Weak Memory Models with Matching Axiomatic and Operational Definitions

Sizhuo Zhang[1] Muralidaran Vijayaraghavan[1] Dan Lustig[2] Arvind[1]
[1]{szzhang, vmurali, arvind}@csail.mit.edu [2]dlustig@nvidia.com
[1]MIT CSAIL [2]NVIDIA

Abstract

Memory consistency models are notorious for being difficult to define precisely, to reason about, and to verify. More than a decade of effort has gone into nailing down the definitions of the ARM and IBM Power memory models, and yet there still remain aspects of those models which (perhaps surprisingly) remain unresolved to this day. In response to these complexities, there has been somewhat of a recent trend in the (general-purpose) architecture community to limit new memory models to being (multicopy) atomic: where store values can be read by the issuing processor before being advertised to other processors. TSO is the most notable example, used in the past by IBM 370 and SPARC-TSO, and currently used in x86. Recently (in March 2017) ARM has also switched to a multicopy atomic memory model, and the new RISC-V ISA and recent academic proposals such as WMM are pushing to do the same.

In this paper, we show that when memory models are atomic, it becomes much easier to produce axiomatic definitions, operational definitions, and proofs of equivalence than doing the same is under non-atomic models. The increased ease with which these definitions can be produced in turn allows architects to build processors much more confidently, and yet the relaxed nature of the models we propose still allows most or all of the performance of non-atomic models to be retained. In fact, in this paper, we show that atomic memory models can be defined in a way that is parametrized by basic instruction and fence orderings. Our operational vs. axiomatic equivalence proofs, which are likewise parameterized, show that the operational model is sound with respect to the axioms and that the operational model is complete: that it can show any behavior permitted by axiomatic model.

For concreteness, we instantiate our parameterized framework in two forms. First, we describe GAM (General Atomic Memory model), which permits intra-thread load-store reorderings. Then, we show how forbidding load-store reordering (as proposed by WMM) allows the operational and axiomatic model to be even further simplified into one based on Instantaneous Instruction Execution (I2E). Under I2E, each processor executes instructions in order and instantaneously, providing an even simpler model still for processors which do not implement load-store reordering. We then prove that the operational and axiomatic definitions of I2E are equivalent as well.

1 Introduction

Interest in weak memory models stems from the belief that such models provide greater flexibility in implementation and thus, can lead to higher performance multicore microprocessors than those that support stronger memory models like *Sequential Consistency* (SC) or *Total Store Order* (TSO). However, extremely complicated and contentious definitions of POWER and ARM ISAs, which are the most important modern examples of industrially supported weak memory models, have generated somewhat of a backlash against weak memory models. As recently as 2017, a trend is emerging in which general-purpose processors are moving away from extremely weak (so-called "non-atomic") memory models and back towards simpler options which are much more tractable to understand and analyze.

Over the years, two competing memory model definition approaches have emerged. One form is the *operational model*, which is essentially an abstract machine that can run a program and directly produce its

legal behaviors. The other form is the *axiomatic model*, which is a collection of constraints on legal program behaviors. Each type has its own advantage. Axiomatic models can use general-purpose combinatorial search tools like model checkers and SMT solvers to check whether a specific program behavior is allowed or disallowed, and they are useful for building computationally-efficient tools [8, 55, 36]. However they are not as suitable for inductive proofs that aim to build up executions incrementally, and many architects find them rather non-intuitive and a big departure from actual hardware. On the other hand, operational models are very natural representations of actual hardware behavior, and their small step semantics are naturally very well suited to building formal inductive proofs [41].

Given the complementary natures of these two types of definitions, it would be ideal if a memory model could have an axiomatic definition and an operational definition which match each other. Then different definitions can serve different use cases. This is indeed the case for strong memory models like SC and TSO, but unfortunately, not so for weak memory models. The research in weak memory models can then be classified into the following two categories:

1. Build accurate axiomatic and operational models of existing architectures.
2. Specify what memory models ought to look like: proposed memory models should be simple to understand with no obvious restrictions on implementations, and the equivalence of axiomatic and operational models may even be understood intuitively.

While great efforts have been devoted to the first type of research to create models for commercial architectures like POWER and ARM, these models and proofs are still subject to subtle incompatibilities and frequent model revisions that invalidate the efforts [46, 8, 22, 33]. More importantly, the veracity of these models is hard to judge because they are often based on information which is not public. For example, the ARM operational model proposed by Flur et al. [22] allows many non-atomic memory behaviors that cannot be observed in any ARM hardware, and the paper claims that those behaviors are introduced to match the intentions of ARM's architects. However, the recently released ARM ISA manual [10] clearly forbids those behaviors, invalidating the model completely.

This paper falls in the second category, and is motivated by the aim to reduce the complexity of commercial weak memory models. The results in this paper are not purely academic – the growing importance of the open source RISC-V ISA [1] has provided an opportunity to design a clean slate memory model. The memory model for RISC-V is still being debated, and the members of the RISC-V community who are involved in the debate have expressed a strong desire for both axiomatic and operational definitions of the memory model.

In this paper, we present a framework which provides an axiomatic semantics, an operational semantics, and proofs of equivalence, and all in a way that is *parameterized* by the basic instruction and fence orderings in a given model. With our model, specifications and proofs are not nearly as fragile and subject to frequent breakage with every subtle tweak to a memory model definition. Instead, the parameterization allows fence semantics to be simply and easily tweaked as needed.

1.1 Contributions

The main contribution of this paper is GAM, a general memory model for systems with atomic memory. The model is parameterized by fences and basic instruction reorderings. Both its operational and axiomatic definitions can be restricted to provide definitions of other simpler atomic memory models. We provide proofs that the operational definition of GAM is *sound* and *complete* with respect to its *axiomatic* definition. We believe that GAM is the first memory model that allows load-store reordering and for which matching axiomatic and operational definitions have been provided.

On top of GAM, we show that GAM can be further simplified by simply preventing load-store reordering. Such models can be described in terms of Instantaneous Instruction Execution (I2E), a model in which instructions execute instantaneously and in-order, with special memory buffers capturing the weak memory behaviors. Furthermore, I2E models can additionally be parameterized by dependency orderings (under a commonly satisfied constraint), providing even more flexibility. We provide proofs of equivalence for our axiomatic and operational definitions of I2E as well.

Paper organization: In Section 2, we present three issues that complicate the definitions of weak memory models. In Section 3, we presented the related work. In Section 4, we present the axiomatic and operational definitions of our parameterized general atomic memory model GAM, along with the proofs of the equivalence of the two definitions. In Section 5, we present an alternative axiomatic definition of GAM because this definition is better suited for using model checkers. In Section 6, we show how GAM can be restricted to represent other simpler memory models. In Section 7, we show that if Load-Store reordering is disallowed, then the operational models can be described in the Instantaneous Instruction Execution manner and parameterized by dependency orderings. Finally we end the paper with brief conclusions in Section 8.

2 Memory Model Background

In the following, we discuss the three specific challenges in defining a weak memory model such that it has matching operational and axiomatic definitions, and explain briefly how we tackle the challenges.

2.1 Atomic versus Non-atomic Memory

Both ARM (until March 2017) and IBM Power use what is known as *non-atomic memory* which does not have a universally-agreed-upon definition [8, 40]. A major source of complication in weak model definitions stems from the use of non-atomic memory. This lack of consensus makes it difficult to have matching definitions with non-atomic memory. In this paper, we define memory models that use *atomic memory*, or more precisely its variant which is known as *multicopy atomic memory*. By atomic memory we mean a conceptual *multiported monolithic memory* where loads and stores are executed instantaneously and a load a returns the value of the latest store to address a. Multicopy atomic memory lets a processor that generates a store bypass the value of that store to other newer loads in the same processor, before other processors may see that store value. Multicopy atomic memory captures the abstraction of a store buffer in the microarchitecture and is the underlying memory system for the popular TSO memory model used by Intel and AMD [48]. In this paper we will use the term atomic memory and multicopy atomic memory interchangeably.

In the RISC-V debate a strong consensus has emerged that the memory model for RISC-V should depend only on atomic memory and therefore in this paper we will discuss only atomic memory models.

2.2 Instruction Reorderings and Single-thread Semantics

Modern high-performance processors invariably execute instructions out of order (aka OOO processors) but they do it such that this reordering is transparent to a single threaded program. However, in a multithreaded setting these instruction reorderings become visible. A major classification of memory models is along the lines of which (memory) instruction reorderings are permitted. For example, SC does not allow any reordering, while TSO allows a Load to be reordered with respect to previous Stores (i.e., it allows Store-Load reordering). WMM [56], Alpha, ARM, and Power also permit Store-Store and Load-Load reordering, provided the accesses are to different addresses, and all of these, except WMM, also permit Load-Store reordering[10, 29, 2, 56]. The same address Store-Store reordering would clearly destroy the single thread semantics and thus, is prohibited. The reason for disallowing the same address Load-Load reorderings is subtler, and a variation of WMM can be defined that indeed allows such a reordering.

However, it should be noted that all of SC, TSO and WMM have matching axiomatic and operational definitions, while to our knowledge Alpha and RMO have only axiomatic definitions. This difference is likely to be caused by the added complexity of permitting Load-Store reordering, i.e., issuing a Store to the memory before all the previous Loads have completed. A consequence of allowing Load-Store reordering is that the value a load gets in a multithreaded setting can depend upon a future store from the same thread. This complicates operational definitions. Load-Store reordering also complicates axiomatic semantics where a special axiom is often needed to disallow so-called *out-of-thin-air* (OOTA) behavior [16, 2]. These two factors add to the difficulty of matching axiomatic and operational definitions.

