Compound Memory Models

ANDRÉS GOENS, the University of Edinburgh, United Kingdom
SOHAM CHAKRABORTY, TU Delft, Netherlands
SUSMIT SARKAR, University of St Andrews, United Kingdom
SUKARN AGARWAL, the University of Edinburgh, United Kingdom
NICOLAI OSWALD, NVIDIA, Switzerland
VIJAY NAGARAJAN, the University of Edinburgh, United Kingdom

Today's mobile, desktop, and server processors are heterogeneous, consisting not only of CPUs but also GPUs and other accelerators. Such heterogeneous processors are starting to expose a shared memory interface across these devices. Given that each of these individual devices typically supports a distinct instruction set architecture and a distinct memory consistency model, it is not clear what the memory consistency model of the heterogeneous machine should be. In this paper, we answer this question by formalizing "compound" consistency models: we present a compositional operational model describing the resulting model when devices with distinct consistency models are fused together. We instantiate our model with the compound x86TSO/PTX model – a CPU enforcing x86TSO and a GPU enforcing the PTX model. A key result is that the x86TSO/PTX compound model retains compiler mappings from the language-based (scoped) C memory model. This means that threads mapped to the x86TSO device can continue to use the already proven C-to-x86TSO compiler mapping, and the same for PTX.

CCS Concepts: • **Theory of computation** → **Program semantics**; • **Computer systems organization** → **Architectures**; • **Software and its engineering** → *Formal software verification*; • **Computing methodologies** → *Parallel computing methodologies.*

Additional Key Words and Phrases: compound memory models, consistency models, coherence protocols

ACM Reference Format:
Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proc. ACM Program. Lang.* 7, PLDI, Article 153 (June 2023), 35 pages. https://doi.org/10.1145/3591267

1 INTRODUCTION

This is the age of heterogeneity. Across a spectrum ranging from today's mobile phones to high-performance clusters, all have multiple different processors within them, including CPUs, GPUs and other accelerators. Such heterogeneous processors are starting to expose a shared memory interface across these devices.

Consider, for example, the recent announcement by NVIDIA [2022]: its upcoming server machine "Grace Hopper" pairs an ARM-based CPU with an NVIDIA GPU using an inter-device coherence protocol, providing a unified shared address space. In a similar vein, AMD [2022a] has announced

Authors' addresses: Andrés Goens, andres.goens@ed.ac.uk, the University of Edinburgh, Edinburgh, United Kingdom; Soham Chakraborty, TU Delft, Netherlands, s.s.chakraborty@tudelft.nl; Susmit Sarkar, Susmit.Sarkar@st-andrews.ac.uk, University of St Andrews, St Andrews, United Kingdom; Sukarn Agarwal, sagarwa2@ed.ac.uk, the University of Edinburgh, Edinburgh, United Kingdom; Nicolai Oswald, noswald@nvidia.com, NVIDIA, Zürich, Switzerland; Vijay Nagarajan, vijay.nagarajan@ed.ac.uk, the University of Edinburgh, Edinburgh, United Kingdom.

2475-1421/2023/6-ART153
https://doi.org/10.1145/3591267

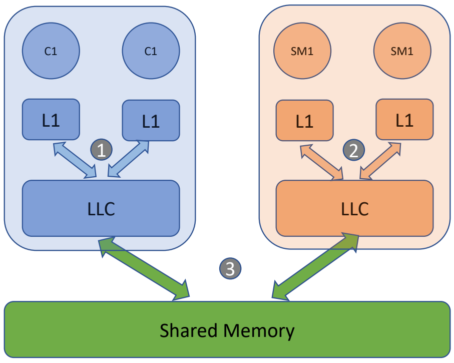

Fig. 1. In today's heterogeneous world, two (or more) devices with distinct consistency models are starting to share memory. This entails, among other things, stitching their intra-device coherence protocols (① and ②) together via an inter-device coherence protocol (③) such as for example CHI. Given that each of these devices support distinct memory consistency models, what memory model should the compound machine satisfy?

a machine pairing its x86 based EPYC CPU with its new Instinct MI200 Series GPU. Given that each of these devices support distinct instruction sets and memory consistency models – e.g., x86TSO [Sewell et al. 2010] and AMD [2022b] GPU – what memory consistency model should the heterogeneous machine support?

More generally, in this new heterogeneous world where devices with distinct memory consistency models are starting to share memory using an inter-device coherence protocol (Figure 1), what consistency model should the heterogeneous machine satisfy? In this paper, we provide a general operational semantics of the memory model of such a machine.

Motivation. Before we discuss our proposal, let us first motivate why addressing the memory model question is important. First, it serves as a good intermediate specification of the heterogeneous machine. Clearly, each of the devices need to be fused at the hardware level, which entails, among other things, fusing each of the intra-device coherence protocols using a global inter-device coherence protocol such as CHI by ARM [2021] or CXL [2022]. Just as clear is that it is possible to do this stitching incorrectly, harming either performance with too strong a model or, worse, correctness if the joint model is too weak. [1] We should guard against this. Having a good intermediate specification is thus necessary to ensure that the stitching is performed correctly.

Second, the surge of such coherent heterogeneous machines raises the question of how to program them. A memory consistency model of the compound machine is necessary to be able to program it using shared-memory paradigms. Even if one contends that getting to a language-level memory model such as OpenCL is the end, having a well-defined instruction-set-level heterogeneous memory model is a necessary means to that end.

1.1 Compound Memory Models

Let us come back to the question we posed: what should be the consistency model of the heterogeneous machine? We call our answer to this question a *compound memory model.* A compound memory model is not a new memory model; it is a compositional amalgamation where, informally speaking, threads from each device continue to adhere to the memory ordering rules of that device's original memory model.

In particular, the compound model resulting from joining two models is different from, say, the weakest of the models. Let us motivate the intuition behind compound models via an example. Suppose an NVIDIA GPU enforcing the PTX[2] memory model is fused with an x86TSO CPU. Consider the messaging passing litmus test where the producer thread is mapped to the PTX core and the consumer thread is mapped to the x86TSO core, as shown in Figure 2a.

[1]A model M1 is stronger than another model M2, iff M1 shows less behaviors than M2.
[2]PTX [Lustig et al. 2019] is a variant of the Release Consistency model with scopes.

$$\begin{array}{r||l} T_1(\text{PTX}) & T_5(\text{x86}) \\ m_1 : X = 1; & m_3 : r_1 = Y; \\ m_2 : Y^{\text{sys}}_{\text{REL}} = 1; & m_4 : r_2 = X; \\ \multicolumn{2}{c}{\text{Outcome: } r_1 = 1, r_2 = 0} \end{array}$$

(a) Message Passing (MP).

$$\begin{array}{r||l} T_1(\text{PTX}) & T_5(\text{x86}) \\ m1 : X = 1; & m3 : r_1 = Y; \\ m2 : Y^{\text{gpu}}_{\text{REL}} = 1; & m4 : r_2 = X; \\ \multicolumn{2}{c}{\text{Outcome: } r_1 = 1, r_2 = 0} \end{array}$$

(b) MP with a GPU-scoped access.

Fig. 2. Examples of message-passing litmus tests with behaviors that are unclear in a compound setting. The tags sys, cta refer to scoped accesses and REL to release accesses.

$$\begin{array}{r||l||l||l} T_5(\text{x86}) & T_1(\text{PTX}) & T_2(\text{PTX}) & T_6(\text{x86}) \\ & m_2 : r_1 = X^{\text{sys}}; & m_5 : r_3 = Y^{\text{sys}}; & \\ m_1 : X = 1; & m_3 : \mathsf{F}^{\text{sys}}_{\text{SC}}; & m_6 : \mathsf{F}^{\text{cta}}_{\text{SC}}; & m_8 : Y = 1; \\ & m_4 : r_2 = Y; & m_7 : r_4 = X; & \\ \multicolumn{4}{c}{\text{Outcome: } r_1 = 1, r_2 = 0, r_3 = 1, r_4 = 0} \end{array}$$

(a) Independent Reads of Independent Writes (IRIW).

$$\begin{array}{r||l||l||l} T_1(\text{PTX}) & T_5(\text{x86}) & T_6(\text{x86}) & T_4(\text{PTX}) \\ m1 : X^{\text{sys}} = 1; & m2 : r_1 = X; & m5 : r_3 = Y; & m8 : Y^{\text{sys}} = 1; \\ & m4 : r_2 = Y; & m7 : r_4 = X; & \\ \multicolumn{4}{c}{\text{Outcome: } r_1 = 1, r_2 = 0, r_3 = 1, r_4 = 0} \end{array}$$

(b) IRIW with x86 reader threads.

Fig. 3. Examples of IRIW litmus tests with behaviors that are unclear in a compound setting. The tags sys and cta refer to scoped accesses and SC refers to sequentially-consistent accesses.

If r_1 reads a 1, can r_2 ever read a 0? The x86TSO/PTX compound memory model would forbid this result. Informally speaking, this is because the ordering of m_1 before m_2 is enforced because m_2 is marked as a release; crucially, this ordering is enforced globally with respect to threads across the whole system because m_2 is system-scoped. In a similar vein, the ordering of m_3 before m_4 is enforced globally because of the ordering rules of x86TSO. Their combination, therefore, forbids this result. It is worth noting that the result is forbidden even though there is no fence between m_3 and m_4 in the x86TSO thread. (The consumer thread would have needed an acquire or a stronger fence in a memory model such as PTX.) This illustrates the fact that the compound x86TSO/PTX memory model is neither x86TSO nor PTX; it is their compositional amalgamation where, in this example, the producer thread behaves like PTX and the consumer behaves like x86TSO.

This compositionality property – with the fact that threads mapped to each device retain the memory ordering rules of their original model – is important for programmability. Indeed, programmers can program each device assuming the memory ordering rules of the device, irrespective of the other devices in the system. This would also mean that programs compiled for each device would continue to work correctly even when the device is fused with other devices.

Let us now consider a variant of this litmus (Figure 2b). Here, because m_2 is GPU-scoped, the ordering between m_1 and m_2 is enforced with respect to the GPU threads only – and crucially, this ordering is not enforced with respect to the CPU threads. Therefore, r_2 can read a 0 even when r_1 reads a 1.

The interaction of multi-copy-atomic memory models (such as x86TSO) with non-multi-copy-atomic memory models (such as PTX) raise more questions. Let us now consider the Independent-read-independent-write (IRIW) litmus test shown in Figure 3a. If r_1 and r_3 both read 1, can r_2 and r_4 both read 0? What if the readers are mapped to x86TSO cores and the writers are mapped to NVIDIA cores, as shown in Figure 3b? These are questions which our proposed model will answer[3].

[3]For the eager reader, both of these outcomes happen to be disallowed in our x86TSO/PTX compound memory model

1.2 Contributions

Addressing the question of memory models for heterogeneous systems, our contributions are:

- We propose an operational specification of a generalized compound memory model of a heterogeneous machine made up of multiple devices with distinct memory consistency models (Sections 3 and 4).
- We show our model can handle a range of MCMs, ranging from x86TSO, a non-scoped MCA model that is relatively strong, to PTX, a scoped non-MCA model that is relatively weak. (In this process, we provide for the first time an operational model of PTX.) Importantly, their combination yields a model of x86TSO/PTX, the resulting model when an x86TSO CPU is fused with a PTX GPU (Section 5).
- We prove that the x86TSO/PTX compound memory model retains compiler mappings from language-based (scoped) C memory model: threads mapped to x86TSO (PTX) device can continue to use the existing C-to-x86TSO (C-to-PTX) compiler mappings. To do so, we propose an axiomatic model CMM, that combines the axiomatic models of x86 and PTX and show that it abstracts our x86TSO/PTX operational model. Based on this axiomatic model, we prove the compiler mappings are correct (Section 6).
- We formalized and tested our model with several x86TSO, PTX, and compound litmus tests.
- Our experiments on the publicly available CPU-GPU model from AMD is consistent with our predicted compound memory model (when fusing x86TSO and AMD GCN3 GPU), and we observe that the soundness of the compound memory model hinges on the inter-device coherence protocol (Section 7).

2 BACKGROUND AND RELATED WORK

Hardware memory consistency models specify how memory must appear to the (systems) programmer. All major commercial (homogeneous) processors specify precisely-defined consistency models as part of the instruction set architecture (ISA) specification. Intel and AMD processors support the x86TSO consistency model [Sewell et al. 2010], whereas ARM [2018], RISC-V [Waterman and Asanovic 2019] and NVIDIA PTX [Lustig et al. 2019] processors support more relaxed models.

Single-/multi-copy atomicity and Scopes. An important feature of memory models concerns the manner in which stores propagate their values to other processors. In single-copy atomic (SCA) memory models, a load returns a value written by a single store and not a mishmash of values from two or more stores [ARM 2011; McKenney 2017]. Almost all memory models are SCA and in this work we restrict our attention to SCA memory models.[4] In *multi-copy-atomic* memory models (MCA) [McKenney 2017; Nagarajan et al. 2020][5], 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. Whereas x86TSO, ARMv8 (and later) and RISC-V support MCA models, models such as IBM Power and NVIDIA PTX do not have this property. Note that another feature of memory models concerns read-modify-write (RMW) instructions that atomically read and write to a location. We do not handle RMWs in our general operational model and leave this for future work, but our axiomatic model considers those. Similarly, we currently do not model mixed-size concurrency, where accesses can be misaligned; updating the compound model with mixed size along the lines of Alglave et al. [2021] and Flur et al. [2017] is also future work.

GPUs and heterogeneous memory models expose the hierarchical nature of the architecture through a thread hierarchy known as scopes [Hower et al. 2014]. The cooperative thread array (CTA)

[4]Note that some works (e.g., Zhang et al. [2017]) use the term SCA to mean a stronger form of MCA. We instead follow the more common terminology [McKenney 2017] which is also followed by ARM [ARM 2011] and RISC-V.

[5]By MCA we mean other-multi-copy-atomicity [McKenney 2017], the flavor of MCA supported by x86, ARM, and RISC-V.

scope, for example, refers to the set of threads that share the same GPU streaming multiprocessor (SM); These threads share the same L1 cache. The GPU scope refers to the set of threads that belong to a GPU and hence share a cache level, L2 for example. The system scope refers to the set of threads that span the whole heterogeneous system.

It is worth noting that our compound memory model specification can handle the composition of scoped, non-scoped, MCA as well as non-MCA memory models. In fact, our case-study considers the composition of x86TSO (which is non-scoped and MCA) with PTX (which is scoped and non-MCA).

Axiomatic and Operational Models. There are two broad approaches to precisely specifying memory models: axiomatic and operational. In the former [Alglave et al. 2014], mathematical predicates on relations between events are used to constrain the behavior of the system. An operational specification [Flur et al. 2016], on the other hand, specifies a memory model using operational semantics, akin to an abstract reference implementation. In this paper, we propose for the first time a generalized operational model with which we can specify and combine scoped, MCA and non-MCA memory models. We also design an axiomatic model as a proof tool to prove correctness of compiler mappings from language-level models.

2.1 Coherence Protocols

As part of supporting a memory consistency model, processors typically implement a coherence protocol that, informally speaking, makes writes and reads visible to all of the processors.

There are many coherence protocols that have appeared in the literature and been employed in real processors. Some of these protocols—especially the ones targeted towards the CPUs—enforce the Single-Writer-Multiple-Reader (SWMR) invariant by invalidating sharers on a write [Nagarajan et al. 2020]. GPU protocols directly enforce relaxed consistency models without SWMR by eschewing writer-initiated invalidations, and instead rely on writebacks and self-invalidations.

