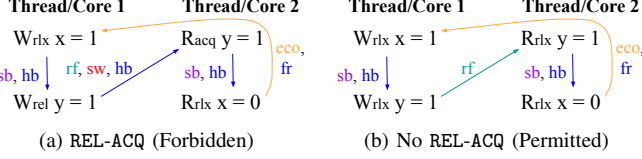


Fig. 5: MP litmus test variants. The REL-ACQ synchronization induces hb in Fig. 5a, resulting in a violation of Coherence.

Writers (IRIW) litmus test, which highlights this requirement and distinguishes orderings enforced by REL-ACQ synchronization (when sw involves at least one non-SC operation) versus SC-SC synchronization (when sw involves two SC operations).

Atomicity forbids intervening writes between any read/write pair related by rmw .

Compared to the original C11 [19], RC11 fixes issues with SC semantics and adds the No-Thin-Air axiom. No-Thin-Air requires that an execution cannot speculatively evaluate a read operation, such that this speculation satisfies itself through a cyclic chain of dependence [20]. That is, values cannot appear “out-of-thin-air.”

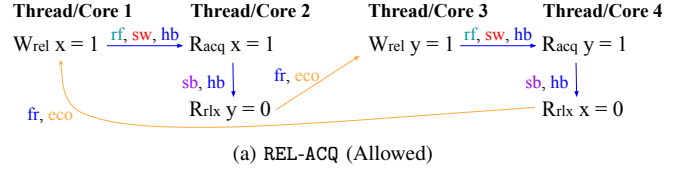
In practice, MEMGLUE implements a slight strengthening of RC11 that uses a *strictly stronger* version of the scb relation, which appears in the SC axiom [18]. RC11 weakened the scb relation to accommodate Power processors, which can produce outcomes that violate the SC axiom under the stronger scb definition when programs mix SC and non-SC operations [46]. For MEMGLUE’s purposes, implementing a strictly stronger variant of RC11 implies that RC11 is upheld. Further, doing so reduces MEMGLUE’s metadata requirement without deviating too far from (the weaker) RC11 behavior (§VI-A).

C. Update-Based Cache Coherence Protocols

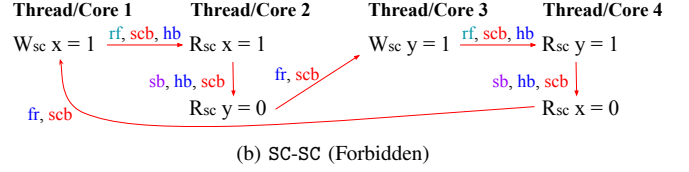
Coherence protocols may be *invalidation-* or *update-*based, and our MEMGLUE implementation adopts the latter approach.

Invalidation-based protocols require caches to send invalidation requests to remote cores when they want to perform a write. When a core wants to read a cache line that has been invalidated, it must request access to it through the protocol. Invalidation-based protocols often maintain the *single-writer, multiple-reader* (SWMR) invariant by requiring that all remote copies of a line be invalidated *before* a write may perform [59].

Update-based protocols [13], [74] replace invalidations with updates that propagate writes to remote cores as soon as they perform locally, trading off lower read latency for higher network traffic. In general, each cache write results in a message sent to all sharers, but remote cores always have the most up-to-date data and can thus perform reads immediately [35].



(a) REL-ACQ (Allowed)



(b) SC-SC (Forbidden)

Fig. 6: IRIW litmus test that is forbidden by C11 iff all the operations are SC, due to a cycle in scb that violates SC.

III. MEMGLUE PRELIMINARIES

We now give an overview of the MEMGLUE consistency protocol, before presenting two implementations (in §IV and §V) that make different assumptions about interconnection network ordering guarantees.

A. MEMGLUE Overview

MEMGLUE is an update-based consistency protocol that coordinates correct shared memory interactions among a heterogeneous set of compute clusters, which have been augmented with MEMGLUE translation shims (Fig. 7).

The MEMGLUE protocol operates within a fragment of C11 that includes RLX, ACQ, REL, and SC memory operations (including the read and write components of RMWs) and SC fences. We omit support for the strictly weaker REL, ACQ, and ACQREL fences for now. The semantics of RMWs is described in our manual proof in our open-source repository [1]. However, RMWs are not implemented in our Mur ϕ models, nor are they discussed in the paper for space reasons.

MEMGLUE shims intercept relevant coherence protocol messages internal to their local clusters and, based on their local ISA MCMs, generate C11-style messages (§III-B) to be handled by the MEMGLUE protocol.

We justify our decision to implement MEMGLUE as a novel (§III-A2) update-based (§III-A1) protocol below.

1) *Why Update-based Consistency Protocols?:* Recall our goal of designing a heterogeneous consistency protocol that is *polite* (§I). That is, MEMGLUE should not overly-restrict clusters’ (i) coherence protocols (invalidation- or update-based variants should be supported), (ii) MCMs (any MCM should be supported), (iii) performance on intra-cluster shared memory communication (operations on memory locations shared within a single cluster should perform comparably to when the cluster is not plugged into a MEMGLUE system), and (iv) performance on inter-cluster shared memory communication.

Requirement (iv) precludes consistency protocols that enforce SWMR (§II-C) among clusters, which subject inter-cluster communication to sequentially consistent ordering constraints [47], [59], [57]. Update-based protocols generally do not uphold SWMR [35], nor do certain invalidation-based

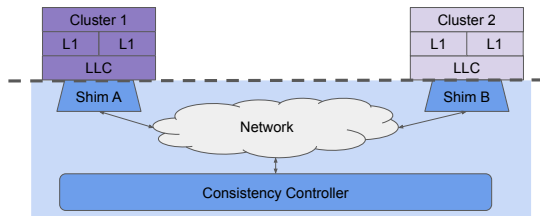


Fig. 7: MEMGLUE system with two heterogeneous clusters. MEMGLUE operates below the dashed line.

protocols, such as those that permit delayed invalidations [43], [44]. Such protocols are reasonable options for exploiting permissible relaxed ordering behaviors between clusters. However, for the reasons below, we elect to implement MEMGLUE as an update-based protocol.