The *General Atomic Memory* (GAM) model defined in this paper takes the challenge and allows all four reorderings (i.e., including Load-Store reordering). To model Load-Store reordering, this paper provides an operational definition of GAM using unbounded Reorder Buffer (ROB) with speculative execution and atomic memory. (The memory system itself is not speculative, i.e., once a store has been issued it cannot be retracted.) A similar mechanism has been used in the past to define the operational model for Power [46], but there are separate concerns about that model which are discussed in Section 3.

2.3 Fences for Writing Multithreaded Programs

If we classify memory models based on instruction reordering only then for a given program, GAM allows more program behaviors than WMM, WMM allows more behaviors than TSO, and TSO allows more behaviors than SC. More behaviors generally mean more flexibility in hardware implementation, however, a programmer needs a way to control instruction reorderings in order to write shared memory multithreaded programs. The foundations of all multithreaded programming, from Dijkstra [20] and Lamport [34] to current Java multithreaded libraries, is based on SC, that is, order-preserving interleaving of instructions in a multithreaded program. Hence as a minimum, any ISA supporting a memory model weaker than SC must provide *fence* instructions to make it possible to disallow instruction reorderings to enforce SC, if desired. Not surprisingly, different models require different types of fences and the execution cost of a fence varies from implementation to implementation.

Fences are often explained in two entirely different ways. One way is to define a fence simply to prevent reordering between loads and stores. For example, RMO and RISC-V have four individual fence components, FenceSS, FenceLS, FenceSL and FenceLL, to prevent reorderings between Store-Store, Load-Store, Store-Load and Store-Store, respectively (the actual names of fence instructions are different), and as many as fifteen fences can be formed by composing these options. Such fences specify when two instructions in a dynamic instruction stream in a processor may not be reordered. Of course, for complete specification, one also has to specify how fences may be reordered with respect to each other or how/whether, for example, FenceSS may be reordered with respect to a Load. This view of fences is only about reordering with in a processor and has nothing to do with the memory system.

Specifying how fences control instruction reordering is not sufficient to understand how programs behave. We need to specify what the meaning of "a store has completed", i.e., when the value of a store becomes visible to loads in other processors or to a load in the same processor. One needs to understand the details of the memory subsystem, such as presence of store buffers, write through caches, etc., to give precise meaning to fences.

The second type of fence definitions is usually explained in terms of their effect on memory. For example, a Store-Release fence (alternatively known as a Commit) blocks the execution of the following stores until all the preceding instructions have completed. Similarly, Load-Acquire fence (or Reconcile) blocks the execution of following instructions until all preceding loads are satisfied. Similarly, there is Full-fence instruction that blocks the execution of all subsequent memory instructions until all the preceding memory instructions have completed.

In addition to subtle differences in the semantics of fences, there can be huge differences in performance penalty of using different types of fences. For example, a full fence may be overkill in an algorithm where it may be sufficient to keep two sequential stores from being reordered. Insertion of fences in a multithreaded program by the programmer or the compiler writer is one of the thorniest problems related to weak memory models. If too many unnecessary fences are inserted in a program then it would show poor performance, and in the extreme case the whole purpose of having a weak memory model would be lost. If too few fences are inserted, then the meaning of a program may change by admitting new behaviors which may not be acceptable. The debugging of multithreaded programs is a difficult task in the best of times, insertion of fences creates the possibility of including even more silent bugs which may manifest under very peculiar scheduling conditions. Automatic insertion of fences by a compiler for the programming model such as the one embodied in C11 may be feasible but that memory model of C11 is already based on some cost assumptions of various fences, creating a catch-22 situation [30].

The lack of agreement on the set of fences and the nuances between different fences all add to the difficulty of matching axiomatic and operational definitions. To address these problems, GAM restricts itself to a very simple atomic memory model where there is no ambiguity about when a value is visible to other processors. Such atomic memory automatically avoids many of the thorniest difficulties (such as cumulativity [8]) in the definitions of fences. Since there is still no clear consensus on which set of fences gives the best tradeoff between ease of use and performance, we have parameterized the GAM model with the type of fences. The axiomatic definition, operational definition and the proofs of equivalence are all also parameterized by the type of fences.

3 Related Work

Lamport's paper on SC [34] is probably the first formal definition, both axiomatic and operational, of a memory model. In the nineties, three different weak memory models were defined axiomatically for SUN's Sparc processors: TSO, PSO and RMO [52, 53]. A weak memory model for DEC Alpha was also specified axiomatically in the same time frame [2]. Until a decade ago, however, there was no effort to specify weak memory models operationally or match axiomatic specifications to operational models. In this context, papers by Sarkar et al.[47], Sewell et al.[48] and Owens et al. [43] are very important because they showed that the axiomatic specification of TSO is exactly equivalent to an operational model using store buffers connected to I2E processors and atomic memory.

Until recently, weak memory models have not been defined prior to ISA implementation and have been documented by manufacturers only in an ad hoc manner using a combination of natural language and litmus tests. Not surprisingly, such "definitions" have had to be revised as implementations have changed, revealing new corner cases of behaviors. Over the last decade, several studies have been performed, mostly by academic researchers, to determine the allowed and disallowed behavior of several commercial microprocessors, with the goal of creating formal models to explain the observed behaviors. These studies have been done on real machines by running billions of instructions and recording the observations (just like studying any natural phenomenon). Then, with extra inputs from hardware designers, a model is constructed that tries to satisfy all these observations. For example, Sarkar et al. specified an operational model for POWER [46, 45], using a non-atomic memory. Later, Mador-Haim et al. [37] developed an axiomatic model for POWER and proved that it matches the earlier operational model. Alglave et al. [5, 7, 4, 8, 6] give axiomatic specifications for ARMv7 and POWER using the Herd framework; Flur et al. [22] give operational specification for ARMv8.

However, there has been some dispute if the operational model of POWER models actual POWER processors accurately [8]. We attribute the reason for the potential errors to be the inherent complexity of the operational model because of the use of non-atomic memory. Alglave models are not sufficiently grounded in operational models and face the problem of being too liberal. The model may admit behaviors which cannot be observed in any implementation. Such models can also lead to insertion of unnecessary fences in a program. We think it is important to have matching operational and axiomatic models.

Researchers have also proposed several other consistency models: Processor Consistency [26], Weak Consistency [21], RC [24], CRF [49], Instruction Reordering + Store Atomicity [11]. The tutorials by Adve et al. [3] and by Maranget et al. [40] provide relationships among some of these models.

Researchers have also proposed architectural mechanisms for implementing SC [34] efficiently [23, 44, 28, 25, 19, 54, 14, 50, 35, 27]. Several of these architectural mechanisms are interesting in their own right and applicable to reducing power consumption, however, so far commercial processor vendors have shown little interest in adopting stricter memory models.

Recently, there is a splurge of activity in trying to specify semantics of concurrent languages: C/C++ [51, 15, 13, 12, 32], Java [39, 18, 38]. These models are specified axiomatically, and allow load-store reordering. For C++, there has been work to specify an equivalent operational model [42].

4 General Atomic Memory Model (GAM)

In this section, we introduce GAM, an atomic memory model framework parametrized by how the memory model enforces following two types of orderings:

1. Memory instruction ordering: the ordering between two memory instructions, i.e., the commonly referred load-load, load-store, store-store and store-load orderings.
2. Fence ordering, the ordering between a fence and a memory instruction or between two fences.

We refer to the combination of the above two orderings as *memory/fence ordering*. GAM uses a function $\mathsf{ordered}(I_{old}, I_{new})$ to represent memory/fence ordering, and this function is used in both the axiomatic and operational definitions of GAM. $\mathsf{ordered}(I_{old}, I_{new})$ returns true when the older instruction I_{old} should be ordered before the younger instruction I_{new} according to the memory instruction ordering or fence ordering enforced by the memory model. For example, Table 1 shows the $\mathsf{ordered}(I_{old}, I_{new})$ table for TSO, which has only one type of fence. The only memory ordering that is not enforced by TSO is the store-load ordering, as represented by the false entry (St, Ld). As a more complex example, Table 2 shows the $\mathsf{ordered}(I_{old}, I_{new})$ table for RMO. In RMO, all four memory instruction orderings are relaxed, as shown by the false entries (Ld, Ld), (Ld, St), (St, Ld) and (St, St). The four fences are used to enforce each type of orderings respectively. For example, the true entries (FenceLS, St) and (Ld, FenceLS) means that FenceLS is ordered before younger stores and is ordered after older loads, thus enforce load-to-store ordering. The fences are even unordered with respect to each other. As a framework, given an ordered function (such as Table 1 or 2), GAM can produce equivalent axiomatic and operational models that enforce the memory/fence orderings represented by the ordered function.

I_{old} \ I_{new}	Ld	St	Fence
Ld	True	True	True
St	False	True	True
Fence	True	True	True

Table 1: Orderings for TSO memory instructions and fences: $\mathsf{ordered}_{TSO}(I_{old}, I_{new})$

I_{old} \ I_{new}	Ld	St	FenceLL	FenceLS	FenceSL	FenceSS
Ld	False	False	True	True	False	False
St	False	False	False	False	True	True
FenceLL	True	False	False	False	False	False
FenceLS	False	True	False	False	False	False
FenceSL	True	False	False	False	False	False
FenceSS	False	True	False	False	False	False

Table 2: Orderings for RMO memory instructions and fences: $\mathsf{ordered}_{RMO}(I_{old}, I_{new})$

It should be noted that the memory/fence ordering cannot fully describe a memory model. The following three aspects are not captured by the ordered function:

1. Load value: a memory model must specify which store values a load may read.
2. Dependency ordering: most memory models order two instructions if the younger instruction is dependent on the older instruction in certain ways.
3. Same-address ordering: even when the memory/fence ordering does not apply to two memory instructions for the same address, a memory model may still order them for the correctness of single-threaded programs.