Coherence protocols are the key to supporting heterogeneous shared memory. In fact, industry has been developing coherence protocol standards such as CAPI [The OpenCAPI Consortium 2021], CHI [ARM 2021] and CXL [2022] for fusing the intra-device coherence protocols together to create heterogeneous shared memory. In this work, we answer the question of what would be the resulting memory consistency model when individual devices that enforce distinct consistency models are fused together using standardized coherence interfaces such as CHI and CXL.

2.2 Consistency for Heterogeneous Processors

Hower et al. [2014] introduce heterogeneous race free (HRF) memory models that accommodate synchronization operations with different scopes; specifically, HSA [HSA Foundation 2012] is a heterogeneous consistency model that is based on HRF. However, HRF does not address the composition of distinct constituent consistency models: e.g., it does not answer the question of what happens when a non-scoped model is fused with a scoped memory model.

Compound memory models were introduced informally by Nagarajan et al. [2020], and formalized subsequently by Oswald et al. [2022] under the assumption that every constituent memory model is both MCA and non-scoped. However, GPU memory models involve scopes, and can often be non-MCA. We are the first to formalize compound consistency for scoped, non-MCA models.

Iorga et al. [2021] formally specify the memory model of heterogeneous CPU/FPGA systems axiomatically as well as operationally and validate their models. In contrast, in this work we specify more generally how different memory models can be composed together. In his position paper, Batty [2017] argues for a compositional approach towards relaxed memory consistency. Our compound consistency models, and specifically the fact that they preserve compiler mappings within each cluster, are a step in this direction.

3 OVERVIEW: COMPOUND MEMORY MODELS

3.1 Problem Formulation

Let us recall the question we are asking. When two devices with distinct memory consistency models are fused together via an inter-device interconnect, what model should the compound machine satisfy? How does one precisely specify this compound memory model?

Note that it is easy to come up with vacuous answers to the question, and indeed, there is a large design space of possible models for the compound system. As an extreme example, one could mandate that the result of fusing two different memory models is always sequential consistency (SC)[Lamport 1979]). Clearly, such a specification is not very useful as that would mandate wholesale changes in the implementation of the machines, in addition to imposing performance penalties. On the other extreme, one could mandate that the result of fusing two different memory models is the weakest of the two. Aside from the fact that two memory models can be incomparable, this could break the correctness of programs compiled to the stronger model.

Consider the scenario when two machines with memory models M1 and M2 are fused together with M1 strictly stronger than M2. Consider a multi-threaded code sequence written for M1 that is intended to be run entirely in the device satisfying M1. Clearly, it is desirable for that code to work correctly in the fused machine as well. Otherwise, legacy code written for M1 will have to be rewritten or recompiled every time M1 is fused with another machine. In a similar vein, it is desirable that code running only on the machine satisfying M2 should not suffer any performance penalties simply because it is fused with M1; nor should the compound memory model specification mandate that the architects of M2 redesign the internals of the machine and make it stronger simply because it is being fused with the stronger machine.

We want the compound memory model to be *compositional*. We want every memory operation from a device with memory model M to continue to behave as specified by M, even when that device is fused with other devices of different memory models. An x86TSO load, for example, should continue to behave like an x86-TSO load when an x86TSO device is fused with, say, a PTX GPU. We saw this in the example from Figure 2a, where the x86TSO consumer thread did not require any additional fences because an x86TSO load has acquire semantics. We want to define our compound memory model in a way that ensures this property.

It is clear how we can tell whether the compound memory model is weak enough: the behavior of a program that runs entirely on one of the individual machines is precisely the same as if running on that machine in isolation. But, how can we tell that the semantics we propose is also strong enough? We rely on mappings to language-level memory models. Given a language-level memory model such as OpenCL or C, every instruction-set-architecture (ISA) has mappings from language-level synchronization primitives to memory operations in that ISA [Sewell 2022]. We want to come up with a general semantics for compound memory models that *retains* these mappings: a compiler that generates code for the compound machine should be able to simply use the pre-existing mappings.

In summary, the semantics of the compound memory model needs to be strong enough to *retain compiler mappings* while being weak enough that each of the subsystems representing the different architectures is *as weak as the original architecture*. Intuitively, the compound memory model implicitly places constraints on the inter-device interconnect used to stitch the machines together.

3.2 A Compositional Memory Model Semantics

We now provide the high-level intuition behind our compound memory model semantics, and we also explain how it satisfies compositionality.

We base our approach on the POP operational model [Flur et al. 2016], which was originally used to describe an ARM memory model. Roughly speaking, the model consists of a set of threads, each

T_1(x86)	T_2(ARM)
	m_3 : `ld` r_1 Y;
m_1 : `MOV` X 1;	`cmp;bc;isb;`
m_2 : `MOV` Y 1;	m_4 : `ld` r_2 X;
	`cmp;bc;isb;`

T_5(x86)	T_1(Power)	T_2(Power)	T_6(x86)
	m_2 : `ld` r_1 X;	m_5 : `ld` r_3 X;	
	`cmp;bc;isync;`	`cmp;bc;isync;`	
m_1 : `MOV` X 1;	m_3 : `hwsync;`	m_6 : `hwsync;`	m_8 : `MOV` Y 1;
	m_4 : `ld` r_2 Y;	m_7 : `ld` r_4 Y;	
	`cmp;bc;isync;`	`cmp;bc;isync;`	

Fig. 4. The MP and IRIW litmus tests compiled from C11 to x86 + ARM/Power with the mappings in Lahav et al. [2017].

of which propagates reads and writes to the other threads in an order that satisfies the memory model constraints. We generalize and make extensive adaptations to POP so that it can be used to compositionally specify a range of different memory models.

There are several aspects that characterize memory models, such as ordering, multi-copy-atomicity (MCA), and scoped operations. Our key insight is that these can all be modeled operationally in a compositional manner using a principle we call LOST: **L**ocal **O**rderings by **S**talling on **T**hreads. Aside from memory-model-agnostic actions that involve propagating reads and writes, all of the memory-model-specific actions are in the form of thread-local operations and stalls. This allows us to instantiate different memory models by simply configuring their respective thread subsystems, leading to a memory model that is naturally compositional.

In the following we explain the intuition of how our model enforces ordering, MCA, and scopes using the LOST principle.

3.2.1 Ordering. Arguably the defining aspect of weak memory models is the relative orderings between memory operations. For certain pairs of memory operations (i.e., reads or writes), the program order has to be preserved globally.

We assert that for a memory model to be compositional, it is necessary that the order relations in each memory model are preserved in threads across all machines in the system. To understand why this condition is necessary, consider the message passing litmus test (cf. Figure2a) written in C, with a write-release write on the producer's side (m_2) and read-acquire on the consumer's side (m_3). Say that the producer thread is mapped to an x86TSO machine and the consumer thread is mapped to an ARM one. Based on the mappings described in Lahav et al. [2017], the assembly code shown on the left in Figure 4 is generated. For existing mappings (from C to each of the constituent memory models) to be preserved, this litmus test should continue to be disallowed, since the original C litmus is disallowed. This, in turn, mandates that $m_1 \rightarrow m_2$ from the x86TSO thread T_1 should be preserved with respect to the consumer thread T_2 from the ARM machine. Likewise, $m_3 \rightarrow m_4$ should be preserved with respect to the producer thread from the x86TSO machine as well.

To ensure this is the case in our model, the orderings $m_1 \rightarrow m_2$ and $m_3 \rightarrow m_4$ are both enforced by the x86TSO and ARM threads in a thread-local fashion, and thus independent of the other thread following the LOST principle. Specifically, in the x86TSO thread, the second write m_2 is stalled until the first write m_1 is globally propagated. Likewise, in the ARM thread, the second read after the fence is stalled until the first read completes.

To enforce this, the threads need to know whether their operations (or other operations they have seen) have been globally executed, i.e. seen by all threads in the system. Importantly, while this information is not local to the thread, it is architecture-independent.

3.2.2 Multi-copy atomicity. MCA is about whether a load is allowed to read the value of a store from another thread early, i.e., before the store becomes globally visible. MCA loads (e.g., x86TSO loads) are forbidden from doing so; in effect, the store (from which the load reads the value) is globally ordered before the load. On the other hand, non-MCA loads (e.g., in IBM Power) are allowed to read early. In memory models that are not MCA, there are special fences that restore the MCA property on demand; when such a fence is executed, it globally orders stores before the fence, if the values produced by the stores had been read by loads from that fence's thread.

MCA loads, non-MCA loads, and fences can all be modeled using the LOST principle. Non-MCA loads are modeled by removing the read as soon as it is satisfied (as will be explained in Section 4.2), thereby making it possible for the successors of the load to propagate before the store (from which the load read its value) propagates. How about MCA fences? When a non-MCA read is satisfied, we still remember the write (from which the read returns) as a "predecessor" of the reading thread, so that when an MCA fence is encountered, that fence stalls until each of the predecessors accumulated thus far has been globally propagated. Note that each of these cases – MCA read, non-MCA read, and MCA fence – are modeled via local operations, thereby adhering to the LOST principle.

Consider the IRIW litmus test (cf. Figure 3a) assuming a non-MCA model such as Power. Note that the two fences m_3 and m_6 are MCA fences (e.g., `hwsync`). Suppose each of the two loads m_2 and m_5 read a 1, while load m_4 reads a 0; is it possible for load m_7 to read a 0? If m_2 from T_2 reads a 1, it would remember the store m_1 as a predecessor; when the MCA fence m_3 is encountered, it stalls the fence until the predecessors are globally propagated. This ensures that m_1 becomes visible to T_4, and hence load m_7 would read a 1.

Now consider the compound case. The model is compositional only if the reads continue to function like before: i.e., MCA if they are in an MCA-thread, and non-MCA otherwise. Consider the same IRIW litmus test for the case in which an x86TSO device (that is MCA) is fused with Power (which is non-MCA). Suppose the store threads are compiled to two threads from x86TSO and the load threads are mapped to Power. Based on the mappings in Lahav et al. [2017], the assembly code shown on the right of Figure 4 is generated. Note that in order to preserve the correctness of the compiler mappings, the compound model should continue to disallow the outcome. This, in turn, mandates that the two `hwsyncs` (m_3 and m_6) should make the stores (m_1 and m_8) globally visible even though the 2 stores are from a different device. Note that this is naturally obtained in our model owing to the LOST principle. Specifically, we mark the two stores m_1 and m_8 as *predecessors* of thread T_2 and T_3 respectively. The `hwsyncs` take these predecessors into account when stalling; specifically m_4 (m_7) stalls until m_1 (m_8) is propagated globally.

3.2.3 *Scoped Operations.*

Thus far we have not explicitly considered scoped memory models. In the following we will show how to model scoped requests and how to compose a scoped model with a non-scoped model.

We model scopes by associating a scope with every order constraint: operations only stall until certain preceding operations are visible at a given scope. We address compositionality by treating every non-scoped order constraint coming from a non-scoped memory model as system scoped (i.e., global scope). We explain these additions with the help of a running example.

Consider a CPU-GPU system similar to the one shown in Figure 1, the GPU satisfying PTX [Lustig et al. 2019] (a scoped memory model) and the CPU satisfying x86TSO (a non-scoped memory model). Let's assume that hardware threads T_1 and T_2 are mapped to the streaming multiprocessor SM_1, T_3 and T_4 are mapped to SM_2; and CPU threads T_5 and T_6 are mapped to CPU cores C_1 and C_2 respectively. Scopes refer to a thread hierarchy where each scope refers to a set of threads. Consequently, threads T_1 and T_2, which are guaranteed to be mapped to SM_1, belong to one CTA: $\text{CTA}_1 \supseteq \{T_1, T_2\}$; threads T_3 and T_4 which are guaranteed to be mapped to SM_2 potentially belong to a different CTA: $\text{CTA}_2 \supseteq \{T_3, T_4\}$. The GPU scope refers to the set of threads within the GPU: $\text{GPU} \supseteq \{T_1, T_2, T_3, T_4\}$. The system scope refers to the set of threads within the whole system, i.e. $\text{Sys} \supseteq \{T_1, T_2, T_3, T_4, T_5, T_6\}$.

Consider the message passing litmus test shown in Figure 2a where the producer is mapped to PTX thread T_1 and the consumer is mapped to x86TSO thread T_5. Note that memory operation m_2 is a system-scoped release, and so it induces an ordering $m_1 \longrightarrow m_2$ across the whole system. In particular, m_1 and m_2 are seen by thread T_5 in order. Consider now the two loads m_3 and m_4 from x86TSO thread T_5. The two loads are ordered as per the x86TSO memory model. Because non-scoped accesses in other machines are treated like system-scoped accesses, these two loads are propagated to all threads in the order: $m_3 \longrightarrow m_4$. In particular, the two loads reach T_1 in order, ensuring that the outcome $r_1 = 1, r_2 = 0$ is forbidden.

Suppose the acquire is tagged CTA scope (and not system scope). This would relax the ordering guarantee of the two stores m_1 and m_2 only to threads that are in CTA scope, i.e., to threads T_1 and T_2. In particular, there is no guarantee that the two stores would reach thread T_5 in order, and this would imply that the outcome $r_1 = 1, r_2 = 0$ is allowed.

It is worth noting that all of this can be modeled without violating LOST as long as threads "know" for each operation the set of threads that operation has propagated to. Just as with the global case, this information is architecture-independent, as it only concerns which operations have been seen in which scope, not any architecture-specific properties of the operations.

4 THE LOST-POP MODEL

In this section, we discuss our operational model in detail. It is based on POP but uses the **L**ocal **O**rderings by **S**talling on **T**hreads (LOST) principle. (We summarize the technical differences between POP and LOST-POP in Section 4.5.) After introducing the general state and transitions behind LOST-POP, which are the memory-model-agnostic components, we discuss how we concretely implement the LOST principle outlined in Section 3. In particular, we show how we can model MCA and non-MCA reads, different fence orderings, and scopes. We conclude by discussing how two or more instances of LOST-POP can be composed.

4.1 Basic Model

The original partial-order propagation (POP) [Flur et al. 2016] model was used to model a concrete memory consistency model (MCM), a non-MCA version of ARMv8. The key insight of POP is modeling the propagation of each memory request individually to the different threads in the system. When a request propagates to a thread, it defines a partial order with memory requests in that thread. This partial order constraints the future propagation of requests to other threads. Our generalization takes this insight and applies the LOST principle to create a model that can be used to describe different MCMs in a composable, thread-local fashion.

The propagation of memory requests is a key step in our model: when a write propagates to a thread, it means its effects are visible to that thread. Counter-intuitively, thus, LOST-POP has no explicit model of memory (just like POP). When a write propagates to all threads, it has effectively been "written to memory". More generally, the set of threads to which the request has propagated intuitively corresponds to a part of the memory subsystem, e.g. a cache level. Similarly to writes, reads also propagate to the different threads in the system. Intuitively, the idea is that a read gets its value from a subset of the memory subsystem, which corresponds to the threads it has propagated to. The behavior of the model is, in essence, defined by thread-local constraints on propagation, implementing the LOST principle.

Consider the example in Figure 5, which represents the message-passing litmus test (c.f. Figure 2a). A thread (T_1) accepts a write request $X = 1$. This request starts by propagating to its own thread T_1 (1). Then, another thread (T_2) accepts a read request $b = X$, which similarly starts by propagating to its own thread (2). The write request of T_1 then propagates to its adjacent thread T_2 (3). When this happens, since the two requests are to the same address, they have to be ordered. The incoming

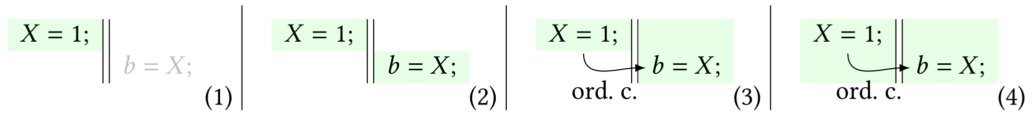

Fig. 5. An example of request propagation in the LOST-POP model. After accepting the write $X = 1$ in T1 (2) and the read $b = X$ in T2 (2), the write $X = 1$ propagates to T2 (3), where it gets ordered with the read $b = X$. The green cells represent propagation, whereas the arrow represent the order constraints. Finally, the read request also propagates to the first thread (4), represented by the green-colored cell in T1.

request (in this case the write request) is ordered before the other request (the read). Finally, the read request propagates to $T1$ (4), and now the read can be satisfied by the write.