Related to requirements (i) and (ii), we find that update-based consistency protocols easily support both update- and invalidation-based cluster coherence protocols (§VI-A demonstrates the latter), and enable a C11-centric design that can accommodate arbitrary cluster MCMs.

Related to requirement (iv), MEMGLUE orchestrates inter-cluster communication, which we anticipate to largely feature producer-consumer access patterns (e.g., a producer/consumer cluster writes to/reads from a shared queue [79]). For this style of communication, update-based protocols have been shown to perform better than invalidation-based alternatives [23]. One may initially worry about increased memory traffic between clusters. However, MEMGLUE is compatible with several performance optimizations for update-based protocols that reduce network traffic (e.g., an exclusive state [13], [74], competitive updates [35]). Plus, thread migration, which exacerbates update traffic in homogeneous shared memory systems [33], [35], is unlikely across heterogeneous MEMGLUE clusters.

2) *Why a Novel Protocol?*: We implement MEMGLUE as a novel protocol for two main reasons. First, our MEMGLUE implementation can be viewed as an abstract-machine operational model of (a slight strengthening of) C11. Due to its complexity, such a model for C11 has not yet been developed [30], [41], [60], and no existing coherence protocol comes close to approximating C11 behavior. Second, many prior update-based coherence protocols make restrictive assumptions about the orderedness of the network through which messages travel [74], [13], [81], [33], which we wish to avoid.

B. MEMGLUE Hardware Primitives

MEMGLUE introduces two types of hardware structures to mediate communication between clusters: per-cluster *shims* and a single system-wide *consistency controller* (CC).

Shims interface between local clusters and the MEMGLUE system. Within their local clusters, shims intercept relevant coherence protocol messages that are exchanged on behalf of ISA write, read, and fence instructions; translate them into their C11 analogs (§IV-D); and send WRITE, RREQ, and FREQ MEMGLUE messages, respectively, to the CC. From the CC, shims can receive WRITE, WRITE_ACK, RRESP, FREQ, and

	Remote Propagation	Local Propagation
MSI + TSO $ppo = sb \setminus (w, R)$	L1 cache write hit	Invalidate all locally cached copies. Write data to LLC.
Firefly + SC $ppo = sb$	Shared bus write	Place write update onto the shared bus.
RCC + RC $ppo = sb \setminus ((RLX, RLX) \cup (REL, RLX) \cup (RLX, ACQ))$	Shared L2 cache write back	Write data to shared L2 cache.

TABLE I: Local and remote write propagation strategies for shim integration for several protocols: MSI (invalidation-based) [59], Firefly (update-based) [74], and RCC (self-invalidation-based) [59]. Remote Propagation provides the local coherence actions that trigger a shim to send a WRITE update to the CC; Local Propagation provides the actions that a shim performs to propagate WRITE updates within its cluster.

FRESP messages (discussed in §IV-A). MEMGLUE messages contain different metadata to ensure they are correctly ordered by the protocol, such as C11-style strengths (RLX, REL, ACQ, or SC). The CC acts as the directory structure within MEMGLUE: all messages from the shims are sent to the CC, which orders, responds to, and reroutes them appropriately.

The shims and CC track additional metadata per valid cache line in existing cluster caches. A shim maintains a metadata cache that shadows its cluster’s shared last-level cache (LLC). Without loss of generality, we assume inclusive cluster LLCs. The CC acts as a directory for the full heterogeneous MEMGLUE system, maintaining data, metadata, and cluster-granularity sharer lists per cache line present in any of its clusters’ LLCs. A cache line tracked by a shim is invalid (valid) if its corresponding LLC cache line is invalid (valid). A cache line in the CC is invalid (valid) if it is invalid (valid) in every (some) cluster’s LLC. The shims and CC also track a timestamp per cache line (§IV-B1).

C. Write Propagation and Shim Integration

Equipping a cluster with a MEMGLUE shim requires determining (i) when intra-cluster operations should be communicated to remote clusters, and (ii) how MEMGLUE operations arriving from a remote cluster should be propagated internally.

The answer to (i), in short, depends primarily on what intra-thread write-write orderings the local MCM globally enforces.

The cluster actions which require external communication are cache updates (shims must update remote clusters), cache misses (shims must retrieve data and/or metadata), and fences (shims must synchronize with remote clusters). When the shims observe local coherence protocol messages indicative of these actions, they send WRITE, RREQ, and FREQ messages to the CC, respectively. Usually, WRITE and FREQ messages correspond to (committed) ISA instructions within the local cluster, so if a cluster’s MCM globally orders a pair of such instructions, its shim must send their generated MEMGLUE messages to the CC in the same order. For most cluster coherence protocols, where ISA fences do not generate protocol messages, the main task of a MEMGLUE shim is to preserve *globally-enforced* orderings among its cluster’s ISA writes.

Such globally-enforced orderings may order writes in different threads (inter-thread) or the same thread (intra-thread). For clusters with MCA MCMs, intra-thread orderings are typically captured by a *preserved program order* (ppo) relation [8]. For clusters with nMCA MCMs, they are often embedded in more subtle causality relations [50]. A shim can observe *inter-thread* write-write orderings (e.g., mo) at a cluster’s coherence ordering point; however, globally-enforced *intra-thread* write-write orderings may require shims to be placed higher (closer to cluster cores) within the memory hierarchy.

To see how a cluster’s intra-thread write-write ordering requirements inform shim placement, consider the following *total store order* (TSO) [70] and *release consistency* (RC) [59], [32] examples from the *Remote Propagation* column of Table I. For a TSO cluster, intra-thread $W \rightarrow_{sb} W$ ordering is preserved globally. Hence, its shim must send out a WRITE message upon each L1 write hit and therefore monitor all L1 cache interfaces. In contrast, an RC cluster preserves intra-thread $W \rightarrow_{sb} W_{rel}$ order globally, but not intra-thread order among non-*rel* writes.⁴ When a W_{rel} is performed at a core, all dirty data in the L1 are written back to the shared L2 (the cluster’s LLC) before the W_{rel} itself is written back to the L2. Thus, the shim need only monitor the cluster’s L2 interface.