The GAM definition given in this section is not parametrized in terms of the above three aspects. Later in Section 6, we will show how to tweak the definition of GAM to derive memory models with a different

dependency ordering or a different same-address ordering. The way to determine load values should be common across all multicopy atomic memory models, so we do not bother changing that. In the following, we give axiomatic and operation definitions of GAM and the equivalence proof. When we use examples to explain our definitions, we assume the ordered function in Table 2, i.e., with all four memory instruction reorderings and relaxed fences.

We present the axiomatic definition before the operational definition but these definitions can be read in any order.

4.1 Axiomatic Definition of GAM

The axiomatic definition of GAM takes three relations as input: program order ($<_{po}$), read-from relations ($\rightarrow_{rf}$) and memory order ($<_{mo}$). The program order ($<_{po}$) is a per-processor total order and represents the order in which the instructions are committed in that processor. A read-from edge specifies that a load reads from a particular store; $\rightarrow_{rf}$ points from a store instruction to a load instruction for the same address, with the load getting the same value that is written by the store. The memory order ($<_{mo}$) is a total order of all memory instructions in all processors. Intuitively, $<_{mo}$ specifies the order of when each memory accesses are performed globally.

The axiomatic model checks $<_{po}$, $\rightarrow_{rf}$ and $<_{mo}$ against a set of axioms. If all the axioms are satisfied, then the program behavior given by $<_{po}$ is allowed by the memory model.

It should be noted that $<_{po}$ is the observable program behavior, while $\rightarrow_{rf}$ and $<_{po}$ are just a *witness* which cannot be observed directly. To justify that a program behavior is allowed by GAM, we only need to find one witness (i.e., $\langle \rightarrow_{rf}, <_{mo} \rangle$) that satisfies all the axioms. To prove that a program behavior is disallowed by GAM, we must show that there is no witness that can satisfy all the axioms simultaneously.

In order to describe the axioms, we first define *preserved program order* ($<_{ppo}$), which is computed from $<_{po}$. $<_{ppo}$ captures the constraints on the out-of-order (OOO) execution of instructions in each processor (locally). Thus, a property of $<_{ppo}$ is that if $I_1 <_{ppo} I_2$ then $I_1 <_{po} I_2$ where I_1 and I_2 are instructions.

As will become clear $<_{po}$ by itself cannot reflect the constraints on the memory system and the interaction between processors. These constraints are expressed by the separate memory axioms of GAM. In the following, we first define how to compute $<_{ppo}$ from $<_{po}$, and then give the memory axioms of GAM.

4.1.1 Definition of Preserved Program Order $<_{ppo}$ for GAM

We define $<_{ppo}$ in three parts. The first part is the *preserved memory/fence order* ($<_{ppomf}$) which is captured by the ordered function. The second part is the *preserved dependency order* ($<_{ppod}$), which includes branch dependencies, address dependencies, data dependencies, etc. The last part is the *preserved same-address order* ($<_{pposa}$), i.e., the ordering of memory instructions for the same address. Finally $<_{ppo}$ is defined as the transitive closure of $<_{ppomf}$, $<_{ppod}$ and $<_{pposa}$.

Definition of preserved memory/fence order $<_{ppomf}$ for GAM: The preserved memory/fence order is fully described by the ordered function.

Definition 1 (Preserved memory/fence order $<_{ppomf}$)**.** *$I_1 <_{ppof} I_2$ iff I_1 and I_2 both are memory or fence instructions, and $I_1 <_{po} I_2$, and* ordered(I_1, I_2) *is true.*

Definition of preserved dependency order $<_{ppod}$ for GAM: We first give some basic definitions that are used to define dependency orderings precisely (all definitions ignore the PC register and the zero register):

Definition 2 (RS: Read Set)**.** *$RS(I)$ is the set of registers an instruction I reads.*

Definition 3 (WS: Write Set)**.** *$WS(I)$ is the set of registers an instruction I can write.*

Definition 4 (ARS: Address Read Set)**.** *$ARS(I)$ is the set of registers a memory instruction I reads to compute the address of the memory operation.*

Definition 5 (data-dependency $<_{ddep}$)**.** *$I_1 <_{ddep} I_2$ if $I_1 <_{po} I_2$ and $WS(I_1) \cap RS(I_2) \neq \emptyset$ and there exists a register r in $WS(I_1) \cap RS(I_2)$ such that there is no instruction I such that $I_1 <_{po} I <_{po} I_2$ and $r \in WS(I)$.*

Definition 6 (addr-dependency $<_{adep}$). *$I_1 <_{adep} I_2$ if $I_1 <_{po} I_2$ and $WS(I_1) \cap ARS(I_2) \neq \emptyset$ and there exists a register r in $WS(I_1) \cap ARS(I_2)$ such that there is no instruction I such that $I_1 <_{po} I <_{po} I_2$ and $r \in WS(I)$.*

Note that data-dependency includes addr-dependency, i.e., $I_1 <_{adep} I_2 \implies I_1 <_{ddep} I_2$.

Now we define $<_{ppod}$, which essentially says that the data-dependencies must be observed, stores should not execute until the preceding branches have been resolved, the execution of stores should be constrained by instructions on which prior memory instructions are address dependent, and in the case of a load following a store to the same address, the execution of the load should be constrained by instructions which produce the store's data.

Definition 7 (Preserved dependency order $<_{ppod}$). *$I_1 <_{ppod} I_2$ if either*

1. *$I_1 <_{ddep} I_2$, or*
2. *$I_1 <_{po} I_2$, and I_1 is a branch, and I_2 is a store, or*
3. *I_2 is a store instruction, and there exists a memory instruction I such that $I_1 <_{adep} I <_{po} I_2$, or*
4. *I_2 is a load instruction, and there exists a store S to the same address such that $I_1 <_{ddep} S <_{po} I_2$, and there is no other* store *for the same address between S and I_2.*

In the above definition, cases 1 and 2 are straightforward; we discuss the rest of the cases below.

Case 3 is about a subtle dependency caused by an address dependency and is illustrated by the example in Figure 1. If I_3 (store) is allowed to be issued before I_1, then the earlier load (I_2) may end up reading its own future store in case I_1 returns value $r_1 = b$.

$I_1 : r_1 = \text{Ld } a$
$I_2 : r_2 = \text{Ld } r_1$
$I_3 : \text{St } b = 1$

Figure 1: Example for case 3

Case 4 is about another subtle dependency when data is transfered not by registers but by local bypassing. In Figure 2, I_3 must be issued after I_1. Otherwise, in case I_3 is issued before I_1, I_3 must bypass from I_2. However, the data of I_2 is still unknown at that time.

$I_1 : r_1 = \text{Ld } a$
$I_2 : \text{St } b = r_1$
$I_3 : r_2 = \text{Ld } b$

Figure 2: Example for case 4

Definition of preserved same-address order $<_{pposa}$ for GAM: Next we give the definition of $<_{pposa}$, which captures the orderings between memory instructions for the same address.

Definition 8 (Preserved same-address order $<_{pposa}$). *$I_1 <_{pposa} I_2$ if either*

1. *$I_1 <_{po} I_2$, and I_1 is a load and I_2 is a store to the same address, or*
2. *$I_1 <_{po} I_2$, and both I_1 and I_2 are store instructions for the same address, or*
3. *$I_1 <_{po} I_2$ and both I_1 and I_2 are load instructions for the same address with no intervening* store *to the same address.*

The above definition explicitly excludes the enforcement of ordering of a store followed by a load to the same address. Otherwise our model would be stricter than TSO in some cases. Case 3 requires that loads for the same address without store to the same address in between to be issued in order. For example, all instructions in Figure 3 must be issued in order.

It should be noted that the choice to enforce this same-address load-load ordering in GAM is kind of arbitrary, because we do not see any decisive argument to support either enforcing or relaxing this ordering.

$$\boxed{\begin{array}{l} I_1 : r_1 = \text{Ld } a \\ I_2 : r_2 = \text{Ld } (b + r_1 - r_1) \\ I_3 : r_3 = \text{Ld } b \\ I_4 : r_4 = \text{Ld } (c + r_3 - r_3) \end{array}}$$

Figure 3: Example for case 3

On the one hand, implementations that execute loads for the the same address out of order will not violate single-thread correctness, and do not need the extra hardware to enforce this load-load ordering. On the other hand, programmers may expect memory models to have the per-location SC property [17], i.e., all memory accesses for a single address appear to be sequentially consistent, and enforcing this same-address load-load ordering is an easy way to provide the per-location SC property. The Alpha memory model [2] is the same as GAM in enforcing this ordering, while the RMO memory model [53] chooses to relax this ordering completely. In Section 6.3 programmers would like memory models to have the per-location SC property [17]. It should be noted that ARMv8.2 makes yet another choice in same-address load-load ordering which we will explain in Section 6.5.

Finally, we define $<_{ppo}$ as the transitive closure of $<_{ppod}$, $<_{pposa}$ and $<_{ppomf}$.

Definition 9 (Preserved program order $<_{ppo}$)**.** *$I_1 <_{ppo} I_2$ if either*

1. $I_1 <_{ppomf} I_2$, or
2. $I_1 <_{ppod} I_2$, or
3. $I_1 <_{pposa} I_2$, or
4. there exists an instruction I such that $I_1 <_{ppo} I$ and $I <_{ppo} I_2$.