More generally, the building blocks of the LOST-POP model are *requests*, which can be either *memory* requests (*reads* or *writes*) or *fences*. These are general classes of requests, the concrete instructions are architecture-dependent. Besides requests, an instance of the model also consists of a system, which has a set of *threads*. All requests have an associated thread, while memory requests also have an associated address (to read from or write to).

We have a set of *order constraints*, a binary relation on requests labeled with a scope. We write $r \longrightarrow r'$ to say that there is an order constraint from (r, r'). Intuitively, these order constraints represent a causality relation, ordering some requests before the other. Order constraints are updated according to so-called *order rules*, which say for a given r, r' whether to add the constraint $r \longrightarrow r'$. These order rules relate two types of instruction and are accordingly also architecture-specific.

Concretely, the model has three basic kinds of transitions: accepting a request in a thread, propagating a request to a thread or satisfying a read request.

Accepting a request. Accepting a request r is simple: the request is just added to the system state and propagated to the thread t that accepted it. If other requests have already been accepted to that thread, we order them before the incoming request (respecting the order rules). More precisely, we add an order constraint r', r for each r' that has been accepted by t, such that $\text{order}(r', r)$ holds.

Propagating to a thread. A request r can propagate to any thread t it has not propagated to (in arbitrary order), on one condition: no previous (according to the order constraints) request r' can be stalling it. If request r' originates from a different thread and there is an order constraint $r' \longrightarrow r$, then r' has to propagated to t before r. This ensures causality is preserved by the model. According to the LOST principle, model-specific stalling has to come from another request r' *in the same thread*. We will discuss the concrete variants of stalling semantics below.

Once the request propagates, the order constraints are updated again, but the incoming request gets ordered to be before the others. By the LOST principle, however, contrary to the orderings generated when accepting a request, these orderings are not architecture-specific. Concretely, for every request r' in thread t that is not already ordered with r, we add an order constraint r, r' iff r' is a memory request to the same address and they are not both reads. An ordering between two writes enforces a write serialization (coherence) ordering on the entire system[6], while orders between reads and writes just define whether a read happens before or after a write. Such edges correspond roughly to the axiomatic notions of `co` (or `ws`), `rf` and `fr` edges. Note that these edges can go either way, depending on which request (non-deterministically) propagates first. They correspond to *observations* of the execution, not constraints placed by the architecture.

[6]Note that this implies that our operational model always enforces a causality of write serialization constraints

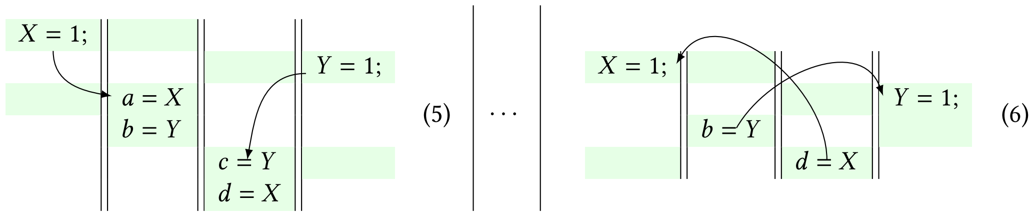

Fig. 6. An IRIW test in POP. First the write $X =_1$ propagates to T_2 and, correspondingly, the write $Y = 1$ to T_4, after which the reads can proceed to propagate back to the write's threads (5). After the reads are satisfied and removed, the constraints are removed. The second reads in each thread, $b = Y$ and $d = X$, can propagate to T_4 and T_1. They are thus ordered before the writes $X = 1$ and $Y = 1$ (6). The outcome $a = 1, b = 0, c = 1, d = 0$ is allowed.

Satisfy a read request. Read requests are satisfied by writes to the same address. In order to respond to a read request r with write w, they both have to have propagated to the exact same set of threads. By construction, this always includes r.thread and w.thread. We also want w to happen before r, i.e., $w \longrightarrow r$. Similarly, w has to be the last request to have written to that address. To model this, we consider all requests w' between r and w, i.e., such that $w \longrightarrow w'$ and $w' \longrightarrow r$. We require w' to be to a different address than r if it is a memory request (w' can also be a read, in which case this rule enforces that same-address read-to-read orderings are also preserved). After being responded to, we mark a read request as removed and ignore its order constraints. This ensures that this rule allows multiple reads to the same address, and more importantly, allows us to model non-MCA behavior.

As we can see from above, order rules play a central role in defining the model. Order rules depend on the concrete types of request and are consequently architecture-specific.

4.2 Non-Multi-Copy Atomicity

Non-MCA reads can be modeled by the propagation principle naturally. They arise from the combination of propagating to threads individually and the fact that we remove reads. To see this, consider the example in Figure 6, which represents the IRIW litmus test (cf. Figure 3a). The two writes propagate only to their adjacent threads, creating an order constraint with the corresponding reads, which can then propagate back to the threads of the writes. Having propagated to the same sets of threads, the two reads can be satisfied and are removed from the system, including their order constraints. The removal of the reads frees up the subsequent reads, and they can thus propagate to the entire system and read the uninitialized values (morally, from main memory), allowing the behavior $a = 1, b = 0, c = 1, d = 0$.

4.3 Fences

So far we have discussed read and write requests, but there is a third type of request that is central to the model: fences. In contrast with reads and writes (and contrary to POP), fences are thread-local in our model: they do not propagate; they enforce orderings only by stalling. The semantics of fences in our model are built by composing the following six kinds of orderings (where R refers to a read, W refers to a write, and pred refers to predecessor writes from other threads which is used for modeling cumulative fences):

$$\begin{array}{lll} R + \mathtt{pred} \longrightarrow R, & R \longrightarrow R, & R + \mathtt{pred} \longrightarrow W, \\ R \longrightarrow W, & W \longrightarrow R \text{ and} & W \longrightarrow W \end{array}$$

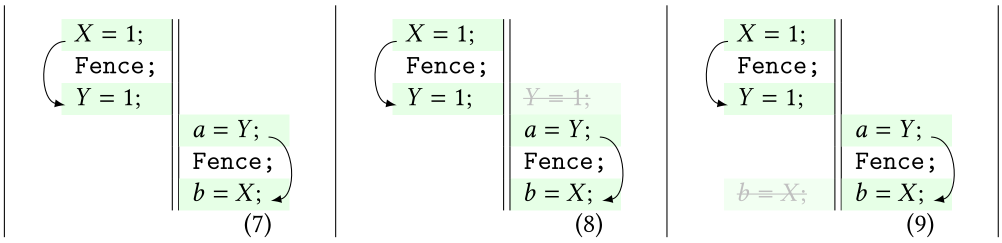

Fig. 7. Message Passing with fences in LOST-POP. We assume that these fences enforce $R \longrightarrow R$ and $W \longrightarrow W$, adding the corresponding edges (7). This means that the write $Y = 1$ cannot propagate before the write $X = 1$ (8), nor the read $b = X$ before the read $a = Y$ (9).

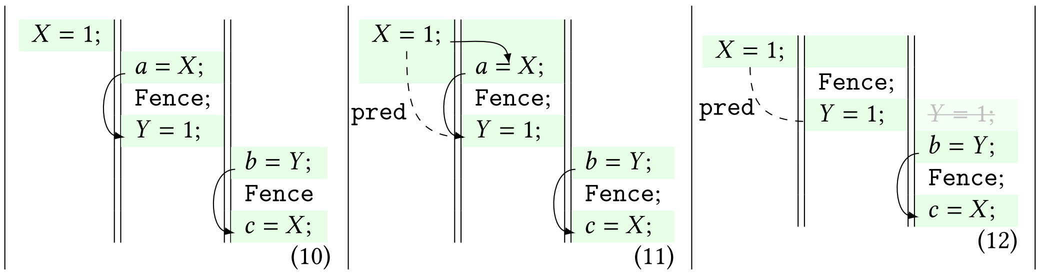

Fig. 8. WRC in LOST-POP. We assume these fences enforce $R + \texttt{pred} \longrightarrow W$ and $R \longrightarrow R$ (10). After the read write $X = 1$ propagates to the middle thread and then $a = X$ to the first thread, this read can be satisfied with the write (11). Once the read is gone, we need the ordering from the predecessor write $X = 1$ to block the write $Y = 1$ from propagating to the last thread (12).

Consider the example of the MP litmus test above with fences that enforce $W \longrightarrow W$ and $R \longrightarrow R$, as depicted in Figure 7. Here, the fences respectively block the propagation of both $Y = 1$ and $b = X$ until $X = 1$ and $a = Y$ have propagated before. This forbids the behavior where $a = 1, b = 0$, but if one of the two fences is removed and either $Y = 1$ or $b = X$ can propagate first, then the behavior is allowed.

The difference between R and R+`pred` models the cummulativity of the fence. Since satisfied reads are removed from the model, they cannot continue to transitively enforce any order constraints. To see this more concretely, consider the WRC example in LOST-POP, in Figure 8. If the fence between $a = X$ and $Y = 1$ only enforces $R \longrightarrow W$ it is not sufficient to disallow the behavior $a = 1, b = 1, c = 0$. This is because, as we can see in the figure, after satisfying the read $a = X$, this read is removed from the thread and the fence would stop blocking the propagation of $Y = 1$. If the fence enforces $R + \texttt{pred} \longrightarrow W$ instead, then it also considers the *predecessor write*, $X = 1$, and requires this to be propagated to the entire system before allowing $Y = 1$ to propagate. More generally, whenever we add an edge $w \longrightarrow r$ from a write request w to a read request r (when the write propagates to the read's thread) we also mark the write as a predecessor of the read's thread, $r.\texttt{thread}$. A fence that enforces $R + \texttt{pred} \longrightarrow R/W$ considers all such predecessors of its thread when blocking.

4.4 Scoped Requests

Recall that some memory models have scoped requests, which restrict the scope to which certain orderings are enforced. We model this by extending our LOST-POP model to be scoped as well. Threads in a system are grouped into subsets of scopes, partially ordered by set inclusion. The order constraints are also labeled by their scope. We write $r \xrightarrow{s} r'$ to denote the order constraint $r \longrightarrow r'$ is enforced at scope s. If the scope s is the set of all threads (i.e., the whole system), we can omit s and write $r \longrightarrow r'$, as before.

Scopes represent the locality-scope semantics of a request, like a single Compute Thread Array (CTA) or the entire system. The set of scopes forms a join semi-lattice (with join being the usual set operation), where the join of all threads is called the *system scope*. We also define an architecture-specific meet $\wedge$, but in general it need not form a lattice[7]. Importantly, all requests in LOST-POP have a scope. Consequently, in an unscoped LOST-POP model, all requests should be system-scoped.

Scopes refine the model in several ways. When **accepting** a request r at thread t: we update the order constraints in a scoped fashion. We add the labeled constraint $r' \xrightarrow{s} r$ with the scope s such that $t \in s = r' \wedge r$, and $\text{order}(r', r)$ holds. When **propagating** request r to thread t: the pre-conditions for propagating also become scoped. An edge $r \xrightarrow{s} r'$ only enforces the ordering of r before r' within the scope s. Note that the new edges between requests of different threads are *not* scoped, since these do not represent architecture-specific constraints but rather observations. In this context, not scoped means their scope is the entire system. Given $w \longrightarrow r$, we only add w as a **predecessor** to thread $t = r.\texttt{thread}$ if $w.\texttt{thread}, r.\texttt{thread} \in w \wedge r$.

Consider a scoped version of the MP example with fences (c.f. Figure 2b), and suppose the system has two separate scopes for the two threads, i.e., the join semilattice is given by Scopes $= \{\{T_1\}, \{T_2\}, \{T_1, T_2\}\}$. If either of the fences has $\{T_1\}$ or $\{T_2\}$ as its scope, then the $R \longrightarrow R$ or $W \longrightarrow W$ orders are enforced within that scope. This means that the write $Y = 1$ or the read $b = X$ can propagate to a different scope first, if the edge is only enforced within the scope $\{T_1\}$ or $\{T_2\}$ respectively. In either case, the outcome $a = 1, b = 0$ is allowed; to be disallowed, both fences need to be system-scoped.

4.5 Differences to POP

As we have seen, our LOST-POP model is a generalization of the POP [Flur et al. 2016] model from ARM to multiple architectures. It does have some additional changes, mostly necessary for the LOST principle. For the complete differences, refer to the technical appendices of Flur et al. [2016] and this paper. Here we discuss these differences at a high level.

The main difference between POP and LOST-POP is that fences do not propagate in LOST-POP. Their semantics are enforced in a thread-local fashion. For this, we also introduced predecessors (see Section 4.3 above). The POP model is also unscoped, whereas LOST-POP is scoped (cf. 4.4). POP uses reorder constraints, whereas LOST-POP uses order constraints, their negation.

Another difference is that LOST-POP does not add same-address read-to-read constraints between threads, which are less intuitive. This change, however, could introduce reorderings in third threads, which is why in LOST-POP, order constraints between requests are only added when propagating to one of the two event's originating thread. In POP this can happen at any thread.

4.6 Composing LOST-POP Models

Having defined LOST-POP models in general, we can proceed to define compound models. The key idea is that, by the LOST principle, the behavior of a thread does not depend on the architecture of other threads. Thus, a compound LOST-POP model is simply a LOST-POP model where different threads have different architectures. These will accordingly have their corresponding request types, order relations and scope intersections, which are the architecture-specific aspects of the model. Note that all these only affect the thread-local stalling, implementing the LOST principle. Thus,

[7]In particular, the meet $\wedge$ can be asymmetrical, which is necessary to model certain fence types that can be asymmetrical in its scoping.

different kinds of threads compose seamlessly, yielding a model that is compositional in the sense described in Section 3. (The full model, defined more formally, can be found in Appendix A.)

5 MODELING X86TSO AND PTX

In this section we show how to instantiate our general operational model to model x86TSO, PTX, and their composition.

x86TSO. x86TSO provides instructions to load from and store to memory locations. Additionally, x86TSO provides the MFENCE instruction to prevent the reordering of the memory accesses around it and flush the writes from the write buffers to the main memory. The x86 model is unscoped, which means all requests are system-scoped (i.e., their scope is the set of all threads in the system). Reads, writes and fences have no variants with different strengths. The semantics of x86 memory operations are $R + \texttt{pred} \longrightarrow R/W$ and $W \longrightarrow W$, corresponding to the preserved-program-order in x86TSO. Accordingly, the order condition is defined such that we always order $r \longrightarrow r'$ in a thread, unless r is a write and r' a read. Fences additionally enforce $W \longrightarrow R$.

PTX. The PTX model is defined as a virtual ISA for GPUs [Lustig et al. 2019; NVIDIA 2019]. The model also provides read, write, and fence accesses. The accesses also have two additional parameters: semantics and scope.

The memory strength semantics in PTX are: weak (WK), relaxed (RLX), acquire (ACQ), release (REL), acquire-release (ACQ-REL), and sequentially-consistent (SC). These memory semantics are partially ordered by $\textsc{wk} < \textsc{rlx} < \{\textsc{acq}, \textsc{rel}\} < \textsc{acq-rel} < \textsc{sc}$. Release and acquire are incomparable, but both are stronger than relaxed and weaker than acquire-release (RA), the rest are totally-ordered.