To question (ii), MEMGLUE propagates incoming WRITE messages (which carry updates from remote clusters) within a cluster by leveraging its local coherence protocol. The *Local Propagation* column of Table I gives examples.

IV. ORDERED MEMGLUE CONSISTENCY PROTOCOL

We first present Ordered MEMGLUE (MEMGLUE_O), which assumes an ordered interconnection network (i.e., messages from the same sender to the same receiver arrive in the order they were sent). A complete specification of the protocol can be found in our open-source repository [1].

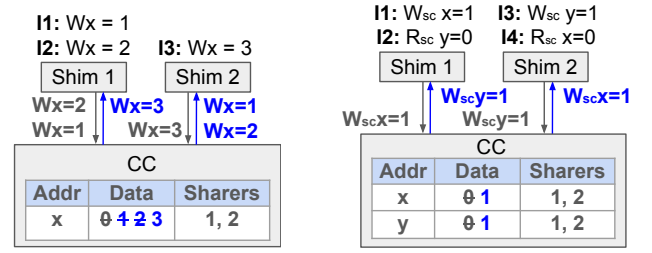
A. MEMGLUE_O Protocol

In this section, we present a simplistic view of MEMGLUE’s actions upon observing cluster-local instructions via their induced coherence protocol messages. In §IV-B we refine the MEMGLUE protocol to maintain the C11 axioms (§II-B).

Cluster writes. When a shim sees a cluster write (via a write hit or write-back, §III-C), it immediately sends a WRITE to the CC and updates its cache line’s state to valid within the shim (if it is not already). When the CC receives this WRITE, it writes its data into its own cache and forwards the WRITE to each cluster that is registered as a sharer of the updated cache line. The cluster whose shim sent the original WRITE message is added as a sharer. When remote sharers receive the WRITE, they propagate it within their clusters (Table I).

Cluster reads. Clusters may always service reads with data they have cached locally. On a read miss at the LLC, the shim sends a RREQ to the CC and does not service local cluster instructions until it receives back a RRESP. Upon receiving

⁴Note that RC’s W_{rel} operations have a slightly different semantics compared to C11’s W_{rel} operations, but are similar in spirit.



(a) Concurrent writes violate coherence. (b) SC writes violate the SC axiom.

Fig. 8: Motivating refinements to the MEMGLUE_O protocol.

the RRESP, the shim services the cluster read by supplying this data to its LLC and updating its state in the shim to valid.

Cluster fences. The shim sends a FREQ to the CC and stalls handling all cluster requests. The CC responds with a FRESP.

B. Refining the Protocol

We refine the §IV-A protocol in two ways to maintain C11.

1) *Timestamps:* Recall that the C11 Coherence axiom enforces SC-per-location (§II-B), which requires that all threads agree on a total order for same-address memory operations. The simple MEMGLUE protocol described in the previous section violates this notion.

Consider the example in Fig. 8a. Note the omission of instruction strengths; the problematic behavior of this example is present under any mapping of instructions to strengths. Without loss of generality, suppose the WRITES arrive in ascending order at the CC, and x is initially 0. To maintain SC-per-location, as required by Coherence, both shims must observe these same-address writes in the same order. However, under our current simple protocol, Shim 1 observes the write order to be 1, 2, 3, while Shim 2 observes 3, 1, 2, because each shim indiscriminately overwrites its local data with the forwarded WRITE updates it receives. MEMGLUE_O corrects this behavior with *timestamps*.

Each cache line’s metadata in the shims and CC is extended with a timestamp (TS) (Fig. 10). Each time the shims or CC process a cluster write hit / write-back (shims) or MEMGLUE WRITE (shims or CC), they increment the TS they track for the target cache line. On a write miss, a shim *synchronizes* its TS for the write’s cache line with the CC, by acquiring the CC’s TS and setting its local TS equal to it. Any WRITE sent from the CC to a shim is tagged with the CC’s TS. When the shims receive a WRITE, they perform a *timestamp check* to determine whether to propagate the WRITE’s data within their clusters.

Definition IV.1. For WRITE w , shim S , and address a , the timestamp check *determines whether w ’s timestamp exceeds the shim’s timestamp at a , i.e. $w.TS > shim[a].TS$.*

The WRITE is propagated within the local cluster *only if* the timestamp check passes; otherwise, its data is stale and must be discarded. In either case, the shim increments its local timestamp. Now, in Fig. 8a, when $Wx = 1$ arrives to Shim 2, its timestamp and the shim’s timestamp for x will be both be 1. Thus, the timestamp check will fail and the shim’s timestamp

will be incremented to 2 without overwriting the value 3. The same happens when $Wx = 2$ with timestamp 2 arrives at Shim 2. However, when $Wx = 3$ with timestamp 3 arrives at Shim 1, the timestamp check passes, and 3 is written. This means both shims will have data 3 and TS 3 in their caches at the end of the exchange. Using these timestamps, we prove that MEMGLUE_O upholds SC-per-location:

Theorem IV.1. *For all addresses a , $\exists \prec_a$ a total order on all writes to a , such that for all shims S , and any pair of reads $R \rightarrow_{sb} R'$ on S which read values w and w' , $w \preceq_a w'$.*

2) *SC Writes:* Consider the motivating example in Fig. 8b. Suppose both shims initially cache x and y , both with value 0. Given the current simple protocol, both WRITES are sent to the CC, and then both reads immediately read the cached data (0) before the remote WRITES arrive. The outcome would therefore be observable, despite being forbidden by C11’s SC axiom. To address this, MEMGLUE_O requires that a shim stop servicing local cluster requests after outputting an SC WRITE until it has received a WRITE_ACK back from the CC. This requirement forces prior SC WRITES (that reached the CC before the shim’s SC WRITE) to propagate to the shim before it may service future instructions. Doing so ensures that SC reads observe a total order for SC writes, as C11 requires (§II-B).