4.1.2 Memory Axioms of GAM

GAM has the following two axioms (the notation $\max_{mo}$ means to find the youngest instruction in $<_{mo}$):

- **Axiom Inst-Order:** If $I_1 <_{ppo} I_2$, then $I_1 <_{mo} I_2$.
- **Axiom Load-Value:**

$$\mathsf{St}\ a\ v \rightarrow_{rf} \mathsf{Ld}\ a \Rightarrow \mathsf{St}\ a\ v = \max_{mo}\{\mathsf{St}\ a\ v' \mid \mathsf{St}\ a\ v' <_{po} \mathsf{Ld}\ a\ \vee\ \mathsf{St}\ a\ v' <_{mo} \mathsf{Ld}\ a\}$$

The first axiom says that $<_{mo}$ must respect $<_{ppo}$. An interpretation of this axiom is that the local ordering constraints on executing two memory instructions in the processor must be preserved when these two memory accesses are performed globally. The second axiom specifies the store that a load should read given $<_{mo}$ and $<_{po}$. Intuitively, each store overshadows previous stores to the same address and thus, a load should not be able to read overshadowed values. The only complication is because of bypassing: a load may read one of its own store values before it is advertised, which means a later load in other processors may still read the globally advertised store value in the memory. More precisely, the set of stores that are visible to a load consists of stores that either precede the load in $<_{po}$ or perform globally before the load does (i.e., precede the load in $<_{mo}$). The store read by the load must be visible to the load, and cannot be overshadowed (in $<_{mo}$) by another store which is also visible to the load.

4.2 An Operational Definition of GAM

The operational model of GAM consists of n processors $P_1 \ldots P_n$ and a monolithic memory m. Each processor P_i consists of an ROB and a PC register. The PC register contains the address of the next instruction to be fetched into ROB. When an instruction is fetched, if the instruction is a branch, we predict the branch target address and update the PC register speculatively; otherwise we simply increment the PC register. Each instruction in the ROB has a *done* bit. (We refer to an instruction as done if the done bit is true, and as not done otherwise.) Though instructions that have been marked as done can be removed from the ROB, we will not bother with this detail.

At each step of the execution one of the instructions marked as not-done in the ROB of a processor P_i is selected and executed and (sometimes) marked as done. There is often a *guard* condition associated with the execution of an instruction, and an instruction can be executed only if the guard is true. As will become clear soon that sometimes the execution of an instruction cannot proceed even when its guard is true.

Our axiomatic model, permits very aggressive execution of load instructions but it also requires that consecutive loads to the same address be done in order. If the operational model executed load instructions only when the address for its preceding memory instructions were known, then we will not be able to capture all the behaviors allowed by the axiomatic model. Thus, in the operational model, we let a load execute even before all the addresses of preceding memory instructions are known, and then later kill a done load if an older memory instruction happens to get the same address. The kill of a load instruction means that all the instruction younger than the killed load, including that load itself, are discarded from the ROB, and the PC register is updated to make instruction fetch begin by refetching the killed load instruction.

In order to implement these speculative loads, we need an additional *address-available* state bit in the ROB for each memory instruction. This bit indicates when the address calculation has been completed. Initially this bit is not set.

We need to know if the source operands of an instruction are available in order to execute the instruction. If the operand is specified as a source register r, then its availability is determined by searching the ROB from the current instruction slot towards older instructions until the first slot containing r as the destination register. (The search always terminates because we assume that the ROB has been initialized with instructions that set initial register values). If the slot containing the destination register is marked as done then the operand is assumed to be available, otherwise not.

This operational model is also parametrized by the memory/fence ordering. That is, it uses $\mathsf{ordered}$ function to control when a memory or fence instruction can be marked as done.

In the following we specify how to execute an instruction in the ROB according to the preserved program order definition given in Section 4.1.1. Each operational rule has a guard and a specified action.

- **Rule Fetch:**
  Guard: True.
  Action: Fetch a new instruction from the address stored in the PC register. Add the new instruction into the tail of ROB. If the new instruction is a branch, we predict the branch target address of the branch, update PC to be the predicted address, and record the predicted address in the ROB entry of the branch; otherwise we increment PC.
- **Rule Execute-Reg-to-Reg:** Execute a reg-to-reg instruction I.
  Guard: I is marked not-done and all source operands of I are ready.
  Action: Do the computation, record the result in the ROB slot, and mark I as done.
- **Rule Execute-Branch:** Execute a branch instruction I.
  Guard: I is marked not-done and all source operands of I are ready.
  Action: Compute the branch target address and mark I as done. If the computed target address is different from the previously predicted address (which is recorded in the ROB entry), then we kill all instructions which are younger than I in the ROB (excluding I). That is, we remove those instructions from the ROB, and update the PC register to the computed branch target address.
- **Rule Execute-Fence:** Execute a fence instruction I.
  Guard: I is marked not-done, and for each older (memory or fence) instruction I' such that $\mathsf{ordered}(I', I)$ is true, I' is done.
  Action: Mark I as done.
- **Rule Execute-Load:** Execute a load instruction I for address a.
  Guard: I is marked not-done, and the address-available bit is set to available, and for each older (memory or fence) instruction I' such that $\mathsf{ordered}(I', I)$ is true, I' is done.
  Action: Search the ROB from I towards the oldest instruction for the first not-done memory instruction with address a:
  1. If a not-done load to a is found then instruction I cannot be executed, i.e., we do nothing.
  2. If a not-done store to a is found then if the data for the store is ready, then execute I by bypassing the

data from the store, and mark I as done; otherwise, I cannot be executed.
3. If nothing is found then execute I by reading $m[a]$, and mark I as done.

- **Rule Compute-Store-Data:** compute the data of a store instruction I.
  Guard: the source registers for the data computation are ready.
  Action: Compute the data of I and record it in the ROB slot.
- **Rule Execute-Store:** Execute a store I for address a.
  Guard: I is marked not-done and in addition all the following conditions must be true:
  1. The address-available flag for I is set,
  2. The data of I is ready,
  3. For each older (memory or fence) instruction I' such that $\mathsf{ordered}(I', I)$ is true, I' is done,
  4. All older branch instructions are done,
  5. All older loads and stores have their address-available flags set,
  6. All older loads and stores for address a are done.

  Action: Update $m[a]$ and mark I as done.
- **Rule Compute-Mem-Addr:** Compute the address of a load or store instruction I.
  Guard: The address-available bit is not set and the address operand is ready with value a
  Action: We first set the address-available bit and record the address a into the ROB entry of I. Then we search the ROB from I towards the youngest instruction (excluding I) for the first memory instruction with address a. If the instruction found is a done load, then we kill that load and all instructions that are younger than the load in the ROB. That is, we remove the load and all younger instructions from the ROB, and set the PC register to the instruction-fetch address of the load. Otherwise no instruction needs to be killed.

4.3 Soundness: GAM Operational model $\subseteq$ GAM Axiomatic Model

The goal is to show that for any execution of the operational model, we can construct $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ which satisfies the GAM axioms and has the same program behavior as the operational execution. To do this, we need to introduce some ghost states to the operational model, and show invariants that hold after every step in the operational model.

In the operational model, we assume there is a (ghost) global time which is incremented whenever a rule fires. We also assume each instruction I in an ROB has the following ghost states which are accessed only in the proofs (all states start as $\top$):

- $I.\mathsf{doneTS}$: Records the current global time when a rule R fires and marks I as done.
- $I.\mathsf{addrTS}$: Records the current global time for memory instruction I when a Compute-Mem-Addr rule R fires to compute the address of I.
- $I.\mathsf{sdataTS}$: Records the current global time for a store instruction I, when a Compute-Store-Data rule R fires to compute the store data of I.
- $I.\mathsf{from}$: Records for load I either the not-done store it bypasses from, or the store with the maximum doneTS among all done stores for a.

In the final proof, we will use the states at the end of the operational execution to construct the axiomatic edges. $<_{po}$ will be constructed by the order of instructions in ROB, $\rightarrow_{rf}$ will be constructed by the from states of loads, and $<_{mo}$ will be constructed by the order of doneTS timestamps of all memory instructions.

For convenience, we use $I.\mathsf{ldval}$ to denote the load value if I is a load, use $I.\mathsf{addr}$ to denote the memory access address if I is a memory instruction, and use $I.\mathsf{sdata}$ to denote the store data if I is a store. These fields are $\top$ if the corresponding values are not available.

Given the model state at any time in the execution of the operational model, we can define the program order $<_{po\text{-}rob}$, data-dependency order $<_{ddep\text{-}rob}$, address-dependency order $<_{adep\text{-}rob}$, and a new relation $<_{ntppo\text{-}rob}$ which is similar to the preserved program order. (we add suffix *rob* to distinguish from the definitions in the axiomatic model):

- $<_{po\text{-}rob}$: Instructions $I_1 <_{po\text{-}rob} I_2$ iff both I_1 and I_2 are in the same ROB and I_1 is older than I_2 in the ROB.

- $<_{ddep\text{-}rob}$: $I_1 <_{ddep\text{-}rob} I_2$ iff $I_1 <_{po\text{-}rob} I_2$ and I_2 needs the result of I_1 as a source operand.
- $<_{adep\text{-}rob}$: $I_1 <_{adep\text{-}rob} I_2$ iff $I_1 <_{po\text{-}rob} I_2$, and I_2 is a memory instruction, and I_2 needs the result of I_1 as a source operand to compute the memory address to access.
- $<_{ntppo\text{-}rob}$: $I_1 <_{ntppo\text{-}rob} I_2$ iff $I_1 <_{po\text{-}rob} I_2$ and at least one of the following conditions hold:
  1. $I_1 <_{ddep\text{-}rob} I_2$.
  2. I_1 is a branch, and I_2 is a store.
  3. I_2 is a store, and there exists a memory instruction I such that $I_1 <_{adep\text{-}rob} I <_{po\text{-}rob} I_2$.
  4. I_2 is a load with $I_2.\mathsf{addr} = a \neq \top$, and there exists a store S with $S.\mathsf{addr} = a$, and $I_1 <_{ddep\text{-}rob} S <_{po\text{-}rob} I_2$, and there is no store S' such that $S'.\mathsf{addr} = a$ and $S <_{po\text{-}rob} S' <_{po\text{-}rob} I_2$.
  5. I_1 is a load with $I_1.\mathsf{addr} = a \neq \top$, and I_2 is a store with $I_2.\mathsf{addr} = a$.
  6. Both I_1 and I_2 are stores with $I_1.\mathsf{addr} = I_2.\mathsf{addr} = a \neq \top$.
  7. Both I_1 and I_2 are loads with $I_1.\mathsf{addr} = I_2.\mathsf{addr} = a \neq \top$, and there is no store S such that $S.\mathsf{addr} = a$ and $I_1 <_{po\text{-}rob} S <_{po\text{-}rob} I_2$.
  8. $\mathsf{ordered}(I_1, I_2)$ is true.