The accesses can be marked with different scopes: weak accesses are scoped only to their own their threads, cta for cooperative thread groups, gpu for threads on the same compute device including other thread groups, and sys for all threads in all devices.

The architecture-specific meet and order relation and semantics of PTX are given in detail in Appendix A. They ensure that a request that is marked with $\texttt{sem} \geq \texttt{rel}$ enforces $R + \texttt{pred} \longrightarrow W$ and $W \longrightarrow W$. A request with $\texttt{sem} \geq \texttt{acq}$ enforces $R \longrightarrow R$ and $R \longrightarrow W$, and an sc fence additionally enforces $W \longrightarrow R$.

The Compound x86 and PTX Model. As explained in Section 4, the compound model arises from simply allowing threads to be of different types. There is no need to define synchronization between the two, nor transformations from one architecture to the other. To see how this works, we shall look at some of the trickier cases.

Consider the example in Figure 9. This is coincidentally what would be the result of compiling this test entirely with SC reads and writes from C11 (cf. 3a). This mapping and its correctness in general will be discussed later in Section 6. In this example, the second thread which is the only x86 thread, enforces SC semantics in this litmus test without a fence. The figure shows how the behavior is disallowed on a purely thread-local fashion: the x86 thread enforces $R + \texttt{pred} \longrightarrow R$, which is the necessary condition for this test behave like the C11 SC version. In particular, T_2 does not rely on the architecture-specific semantics of requests in other threads.

Scopes can cause complex interactions between requests as well. Consider the example in Figure 10. As can be seen in the sequence described in the figure, crucial for this litmus test is whether the write $X = 1$ is made a predecessor of T_2. However, the PTX thread T_2 makes this decision on an local, architecture-independent basis. It only consider the basic type (write) and scope of the incoming write $X = 1$, and information about its propagation. This is all independent of the architecture of T_1.

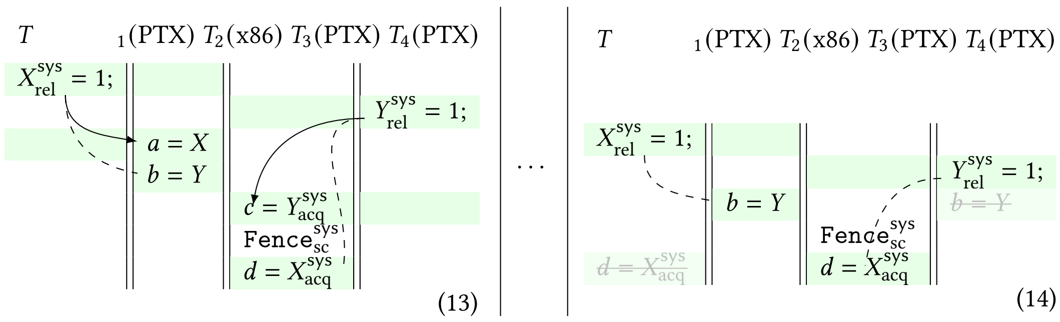
T_1(PTX) T_2(x86) T_3(PTX) T_4(PTX)
$X_{\mathrm{rel}}^{\mathrm{sys}} = 1;$
$a = X$
$b = Y$
$Y_{\mathrm{rel}}^{\mathrm{sys}} = 1;$
$c = Y_{\mathrm{acq}}^{\mathrm{sys}}$
$\mathtt{Fence}_{\mathrm{sc}}^{\mathrm{sys}}$
$d = X_{\mathrm{acq}}^{\mathrm{sys}}$
(13)
...
T_1(PTX) T_2(x86) T_3(PTX) T_4(PTX)
$X_{\mathrm{rel}}^{\mathrm{sys}} = 1;$
$b = Y$
$Y_{\mathrm{rel}}^{\mathrm{sys}} = 1;$
~~$b = Y$~~
$\mathtt{Fence}_{\mathrm{sc}}^{\mathrm{sys}}$
~~$d = X_{\mathrm{acq}}^{\mathrm{sys}}$~~
$d = X_{\mathrm{acq}}^{\mathrm{sys}}$
(14)

Fig. 9. A compound IRIW test. After the write $X =_1$ propagates to T_2 it adds a predecessor edge, just as the write $Y = 1$ does when propagating to T_4. After that, the reads can proceed to propagate accordingly (13). After the reads are satisfied and removed, the read $b = Y$ cannot propagate to T_4 and be ordered before the write $Y = 1$ because of the predecessor, since x86TSO enforces $R + \mathtt{pred} \rightarrow R$. This order is also enforced by the SC fence in PTX, which prevents the read $d = X$ from propagating to thread T_1 (14). The outcome $a = 1, b = 0, c = 1, d = 0$ is disallowed.

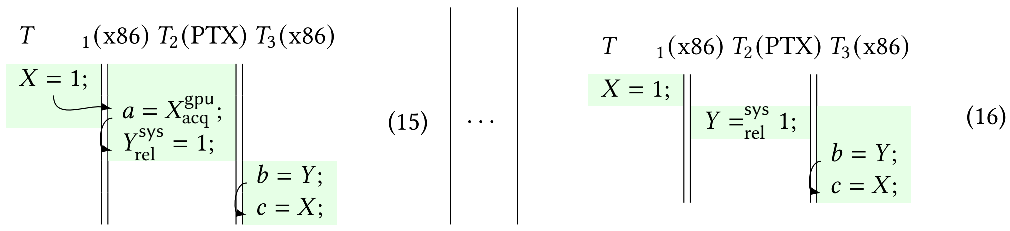
T_1(x86) T_2(PTX) T_3(x86)
$X = 1;$
$a = X_{\mathrm{acq}}^{\mathrm{gpu}};$
$Y_{\mathrm{rel}}^{\mathrm{sys}} = 1;$
$b = Y;$
$c = X;$
(15)
...
T_1(x86) T_2(PTX) T_3(x86)
$X = 1;$
$Y =_{\mathrm{rel}}^{\mathrm{sys}} 1;$
$b = Y;$
$c = X;$
(16)

Fig. 10. A compound WRC test. Since the read $a = X$ is scoped only to the gpu, it is not scope inclusive with the write $X = 1$, even if this write is scoped to the full system, as a TSO operation. Thus, the write does not get added as a predecessor to T_2 (15). This means that after the read is satisfied and removed, the write of $Y = 1$ can propagate to T_3 before the write $X = 1$ does (16). The outcome $a = 1, b = 1, c = 0$ is thus allowed.

5.1 Testing our Models

To validate our operational model, we have implemented the model within the Lean4 theorem prover and language [de Moura and Ullrich 2021] along with the corresponding instances of the x86TSO and PTX models [8].

By (guided) exhaustive exploration and manual interactive inspection, we ran the x86TSO model on 34 litmus tests, each of which agree with the axiomatic model outcomes [Owens et al. 2009]. We verified this by modeling the axiomatic TSO model in the Alloy tool [Jackson 2012] and programatically exporting our litmus tests to verify with Alloy. The litmus tests included variants of MP, SB, LB, N7, WRC, ISA2, IRIW, 2+2W and some other unnamed tests that served as basic sanity checks. The variants also featured fences in different places.

Similarly, for the PTX model we used the published Alloy model[9] from [Lustig et al. 2019] and exported the litmus tests to test them there. We explored 73 variants with different fence semantics and scopes of the same litmus tests mentioned above. We find that in all cases, our operational model is at least as strong as the reference axiomatic PTX model. There are two ways in which our operational model is stronger than the axiomatic PTX model:

[8] https://github.com/goens/lost-pop-lean
[9] https://github.com/nvlabs/ptxmemorymodel

$$\begin{array}{l||l} X^{\mathsf{sys}}_{\mathrm{REL}} = 1; & Y^{\mathsf{sys}}_{\mathrm{REL}} = 1; \\ Y^{\mathsf{sys}}_{\mathrm{REL}} = 2; & X^{\mathsf{sys}}_{\mathrm{REL}} = 2; \\ {r_1}^{\mathsf{sys}}_{\mathrm{ACQ}} = Y & r_2 = X^{\mathsf{sys}}_{\mathrm{ACQ}} \end{array}$$

(a) 2+2W.

$$\begin{array}{l||l||l} X^{\mathsf{sys}} = 1; & r_1 = Y^{\mathsf{sys}}; & r_2 = Z^{\mathsf{sys}}; \\ F^{\mathsf{sys}}_{\mathrm{REL}}; & F^{\mathsf{sys}}_{\mathrm{ACQ}}; & F^{\mathsf{sys}}_{\mathrm{ACQ}} \\ Y^{\mathsf{sys}} = 1 & Z^{\mathsf{sys}} = 1 & r_3 = X^{\mathsf{sys}}; \end{array}$$

(b) ISA2 with Acq Fence

$$\begin{array}{l||l||l} X^{\mathsf{sys}} = 1; & r_1 = Y^{\mathsf{sys}}; & r_2 = Z^{\mathsf{sys}}; \\ F^{\mathsf{sys}}_{\mathrm{REL}}; & F^{\mathsf{sys}}_{\mathrm{REL}}; & F^{\mathsf{sys}}_{\mathrm{ACQ}}; \\ Y^{\mathsf{sys}} = 1 & Z^{\mathsf{sys}} = 1 & r_3 = X^{\mathsf{sys}}; \end{array}$$

(c) ISA2 with Rel Fence

Fig. 11. Our operational model is stronger than the axiomatic PTX. In 2+2W, PTX allows $r_1 = 1, r_2 = 1$ but this behavior is not allowed our operational model. Similarly, in ISA2 with an Acq (11b) or Rel (11c) fence, the outcome $r_1 = 1, r_2 = 1, r_3 = 0$ is allowed in PTX but disallowed in our model. PTX requires the fence to be acqrel for the outcome to be disallowed, whereas our model disallows it with either of the three.

(1) Write serialization is enforced within the causality constraints by our operational model, but not by the axiomatic PTX model.
(2) Release and acquire fences are slightly stronger in cumulative chains of events, downstream of a release fence (so-called B-cumulativity [Alglave et al. 2014]). In our operational model, events can be transitively ordered using either release or acquire fences, though the axiomatic PTX model would only order if the fence is at least acqrel.

Figure 11 shows two litmus tests that witness this difference, 2+2W and ISA2 with a single release or acquire fence in the middle thread.

Finally, for the compound axiomatic model defined in Section 6, we included compound variants of the the same tests, with different thread types, and additionally all the tests for the individual models, for a total of 159 tests for the compound x86TSO/PTX model.

6 COMPILER MAPPINGS

In this section, we discuss the compiler mappings from the scoped C/C++ model to the x86TSO/PTX compound memory model. However, in contrast to the POP-style operational model, the scoped C/C++ model is defined axiomatically. Due to this gap, we propose CMM, an axiomatic model that combines the axiomatic models of x86 [Alglave et al. 2021] and PTX [Lustig et al. 2019]. CMM is designed to be a sound abstraction of the x86TSO/PTX operational model, *i.e.* permit all behavior observable in the operational model. We then prove the correctness of a mapping from scoped-C/C++ to CMM.

6.1 Axiomatic Semantics for Concurrency

In axiomatic semantics, a memory model is characterized by a set of predicates describing the allowed execution behaviors of a program. A program, thus, is represented by a set of finite executions, where an execution constitutes a set of events and different relations between the events. An event represents the execution of a shared memory or a fence access and the events are related by various binary relations.

Events. An event is of the form ⟨id, tid, lab⟩ where id, tid, and lab denote unique identifier, thread identifier, and a label based on the respective memory or fence access. A label ⟨op, loc, Val⟩ consists of operation type, location, and read or written value. We use R, W, and F to denote the set of read, write, and fence events. Given an event e, we write e.id, e.tid, e.lab, e.op, e.loc, and e.Val to denote (when applicable) its identifier, thread, label, operation, location, read or written value respectively. We also define a skip event where lab =⊥, that is a placeholder for a no operation (nop).

Notations. We use a notation similar to that of ‘cat’ language [Alglave and Maranget 2022]. Given a binary relation S on events, $\mathrm{dom}(S)$ and $\mathrm{codom}(S)$ are domain and range of S. The inverse relation, as well as the reflexive, transitive, and reflexive-transitive closures of S are denoted by S^{-1}, $S^?$, S^+,

$$\mathrm{acy}(\mathrm{po}_{=\mathrm{loc}} \cup \mathrm{rf} \cup \mathrm{fr} \cup \mathrm{co}) \qquad (\text{sc-per-loc})$$
$$\mathrm{rmw} \cap (\mathrm{fre}; \mathrm{coe}) = \emptyset \qquad (\text{atomicity})$$
$$\mathrm{irr}(\mathrm{xhb}) \qquad (\text{GHB})$$

where

- $\mathrm{ppo} \triangleq (\mathrm{WW} \cup \mathrm{RW} \cup \mathrm{RR}) \cap \mathrm{po}$
  where
  $\mathrm{WW} = \mathrm{W} \times \mathrm{W}$
  $\mathrm{RW} = \mathrm{R} \times \mathrm{W}$
  $\mathrm{RR} = \mathrm{R} \times \mathrm{R}$
- $\mathrm{implied} \triangleq (\mathrm{po}; [A]) \cup ([A]; \mathrm{po})$
  where
  $A = \mathrm{dom}(\mathrm{rmw}) \cup \mathrm{codom}(\mathrm{rmw}) \cup \mathrm{F_x}$
- $\mathrm{xhb} \triangleq (\mathrm{ppo} \cup \mathrm{implied} \cup \mathrm{rfe} \cup \mathrm{fr} \cup \mathrm{co})^+$

$$[\mathrm{W}]; \mathrm{cause}_{=\mathrm{loc}}; [\mathrm{W}] \subseteq \mathrm{co} \qquad (\text{Coherence})$$
$$\mathrm{irr}(\mathrm{sc}; \mathrm{cause}) \qquad (\text{FenceSC})$$
$$\mathrm{irr}((\mathrm{rf} \cup \mathrm{fr}); \mathrm{cause}) \qquad (\text{Causality})$$
$$((\mathrm{ms} \cap \mathrm{fr}); (\mathrm{ms} \cap \mathrm{co})) \cap \mathrm{rmw} = \emptyset \qquad (\text{Atomicity})$$
$$\mathrm{acy}(\mathrm{po}_{=\mathrm{loc}} \cup (\mathrm{ms} \cap (\mathrm{rf} \cup \mathrm{co} \cup \mathrm{fr}))) \qquad (\text{SC-per-loc})$$
$$\mathrm{acy}(\mathrm{rf} \cup \mathrm{dep}) \qquad (\text{No-Thin-Air})$$

where

$$\mathrm{prel} \triangleq ([\mathrm{W}_{\sqsupseteq \mathrm{REL}}]; \mathrm{po}_{=\mathrm{loc}}{}^?; [\mathrm{W}]) \cup ([\mathrm{F}_{\mathrm{REL}}]; \mathrm{po}; [\mathrm{W}])$$
$$\mathrm{pacq} \triangleq ([\mathrm{R}]; \mathrm{po}_{=\mathrm{loc}}{}^?; [\mathrm{R}_{\sqsupseteq \mathrm{ACQ}}]) \cup ([\mathrm{R}]; \mathrm{po}; [\mathrm{F}_{\mathrm{ACQ}}])$$
$$\mathrm{obs} \triangleq (\mathrm{ms} \cap \mathrm{rf}) \cup (\mathrm{obs}; \mathrm{rmw}; \mathrm{obs})$$
$$\mathrm{sw} \triangleq (\mathrm{ms} \cap (\mathrm{prel}; \mathrm{obs}; \mathrm{pacq})) \cup \mathrm{sc}$$
$$\mathrm{causeb} \triangleq (\mathrm{po}^?; \mathrm{sw}; \mathrm{po}^?)^+$$
$$\mathrm{cause} \triangleq \mathrm{causeb} \cup (\mathrm{obs}; (\mathrm{causeb} \cup \mathrm{po}_{=\mathrm{loc}}))$$

Fig. 12. x86TSO and PTX axiomatic models.

and S^* respectively. Binary relations S_1 and S_2 are composed by $S_1; S_2$. $[A]$ is the identity relation on the set A. The relation S is acyclic if S^+ is irreflexive. We denote irreflexivity and acyclicity of relation S by $\mathrm{irr}(S)$ and $\mathrm{acy}(S)$ respectively. Given a relation S, $S_{=\mathrm{loc}}$ denotes the relation S between same-location events, that is, $S_{=\mathrm{loc}} \triangleq \{(a, b) \mid a.\mathrm{loc} = b.\mathrm{loc}\}$. Also $S_{\neq\mathrm{loc}}$ denotes relation S between events on different locations, that is, $S_{\neq\mathrm{loc}} \triangleq S \setminus S_{=\mathrm{loc}}$ holds.