C. System-wide Proof of MEMGLUE_O

For MEMGLUE to uphold C11, the program executions observable in MEMGLUE must be a subset of those allowed by C11 (i.e., $\text{MEMGLUE} \subseteq \text{C11}$). In this section, we sketch three out of the four proofs that we conduct to verify that MEMGLUE_O upholds the C11 axioms for all programs. All four proofs—one corresponding to each C11 axiom (§II-B)—can be found in our open-source repository [1].

Each proof proceeds as follows. First, we assume the existence of an axiom-violating program execution. Then, we establish an order \prec_{CC} in which messages must have hit the CC for this execution to have been observable in MEMGLUE_O (e.g., $m0 \prec_{CC} m1$ means $m0$ hits the CC before $m1$). Then, we derive a contradiction ($\Rightarrow \Leftarrow$) that \prec_{CC} must contain a cycle, proving that the execution is not observable in MEMGLUE_O .

Coherence (§II-B): $\nexists I1, I2. (I1, I2) \in \text{hb} \wedge (I2, I1) \in \text{eco}$

Pf (sketch). We assume for sake of contradiction that such instructions $I1$ and $I2$ exist. We use the orderedness of the network and Thm. IV.1 to prove that instructions related by hb hit the CC in hb order. We then prove that eco -related instructions must hit the CC in eco -order by casing on each eco edge type. Then, $I1 \prec_{CC} I2$ because $(I1, I2) \in \text{hb}$, but also $I2 \prec_{CC} I1$ because $(I2, I1) \in \text{eco}$. $\Rightarrow \Leftarrow \square$

SC Axiom (§II-B): *acyclic (psc)*

Pf (sketch): Recall that psc orders SC operations with respect to one another (Fig. 2). Assume a cycle of psc edges exists in some program execution. We first prove that any psc cycle must contain at least one write or fence. Then we prove that all WRITES and FREQs generated in this cycle must hit the CC in psc -order, meaning that \prec_{CC} contains a cycle. $\Rightarrow \Leftarrow \square$

x86 instruction	Generated by (C11)	Translated to (MemGlue)
MOV (from memory)	L_{RLX}, L_{ACQ}, L_{SC}	L_{SC}
MOV (into memory)	S_{RLX}, S_{REL}, S_{SC}	S_{SC}
MFENCE	F_{SC}	F_{SC}

(a) TSO.

ARM instruction	Generated by (C11)	Translated to (MemGlue)
LDR	L_{RLX}	L_{RLX}
LDA	L_{ACQ}, L_{SC}	L_{SC}
STR	S_{RLX}	S_{RLX}
STL	S_{REL}, S_{SC}	S_{SC}
DMB ISH LD	F_{ACQ}	F_{SC}
DMB ISH	$F_{REL}, F_{ACQREL}, F_{SC}$	F_{SC}

(b) ARMV8.

Fig. 9: C11 compiler mappings, and MEMGLUE reverse compiler mappings for loads (L), stores (S), and fences (F). Recall that MEMGLUE does not yet support non-SC fences.

No-Thin-Air (§II-B): *acyclic (sb|rf)*

Pf (sketch). Assume a (sb|rf) cycle exists in some program. Then prove: $\forall I_1, I_2. (I_1 \rightarrow_{sb} I_2 \vee I_1 \rightarrow_{rf} I_2) \Rightarrow I_1 \prec_{CC} I_2$. A cycle in (sb|rf) thus implies a cycle in \prec_{CC} . $\Rightarrow \Leftarrow \square$

D. Per-Cluster Proofs

§IV-C presents a proof that MEMGLUE_O upholds correct C11 instruction orderings; it remains to be shown that shims correctly translate local coherence protocol messages to C11-style MEMGLUE messages. We design shim translation units assuming clusters run (correctly) compiled C11 code.

1) *Translation Scheme:* In a nutshell, shims observe cluster coherence protocol messages, determine the ISA instruction(s) that generate these messages, identify the *strongest* C11 operations that generate these instructions [69], and output the matching MEMGLUE messages. That is, we design shims to effectively invert (verified) compiler mappings from C11 to a cluster’s target ISA MCM,⁵ as illustrated in Fig. 9 [69].

Our translation strategy clearly enforces $\text{MEMGLUE} \subseteq \text{C11}$ in the absence of compiler optimizations. However, compilers may perform *legal* optimizations, which guarantee $\text{ISA} \subseteq \text{C11}$ [6], [16], [17], [29], [28], [77]. We sketch a proof by contradiction that in the presence of such optimizations, $\text{MEMGLUE} \subseteq \text{C11}$ still holds. Suppose that a program is compiled to p (unoptimized) and p_{opt} (legally optimized). Shims enforce $\text{MEMGLUE} \subseteq \text{C11}$ iff three conditions hold.

Condition 1: Instructions in p_{opt} are at least as strong as their p analogs. That is, under the mapping $orig : inst \rightarrow inst$ from instructions in p to their counterparts in p_{opt} , $\forall i, i'. (i, i') \in orig \Rightarrow stren(i') \preceq stren(i)$ (recall from §II-B that $RLX \prec REL/ACQ \prec ACQREL \prec SC$). Only a compiler optimization that relaxes the strengths of instructions in p to produce p_{opt} can violate the correctness condition above. But, such a relaxation would also violate $\text{ISA} \subseteq \text{C11}$, contradicting our assumption on legal compiler optimizations. $\Rightarrow \Leftarrow \square$

Condition 2: Source-to-source instruction reorderings, which happen at the C11 level before lowering to machine

⁵In the case of ISAs whose MCMs are not C11-compatible, MEMGLUE can translate all instructions to SC, but cannot exploit their relaxed MCMs.