It should be noted that the way to compute $<_{ntppo\text{-}rob}$ from $<_{po\text{-}rob}$ is almost the same as the way to compute $<_{ppo}$ from $<_{po}$ except for two differences. The first difference is that $<_{ntppo\text{-}rob}$ is not made transitively closed; this is for simplifying the proof to some degree. The second difference is that in case the definition needs the address of memory instructions, $<_{ntppo\text{-}rob}$ ignores memory instructions which have not computed their addresses. Since the address of every memory instruction will be computed at the end of the operational execution, the second difference will diminish by that time. Since $<_{po}$ is defined by the $<_{po\text{-}rob}$ at the end of the operational execution, $<_{ppo}$ will be the transitive closure of $<_{ntppo\text{-}rob}$ at the end of the operational execution.

With the above definitions, we give the invariants of any operation execution in Lemma 1. Invariant 2 is a similar statement to the Inst-Order axiom, and will become exactly the same as that axiom at the end of the operational execution. Invariants 2 and 3 captures the ordering effects of dependencies carried to the computation of memory address and store data. Invariant 4 captures guard 5 of the Execute-Store rule, and is also related to case 3 of Definition 7 for $<_{ppo}$. Invariant 5 is an important property saying that stores are never written to the shared memory speculatively, so the model does not need any system-wide rollback. Invariant 6 constrains the current monolithic memory value. Invariant 7 constrains the store read by a load, and in particular, invariant 7d will become the Load-Value axiom at the end of the operation execution. The detailed proof can be found in Appendix A.

Lemma 1. *The following invariants hold during the execution of the operational model:*

1. *If $I_1 <_{ntppo\text{-}rob} I_2$ and $I_2.\mathsf{doneTS} \neq \top$, then $I_1.\mathsf{doneTS} \neq \top$ and $I_1.\mathsf{doneTS} < I_2.\mathsf{doneTS}$.*
2. *If $I_1 <_{adep\text{-}rob} I_2$ and $I_2.\mathsf{addrTS} \neq \top$, then $I_1.\mathsf{doneTS} \neq \top$ and $I_1.\mathsf{doneTS} < I_2.\mathsf{addrTS}$.*
3. *If $I_1 <_{ddep\text{-}rob} I_2$, and not $I_1 <_{adep\text{-}rob} I_2$, and I_2 is a store, and $I_2.\mathsf{sdataTS} \neq \top$, then $I_1.\mathsf{doneTS} \neq \top$ and $I_1.\mathsf{doneTS} < I_2.\mathsf{sdataTS}$.*
4. *If $I_1 <_{po\text{-}rob} I_2$, and I_1 is a memory instruction, and I_2 is a store, and $I_2.\mathsf{doneTS} \neq \top$, then $I_1.\mathsf{addrTS} \neq \top$ and $I_1.\mathsf{addrTS} < I_2.\mathsf{doneTS}$.*
5. *We never kill a done store.*
6. *For any address a, let S be the store with the maximum* doneTS *among all the done stores for address a. The monolithic memory value for a is equal to $S.\mathsf{sdata}$.*
7. *For any done load L, let $S = L.\mathsf{from}$ (i.e., S is the store read by L). All of the following properties are satisfied:*
   (a) *S still exists in an ROB (i.e., S is not killed).*
   (b) *$S.\mathsf{addr} = L.\mathsf{addr}$ and $S.\mathsf{sdata} = L.\mathsf{ldval}$.*
   (c) *If S is done, then there is no not-done store S' such that $S'.addr = a$ and $S' <_{po\text{-}rob} L$.*
   (d) *If S is done, then for any other done store S' with $S'.\mathsf{addr} = L.\mathsf{addr}$, if $S' <_{po\text{-}rob} L$ or $S'.\mathsf{doneTS} < L.\mathsf{doneTS}$, then $S'.\mathsf{doneTS} < S.\mathsf{doneTS}$.*
   (e) *If S is not done, then $S <_{po\text{-}rob} L$, and there is no store S' such that $S'.\mathsf{addr} = L.\mathsf{addr}$ and $S <_{po\text{-}rob} S' <_{po\text{-}rob} L$.*

With the above invariants, we can finally prove the following soundness theorem.

Theorem 1. *GAM operational model $\subseteq$ GAM axiomatic model.*

Proof. For any execution of the operational model, at the end of the execution, all instructions must be done. We construct $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ using the ending state of the operational execution as follows:

- $<_{po}$ is constructed as the order of instructions in each ROB.
- $<_{mo}$ is constructed by the ordering of doneTS, i.e., for two memory instructions I_1 and I_2, $I_1 <_{mo} I_2$ iff I_1.doneTS $< I_2$.doneTS.
- $\rightarrow_{rf}$ is constructed by the from fields, i.e., for a load L and a store S, $S \rightarrow_{rf} L$ iff $S = L$.from.

Invariant 7b ensures that the constructed $\rightarrow_{rf}$ and $<_{po}$ are consistent with each other (e.g., it rules out the case that $\rightarrow_{rf}$ says a load should read a store with value 1, but $<_{po}$ says the load has value 2).

Since all instructions are done at the end of execution, then invariant 7d becomes the Load-Value axiom. Therefore, the constructed $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ satisfy the Load-Value axiom.

At the end of execution, invariant 1 becomes: if $I_1 <_{ntppo\text{-}rob} I_2$, then I_1.doneTS $< I_2$.doneTS. Note that the $<_{ppo}$ computed from $<_{po}$ is actually the transitive closure of $<_{ntppo\text{-}rob}$. Since instructions are totally ordered by doneTS fields, we have if $I_1 <_{ppo} I_2$, then I_1.doneTS $< I_2$.doneTS. Since $<_{mo}$ is defined by the order of doneTS fields, the Inst-Order axiom is also satisfied. □

4.4 Completeness: GAM Axiomatic model $\subseteq$ GAM Operational Model

Theorem 2. *GAM axiomatic model $\subseteq$ GAM operational model.*

Proof. The goal is that for any legal axiomatic relations $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ (which satisfy the GAM axioms), we can run the operational model to give the same program behavior. The strategy to run the operational model consists of two major phases. In the first phase, we only fire Fetch rules to fetch all instructions into all ROBs according to $<_{po}$. During the second phase, in each step we fire a rule that either marks an instruction as done or computes the address or data of a memory instruction. Which rule to fire in a step depends on the current state of the operational model and $<_{mo}$. Here we give the detailed algorithm that determines which rule to fire in each step:

1. If in the operational model there is a not-done reg-to-reg or branch instruction whose source registers are all ready, then we fire an Execute-Reg-to-Reg or Execute-Branch rule to execute that instruction.
2. If the above case does not apply, and in the operational model there is a memory instruction, whose address is not computed but the source registers for the address computation are all ready, then we fire a Compute-Mem-Addr rule to compute the address of that instruction.
3. If neither of the above cases applies, and in the operational model there is a store instruction, whose store data is not computed but the source registers for the data computation are all ready, then we fire a Compute-Store-Data rule to compute the store data of that instruction.
4. If none of the above cases applies, and in the operational model there is a fence instruction and the guard of the Execute-Fence rule for this fence is ready, then we fire the Execute-Fence rule to execute that fence.
5. If none of the above cases applies, then we find the oldest instruction in $<_{mo}$, which is not-done in the operational model, and we fire an Execute-Load or Execute-Store rule to execute that instruction.

Before giving the invariants, we give a definition related to the ordering of stores for the same address. For each address a, all stores for a are totally ordered by $<_{mo}$, and we refer to this total order of stores for a as $<^{a}_{co}$.

Now we show the invariants. After each step, we maintain the following invariants:

1. The order of instructions in each ROB in the operational model is the same as the $<_{po}$ of that processor in the axiomatic relations.
2. The results of all the instructions that have been marked as done so far in the operational model are the same as those in the axiomatic relations.
3. All the load/store addresses that have been computed so far in the operational model are the same as those in the axiomatic relations.

4. All the store data that have been computed so far in the operational model are the same as those in the axiomatic relations.
5. No kill has ever happened in the operational model.
6. For the rule fired in each step that we have performed so far, the guard of the rule is satisfied the at that step (i.e., the rule can fire).
7. In each step that we have performed so far, if we fire a rule to execute an instruction (especially a load) in that step, the instruction must be marked as done by the rule.
8. For each address a, the order of all the store updates on monolithic memory address a that have happened so far in the operational model is a prefix of $<_{co}^{a}$.

The detailed proof of the invariants can be found in Appendix B. □

5 COM: an Alternative Axiomatic Model

In this section, we present an alternative (but still parameterized) axiomatic formulation that is perhaps less intuitive, but nevertheless in common use due to its computational efficiency. We call this formulation the *COM* model (where "COM" stands for communication, as described below). We first present a proof of equivalence between the GAM axioms and the COM axioms. This in turn implies that COM is also equivalent to the operational definition of GAM. We then implement both axiomatic models in Alloy [31] in order to perform sanity checking and empirical testing of the models and of the proofs.