Relations. Events are connected by various relations: The program-order (po) relation is a strict partial order that captures the syntactic order among the events. Similarly, reads-from (rf) relates a pair of write and read events on the same location and having the same values. The coherence-order (co) relation is a strict total order on same-location write events.

We compose these relations to derive new ones. The from-read relation ($\mathrm{fr} \triangleq \mathrm{rf}^{-1}; \mathrm{co}$) relates read and write events r and w on the same location. In this case, w is co-after the write u where $\mathrm{rf}(u, r)$ holds. A relation is external when it is not between po-related events, e.g. external rf, co, fr relations are: $\mathrm{rfe} \triangleq (\mathrm{rf} \setminus \mathrm{po})$, $\mathrm{coe} \triangleq (\mathrm{co} \setminus \mathrm{po})$, and $\mathrm{fre} \triangleq (\mathrm{fr} \setminus \mathrm{po})$ respectively. For a read-modify-write instruction, we model the read and write as two individual events. The rmw relation connects the corresponding pair of read and write events accessing the same memory location. These two events are $\mathrm{po}_{=\mathrm{loc}}$-related as well as immediate-po-related, i.e. there is no intermediate event. Finally, we write $\mathrm{U} \triangleq \mathrm{dom}(\mathrm{rmw}) \cup \mathrm{codom}(\mathrm{rmw})$ to denote the rmw-related events.

Consistency axioms. The consistency axioms for a model are defined based on these relations and events. If an execution satisfies all the axioms of a model, then the execution is consistent in that model. Given a program $\mathbb{P}$, the behaviors of its consistent executions in a memory model constitute the program behavior in that model.

6.2 Compound Model: Axiomatic Specification

We combine the x86TSO and PTX axiomatic models to define compound memory model CMM. Therefore, before discussing the CMM model we review x86TSO and PTX [Alglave et al. 2021; Lustig et al. 2019].

x86TSO vs PTX: a review. The x86TSO and PTX models are in Figure 12. These models have some similarities, for instance, both the models preserve atomicity and sc-per-location guarantees. However, in addition to the PTX scopes, the models also differ in various aspects. In x86TSO, the accesses in a thread are ordered by the ppo and implied relations. The events in PTX are ordered based on access types, memory orders, weaker fences, as well as dependencies (dep).

The global orderings in x86TSO and PTX, xhb and cause respectively, differ significantly. More specifically, co and fr are part of xhb in x86TSO unlike in cause in PTX. In PTX cause is defined based on synchronization and sc ordering. The role of the global orderings also varies in the axioms in x86TSO and PTX. In x86TSO the global ordering constraint (GHB) uses only xhb whereas in PTX cause is used in the constraints along with different relations.

CMM**: a combination of x86TSO and PTX.** We combine the models in Figure 12 to define the CMM model such that the model is (1) strong enough to ensure that the scoped mapping from x86 and PTX holds, and (2) weak enough to simulate the operational PTX + x86TSO model.

In CMM an execution is of the form $\langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{gsc}\rangle$ where we refine or extend the events and relations as follows. The execution has x86TSO, PTX, and initialization (init) events $\mathsf{E_x}$, $\mathsf{E_g}$, $\mathsf{E_{init}}$ respectively. Note that $\mathsf{E_x}$ and $\mathsf{E_g}$ are never po-related. We write $\mathsf{W_x}$, $\mathsf{R_x}$, $\mathsf{F_x}$ to denote the x86TSO read, write, fence events. Similarly $\mathsf{W_g}$, $\mathsf{R_g}$, $\mathsf{F_g}$ denote PTX read, write, and fence events.

CMM defines additional relations between x86TSO and PTX events to combine the two models. Given a relation S, we write $S^{\mathsf{x}}_{\mathsf{g}}$ and $S^{\mathsf{g}}_{\mathsf{x}}$ to denote relation S from x86TSO-to-PTX and PTX-to-x86TSO respectively. For instance, relation $\mathsf{rfe}^{\mathsf{g}}_{\mathsf{x}} \triangleq [\mathsf{W_g}]; \mathsf{rfe}; [\mathsf{R_x}]$ denotes the PTX-to-x86TSO external-read-from relation. We extend the relations in the individual models with the combinations of the relations and events from other models.

We extend the PTX $\mathsf{sc} \subseteq \mathsf{F_{sc}} \times \mathsf{F_{sc}}$ order to global-SC order $\mathsf{gsc} \subseteq ([\mathsf{F_{sc}} \cup \mathsf{F_x} \cup \mathsf{R_x}] \times [\mathsf{F_{sc}} \cup \mathsf{F_x} \cup \mathsf{R_x}])$ that orders sc fences in PTX as well as fence and read events in x86TSO.

PTX-to-x86TSO. In CMM the $\mathsf{rfe}^{\mathsf{g}}_{\mathsf{x}}$ relation is as strong as an x86TSO rfe, and therefore extends the xhb ordering: $\mathsf{gxhb} \triangleq (\mathsf{rfe}^{\mathsf{g}}_{\mathsf{x}})^?; \mathsf{xhb}$.

x86TSO-to-PTX. Now we consider the relations from x86TSO to PTX. In CMM an x86TSO write and read are at least as strong as a sys-scoped release-write and acquire-read in PTX. Therefore, an x86TSO write and read events may serve as the source and the sink of a synchronization relation with PTX events. So we extend prel, pacq, and sw relations to define gprel, gpacq, and xgsw relations. Based on the xgsw relation we define xgcauseb and xcause relations, analogous to PTX causeb and cause relations respectively.

$$\mathsf{xgsw} \triangleq (\mathsf{ms} \cap (\mathsf{gprel}; \mathsf{obs}; \mathsf{gpacq})) \setminus (\mathsf{E_x} \times \mathsf{E_x}) \text{ where}$$
$$\mathsf{gprel} \triangleq \mathsf{prel} \cup \mathsf{po}^?; [\mathsf{W_x}] \text{ and } \mathsf{gpacq} \triangleq \mathsf{pacq} \cup [\mathsf{R_x}]; \mathsf{po}^?$$
$$\mathsf{xgcause} \triangleq \mathsf{xgcauseb} \cup (\mathsf{obs}; (\mathsf{xgcauseb} \cup \mathsf{po}_{=\mathsf{loc}})) \text{ where } \mathsf{xgcauseb} \triangleq (\mathsf{po}^?; (\mathsf{sc} \cup \mathsf{xgsw}); \mathsf{po}^?)^+$$

Combined Ordering Finally, we combine the ordering in each model along with the interactions between them to define combined-order (cord) relation between a morally strong (ms) event pair. Morally strong means that both events have a scope that includes the other access, and neither of them is weak. A more formal definition can be found in [Lustig et al. 2019]. We also define weak-combined-ordering (wcord) relation where the event pair may not be in morally strong relation.

$$\mathsf{wcord} \triangleq (\mathsf{gxhb} \cup \mathsf{xgcause} \cup \mathsf{gpo}^?; \mathsf{gsc}; \mathsf{gpo}^?)^+ \text{ and } \mathsf{cord} \triangleq \mathsf{ms} \cap \mathsf{wcord} \text{ where } \mathsf{gpo} \triangleq [\mathsf{E_g}]; \mathsf{po}; [\mathsf{E_g}]$$

Note that, unlike the sc relation in the PTX model, the gsc relation is not a component of xgsw. The gsc relation (with optional po-predecessor and successor for PTX events) is used in defining the wcord relation. We also define relation extended-coherence-order (eco) as a combination of the rf, fr and co relations [Lahav et al. 2017] between morally strong events: $\mathsf{eco} \triangleq \mathsf{ms} \cap ((\mathsf{fr} \cup \mathsf{co} \cup \mathsf{fr})^+)$.

C11	x86	PTX	
$R^{sco}_{o_r}$	R_x	$R^{sco}_{o_r}$	where $o_r \in \{\textsc{na/wk}, \textsc{rlx}, \textsc{acq}\}$
$W^{sco}_{o_w}$	W_x	$W^{sco}_{o_w}$	where $o_w \in \{\textsc{na/wk}, \textsc{rlx}, \textsc{rel}\}$
$RMW^{sco}_{o_u}$	RMW	$RMW^{sco}_{o_u}$	where $o_u \in \{\textsc{rlx}, \textsc{rel}, \textsc{acq}, \textsc{acq-rel}\}$
$F^{sco}_{o_f}$	nop	$F^{sco}_{o_f}$	where $o_f \in \{\textsc{rel}, \textsc{acq}, \textsc{acq-rel}\}$
F^{sco}_{SC}	MFENCE	F^{sco}_{SC}	
R^{sco}_{SC}	R_x	$R^{sco}_{ACQ}; F^{sco}_{SC}$	
W^{sco}_{SC}	W_x; MFENCE	$W^{sco}_{ACQ}; F^{sco}_{SC}$	
RMW^{sco}_{SC}	RMW	$RMW^{sco}_{ACQ\text{-}REL}; F^{sco}_{SC}$	

Fig. 13. Mapping schemes: scoped-C/C++ to x86 and PTX.

Axioms. We define the consistency constraints for CMM as follows.

- $([W]; \text{cord}; [W])_{=\text{loc}} \subseteq \text{co}$ (CMM-Coherence)
- $(\text{gsc}; \text{cord})$ is irreflexive. (CMM-FenceSC)
- $\text{cord}; \text{eco}^?$ is irreflexive. (CMM-irrCordECO)
- $((\text{rf} \cup \text{fr}); \text{wcord})$ is irreflexive. (CMM-Causality)
- $(((\text{ms} \cap \text{fr}); (\text{ms} \cap \text{co})) \cap \text{rmw}) = \emptyset$ (CMM-Atomicity)
- $(\text{po}_{=\text{loc}} \cup (\text{ms} \cap (\text{rf} \cup \text{co} \cup \text{fr})))$ is acyclic. (CMM-sc-per-loc)
- $(\text{rf} \cup \text{dep} \cup \text{ppo})$ is acyclic. (CMM-No-Thin-Air)

The axioms are similar to the PTX axioms. The (CMM-Coherence), (CMM-FenceSC) axioms are similar to the (Coherence), (FenceSC) axioms in PTX where the sc and cause relations are replaced by the gsc and cord relations. Axiom (CMM-irrCordECO) is analogous to the (Coherence) axiom in scoped-C/C++. Axiom (CMM-causality) uses wcord instead of cause in the PTX (Causality) axiom. The axiom enforces that an rf or fr related event pair follows weak-global-ordering wcord even when they are not morally strong. The (CMM-Atomicity) and (CMM-sc-per-loc) axioms are same as PTX, but applicable on compound executions. In the (CMM-No-Thin-Air) axiom we augment the PTX (No-Thin-Air) axiom with ppo ordering in x86TSO.

LOST-POP vs CMM. We also encoded the CMM axiomatic model in Alloy and tested all 139 litmus tests, comparing our operational and axiomatic models. The axiomatic model exactly matches the behavior of the axiomatic models for both PTX and x86TSO in all litmus tests, and is weaker than the operational compound model in the exact same way that the axiomatic PTX model is weaker than our operational instance.

6.3 Mapping scoped-C/C++ to CMM

Mapping correctness from programming languages to architectures is a widely studied problem [Batty et al. 2011; Chakraborty and Vafeiadis 2017, 2019; Kang et al. 2017; Lahav et al. 2017; Lustig et al. 2019; Podkopaev et al. 2019; Sarkar et al. 2012]. Following Lustig et al. [2019], we consider a scoped extension to C/C++ which provides non-atomic and atomic accesses which are similar to PTX weak and strong accesses respectively. Scoped-C/C++ provides atomic read, write, RMW, fence accesses similar to the PTX strong accesses with same memory orders and scopes. Scoped-C/C++ defines scope-inclusive (incl) relation which is, as observed by Lustig et al. [2019], essentially the same as the morally-strong relation. Similar to PTX, scoped-C/C++ defines *synchronize-with* relation between release and acquire accesses, and then a defines *happens-before* relation. Scoped C/C++ does not defines any sc relation, but orders sc accesses in an execution with the psc relation adapted from Lahav et al. [2017]. Finally, if a program has a racy execution on non-atomic or non-morally strong accesses then the program has *undefined behavior*. The formal definition of the scoped-C/C++ model is in Appendix B.

Mapping Correctness: Scoped-C/C++ to CMM. In Figure 13 we propose the mapping scheme for scoped-C/C++ to the respective architectures. The mapping scheme to x86 [Sewell 2022] introduces a trailing MFENCE for an SC write access. For the PTX mapping, we note that Lustig et al. [2019] followed the leading-SC-fence mapping scheme from Lahav et al. [2017]. To avoid this incompatibility of placing SC fences, we adapt the trailing-SC-fence mapping scheme from Lahav et al. [2017], which is believed to be an alternative, correct mapping for PTX.

Theorem 6.1. The scoped-C/C++ to CMM mapping scheme in Figure 13 is correct.

We prove Theorem 6.1 in Appendix C. Given a scoped-C/C++ program, we strengthen the non-release-acquire accesses to be mapped to x86 to release-acquire accesses to generate a program $\mathbb{P}_s$. We consider only race-free X_s executions. We then perform the splitting transformations (the SC writes to be mapped to x86 and the SC accesses in PTX are replaced by the respective memory accesses followed by SC fences) followed by mapping from Figure 13 to generate X_t. We show that for each consistent execution X_t of $\mathbb{P}_t$ in compound memory model there exists a scoped C/C++ consistent execution X_s of $\mathbb{P}_s$ such that X_s has the same behavior. The proof transforms X_t to X_s, such that all the consistency conditions of scoped C/C++ hold, or demonstrates a race in $\mathbb{P}_s$, contradicting the race-free assumption.

Theorem 6.1 refers specifically to the mapping from Figure 13, and in particular, to the PTX+x86 combination. In future work, we want to use the LOST-POP operational model to generalize this result to other architectures and combinations.

7 VALIDATING COMPOUND MEMORY MODELS

The NVIDIA Grace Hopper machine is not publicly available yet, but AMD have been offering integrated CPU-GPU chips (or APUs). Furthermore, AMD have made their design publicly available in the cycle-accurate Gem5 simulator [Lowe-Power et al. 2020], and it is clear that the heterogeneous APU is in fact an x86 CPU fused with a GCN3 GPU, using an inter-device coherence protocol that enforces the Single-Writer-Multiple-Reader (SWMR) invariant [Nagarajan et al. 2020].