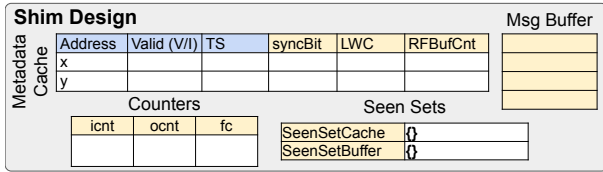


Fig. 10: MEMGLUE shim design. Blue components are those of MEMGLUE_O, yellow are those added by MEMGLUE_U.

code, may not violate C11. Legal compilers perform only source-to-source transformations which uphold C11 [77]. \square

Condition 3: $\forall I1, I2. (I1, I2) \in sb_p \wedge (I2, I1) \in sb_{popt} \implies (I1, I2) \notin sb_{cause}$ (we use sb_{cause} to capture those thread-local orderings that must be globally enforced by the ISA MCM). That is, instructions may only get reordered in the optimized program if such a reordering is permitted by the ISA MCM. Any compiler violating this condition may produce code that violates its ISA MCM. However, if compilers violate their MCM, they lose any provable guarantee that $ISA \subseteq C11$. Thus, such reorderings would not be legal. $\Rightarrow \neq \square$

V. UNORDERED MEMGLUE CONSISTENCY PROTOCOL

MEMGLUE is intended to be implemented over a network with no ordering guarantees, yielding Unordered MEMGLUE (MEMGLUE_U). As a motivating example, recall the programs from Fig. 5; consider what happens if the two writes on Core 1 are sent to Core 2 and arrive out-of-order. In Fig. 5b, MEMGLUE_U should not reconstruct the original ordering because the writes are allowed to be read out-of-order. However, in Fig. 5a, this reordering should not be visible due to the REL-ACQ synchronization between the cores. MEMGLUE_U must track enough metadata to distinguish cases like these and reconstruct the proper ordering of messages when necessary.

A. Reorderings Allowed in MEMGLUE_U

In this section, we distinguish between messages *arriving* versus *accepting* at a destination. A message *arrives* when it reaches its destination after being sent through the network. A message *accepts* (after arriving) once its destination is allowed, per MEMGLUE_U's state transition rules, to *process* it (e.g., update state, send response messages). A message *arrives early* if it reaches its destination before all prior messages from the same sender have been accepted at the destination. A message *accepts early* if it arrives early, and then is accepted before all prior messages from the same sender to the same destination have been accepted. Any MEMGLUE_U message may arrive early; only some may accept early.

MEMGLUE_U permits the following optimizations:

- 1) RLX reads from a cluster may read from WRITES that have arrived early to the shims.
- 2) RLX WRITES and RRESPs may accept early.
- 3) REL WRITES and ACQ RRESPs may accept early.

Each reordering is subject to certain constraints (§V-B).

MEMGLUE_U tracks additional metadata, shown in Fig. 10. To reconstruct the order in which messages were originally

sent to them from each sender, the shims and CC maintain a set of *message counters*: an *icnt* per (incoming) message source and an *ocnt* per (outgoing) message destination. These track the number of messages received at and sent by each shim/CC, respectively. The shims only have one source and destination for all messages, the CC, and thus only have one *icnt* and *ocnt*. A message arrives early if its *cnt* (i.e., the *ocnt* of its sender at the time it was sent) is more than one greater than the destination's *icnt* for its sender. When a message arrives early, but cannot accept early, it is queued in a *message buffer*. Messages are removed from the buffer either when enough prior messages have accepted such that the buffered messages may accept early, or when all messages from the same sender with a lower *cnt* have accepted. A counter *RFBufCnt* is tracked per cache line to maintain SC-per-location under the first optimization (see our open-source repository [1] for details). MEMGLUE_U also tracks *write_ids*, *seen_ids*, *seen_sets*, and *fence_counters* (§V-B2), as well as *local_write_counters* (§V-B1).

B. MEMGLUE_U Protocol

MEMGLUE_U enables significantly more reordering of protocol messages than MEMGLUE_O. We describe how it retains SC-per-location (§IV-B) and hb orderings (§II-B) below.

1) *SC-per-location*: Same-address write updates may arrive to the shims out-of-order, potentially causing a stale write to pass the timestamp check (Def. IV.1). This scenario would occur in the execution in Fig. 8a if the write updates of $x = 1$ and $x = 2$ arrive out-of-order to Shim 2. Therefore, in MEMGLUE_U, the shims must accept all same-address write updates in order. To this end, the shims track a *local_write_counter* (LWC) per address, and the CC tracks a LWC per address, per shim. The LWCs function similarly to the *icnts/ocnts* and ensure that same-address write updates accept in order. With this ordering guarantee, the normal timestamp check (§IV-B) may be used in MEMGLUE_U.

2) *Happens-Before Orderings*: To maintain the hb relation, instructions related by *sw* must correctly enforce orderings induced by release-acquire synchronization, as described in §II-B. This is difficult in MEMGLUE_U, as REL WRITES and ACQ RRESPs can arrive to the shims out-of-order with respect to write updates that happened before them. As an example, consider Fig. 11. Instruction 1 (I1) synchronizes with I2, meaning $I1 \rightarrow_{hb} I3$. However, I1 and I3's write updates may arrive out-of-order to Shim 3. This reordering would render the forbidden outcome in Fig. 11 observable, so it should not be allowed. However, some reordering of REL and ACQ messages should be allowed, if the shim has already seen⁶ the writes that are required in order to maintain *sw*-induced orderings.

To determine whether REL/ACQ reordering is allowed, we add (1) unique *write_ids* per write, assigned at the CC, (2) a *seen_id* per (REL/ACQ) message, to track the highest *write_id* a shim must see before accepting the message, and (3) two

⁶“Seen” is formally defined the proof [1]. Intuitively, a core has “seen” a write once no reads on that core can read a from another write older than it.