5.1 The COM Axioms

The COM model is defined in terms of three basic relations and three derived relations, plus $<_{ppo}$:

- Basic relations:
  - Program order ($<_{po}$), as before
  - Reads-from ($\rightarrow_{rf}$), as before
  - Coherence ($<_{co}$), a total order over the writes to each memory address
- Derived relations:
  - Reads-from external ($\rightarrow_{rfe}$), which is the subset of $\rightarrow_{rf}$ for which both the read and the write are in different threads
  - From-reads ($\rightarrow_{fr} = \rightarrow_{rf^{-1}}; <_{co}$), which relates each read r to every write which follows the $\rightarrow_{rf}$-source of r in $<_{co}$. ($\rightarrow_{rf^{-1}}$ indicates the inverse of $\rightarrow_{rf}$)
  - Program order, same location ($<_{poloc}$), which is the subset of program order that relates memory accesses to the same memory address

Another derived relation $<_{com} = \rightarrow_{rf} \cup <_{co} \cup \rightarrow_{fr}$ is often defined as a convenient shorthand in this style of model (hence our choice of the name "COM"), but we do not use it in this paper.
In the COM formulation, an execution is legal if it satisfies the following two axioms:

- **Axiom SC-per-Location:** acyclic($\rightarrow_{rf} \cup <_{co} \cup \rightarrow_{fr} \cup <_{poloc}$)
- **Axiom Causality:** acyclic($\rightarrow_{rfe} \cup <_{co} \cup \rightarrow_{fr} \cup <_{ppo}$)

5.2 Equivalence of GAM and COM

The complete proofs are provided in Appendix C. We provide an intuition here.

To prove that GAM $\subseteq$ COM, we must do two things: 1) find a suitable choice of $<_{co}$, which does not exist in the GAM model, and 2) prove that if the GAM axioms are satisfied, the COM axioms are satisfied. Of course, the natural choice for $<_{co}$ is to simply take the restriction of $<_{mo}$ that relates only stores to the same address, and that is indeed what we use. It remains to show that for any choice of $<_{mo}$ in the GAM axioms, the two COM axioms are satisfied.

We start with a lemma:

Lemma 2. *All of $\rightarrow_{rfe}$, $<_{co}$, $\rightarrow_{fr}$, and $<_{ppo}$ are contained in $<_{mo}$.*

Proof. Straightforward; see appendix. □

With this lemma, it is easy to show that the Causality axiom is satisfied:

Theorem 3. *The Causality axiom is satisfied.*

Proof. By Lemma 2, the union $\rightarrow_{rfe} \cup <_{co} \cup \rightarrow_{fr} \cup <_{ppo}$ is a subset of $<_{mo}$. Therefore, since $<_{mo}$ is acyclic, $\rightarrow_{rfe} \cup <_{co} \cup \rightarrow_{fr} \cup <_{ppo}$ must also be acyclic. □

The SC-per-Location axiom will take a bit more work to prove. To start, define $<_{eco}$ as the union of the following relations:

- $<_{co}$ (Write to Write)
- $\rightarrow_{fr}$ (Read to Write)
- $<_{co}{}^*; \rightarrow_{rf}$ (Write to Read)
- $\rightarrow_{rf^{-1}}; <_{co}{}^*; \rightarrow_{rf}$ (Read to Read)

Lemma 3. *For all pairs i_1, i_2 of memory accesses to the same address, either $i_1 <_{eco} i_2$ or $i_2 <_{eco} i_1$.*

Proof. By construction; see appendix. □

If i_1 and i_2 are related in program order, then the $<_{eco}$ direction must match:

Lemma 4. *If $i_1 <_{poloc} i_2$, then $i_1 <_{eco} i_2$.*

Proof. The alternative of $i_2 <_{eco} i_1$ results in a contradiction, except for one case where it overlaps $i_1 <_{eco} i_2$. See appendix. □

Theorem 4. *The SC-per-Location axiom is satisfied.*

Proof. (abbreviated; see appendix)

First, by Lemma 4, all $<_{poloc}$ edges involving at least one write can be converted into sequences containing only $\rightarrow_{rf}$, $<_{co}$, and $\rightarrow_{fr}$. So we consider only cycles with $\rightarrow_{rf}$, $<_{co}$, $\rightarrow_{fr}$, and read-to-read $<_{poloc}$ edges. Replace every instance of read-read $<_{poloc}$ in the cycle with $\rightarrow_{rf^{-1}}; <_{co}{}^*; \rightarrow_{rf}$ per Lemma 4. Now, because $<_{co}$ and $\rightarrow_{fr}$ both target writes, every appearance of $\rightarrow_{rf^{-1}}$ must be preceded either by $\rightarrow_{rf}$ or by $\rightarrow_{rf^{-1}}; <_{co}{}^*; \rightarrow_{rf}$. In particular, every appearance of $\rightarrow_{rf^{-1}}$ must be preceded directly by $\rightarrow_{rf}$. Since $\rightarrow_{rf}; \rightarrow_{rf^{-1}}$ is the identity function, all appearances of $\rightarrow_{rf^{-1}}$ in the cycle can be eliminated by simply removing each $\rightarrow_{rf}; \rightarrow_{rf^{-1}}$ pair in the cycle. This leaves a cycle with only $\rightarrow_{rf}$, $<_{co}$, and $\rightarrow_{fr}$, which is a contradiction. □

5.3 COM ⊆ GAM

This direction is easier. Given $<_{po}$, $\rightarrow_{rf}$, and $<_{co}$, we must find a suitable $<_{mo}$. By the Causality axiom, $\rightarrow_{rfe} \cup <_{co} \cup \rightarrow_{fr} \cup <_{ppo}$ is acyclic, and hence there is at least one total ordering compatible with it. We show that any such total ordering satisfies GAM. The Inst-Order axiom is true by construction, and hence we must only show that the Load-Value axiom is satisfied.

Theorem 5. *Any $<_{mo}$ which is a total ordering of $\rightarrow_{rfe} \cup <_{co} \cup \rightarrow_{fr} \cup <_{ppo}$ satisfies the Load-Value axiom.*

Proof. If $w \rightarrow_{rf} r$, then either $w \rightarrow_{rfi} r$ or $w \rightarrow_{rfe} r$. In the first case, $w <_{po} r$, or else it would contradict the SC-per-Location axiom. In the second case, $w <_{mo} r$ by construction of $<_{mo}$. In either case, w must be in the candidate set

$$\{\mathsf{St}\ a\ v' \mid \mathsf{St}\ a\ v' <_{po} \mathsf{Ld}\ a\ \vee\ \mathsf{St}\ a\ v' <_{mo} \mathsf{Ld}\ a\}.$$

It remains to be shown that w is in fact the $<_{mo}$-maximal element of that candidate set.

Suppose that w is not maximal. Then there is some other write w' to the same address a such that $w <_{mo} w'$ and either $w' <_{po} r$ or $w' <_{mo} r$. But then by definition, $r \rightarrow_{fr} w'$, and $\rightarrow_{fr}$ cannot contradict either $<_{po}$ (by SC-per-Location) or $<_{mo}$ (by construction of $<_{mo}$). Hence we have a contradiction. □

5.4 Empirical Validation

We also used model checking to confirm the validity of the proof of equivalence between GAM and COM. We encoded both models into Alloy [31, 55], a relational model finder backed by a SAT solver, and checked for any mismatches. The definition of this model is shown in Appendix D. In keeping with the spirit of the proofs, $<_{ppo}$ is entirely parameterized; there is no explicit notion of fence or dependency in this version of the model. We only assume that Definition 8 always holds. Under these conditions, Alloy verifies in roughly one hour that no counterexamples are found for tests with up to seven instructions.

6 Comparing GAM with Existing Atomic Memory Models

Now that we have defined our three model formulations and completed the proofs of equivalence, we can now show how GAM is related to existing atomic memory models. Most atomic memory models already have the same axioms as GAM, so our commparison will base off from the definitions of $<_{ppo}$. In some cases, the existing memory model can be instantiated from GAM. While in other cases, the dependency ordering or same-address load-load ordering of an existing model does not match that in GAM, and we will explain the difference and possible ways to tweak GAM to match the existing model.

6.1 SC

SC has no fence, the memory/fence ordering enforced by SC is shown in Table 3.

I_{old} \ I_{new}	Ld	St
Ld	True	True
St	True	True

Table 3: Memory/fence orderings for SC: $\mathsf{ordered}_{SC}(I_{old}, I_{new})$

After supplying $\mathsf{ordered}_{SC}$ to GAM, $<_{ppo}$ in the GAM axiomatic instance will order every pair of memory instructions from the same processor. In this case, the Load-Value axiom will reduce to

$$\mathsf{St}\ a\ v \rightarrow_{rf} \mathsf{Ld}\ a \Rightarrow \mathsf{St}\ a\ v = \max_{mo}\{\mathsf{St}\ a\ v' <_{mo} \mathsf{Ld}\ a\}$$

This is because $I_1 <_{po} I_2$ implies $I_1 <_{ppo} I_2 \Rightarrow I_1 <_{mo} I_2$. Thus, the GAM axiomatic instance is equivalent to SC. In the operational instance of GAM, the SC-ordered function makes the guards of Execute-Load and Execute-Store rules to wait for all previous memory instructions to be done. This is also the same as SC.

6.2 TSO

TSO has only one fence, and the memory/fence ordering enforced by TSO is shown in Table 1. The TSO axiomatic model defines a $<_{ppo}$ edge from instructions I_1 to I_2 iff $\mathsf{ordered}_{TSO}(I_1, I_2)$. When supplying GAM with the $\mathsf{ordered}_{TSO}$ function, the $<_{ppo}$ of the resulting GAM axiomatic instance is the same as that of TSO axiomatic model, since $<_{ppod}$ and $<_{pposa}$ are entirely contained within $\mathsf{ordered}_{TSO}$. In other words, all load-load, load-store, and store-store orderings are automatically enforced anyway, so there is no need to worry about any particular subset of such orderings. In the operational instance of GAM, the TSO-ordered function will cause the guard of the Execute-Load rule to wait for all older loads in ROB to be done, and cause the guard of the Execute-Store rule to wait for all older memory instructions to be done.