In this section, we seek empirical answers to the following questions. What memory model does the CPU enforce? What memory model does the GPU enforce? What memory model does the compound CPU-GPU machine satisfy? Crucially, does it enforce the compound memory model obtained by composing the individual CPU and GPU memory model as proposed in this paper? If the compound memory model is indeed satisfied, does it depend on the inter-device coherence protocol at all? Being a simulation, it conveniently allows us to tweak the coherence protocol and observe its effects. In the rest of this section, we answer these questions by running various litmus tests on the simulator.

There are two important caveats with this approach. Despite having industry involvement, a cycle-accurate simulator model may not necessarily show all of the behaviors experienced by a real machine. Secondly, because a cycle-accurate simulation is orders of magnitude slower that the real run, it is hard to expose as many behaviors as in the real run. Nevertheless, hardware not being available, a simulator is the next best way to test our models. Before discussing the results, we briefly discuss the simulator set up and configuration.

7.1 The AMD CPU-GPU Architecture and Litmus Tests

The CPU device consists of 4 cores, each with its private L1 and L2 cache. The GPU device consists of 16 streaming multiprocessors (SMs) with a private L1 per SM and an L2 shared by the SMs. The CPU model supports the x86 instruction set and therefore supports one fence instruction (`mfence`). The GPU memory model is a scoped memory model and it supports release, acquire,

Test	x86-Gem5	x86TSO	XC-Gem5	XC-M.	Test	XC-Gem5	XC-M.
MP-sys	✗	✗	✓	✓	MP-sys-F	✗	✗
SB-sys	✓	✓	✓	✓	MP-cta-F	✓	✓
SB-sys-F	✗	✗	✗	✗			
IRIW-sys	✗	✗	✓	✓	IRIW-sys-F	✗	✗
LB-sys	✗	✗	✗	✗			

Table 1. CPU and GPU Observations for different litmus tests compared with the observations for a scoped relaxed model with an SC fence. The Scoped-XC model is denoted by XC-M. MP-sys means that it is a message passing litmus test without any fences. MP-sys-F (MP-cta-F) means it is a message passing litmus test where there is a system-scoped (cta-scoped) fence between the two writes in the producer and the two reads in the consumer; the producer and consumer are mapped to different SMSs. We do not show the result of these tests for x86 because the result is disallowed even without a fence, as shown in MP-sys. Note that we also do not show several variants of these tests where – like MP-cta-F – the fences are not morally strong, in which case we observe an "allowed" outcome similarly to MP-cta-F.

and SC fences.[10] We ran variants of store buffering (SB), load buffering (LB), message passing (MP) and the independent-read-independent-write (IRIW) litmus tests. Each of the litmus tests were run between 512–2048 times within the simulator; we perturb the simulation using knobs such as delays and cache evictions to explore different executions.

7.2 Observations

Table 1 shows the observed outcomes for the CPU and GPU litmus tests. As we can see, MP, LB, IRIW, SB-F are disallowed in the x86 simulator whereas, SB is allowed. These observations suggest that the CPU does enforce the x86TSO memory model. On the GPU side without any fences, each of the litmus tests (except LB) is allowed. We can see that with any system-scoped fence (release, acquire or full fence), each of the litmus tests is disallowed. In other words, each of the fences behaves like an SC fence. Finally, when the scopes of the operations are not morally strong, then the litmus tests are allowed – for example, when the MP litmus test is run with a CTA-scoped fence on two different CTAs. We do not show the other non-morally-strong litmus tests. These observations suggest a scoped relaxed memory model with a strong SC fence. Furthermore, the memory model additionally preserves the R $\longrightarrow$ W ordering. We refer to this memory model as Scoped XC.

We ran compound litmus tests on the CPU-GPU model. Note that each of the litmus tests gets different variants depending on where the threads are mapped; MP-1, for example, refers to the litmus test where the producer is mapped to the GPU and the consumer is mapped to the CPU. (MP-2 is the converse.) In IRIW-1, the storing threads are mapped to the GPU and loading threads are mapped to the CPU. (IRIW-2 is the converse.) As we can see from Table 2, each of our observations corresponds to the compound operational model arising where the CPU enforcing x86TSO is fused with a GPU enforcing Scoped XC. This suggests that compound memory models are something that can be naturally realized when multiple devices are stitched together using a global inter-device coherence protocol.

To understand the impact of the inter-device coherence protocol on the observed compound memory model, we modified the inter-device protocol as follows: In the baseline protocol, when a GPU SM performs a writeback, the inter-device protocol invalidates any sharers in the CPU side, and only then sends an acknowledgement to the SM performing the writeback. (A subsequent

[10] A note on terminology: the gem5 GPU model actually supports the marking of stores as a release and loads as an acquire, but because releases and acquires are implemented like fences in the simulator, we call these as a release fence and an acquire fence in the rest of the discussion.

Test	SWMR	no-SWMR	Model
MP1-sys	✓	✓	✓
MP1-sys-F	✗	✓	✗
MP1-cta-F	✓	✓	✓
SB-sys	✓	✓	✓
IRIW1-sys	✗	✗	✗
IRIW2-sys-F	✗	✗	✗
MP2-sys	✓	✓	✓
MP2-sys-F	✗	✗	✗
SB-sys-F	✗	✗	✗
IRIW2-sys	✓	✓	✓
LB-sys	✗	✗	✗

Table 2. Observation for different CPU-GPU litmus tests on the gem5 simulator, compared with the results of the compound memory model, fusing x86TSO with Scoped XC. In MP1-sys-F, the producer is mapped to the GPU with a system-scoped fence between the two writes, and the consumer is mapped to the CPU with no fences between the two loads. In MP2-sys-F, the producer is mapped to the CPU with no fences between the two writes, and the consumer is mapped to the GPU with a system-scoped fence between the two reads. Note that relaxing the SWMR property weakens the memory consistency model for the heterogeneous litmus test: MP1-sys-F.

release on that SM waits for all previous writebacks to be acknowledged.) We modify the protocol such that when an SM performs a writeback, an early acknowledgement is sent to that SM before any sharers in the CPU are invalidated. Note that this breaks SWMR (the coherence invariant) of the inter-device protocol. Once the coherence invariant is weakened, one of the tests which was disallowed is then allowed (MP with the producer in the GPU side, shown in red in Table 2). Without inter-device coherence the ordering enforced in the GPU side is not transmitted correctly to the CPU side. This appears to demonstrate that the strength of compound memory model hinges on the guarantees provided by the inter-device coherence protocol.

8 CONCLUSION

When different devices with distinct memory consistency models are glued together using an inter-device coherence protocol, what consistency model should the heterogeneous machine satisfy? In this paper, we have provided an answer to this question in the form of a generalized operational memory model called the compound memory model. The key insight behind our approach is that a wide variety of memory models including scoped and non-multi-copy-atomic memory models can be modeled via the the LOST principle – Local Ordering by Stalling on Threads – leading to a naturally compositional memory model. We instantiate our model with the x86TSO/PTX memory model; we model it axiomatically and show that such a compound model retains compiler mappings. Finally, we have presented evidence to show that compound memory models are naturally realized as long as multiple devices are fused together via a cache-coherent interconnect. The practicality of realizing compound memory models combined with its preservation of compiler mappings makes it an attractive value proposition for heterogeneous machines.

ACKNOWLEDGMENTS

We thank Daniel Lustig, Chengsong Tan, Brandon Lucia (our shepherd) and the anonymous reviewers for their helpful comments and suggestions. This work was funded by the Engineering and Physical Sciences Research Council, through grant references EP/V038699/1 and EP/V028154/1.

ARTIFACT

This paper has an accompanying artifact which includes implementations of both models, LOST-POP as described in Section 4 and the axiomatic models discussed in Section 6, as well as the simulations described in Section 7. The version of the artifact accompanying this paper can be found at https://doi.org/10.5281/zenodo.7711175.

The code for the LOST-POP model in Lean 4 [de Moura and Ullrich 2021] is also available at https://github.com/goens/lost-pop-lean, whereas the axiomatic models, implemented in Alloy [Jackson 2012], can be found individually at https://github.com/goens/cmm-axiomatic. The code for the simulations can be found at https://github.com/sukarnagarwal/PLDI23_Compound_Simulation.

The artifact is packaged in a Docker[11] image. The image can be downloaded from the Zenodo artifact or from DockerHub, at https://hub.docker.com/r/goens/pldi23-ae. Each of the variants listed here has its own README file with more detailed instructions on how to run and modify that part of the artifact.

REFERENCES

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (2021), 54 pages. https://doi.org/10.1145/3458926

Jade Alglave and Luc Maranget. 2022. herd7 consistency model simulator. http://diy.inria.fr/www/.

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

AMD. 2022a. AMD Instinct™ MI200 Series Accelerator and Node Architectures. https://chipsandcheese.com/2022/09/18/hot-chips-34-amds-instinct-mi200-architecture/. Accessed: 9th September 2022.

AMD. 2022b. AMD ROCm Memory model. https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Codeobj-format.html#memory-model. Accessed: 8th August 2022.

ARM. 2011. Atomicity in the ARM Architecture. https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/Application-Level-Memory-Model/Memory-types-and-attributes-and-the-memory-order-model/Atomicity-in-the-ARM-architecture. Accessed: 20th March 2023.

ARM 2018. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.* ARM. Initial v8.4 EAC release.

ARM. 2021. The AMBA CHI Specification. https://developer.arm.com/architectures/system-architectures/amba/amba-5. Accessed: 5th July 2022.

Mark Batty. 2017. Compositional relaxed concurrency. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (September 2017). https://kar.kent.ac.uk/64300/

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL'11.* ACM, 55–66. https://doi.org/10.1145/1926385.1926394

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO '17.* IEEE, 100–110.

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. 3, POPL (2019). https://doi.org/10.1145/3290383

CXL. 2022. Compute Express Link. https://www.computeexpresslink.org/. Accessed: 5th July 2022.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *International Conference on Automated Deduction.* Springer, 625–635.

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *POPL 2016.* 608–621.

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. In *POPL'17.*

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

HSA Foundation. 2012. Heterogeneous System Architecture: A Technical Review.

Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The semantics of shared memory in Intel CPU/FPGA systems. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485497

ISO/IEC 14882. 2011. Programming Language C++.

ISO/IEC 9899. 2011. Programming Language C.

Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis.* MIT press.

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. In *POPL'17* (Paris, France) *(POPL 2017).* Association for Computing Machinery, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

[11] https://www.docker.com/

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. 618–632. https://doi.org/10.1145/3062341.3062352 Technical Appendix Available at https://plv.mpi-sws.org/scfix/full.pdf.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691.

Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 257–270. https://doi.org/10.1145/3297858.3304043

Paul E. McKenney. 2017. Is Parallel Programming Hard, And, If So, What Can You Do About It? (v2017.01.02a). *CoRR* abs/1701.00854 (2017). arXiv:1701.00854 http://arxiv.org/abs/1701.00854

Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00962ED2V01Y201910CAC049

NVIDIA. 2019. CUDA Toolkit Documentation - PTX ISA.

NVIDIA. 2022. NVIDIA debuts Grace CPU Superschip. https://nvidianews.nvidia.com/news/nvidia-introduces-grace-cpu-superchip. Accessed: 22nd August 2022.

Nicolai Oswald, Vijay Nagarajan, Daniel J. Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. 2022. HeteroGen: Automatic Synthesis of Heterogeneous Cache Coherence Protocols. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022.* IEEE, 756–771. https://doi.org/10.1109/HPCA53966.2022.00061

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *TPHOLs*. 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the Gap between Programming Languages and Hardware Weak Memory Models. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290382

Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *PLDI'12*. ACM, 311–322. https://doi.org/10.1145/2254064.2254102

Peter Sewell. 2022. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html. Accessed April 5, 2023.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

The OpenCAPI Consortium. 2021. The OpenCAPI Consortium. https://opencapi.org/. Accessed: 5th July 2022.

Andrew Waterman and Krste Asanovic. 2019. The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. 2017. Weak Memory Models: Balancing Definitional Simplicity and Implementation Flexibility. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017.* IEEE Computer Society, 288–302. https://doi.org/10.1109/PACT.2017.29

Technical Appendix

This is the technical appendix of the article “Compound Memory Models”.

CONTENTS

A THE LOST-POP MODEL

In this appendix we describe the full model of **L**ocal **O**rdering by **S**talling on **T**threads extension to POP. As its name suggests, this operational model is based on the POP model [Flur et al. 2016] but generalizes it via extensive adaptations.

A.1 Basics

We divide the LOST-POP model into different parts: The *architecture* describes a concrete kind of architecture and its memory consistency model, e.g. x86-TSO or PTX. A *system* is defined by a set of threads, where each thread belongs to a given architecture. The threads are grouped hierarchically in scopes (in sets of subsets of all threads). If the system is heterogeneous, i.e. not all threads are of the same architecture, then there must be a subset of scopes partitioning the system into distinct architectures. For example, a compound TSO-PTX system might have 8 TSO threads and 64 PTX threads, split into two distinct device scopes for the TSO and PTX parts each, the PTX subsystem being further divided into more scopes.

In general, the model concerns itself with requests in the memory subsystem. We have three kinds of requests, *reads*, *writes* and *fences*. An architecture can further specialize these, as we will see. More formally, given a system as a set of threads grouped hierarchically in scopes, it defines a transition system that gives the operational semantics. The transitions of this system can be one of three:

(1) Accept a request r at a thread t.
(2) Propagate a request r to a thread t.
(3) Satisfy a request r with a request w.

Just like in the POP model, the LOST-POP model works by defining a set of predicates that state when each of these transitions can be taken by the system. The state of a system is given by

(1) A set `requests` of (active) requests. Each request has a unique id, an originating thread, a list of threads it has propagated to, a list of threads it is a predecessor at, and an architecture-specific request type. Reads and writes additionally have an address and writes additionally a value.
(2) A set `satisfied` of satisfied read requests, which are annotated with the value they read. This set is disjoint from the set of active requests, i.e. satisfied reads are not active.
(3) A labeled `order` relation between active requests, where the labels of the relation correspond to the scopes of the system. This relation represents a causal order in the system, either because of an observation or because of a preserved order. It is comparable to the "happens before" relation in some axiomatic models. The scope label represent until when this order is preserved in a scoped system; observations are observed everywhere and thus are always labeled with the full system's scope.

An *execution trace* is a set of transitions in such a system, which can yield a potential *outcome*, defined by the values read by each of the reads in the trace. It is total if the state after the end of the execution trace cannot take a propagate or satisfy-read transition.

A.2 Transitions

This section discusses the three kinds of transitions, both how they change the state or when the system can take them.

A.2.1 Accept a request r at thread t. A request r can be accepted at any time at any thread t, as long as the architecture type of r and t match. We model the memory subsystem, and hence phenomena like address dependencies are not modeled here explicitly. These would result in additional conditions to accepting a request.

The effect of accepting a request r at thread t is that it gets added to the list of requests in the state, given the next id after all requests accepted previously. The request is also marked as propagated to its originating thread t, and the order constraints of the state are updated as well. A request accepted is always placed *after* request already in the system. Concretely, for a request $r' \neq r$, we add an edge $r' \longrightarrow r$ iff:

(1) If both r, r' are memory operations (reads or writes) to the same address and at least one of them is a write. These edges represent observations in the system and are thus marked to be at the system's scope.
(2) If r' is also a request on thread t, or is marked as a predecessor of t, then we add the edge iff (r', r) satisfies the architecture-specific *order condition*. The edge is marked by the, also architecture-specific, *scope intersection* of r' and r.

A.2.2 Propagate a request r to thread t. The order constraints control when requests can propagate. Concretely, a request r can be propagated to thread t if the following are true

(1) Request r is a(n active) memory request, i.e. read or write, and has not yet propagated to thread t.
(2) No request r' is blocking r from propagating. In this context, we say a request r' *blocks* request r iff
   (a) Both requests r, r' originate from the same thread t', there's an order constraint $r' \longrightarrow r$ marked with scope s and r' has not propagated to every thread in s *or*
   (b) Request r' originates from a different thread, there is an order constraint $r' \longrightarrow r$, and r' has not propagated to t.
(3) Any architecture-specific constraints.