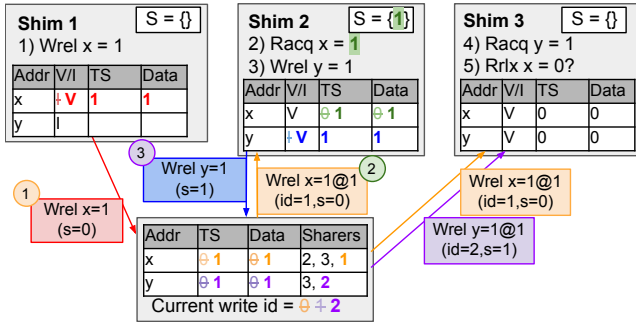


Fig. 11: Example of MEMGLUE_U forbidding an execution of a litmus test with REL-ACQ synchronization. Events unfold in rainbow order. Shim and CC structures have been simplified.

seen sets per shim, to track what writes each shim has seen. For simplicity, we elide details of the distinction between each *seen_set*; details can be found in our open-source repository [1]. When REL WRITES are sent to the CC, they carry with them the highest *write_id* that has previously arrived at the sending shim. In Fig. 11, for example, there are no writes in Shim 1’s *seen_set* (*S*) when I1 performs, so I1 carries *seen_id* = 0 with its REL WRITE update. This write gets assigned *write_id* = 1 at the CC and is then sent to Shim 2 as an update. When this update arrives and is accepted at Shim 2, *write_id* = 1 is added to Shim 2’s *seen_set*, signifying that any remote instruction that synchronizes-with any later instruction at Shim 2 must see I1. So, when I3 performs, its update to the CC carries *seen_id* = 1, and the forwarded update to Shim 3 carries this *seen_id* as well. Crucially, Shim 3 *cannot* accept this update until *write_id* = 1 is present in Shim 3’s *seen_set*. If Shim 3 is not registered as a sharer of the cache line associated with the update’s *seen_id* (i.e., it will never be forwarded an update with *write_id* = 1), then it will not be able to accept the update early. Hence, MEMGLUE_U will not allow I5 to read 0, which would violate Coherence. This “seen” logic prevents write updates from arriving before *hb*-prior messages, ensuring MEMGLUE_U honors *sw*.

Fences may also be related by *sw* (§II-B). As such, messages must not get reordered across them: all writes that happened before them must be seen by any read or fence that synchronizes with them. While trivial to achieve in MEMGLUE_O due to the orderedness of the network, in MEMGLUE_U we must add additional *fence counters* (*fcs*) to preserve these orderings.

When the CC receives a *FREQ*, it forwards it to all other shims. The CC’s *fcs* count how many *FREQs* are sent to each shim, and a shim’s *fcs* counts how many *FREQs* it has received. Each CC message to the shims is tagged with the CC’s *fcs* for that shim; when a message *msg* arrives to a shim, if *msg.fcs* ≠ *shim.fcs*, then *msg* has arrived before a prior fence. The shim buffers *msg* until it has seen *msg.fcs* total *FREQs*.

C. System-wide Proof of MEMGLUE_U

Coherence (§II-B): This proof proceeds exactly as the original proof (§IV-C), but with MEMGLUE_U’s ordering relaxations

factored in. For instance, to reason about orderings involving ACQ reads and REL writes, we introduce the “seen” relation in the proof, which is inspired by the intuitive definition we presented in §V-B2. We prove that if $I1 \rightarrow_{hb} I2$, then $I2$ “saw” $I1$, and that if $I2 \rightarrow_{eco} I1$, then $I1$ “saw” $I2$. This inverse seen relation presents our contradiction.

SC Axiom (§II-B): Since SC instructions are always accepted in order in MEMGLUE_U, this proof is nearly identical to the ordered proof. However, we reason differently about fences; when a fence is involved in an *sw* edge, it is necessary to prove that instructions that happen before a release fence are seen by all instructions that happen after an acquire fence (§II-B). We prove this via fence counters.

VI. VERIFYING MEMGLUE’S CORRECTNESS

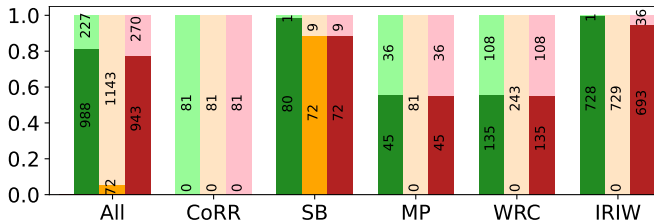
To verify MEMGLUE upholds C11, we (1) implement it in a model checker, and (2) complete a manual correctness proof.

A. Model Checking

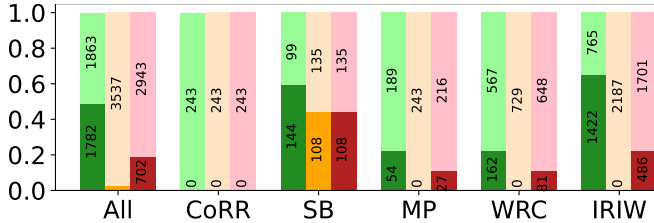
Mur ϕ is an explicit-state model checker for concurrent systems commonly used to verify cache coherence protocols [27]. We first implement MEMGLUE_O and MEMGLUE_U in Mur ϕ , and verify these implementations with respect to a test suite derived from the CoRR, SB, MP, WRC, and IRIW litmus tests [7]. The first suite of 1,215 tests features all variations of these litmus tests produced by assigning each instruction with each relevant C11 memory order (§III-A). The second suite of 3,645 tests is derived from the first by considering all possible placements of SC fences. For all tests, we treat clusters as black-boxes that emit MEMGLUE operations as defined in their assigned litmus test thread. Our goal is to verify the MEMGLUE protocol itself independent of shim translation.

We run each test through Mur ϕ to determine its observability in MEMGLUE, and through the herd tool [8] using the RC11 model [45] (axiomatically defined in the cat language [8]) to determine its allowability in C11. Fig. 12a shows the results of running these tests with Mur ϕ . For each implementation, no test forbidden by C11 is observable in either MEMGLUE variant—both uphold C11 with respect to the litmus tests. Also, MEMGLUE_U allows most of the behavior that C11 does, meaning that MEMGLUE_U’s reordering optimizations are indeed leveraging the reordering behavior that is allowable by C11 (and thus the weak MCMs C11 accommodates). While not a performance study per se, this result suggests that the MEMGLUE protocol itself should not overly restrict heterogeneous shared memory performance.