6.3 SPARC RMO

RMO has various fences, and the memory/fence orderings enforced by RMO are shown in Table 2. RMO also enforces the ordering between dependent instructions. However, there is a bug in the dependency definition in

RMO [56]. For the sake of comparison, we consider this to be a mistake rather than an intentional deviation, and hence we simply assume a corrected version of RMO that has the same definition of dependency ordering as GAM does.

When we supply the $\mathsf{ordered}_{RMO}$ function to GAM, the resulting GAM axiomatic instance is very close to but slightly different from the RMO axiomatic model. The difference is that RMO does not order loads for the same address. Same-address load-load reordering is a subtle issue (see Section 6.5) and a common source of implementation bugs [9], but by modern standards RMO's approach is considered overly aggressive. Nevertheless, for completeness we describe how GAM could be tweaked to allow same-address load-load reordering: we can simply tweak the axiomatic definition of GAM by removing case 3 from the definition of $<_{pposa}$ (Definition 8). After this removal, the GAM axiomatic instance becomes exactly the same as RMO.

The challenge is then to tweak the GAM operational instance to keep it equivalent to the axiomatic instance. In the GAM operational instance, we relax the Execute-Load rule by making the ROB search ignore loads for the same address. Also, in the Compute-Mem-Addr rule that computes the address of a *load*, the ROB search in the rule should ignore younger loads for the same address. These two changes relax the ordering between loads for the same address, making the operational instance of GAM still match the axiomatic instance of GAM.

6.4 WMM

WMM [56] has two fences: Commit and Reconcile, and the memory/fence orderings enforced by WMM are shown in Table 4. The ordering between Commit and Reconcile is particularly important in WMM, as preventing store-load reordering requires the combination of a Commit and a Reconcile.

I_{old} \ I_{new}	Ld	St	Commit	Reconcile
Ld	False	True	True	True
St	False	False	True	False
Commit	False	True	True	True
Reconcile	True	True	True	True

Table 4: Memory/fence orderings for WMM: $\mathsf{ordered}(I_{old}, I_{new})$

WMM does not enforce any dependency ordering (although all load-store ordering is automatically enforced). Therefore, in order to make GAM match WMM, we must first tweak the axiomatic definition of GAM by dropping $<_{ppod}$. After this change, one subtle difference still remains. Consider a scenario in which $L_1 <_{po} S <_{po} L_2$, where L_1 and L_2 are both loads for address a and S is store for a. GAM does not directly require L_1 and L_2 to be ordered in $<_{mo}$ due to the intervening store, but WMM does require $L1 <_{mo} L_2$. However, it turns out that the two are actually equivalent in this case, because we can transform the $<_{mo}$ in GAM to a legal $<_{mo}$ in WMM (i.e., one that obeys $<_{ppo}$ in WMM).

During the transformation, the store read by each load determined by the Load-Value axiom will not change. In each transformation step, for a processor i, we find one such potential counterexample scenario: a load L_2 which is the youngest load in the $<_{po}$ of processor i which is $<_{mo}$-before an older load L_1 from processor i for the same address. This gives us $L_1 <_{po} L_2$ and $L_2 <_{mo} L_1$. Since this reordering is allowed by GAM, there must be store for a between L_1 and L_2 in the $<_{po}$ of processor i. The transformation is to move L_2 to be right after L_1 in $<_{mo}$. This is legal because no WMM ordering primitive can cause an instruction to be ordered after L_2 without also being ordered after L_1. It also does not affect the value returned by L_1, nor does it affect any instruction originally older than L_2 in $<_{mo}$. By repeating the above steps, we can complete the transformation until no such apparent contradictions remain. Therefore, the axiomatic instance of GAM is equivalent to WMM.

We also need to tweak the GAM operational instance to keep it equivalent to the axiomatic instance. In the operational instance of GAM, if a Fetch rule fetches a load into the ROB, we predict the load value and record it in the new ROB entry. Younger instructions in ROB can read the predicted load value for

computation. In the Execute-Load rule, if the load is marked as done, then we compare the read value with the previously predicted value. In case they are not equal, we kill all instructions younger than the load in ROB. Introducing load-value prediction relaxes dependency ordering, making the operational instance of GAM still match the axiomatic instance of GAM.

6.5 ARM v8.2

As of March 2017, ARM completely revamped its memory consistency model. The end result looks very similar to GAM, with $<_{ppof}$ defined to include `DMB LD` (load-to-load/store), `DMB ST` (store-to-store), Load-Acquire (ordered with subsequent loads/stores), and Store-Release (ordered with prior loads/stores). There is, however, one main exception: ARM allows read-same-write (RSW) behavior (Figure 4): two loads which return the value written by the same write are *not* ordered in $<_{ppo}$. In particular, in Figure 4, the two loads of z are not ordered on ARM, even though they are two loads of the same address with no intervening store. If the two loads read from *different* stores (e.g., the RDW behavior in Figure 4), however, the outcome is forbidden.

St x, 1	Ld r1, y (=1)		St x, 1	Ld r1, y (=1)	
Fence	Ld r2, $z + r1 - r1$ (=0)		Fence	Ld r2, $z + r1 - r1$ (=1)	
St y, 1	Ld r3, z (=0)		St y, 1	Ld r3, z (=2)	
	Ld r4, $x + r3 - r3$ (=0)			Ld r4, $x + r3 - r3$ (=0)	

Figure 4: The read-same-writes (RSW, left) litmus test is forbidden under GAM but permitted by ARM. The read-different-writes (RDW, right) litmus test is forbidden under both ARM and GAM. Both tests are the same, but ARM makes a distinction based on the values returned by the loads.

We feel the subtlety in allowing the RSW behavior while forbidding the RDW behavior may lead to confusion. Besides, there is no published evidence showing that having this subtlety can lead to higher performance in implementations. Therefore, definition 8.3 of GAM was carefully chosen to forbid the RSW behavior, while still allowing so-called "fri-rfi" behavior (Figure 5) which can result from local store forwarding in implementations.

St x, 1	Ld r1, y (=1)
Fence	St y, 2
St y, 1	Ld r2, y (=2)
	Ld r3, $x + r2 - r2$ (=0)

Figure 5: The MP+fence+fri-rfi-addr litmus test.

6.6 Alpha

Alpha's memory model is similar to GAM with one single fence, but it is strictly weaker in that it does not enforce any dependencies, including even load-store dependencies. Alpha therefore allows the behavior in Figure 6, while GAM does not.

It is possible to remove all dependency orderings from the GAM axiomatic model in order to account for this behavior, but doing so in the operational model would be a substantial challenge (just as it would be in any real microarchitecture). It is not generally possible to perform speculative stores, as there is no way to undo a failed speculation, so the operational model that produces such behaviors would necessarily be somewhat contrived. In any case, such behaviors are no longer produced in more modern memory model definitions, and so we do not attempt to adapt the GAM operational model to account for speculative load-store dependency reordering.

Ld r1, x (=1) If r1 == 0 then St y, 1 else St y, 1	Ld r2, y (=1) St x, r2

Figure 6: Alpha is more relaxed to reorder stores before branches

6.7 RISC-V

The RISC-V model is not yet finalized, but it is likely to use a model very similar to GAM. For comparison, we include the basics of the expected model below. Note in particular that Release is not ordered with Acquire, in contrast to how WMM does order Commit with Reconcile.

I_{old} \ I_{new}	Ld	St	Release	Acquire	Full
Ld	False	False	True	True	True
St	False	False	True	False	True
Release	False	True	True	False	True
Acquire	True	True	True	True	True
Full	True	True	True	True	True

Figure 7: Memory/fence orderings for RISC-V: $\mathsf{ordered}_{RISC\text{-}V}(I_{old}, I_{new})$

This model presents all of the best features of GAM: a minimal set of dependency orderings that are nevertheless up to modern standards, a flexible and performant yet easy-to-define set of fences, and a reasonably-minimal set of same-address orderings. It can also be adapted to the needs of any subtle variant or modification by simply changing the set of fences that are included in Table 7. As such, if RISC-V adopts GAM, it will be the first modern architecture allowing load-store reordering to come complete with a proper axiomatic model, a proper operational model, and a full proof of equivalence.

7 GAM-I2E: Parameterizing Dependency Ordering

In previous sections, we have seen that GAM is not parameterized by dependency orderings, and requires manual tweak on the definitions to produce memory models with a different dependency ordering. The major reason is that GAM is designed to be able to allow load-store reordering. As stated in Section 2.2, allowing load-store reordering means a store may indirectly affect an older load in the same processor. This implies that no matter what mechanism an operational model uses, it cannot execute instructions in order. Thus, when an operational model wants to execute an instruction I, it may not have all the information (e.g., memory access addresses) of instructions that are older than I in the same processor. However, in the axiomatic model, such information is always available, and will used in the computation of $<_{ppod}$ edges that point to I. The lack of information in the operational model makes it difficult to parameterize dependency ordering while keeping the operational and axiomatic models equivalent.