Note that only memory operations (reads or writes) propagate, fences do not. Fences are used to enforce order constraints. Propagation adds Thread t to the list of threads r has propagated to. It also updates the order constraints after this. Propagation can only yield observations, not enforced constraints, these are all locally enforced on their own threads. Contrary to accepting, a request propagated is placed *before* requests already in that thread. Concretely, for a request $r' \neq r$, we add an edge $r \longrightarrow r'$ iff:

(1) Thread t is the originating thread of r', i.e. $t = r.\texttt{thread}$.
(2) r' has not propagated to Thread $r.\texttt{thread}$, the originating thread of r.
(3) Both r, r' are memory operations (reads or writes) to the same address and at least one of them is a write.

As it again represents an observation, this edge is marked with the full system's scope. There is one exception where propagation also adds edges from architecturally-enforced order constraints, not just observations. If r is a write and by propagating we add an edge $r \longrightarrow r'$ with a read r', then we also mark r as a predecessor of thread t. An architecture might impose additional conditions for marking such a request as a predecessor. The idea of this is to avoid breaking the causality chain if read r' is potentially removed later when satisfying a request. If we mark r as a predecessor of t, then we also add edges $r \longrightarrow r''$ to any other requests r'' in Thread t according to the architecture-specific order constraints.

A.2.3 Satisfy a request r with a request w. We can satisfy a read request r with a write request w if the following conditions hold:

(1) r is not already satisfied.

(2) r and w are, respectively, a read and a write to the same address.
(3) r and w have been propagated to exactly the same set of threads.
(4) There is an order constraint $w \longrightarrow r$.
(5) There is no other memory request r' between r and w to the same address as r and w, i.e. such that $w \longrightarrow r'$ and $r' \longrightarrow r$.

Note that there is no reading from memory, as the model does not explicitly have a concept of memory. Instead, we model a request reading from memory as a request being propagated to all threads in the system. To model the initial value in memory, we might thus start the system state with fully-propagated writes to each address with those initial values.

When we satisfy r with w, we change the state in the following way:

(1) We remove r from the active requests in the state.
(2) We mark r as a satisfied request and set its value to the value of w.

A.3 Order Constraints

The architecture-specific order constraints can be one of the following types

(1) read to write with predecessors
(2) read to write without predecessors
(3) read to read with predecessors
(4) read to read without predecessors
(5) write to read
(6) write to write

Fences are always ordered with one another. As fences are not memory operations and do not propagate, this can again be seen as an observation, rather than a constraint. It just tells us the program order of the fences.

A.4 Memory and Initialization

We can intuitively think of the value of a write being in memory when it has propagated to the whole system. Consequently, a read cannot be satisfied by memory, it has to be satisfied by a write. To model this, we simply consider (implicit) initial writes to every address that start by being propagated to every thread.

Consider the message passing program depicted in Figure 14. In it, two values are written in the first thread ($T1$), $X = 1; Y = 1$ and read by the second thread ($T2$) in the opposite order, $a = Y; b = X$. The two addresses X and Y are initialized to 0 with (fully-propagated) implicit writes. In the example depicted, $T2$ accepts the two requests first, before $T1$ accepts anything. The second read, $b = X$ can propagate to $T1$ before $a = Y$, as they are not ordered. When this happens, an order constraint is added between the (implicit) initialization write $X = 0$ and the read $b = X$. The read request can be satisfied, yielding the result $b = 0$ and removing the read request from the system state. Then, in multiple (elided) steps, the write requests are accepted by $T1$ and propagated to $T2$. This adds order constraints between the writes and the unsatisfied read, $a = Y$. This read can only be satisfied by the last write, $Y = 1$, since there is an order constraint between $Y = 0$ and $Y = 1$. Satisfying with $Y = 0$ would be forbidden by the rule described above, where $Y = 1$ corresponds to the request w' between $r : a = Y$ and $w : Y = 0$. Note that in the last depicted state, this read could not yet be satisfied, as $a = Y$ has not yet propagated to $T1$.

A.5 Architectures

Our model is "architecture polymorphic", in the sense that we can instantiate different types of architectures. An architecture is mainly defined by its concrete request types. These can be used

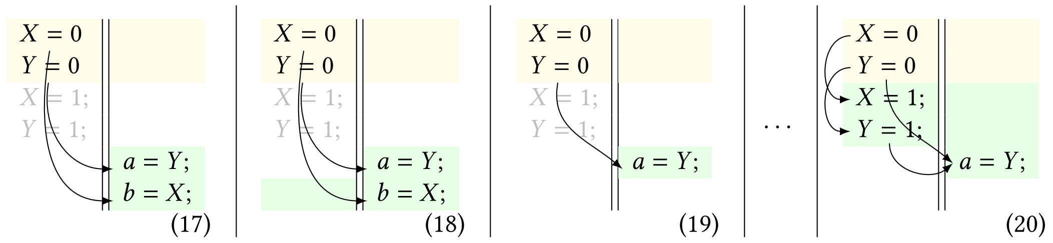

Fig. 14. A Message Passing Example in LOST-POP, showing the initial writes explicitly. These initial writes start fully propagated (to all threads), so when the two reads are accepted (17) they get ordered with the initial writes. The read of $b = X$ can propagate first to the other thread, if there is no ordering between the two reads (18), and be satisfied with the initial write (19). Then the writes are accepted and propagate, allowing $a = Y$; to read from $Y = 1$ (20).

to distinguish between different types of reads, writes or fences (e.g. relaxed vs. release reads, or acquire vs SC fences). Correspondingly, for each pairs of requests types, an architecture has to specify an order condition. The order condition is built using the rules described above.

A.5.1 x86 TSO. In x86 there's no scopes nor special types of reads or writes, so the type of requests is a trivial (unit) type. The order condition in x86 TSO basically enforces the preserved program order, which means

(1) read to write with predecessors
(2) read to read with predecessors
(3) write to write

Write to read is also enforced for same-address requests.

A.5.2 PTX. The PTX memory model is more complex than x86 TSO. Memory operations can be marked with both a scope and different kinds of semantics. The scope markings can be one of `system`, gpu and `cta`. The semantics markings can be one of `weak`, `relaxed`, `release`, `acquire`, `acquire-release` and `sc`. The semantics markings are partially ordered, where `weak` < `relaxed` < `release`, `acquire` < `acquire-release` < `sc`, only `release` and `acquire` being incomparable. For the scopes we consider the intersection of the scopes, which we define as follows. Given two requests r, r' marked with scopes s and s' respectively, then the intersection is given by

$$s \wedge s' := \begin{cases} s', & \text{if } r'.\texttt{sem} \geq \texttt{release} \\ s, & \text{if } r.\texttt{sem} \geq \texttt{acquire} \\ \min(s, s'), & \text{otherwise} \end{cases}$$

We say that the scopes of two requests r, r' match if

$$\{r.\texttt{thread}, r'.\texttt{thread}\} \subseteq r.\texttt{scope} \wedge r'.\texttt{scope}.$$

The order relation of PTX thus adds an edge $r \longrightarrow r'$ at scope $s = r.\texttt{scope} \wedge r'.\texttt{scope}$ if the scopes of r and r' match and *any* of the following holds:

(1) r or r' is a fence and $r.\texttt{thread} = r'.\texttt{thread}$.
(2) r and r' are both reads and $r.\texttt{address} = r'.\texttt{address}$.
(3) $r.\texttt{sem} \geq \texttt{acquire}$ and $r.\texttt{thread} = r'.\texttt{thread}$.
(4) $r'.\texttt{sem} \geq \texttt{acquire}$, r is a read and $r.\texttt{thread} = r'.\texttt{thread}$.
(5) $r'.\texttt{sem} \geq \texttt{release}$ and ($r.\texttt{thread} = r'.\texttt{thread}$ or r is a predecessor at $r'.\texttt{thread}$.
(6) $r.\texttt{sem} \geq \texttt{release}$, r' is a write and $r.\texttt{thread} = r'.\texttt{thread}$.

In PTX we add a predecessor write w to the thread of read r only if the scopes of r and w match. We say that a request is fence-like if it is a fence or its semantics are $\geq$ `release`, `acquire`. A fence-like request r' is blocked on request r iff all of the above hold:

(1) r either originates or is a predecessor in r'.`thread`.
(2) r is order-constraints before r' in the scope that encompasses both r.`thread` and r'.`thread`
(3) r has not propagated to every thread in the scope $s = r$.`scope` $\wedge$ r'.`scope`.
(4) if r is a read then r' not all order-constraints-preceding writes have propagated to all threads s.
(5) if r is a write and r'.`sem` $\geq$ `release` then not all order-constraints-preceding reads, writes and predecessors (of r'.`thread`) have propagated to all threads in s.

A request r can propagate to thread t in PTX iff, in addition to the base model's conditions, for any request r' such that r is blocked on r' the following hold:

(1) if r is a read and r'.`sem` $\geq$ `acquire` then all preceding reads in a thread and their corresponding order-constraints-preceding writes have propagated to all threads in $s = r$.`scope` $\wedge$ r'.`scope`.
(2) if r is a read and r'.`sem` = `sc` then all preceding reads, writes and predecessors (of r.`thread`) have propagated to all threads in $s = r$.`scope` $\wedge$ r'.`scope`.
(3) if r is a write and r'.`sem` $\geq$ `acquire` then all preceding reads in a thread and their corresponding order-constraints-preceding writes have propagated to all threads in $s = r$.`scope` $\wedge$ r'.`scope`.
(4) if r is a write and r'.`sem` $\geq$ `release` then all preceding reads, writes and predecessors (of r.`thread`) have propagated to all threads in $s = r$.`scope` $\wedge$ r'.`scope`.

B AXIOMATIC MODELS

This appendix summarizes the original axiomatic models for PTX and scoped-C11, as presented in [Lustig et al. 2019], as well as x86TSO, as presented in [Alglave et al. 2021].

B.1 PTX

$$\mathsf{prel} \triangleq ([\mathsf{W}_{\sqsupseteq\mathrm{REL}}]; \mathsf{po}_{=\mathsf{loc}}{}^{?}; [\mathsf{W}]) \cup ([\mathsf{F}_{\mathrm{REL}}]; \mathsf{po}; [\mathsf{W}]) \qquad \mathsf{obs} \triangleq (\mathsf{ms} \cap \mathsf{rf}) \cup (\mathsf{obs}; \mathsf{rmw}; \mathsf{obs})$$

$$\mathsf{pacq} \triangleq ([\mathsf{R}]; \mathsf{po}_{=\mathsf{loc}}{}^{?}; [\mathsf{R}_{\sqsupseteq\mathrm{ACQ}}]) \cup ([\mathsf{R}]; \mathsf{po}; [\mathsf{F}_{\mathrm{ACQ}}]) \qquad \mathsf{sw} \triangleq (\mathsf{ms} \cap (\mathsf{prel}; \mathsf{obs}; \mathsf{pacq})) \cup \mathsf{sc}$$

$$\mathsf{cause} \triangleq \mathsf{causeb} \cup (\mathsf{obs}; (\mathsf{causeb} \cup \mathsf{po}_{=\mathsf{loc}})) \text{ where } \mathsf{causeb} \triangleq (\mathsf{po}^{?}; \mathsf{sw}; \mathsf{po}^{?})^{+}$$

With these definitions, the PTX memory model is given by the following propositions:

- $[\mathsf{W}]; \mathsf{cause}_{=\mathsf{loc}}; [\mathsf{W}] \subseteq \mathsf{co}$ (Coherence)
- $(\mathsf{sc}; \mathsf{cause})$ is irreflexive (FenceSC)
- $((\mathsf{rf} \cup \mathsf{fr}); \mathsf{cause})$ is irreflexive (Causality)
- $((\mathsf{ms} \cap \mathsf{fr}); (\mathsf{ms} \cap \mathsf{co})) \cap \mathsf{rmw} = \emptyset$ (Atomicity)
- $\mathsf{po}_{=\mathsf{loc}} \cup (\mathsf{ms} \cap (\mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr}))$ is acyclic (SC-per-Location)
- $(\mathsf{rf} \cup \mathsf{dep})$ is acyclic (No-Thin-Air)

B.2 x86TSO

The x86TSO memory model is given as follows:

- $(\mathsf{po}_{=\mathsf{loc}} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co})$ is acyclic. (sc-per-loc)
- $\mathsf{rmw} \cap (\mathsf{fre}; \mathsf{coe}) = \emptyset$ (atomicity)
- xhb is irreflexive where (GHB)
  - $\mathsf{ppo} \triangleq ((\mathsf{W} \times \mathsf{W}) \cup (\mathsf{R} \times \mathsf{W}) \cup (\mathsf{R} \times \mathsf{R})) \cap \mathsf{po}$
  - $\mathsf{implied} \triangleq (\mathsf{po}; [A]) \cup ([A]; \mathsf{po})$ where $A = \mathrm{dom}(\mathsf{rmw}) \cup \mathrm{codom}(\mathsf{rmw}) \cup \mathsf{F}_{\mathsf{x}}$
  - $\mathsf{xhb} \triangleq (\mathsf{ppo} \cup \mathsf{implied} \cup \mathsf{rfe} \cup \mathsf{fr} \cup \mathsf{co})^{+}$

B.3 scoped-C/C++

Scoped-C/C++ is a scoped extension to C/C++. It is given by:

- $(\mathsf{hb}; \mathsf{eco}^{?})$ is irreflexive (Coherence)
- $\mathsf{rmw} \cap (\mathsf{fr}; \mathsf{co}) = \emptyset$ (Atomicity)
- $(\mathsf{psc} \cap \mathsf{ms})$ is irreflexive. (SC)

where

$$\mathsf{rs} \triangleq [\mathsf{W}]; \mathsf{po}_{=\mathsf{loc}}{}^{?}; [\mathsf{W}_{\sqsupseteq\mathrm{RLX}}]; ((\mathsf{ms} \cap \mathsf{rf}); \mathsf{rmw})^{*} \qquad \text{(release-sequence)}$$

$$\mathsf{sw}_{\mathsf{cc}} \triangleq [\mathsf{E}_{\sqsupseteq\mathrm{REL}}]; ([\mathsf{F}]; \mathsf{po})^{?}; \mathsf{rs}; (\mathsf{ms} \cap \mathsf{rf}); [\mathsf{R}_{\sqsupseteq\mathrm{RLX}}]; (\mathsf{po}; [\mathsf{F}])^{?}; [\mathsf{E}_{\sqsupseteq\mathrm{ACQ}}] \qquad \text{(synchronization-with)}$$

$$\mathsf{hb} \triangleq (\mathsf{po} \cup (\mathsf{ms} \cap \mathsf{sw}_{\mathsf{cc}}))^{+} \qquad \text{(happens-before)}$$

$$\mathsf{scb} \triangleq \mathsf{po} \cup (\mathsf{po}_{\neq\mathsf{loc}}; \mathsf{hb}; \mathsf{po}_{\neq\mathsf{loc}}) \cup \mathsf{hb}_{=\mathsf{loc}} \cup \mathsf{co} \cup \mathsf{fr} \qquad \text{(SC-before)}$$

$$\mathsf{psc}_{\mathsf{base}} \triangleq ([\mathsf{E}_{\mathsf{sc}}] \cup [\mathsf{F}^{\mathsf{sc}}]; \mathsf{hb}^{?}); \mathsf{scb}; ([\mathsf{E}_{\mathsf{sc}}] \cup \mathsf{hb}^{?}; [\mathsf{F}_{\mathsf{sc}}]) \qquad \text{(partial-SC-base)}$$

$$\mathsf{psc}_{\mathsf{F}} \triangleq [\mathsf{F}^{\mathsf{sc}}]; (\mathsf{hb} \cup \mathsf{hb}; \mathsf{eco}; \mathsf{hb}); [\mathsf{F}_{\mathsf{sc}}] \qquad \text{(partial-SC-fence)}$$

$$\mathsf{psc} \triangleq \mathsf{psc}_{\mathsf{base}} \cup \mathsf{psc}_{\mathsf{F}} \qquad \text{(partial-SC)}$$

Races. There are two types of races in scoped-C/C++: data race and heterogeneous race. In a consistent execution, a *data race* happens when the execution contains a pair of concurrent events which access the same memory location, at least one of them is a write and at least one of the access is non-atomic. While a data race may take place in the non-scoped C/C++ model [ISO/IEC 14882 2011; ISO/IEC 9899 2011] as well, a heterogeneous race is specific to the scoped C/C++ model [Lustig et al. 2019]. In a consistent execution a pair of event is in *heterogeneous race* if the events access same memory location, at least one is write, and the events are not morally strong hb-related. If a program has a racy execution then the program has *undefined behavior*.