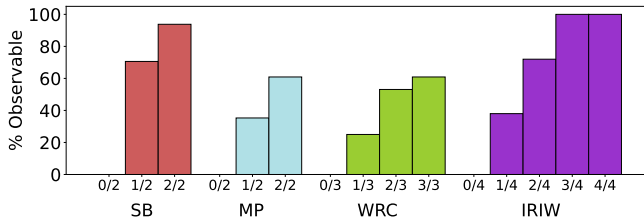
Next, we run a suite of 1,878 tests on MEMGLUE_U, in which we map all variations of the five tests across a set of “strong” and “weak” clusters. The “strong” clusters implement a standard MSI protocol locally [59], and we reverse-compile all instructions to SC to model a TSO cluster with a standard MSI coherence protocol (§IV-D1). The “weak” clusters are again black boxes in that they do not contain a local protocol, and reverse compilation could emit any combination of the atomics (modeling a cluster which maximally exploits the strengths offered by C11). The goal of these experiments is to



(a) Ordered (yellow) and Unordered (red) MEMGLUE results. Green columns show what is permitted in C11. Dark (light) colors are the portion of observable (unobservable) tests.



(b) Results of tests with all distributions of fences.



(c) Results of tests run on different fractions of weak (versus strong) cores.

Fig. 12: Litmus testing results.

demonstrate that MEMGLUE_U permits more relaxed behavior as the clusters it unifies become weaker.

Fig. 12c shows that as more litmus test threads are mapped to weak clusters, more reordering is allowed by MEMGLUE_U.

B. Proof

Since our Mur ϕ model checking results represent *bounded* proofs of MEMGLUE’s correctness guarantees, we construct a manual proof that for any program, none of its C11-forbidden executions are observable in a MEMGLUE system (as sketched in §IV-C and §V-C). This is a particularly important result of this work—proving that a cache coherence protocol implements a particular MCM is notoriously difficult, even with protocols and MCMs that are significantly simpler than MEMGLUE and C11 [15]. This proof is also reusable across all MEMGLUE-enabled systems; only the shim-local proofs need to be re-done for each new cluster.

VII. RELATED WORK

Coherence Interfaces: Some works propose novel coherence interfaces to support fine-grained heterogeneous coherence.

Crossing Guard [62] provides a MESI-style coherence interface between a host CPU and accelerators. Beyond correctness, the main goal of Crossing Guard is to ensure safe and reliable interactions of untrusted accelerators with the host, by defending against unauthorized data access, deadlock, and denial of

service attacks. However, to achieve some of these guarantees, Crossing Guard may require host coherence protocol changes.

Spandex [9] provides a richer coherence interface based on the DeNovo coherence protocol [24], with the primary goal of high-performance integration of heterogeneous devices with a wider range of coherence protocol demands. Spandex’s device-side logic and integration logic are comparable to MEMGLUE’s shims and CC, respectively. The authors discuss on how Spandex could be extended to account for inter-device MCM mismatches, but do not implement these extensions.

Instead of a coherence interface per se (like above or industrial approaches [5], [71], [73], [12], [31], [2]), MEMGLUE provides an MCM interface and adopts an update-based consistency protocol design to intercept and propagate relevant cluster operations, according to their ISA MCM requirements.

Consistency Protocols: HeteroGen [63], which synthesizes a consistency protocol for a particular set of heterogeneous clusters, is the first work to explicitly address MCM mismatches among clusters in heterogeneous coherence protocol design. §I discusses the trade-offs associated with this approach.

Follow-up work [34] presents a compositional operational model for defining *compound memory models*, which result from merging together per-cluster MCMs via a HeteroGen-style approach. The operational model can handle scoped and non-MCA cluster MCMs (unlike HeteroGen) by leveraging ordering relaxation in message propagation and predecessor tracking of memory operations—similar in spirit to MEMGLUE’s unordered update propagation and seen sets, respectively. However, the model assumes that threads have a global knowledge of where instructions have propagated in order to maintain correct instruction orderings, challenging its transformation into a concrete implementation. MEMGLUE, in contrast, is designed to be implementable in hardware.

VIII. CONCLUSIONS

MEMGLUE is an update-based consistency protocol that facilitates cache-coherent shared memory among heterogeneous clusters with diverse MCMs. To do so, it equips each cluster with a hardware shim that translates relevant cluster coherence protocol messages to C11-style MEMGLUE messages, and then coordinates the exchange of MEMGLUE messages among shims. We prove that MEMGLUE upholds C11 with respect to several thousand litmus tests (using model checking) and for all programs (with a manual proof).

ACKNOWLEDGMENT

We thank Grigory Chirkov and the anonymous reviewers for their constructive comments and feedback. This work was supported by the National Science Foundation (NSF) under the Graduate Research Fellowship Program and award number CAREER CCF-2236855. Rachel would also like to acknowledge and celebrate her late father, Dr. Rance Cleaveland, for his guidance, encouragement, and support throughout this project and all of her research work. He is dearly missed by his family, his friends, and the academic community.