C PROOF OF COMPILER MAPPINGS

In this appendix, we show the mapping scheme discussed in the paper (in Section 6) is correct for the axiomatic compound memory model CMM. This means that any observable behavior in the compiled program is also a behavior of the original scoped C/C++ program.

Definition C.1. Given an execution, its *behavior* is defined by the final values of all memory locations. These values written by the writes which have no co-successor in the execution.

$$\mathsf{Behav}(\mathsf{X}) \triangleq \{\langle \mathsf{e.loc}, \mathsf{e.Val}\rangle \mid \mathsf{e} \in \mathsf{X.W} \wedge [\{\mathsf{e}\}]; \mathsf{X.co} = \emptyset\}$$

We strengthen the non-release-acquire accesses to be mapped to x86TSO to release-acquire accesses to generate a program $\mathbb{P}_s$. We consider only race-free X_s executions. We then perform the splitting transformations (the SC writes mapped to x86TSO writes and the SC accesses in PTX are replaced by the respective memory accesses followed by SC fences) following the mapping from Figure 13 to generate X_t.

Next, we prove the mapping correctness of program $\mathbb{P}_s$ on CMM. Suppose the mapping of $\mathbb{P}_s$ generates $\mathbb{P}_t$ program on CMM. We prove that for each consistent execution X_t of $\mathbb{P}_t$ in CMM there exists a scoped C/C++ consistent execution X_s of $\mathbb{P}_s$ such that X_s is racy or race-free with the same behavior. If it is racy then the transformations are allowed. Hence we consider only race-free X_s execution. We prove this step in the following sub-steps.

(1) Define X_s from X_t, We map the events one-to-one following the mapping schemes in Figure 13. Moreover, the po, rf, co relations in X_s match the corresponding relations in X_t. Suppose event a and b in $\mathsf{X}_s.\mathsf{E}$ are mapped to a' and b' in $\mathsf{X}_t.\mathsf{E}$. If $(a, b) \in \mathsf{X}_s.R$ then $(a', b') \in \mathsf{X}_t.R$ and vice versa where R is in $\{\mathsf{po}, \mathsf{rf}, \mathsf{co}, \mathsf{rmw}\}$.

(2) Relate the relations in X_s and X_t,

Analyze sw_{cc} *in* X_s. Inlining rs in sw_{cc} we get:

$$\begin{aligned}\mathsf{sw}_{cc} \triangleq\ & [\mathsf{E}_{\sqsupseteq\mathrm{REL}}]; ([\mathsf{F}]; \mathsf{po})^?; [\mathsf{W}]; \mathsf{po}_{=\mathsf{loc}}{}^?; [\mathsf{W}_{\sqsupseteq\mathrm{RLX}}];\\ & ((\mathsf{ms} \cap \mathsf{rf}); \mathsf{rmw})^*;\\ & (\mathsf{ms} \cap \mathsf{rf}); [\mathsf{R}_{\sqsupseteq\mathrm{RLX}}]; (\mathsf{po}; [\mathsf{F}])^?; [\mathsf{E}_{\sqsupseteq\mathrm{ACQ}}]\end{aligned}$$

It implies $\mathsf{sw}_{cc} \triangleq \mathsf{Wrl}; \mathsf{rseq}^*; \mathsf{Raq}$ where $\mathsf{Wrl} = [\mathsf{E}_{\sqsupseteq\mathrm{REL}}]; ([\mathsf{F}]; \mathsf{po})^?; [\mathsf{W}]; \mathsf{po}_{=\mathsf{loc}}{}^?; [\mathsf{W}_{\sqsupseteq\mathrm{RLX}}]$, $\mathsf{rseq} \triangleq ((\mathsf{ms} \cap \mathsf{rf}); \mathsf{rmw})^*; (\mathsf{ms} \cap \mathsf{rf})$, and $\mathsf{Raq} = [\mathsf{R}_{\sqsupseteq\mathrm{RLX}}]; (\mathsf{po}; [\mathsf{F}])^?; [\mathsf{E}_{\sqsupseteq\mathrm{ACQ}}]$ hold.

We analyze two cases of rseq-related memory events for heterogeneous race.

Case rseq $\not\subseteq$ ms:
We know dom(rseq) consists of write operations. It implies there is a heterogeneous race between dom(rseq) and codom(rseq) and the program behavior is undefined. In this case, any mapping is possible.

Case rseq $\subseteq$ ms: rseq-related memory events are not in heterogeneous race. In this case we analyze the correctness.

Following the mapping schemes and the definitions:

- Wrl in X_s maps to gprel in X_t.
- rseq in X_s maps to $\mathsf{ms} \cap \mathsf{obs}$ in X_t as $\mathsf{rseq} \subseteq \mathsf{ms}$.
- Raq in X_s maps to gpacq in X_t.

Therefore, $\mathsf{X}_s.\mathsf{sw}_{cc}$ results in $(\mathsf{gprel}; (\mathsf{ms} \cap \mathsf{obs}); \mathsf{gpacq})$ in X_t.
It implies that sw_{cc} in X_s results in xgsw relation in X_t.

Analyze hb *in* X_s. From the definition we know that hb in X_s implies $(\mathsf{po} \cup (\mathsf{ms} \cap \mathsf{sw}_{\mathsf{cc}}))^+$ in X_s.
It implies $\mathsf{po} \cup (\mathsf{po}^?; (\mathsf{ms} \cap \mathsf{sw}_{\mathsf{cc}}); \mathsf{po}^?)^+$ in X_s.
It results in $\mathsf{po} \cup (\mathsf{po}^?; \mathsf{xgsw}; \mathsf{po}^?)^+$ in X_t.
It implies $\mathsf{po} \cup \mathsf{xgcauseb}$ in X_t as $\mathsf{xgsw} \subseteq \mathsf{xgcauseb}$.
It implies $\mathsf{po} \cup \mathsf{cord}$ in X_t as $\mathsf{xgcauseb} \subseteq \mathsf{cord}$.

(2) X_s preserves consistency in scoped C/C++ model, We prove this by contradiction.
Assume X_s violates the (Coherence) axiom.
It implies X_s contains an $\mathsf{hb}; \mathsf{eco}^?$ cycle.
It results in an $(\mathsf{po} \cup \mathsf{cord}); \mathsf{eco}^?$ a cycle in X_t.
We know po is irreflexive and po; eco? a cycle in X_t violates (CMM-sc-per-loc) which is a contradiction.
Also a $\mathsf{cord}; \mathsf{eco}^?$ cycle violates (CMM-irrCordECO) which is a contradiction.
Hence X_s has no $\mathsf{hb}; \mathsf{eco}^?$ cycle.

Assume X_s violates (Atomicity).
It implies a $\mathsf{rmw} \cap (\mathsf{fr}; \mathsf{co}) = \emptyset$ holds in X_s.
Suppose $\mathsf{rmw}(r, w)$ and there exists w' such that $\mathsf{fr}(r, w')$ and $\mathsf{co}(w', w)$ in X_s.
We analyze the two possibilities considering the ms relation:

Case $\mathsf{ms}(r, w')$ and $\mathsf{ms}(w', w)$ holds:
It implies $(((\mathsf{ms} \cap \mathsf{fr}); (\mathsf{ms} \cap \mathsf{co})) \cap \mathsf{rmw}) = \emptyset$ holds in X_t which violates (CMM-Atomicity).
Hence a contradiction as X_t is CMM-consistent.
So, X_s preserves (Atomicity).

Case Otherwise:
X_s contains heterogeneous race (r, w') and (w', w) and the scoped C/C++ program behavior is undefined. Thus, any transformation is correct.

Assume X_s violates (SC).
It implies that X_s contains a $(\mathsf{psc} \cap \mathsf{ms})$ cycle with a set of SC events.
We categorize the events on the psc cycle in two exclusive sets: A and B where

- Set A be the set of SC events A whose po-following events are on the psc path.
- Set B be the set of SC events A whose po-following events are NOT on the psc path.

In this case $A \neq \emptyset$ as in that case the psc cycle would be on same location events. In that case, X_t also has a cycle on same-location events, which is a contradiction.

Thus, the cycle consists sequence of events from A which may have an intermediate sequence of events from B.

Given an event $a \in A$, let e_1 and e_2 be the SC-events which are the immediate psc predecessor and successor of a respectively. We can apply a splitting transformation with a trailing-fence scheme [Lahav et al. 2017]. In that case, a (with changed memory order) will have an immediate po-successor F_{SC} event f which will be placed on the psc path. Therefore, $\mathsf{psc}(e_1, f)$ and $\mathsf{psc}(f, e_2)$ hold on the cycle.

Thus we apply the splitting transformations on all events in A except the read and RMW events, which would be mapped to x86TSO and come up with an alternative psc cycle. We re-compute A where we remove the SC memory events and include the introduced fences.

The psc cycle contains the following events:

- (Type 1) R_{SC} events which will be mapped to x86TSO.
- (Type 2) $(\mathsf{R}_{\mathsf{SC}} \cup \mathsf{W}_{\mathsf{SC}}) \in B$ events which has fre and coe successor edges on psc cycle.

- (Type 3) $\mathsf{F_{sc}} \in A$ events.

Without loss of generality, consider a sequence of Type 2 events $a_1, a_2, \dots, a_n$. It is reduced to a $(\mathsf{co} \cup \mathsf{fr})$ relation based on the start of the sequence.

Without loss of generality, consider e_5 and e_6 as the immediate predecessor and successor SC events on the psc path.

- $(e_5, a_1) \in [\mathsf{R_{sc}}]; \mathsf{scb}; \mathsf{hb}^?$,
- $(a_1, a_n) \in (\mathsf{co} \cup \mathsf{fr}); [\mathsf{W_{sc}}]$, and
- $(a_n, e_6) \in [\mathsf{W_{sc}}]; (\mathsf{rfe}; \mathsf{hb}_{=\mathsf{loc}}{}^?); [\mathsf{R_{sc}} \cup (\mathsf{hb}^?; [\mathsf{F_{sc}}])]$.

This yields two possibilities:

(1) The cycle is only of $\mathsf{R_{sc}}$ event or (2) otherwise.

Case The cycle is only of $\mathsf{R_{sc}}$ event:

Now we consider the mapping of the cycle to $\mathsf{X_t}$. Following the mapping schemes

- The $\mathsf{R_{sc}}$ events are mapped to x86TSO.
- The $\mathsf{F_{sc}}$ events are mapped to x86TSO or PTX.

Now consider the corresponding relations on $\mathsf{X_t}$ where $[\mathsf{R_{sc}}]; \mathsf{psc}; [\mathsf{R_{sc}}]$ and the $\mathsf{R_{sc}}$ events are mapped to x86TSO.

Hence, $[\mathsf{R_{sc}}]; \mathsf{psc}; [\mathsf{R_{sc}}]$

$\subseteq [\mathsf{R_{sc}}]; (\mathsf{po} \cup \mathsf{po}_{\neq\mathsf{loc}}; \mathsf{hb}; \mathsf{po}_{\neq\mathsf{loc}} \cup \mathsf{hb}_{=\mathsf{loc}} \cup \mathsf{co} \cup \mathsf{fr}); [\mathsf{R_{sc}}]$ on $\mathsf{X_s}$.

$\subseteq [\mathsf{R_{sc}}]; (\mathsf{po} \cup \mathsf{hb}); [\mathsf{R_{sc}}]$ on $\mathsf{X_s}$ as $[\mathsf{R_{sc}}]; (\mathsf{co} \cup \mathsf{fr}); [\mathsf{R_{sc}}] = \emptyset$.

It implies $[\mathsf{R_x}]; (\mathsf{po} \cup \mathsf{cord}); [\mathsf{R_x}]$ on $\mathsf{X_t}$.

It implies $[\mathsf{R_x}]; \mathsf{cord}; [\mathsf{R_x}]$ on $\mathsf{X_t}$ as $[\mathsf{R_x}]; \mathsf{po}; [\mathsf{R_x}] \subseteq \mathsf{xhb} \subseteq \mathsf{cord}$.

In this case $\mathsf{X_t}.\mathsf{gsc}$ is consistent with $\mathsf{X_s}.\mathsf{psc}$, as it would otherwise create a $\mathsf{gsc}; \mathsf{cord}$ cycle, a contradiction.

Case Otherwise:

Without loss of generality, suppose the psc cycle contains $f \in \mathsf{F_{sc}}$, $r \in \mathsf{R_{sc}}$, and $a \in (\mathsf{R_{sc}} \cup \mathsf{F_{sc}})$ such that $\mathsf{psc}(f, r)$ and $\mathsf{psc}(r, a)$.

In this case, $(f, a) \in [\mathsf{F_{sc}}]; \mathsf{hb}^?; \mathsf{scb}; [\mathsf{R_{sc}}]; \mathsf{scb}; [\mathsf{R_{sc}} \cup (\mathsf{hb}^?; [\mathsf{F_{sc}}])]$

$\subseteq [\mathsf{F_{sc}}]; \mathsf{hb}; [\mathsf{R_{sc}}]; \mathsf{scb}; [\mathsf{R_{sc}} \cup (\mathsf{hb}^?; [\mathsf{F_{sc}}])]$ as $\mathsf{R_{sc}} \notin \mathsf{codom}(\mathsf{co} \cup \mathsf{fr})$.

$\subseteq [\mathsf{F_{sc}}]; \mathsf{hb}; \mathsf{scb}; [\mathsf{R_{sc}} \cup (\mathsf{hb}^?; [\mathsf{F_{sc}}])]$

$\subseteq [\mathsf{F_{sc}}]; \mathsf{psc}$

Therefore, we derive an alternative psc cycle on $\mathsf{X_s}$ without passing through $r \in \mathsf{R_{sc}}$.

Applying the above step we derive a psc cycle of only $\mathsf{F_{sc}}$ events which are in $\mathsf{psc_F}$ relations.

Following Lustig et al. [2019] we can add gsc edges consistently with the $\mathsf{psc_F}$ relations.

The $\mathsf{X_s}.\mathsf{psc_F}$ cycle results in a $\mathsf{X_t}.\mathsf{gsc}$ cycle, which is a contradiction.

Hence (SC) holds.

(4) $\mathsf{X_s}$ and $\mathsf{X_t}$ has same behaviors. We know that the co relations in $\mathsf{X_s}$ and $\mathsf{X_t}$ match. Hence $\mathsf{X_s}$ and $\mathsf{X_t}$ has same behaviors.

Hence the mapping scheme in Figure 13 from scoped C/C++ to CMM is correct.