REFERENCES

- [1] <https://github.com/rachelcleaveland/memglue-litmus-testing>.
- [2] Cache coherent interconnect for accelerators (ccix). <https://www.ccixconsortium.com/>. Accessed: 2023-08-14.
- [3] Heterogeneous integration roadmap 2021 edition. <https://eps.ieee.org/technology/heterogeneous-integration-roadmap/2021-edition.html>. Accessed: 2023-11-09.
- [4] memory_order. https://en.cppreference.com/w/c/atomic/memory_order. Accessed: 2023-07-11.
- [5] Nvidia grace hopper superchip architecture. Technical report, Nvidia Corporation, Santa Clara, CA, 2022.
- [6] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2007.
- [7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.
- [8] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014.
- [9] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. Spandex: A flexible interface for efficient heterogeneous coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.
- [10] Arm. Architecture reference manual, Armv7-A and Armv7-R edition, 2008.
- [11] Arm. Arm architecture reference manual, Armv8, for Armv8-A architecture profile, 2013.
- [12] Arm. Amba chi architecture specification, 2024. Accessed 31 July 2024.
- [13] Russell R Atkinson and Edward M McCreight. The dragon processor. *ACM SIGOPS Operating Systems Review*, 1987.
- [14] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M Balakrishnan, and Peter Marwedel. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, 2002.
- [15] Christopher J. Banks, Marco Elver, Ruth Hoffmann, Susmit Sarkar, Paul Jackson, and Vijay Nagarajan. Verification of a lazy cache coherence protocol against a weak memory model. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017.
- [16] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.
- [17] Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Interprocedural strength reduction of critical sections in explicitly-parallel programs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.
- [18] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. *43rd Symposium on Principles of Programming Languages (POPL)*, 2016.
- [19] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. *29th Conference on Programming Language Design and Implementation (PLDI)*, 2008.
- [20] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, 2014.
- [21] Sebastian Burckhardt, Rajeev Alur, and Milo MK Martin. Verifying safety of a token coherence implementation by parametric compositional refinement. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 2005.
- [22] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiu, and Doug Burger. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.
- [23] Liquan Cheng and John B Carter. Extending cc-numa systems to support write update optimizations. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
- [24] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.
- [25] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 2020.
- [26] Christina Delimitrou and Christos Kozyrakis. Quality-of-service-aware scheduling in heterogeneous data centers with paragon. *IEEE Micro*, 2014.
- [27] David L Dill. The mur ϕ verification system. In *Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8*, 1996.
- [28] Johannes Doerfert and Hal Finkel. Compiler optimizations for openmp. In *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings 14*, 2018.
- [29] Johannes Doerfert and Hal Finkel. Compiler optimizations for parallel programs. In *International Workshop on Languages and Compilers for Parallel Computing*, 2018.
- [30] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. Verifying c11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.
- [31] HSA Foundation. Heterogeneous system architecture: A technical review, 2012. Accessed 31 July 2024.
- [32] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *17th International Symposium on Computer Architecture (ISCA)*, 1990.
- [33] David B Glasco, Bruce A Delagi, and Michael J Flynn. Update-based cache coherence protocols for scalable shared-memory multiprocessors. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, 1994.
- [34] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. Compound memory models. *Proceedings of the ACM on Programming Languages*, 2023.
- [35] Håkan Grahn, Per Stenström, and Michel Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 1995.
- [36] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 2019.
- [37] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *Commun. ACM*, 2021.
- [38] IBM. Power ISA version 2.07, 2013.
- [39] ISO/IEC. Information technology – programming languages – C. International standard 9899:2011, 2011.
- [40] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.
- [41] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. *ACM SIGPLAN Notices*, 2017.
- [42] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE micro*, 2011.
- [43] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, 1992.
- [44] Leonidas I Kontothanassis, Michael L Scott, and Ricardo Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, 1995.
- [45] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. <https://plv.mpi-sws.org/scfix/>.
- [46] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *38th Conference on Programming Language Design and Implementation (PLDI)*, 2017.
- [47] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computing*, 1979.

- [48] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.
- [49] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, 2014.
- [50] Daniel Lustig, Sameer Sahasrabbudhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.
- [51] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhat-tacharjee. COATCheck: Verifying memory ordering at the hardware-OS interface. *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [52] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. *42nd International Symposium on Computer Architecture (ISCA)*, 2015.
- [53] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated memory consistency proofs for microarchitectural specifications. *Proceedings of the 51st International Symposium on Microarchitecture (MICRO)*, 2018.
- [54] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLCheck: Verifying the memory consistency of RTL designs. *Proceedings of the 50th International Symposium on Microarchitecture (MICRO)*, 2017.
- [55] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μ hb graphs to verify the coherence-consistency interface. *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.
- [56] Jeremy Manson, William Pugh, and Sarita V Adve. The java memory model. *ACM SIGPLAN Notices*, 2005.
- [57] A. Meixner and D.J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *International Conference on Dependable Systems and Networks (DSN'06)*, 2006.
- [58] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. Finepack: Transparently improving the efficiency of fine-grained transfers in multi-gpu systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.
- [59] Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.
- [60] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for c/c++ 11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.
- [61] NVIDIA. Parallel thread execution ISA version 6.0., 2017. <http://docs.nvidia.com/cuda/parallel-thread-execution/index.html>.
- [62] Lena E. Olson, Mark D. Hill, and David A. Wood. Crossing guard: Mediating host-accelerator coherence interactions. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [63] Nicolai Oswald, Vijay Nagarajan, Daniel J Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. Heterogen: Automatic synthesis of heterogeneous cache coherence protocols. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.
- [64] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, 2009.
- [65] Fong Pong, Andreas Nowatzky, Gunes Aybay, and Michel Dubois. Verifying distributed directory-based cache coherence protocols: S3. mp, a case study. In *EURO-PAR'95 Parallel Processing: First International EURO-PAR Conference Stockholm, Sweden, August 29-31, 1995 Proceedings 1*, 1995.
- [66] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.
- [67] Oxford University Press. Oxford advanced learner's dictionary, 2024. <https://www.oxfordlearnersdictionaries.com/>.
- [68] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages*, 2017.
- [69] Peter Sewell. C/c++11 mappings to processors. <https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html>. Accessed: 2023-07-11.
- [70] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 2010.
- [71] Debendra Das Sharma and Siamak Tavallaei. Compute express link 2.0 white paper. *CXL*. Retrieved October, 31:2021, 2020.
- [72] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 1990.
- [73] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. Capi: A coherent accelerator processor interface. *IBM Journal of Research and Development*, 2015.
- [74] Charles P Thacker and Lawrence C Stewart. Firefly: a multiprocessor workstation. *ACM SIGARCH Computer Architecture News*, 1987.
- [75] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [76] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.
- [77] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Moriset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the c11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.
- [78] Andrew Waterman and Krste Asanović. The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20190608-base-ratified. Technical report, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, June 2019.
- [79] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. Cohort: Software-oriented acceleration for heterogeneous socs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.
- [80] Will Deacon. Formalising the armv8 memory consistency model. <https://www.csm.ornl.gov/workshops/openshmem2018/presentations/mm-openshmem2018.pdf>, August 2018.
- [81] Andrew W. Wilson and Richard P. LaRowe. Hiding shared memory reference latency on the galactica net distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 1992.
- [82] Meng Zhang, Alvin R Lebeck, and Daniel J Sorin. Fractal coherence: Scalably verifiable cache coherence. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.
- [83] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012.
- [84] Zhen Zhuang, Bei Yu, Kai-Yuan Chao, and Tsung-Yi Ho. Multi-package co-design for chiplet integration. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022.