Recently, Zhang et al. have shown that some memory models can be expressed in the form of instantaneous instruction execution (I2E) when load-store reordering is forbidden. I2E means that each processor in the operational model executes instructions instantaneously and in order. Here we apply that idea, i.e., we force $\mathsf{ordered}$(Ld, St) to be true (i.e., forbid load-store reordering) to make it possible to express the GAM operational model in I2E. In the I2E operational model, for the next instruction I to execute on a processor i, we know all the information of all the instructions older than I in processor i. Thus, the I2E operational model can compute the $<_{ppo}$ edges pointing to I using the same way as the axiomatic model does. Hence, the I2E operational model knows the same constraint of executing I (i.e., the constraint on placing I in the global order) as the axiomatic model does. I2E eliminates the difference in information available to

the axiomatic and operational models, making it possible to parameterize the memory model by any form of $<_{ppod}$. The model is not parametrized by same-address ordering because some same-address ordering are required by single-thread correctness. It should be noted that computing the $<_{ppo}$ edges pointing to instruction I should not require knowing the execution result of I. This is true for computing the $<_{ppomf}$ and $<_{pposa}$ edges defined in GAM. This should also be true for most definitions of preserved dependency ordering (i.e., $<_{ppod}$).

We refer to this new model as *GAM-I2E*. In the following, we give the axiomatic and operational definitions of GAM-I2E, which are parametrized by $<_{ppomf}$ and $<_{ppod}$, as well as the equivalence proof.

7.1 Axiomatic Model of GAM-I2E

The axiomatic model of GAM-I2E is exactly the same as that of GAM in Section 4.1. The only additional requirement is that ordered(Ld, St) must be true. Given this requirement, one can slightly simplify the definition of $<_{pposa}$ by removing case 1 from Definition 8, because that load-store ordering is already enforced by $<_{ppomf}$.

7.2 Operational Model of GAM-I2E

The operational model consists of n processors. Each processor executes instructions instantaneously, and contains a local buffer to temporarily keep executed stores and fences. The memory system is a list $<_{mo\text{-}i2e}$ of load and store instructions. (we use suffix $i2e$ to distinguish from the definitions in the axiomatic model). In the following, we will also use $<_{mo\text{-}i2e}$ as a total order of memory instructions in the memory system, i.e., $I_1 <_{mo\text{-}i2e} I_2$ means that instruction I_1 is closer to the list head than I_2.

Assume the next instruction to execute on a processor is I. Let $<_{po\text{-}i2e}$ be the execution order of I and all instructions already executed by the processor (i.e., I is the youngest). If we treat $<_{po\text{-}i2e}$ as a program order, then we can follow the definitions of preserved program order (Section 4.1.1) to compute $<_{ppo\text{-}i2e}$ from $<_{po\text{-}i2e}$. $<_{ppo\text{-}i2e}$ is the preserved program order among I and all instructions executed by the processor. Note that to make this definition meaningful, computing $<_{ppo\text{-}i2e}$ should not require knowing the load value of I. It should also be noted that $<_{ppo\text{-}i2e}$ edges grow monotonously. To be specific, consider the case that a processor has executed k instructions $I_1 <_{po\text{-}i2e} I_2 <_{po\text{-}i2e} \cdots <_{po\text{-}i2e} I_k$, and we have computed the $<_{ppo\text{-}i2e}$ edges for $I_1 \ldots I_k$. If the processor executes a new instruction I_{k+1}, then the $<_{ppo\text{-}i2e}$ edges for $I_1 \ldots I_{k+1}$ will contain all the previously computed $<_{ppo\text{-}i2e}$ edges for $I_1 \ldots I_k$, and all the newly added edges will point to I_{k+1}. This is because whether instructions I and I' are ordered by preserved program order is fully determined by I, I' and instructions between I and I' in the program order.

With the above definitions, now we give the rules for the operational moddel of GAM-I2E.

- **Rule Execute-Reg-Branch:** Execute a reg-to-reg or branch instruction I.
  Guard: True.
  Action: Execute I and update local register states.
- **Rule Execute-Store-Fence:** Execute a store or fence instruction I.
  Guard: True.
  Action: Insert I into the local buffer.
- **Rule Execute-Load:** Execute a load L for address a.
  Guard: There is no instruction I in the local buffer that is ordered before L in $<_{ppo\text{-}i2e}$.
  Action: Insert L into an arbitrary place in list $<_{mo\text{-}i2e}$ such that for any memory instruction I which is ordered before L in $<_{ppo\text{-}i2e}$, L is after I in $<_{mo\text{-}i2e}$. With the updated $<_{mo\text{-}i2e}$, we can determine the load value of L in the following way:
  1. If the local buffer contains any store for a, then L reads from the youngest (i.e., most recently inserted) store for a in the local buffer.
  2. Otherwise, L reads from the youngest store for a in $<_{mo\text{-}i2e}$ that is from the same processor of L or is older than L in $<_{mo\text{-}i2e}$.

- **Rule Dequeue-Store:** Dequeue a store S from the local buffer to the memory system.
  Guard: There is no instruction in the local buffer that is ordered before S in $<_{ppo\text{-}i2e}$.
  Action: Remove S from the local buffer, and append S to the end of $<_{mo\text{-}i2e}$ (i.e., S becomes the youngest in $<_{mo\text{-}i2e}$).
- **Rule Dequeue-Fence:** Dequeue a fence F from the local buffer.
  Guard: There is no instruction in the local buffer that is ordered before F in $<_{ppo\text{-}i2e}$.
  Action: Remove F from the local buffer.

7.3 Soundness: GAM-I2E Operational Model $\subseteq$ GAM-I2E Axiomatic Model

Theorem 6. *GAM-I2E operational model $\subseteq$ GAM-I2E axiomatic model*

Proof. The goal is to show that for any execution of the GAM-I2E operational model, we can construct $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ which satisfies the GAM-I2E axioms and has the same program behavior as the operational execution. $<_{po}$ is the order of executing instructions in each processor of the operational model. $\rightarrow_{rf}$ is constructed according to the Execute-Load rule, i.e., if the Execute-Load rule picks store S to satisfy a load L, then $S \rightarrow_{rf} L$. $<_{mo}$ is the $<_{mo\text{-}i2e}$ at the end of the operational execution. We need to show that $\langle <_{po}, \rightarrow_{rf}, <_{mo} \rangle$ satisfies the axioms. It should be noted that $<_{po\text{-}i2e}$ and $<_{ppo\text{-}i2e}$ always matche $<_{po}$ and $<_{ppo}$ respectively during the operational execution. That is, when an instruction I of processor i is executed in the operational execution, $<_{po\text{-}i2e}$ and $<_{ppo\text{-}i2e}$ of instructions executed by processor i (including I) satisfies the following invariants:

- $<_{po\text{-}i2e}$ is a prefix of $<_{po}$ (of processor i) up to I (including I).
- For any instructions $I_1 <_{ppo} I_2$ from processor i, if I_1 and I_2 are not ordered after I in $<_{po}$ (i.e., I_2 may be equal to I), then $I_1 <_{ppo\text{-}i2e} I_2$.
- For any instructions I_1 and I_2, if $I_1 <_{ppo\text{-}i2e} I_2$, then $I_1 <_{ppo} I_2$.

With above invariants, we prove that the Inst-Order axiom is satisfied by contradiction, i.e., we assume there are two memory instructions I_1 and I_2 from processor i such that $I_1 <_{ppo} I_2$ but $I_2 <_{mo} I_1$. In the operational model, when I_2 is executed, I_1 must have been executed, and I_1 is ordered before I_2 in $<_{ppo\text{-}i2e}$ according to the invariants. At the time when I_2 is executed, I_1 can only be in one of the following two places:

1. I_1 is already in the memory system: In this case, if I_2 is a load, then the Execute-Load rule ensures that I_2 is placed after I_1 in $<_{mo\text{-}i2e}$. If I_2 is a store, it can only be appended to the end of $<_{mo\text{-}i2e}$, and is still after I_1 in $I <_{mo\text{-}i2e}$.
2. I_1 is in the local buffer: In this case, I_1 must be a store. I_2 must also be a store (otherwise if I_2 is a load, the guard of Execute-Load rule will be false due to I_1 in the local buffer). And I_2 is inserted into the local buffer. The Dequeue-Store rule ensures that I_1 will be appended to $<_{mo\text{-}i2e}$ before I_2, so I_1 is still before I_2 in $<_{mo\text{-}i2e}$.

$I_1 <_{mo\text{-}i2e} I_2$ implies that $I_1 <_{mo} I_2$, contradicting with the initial assumption. Thus the Inst-Order axiom is satisfied.

Now we show that the Load-Value axiom is also satisfied. Consider a load L for address a from processor i which reads from a store S in the operational execution. When the Execute-Load rule executes L, we consider where S resides:

1. S is in the local buffer of processor i: S will be appended to $<_{mo\text{-}i2e}$ later, so S must be after L in $<_{mo}$. Since we already have $S <_{po} L$ and $L <_{mo} S$, the Load-Value axiom will only pick stores that are before L in $<_{po}$. Now we consider such a store S' ($\neq S$) for a which is before L in $<_{po}$. Note that S is the most recently inserted store for a when L is executed. Thus, when S is executed by processor i, S' must have been executed, and we have $S' <_{po\text{-}i2e} S \Rightarrow S' <_{ppo\text{-}i2e} S$ at that time (according the definition of same-address ordering). Therefore, S' is appended to $<_{mo\text{-}i2e}$ before S, and thus $S' <_{mo} S$. As a result, the Load-Value axiom also agrees on $S \rightarrow_{rf} L$.
2. S is already in $<_{mo\text{-}i2e}$: When L is executed, the local buffer of processor i cannot contain any store for a according to the guard of the Execute-Load rule. Thus, all stores for a that are before L in $<_{po}$ are

already in $<_{mo\text{-}i2e}$ at that time. Since stores can only be appended to the end of $<_{mo\text{-}i2e}$, all stores for a that are before L in $<_{mo}$ are also in $<_{mo\text{-}i2e}$ by the time when L is executed. Then the way that the Execute-Load rule determines the load value of L is exactly the same as the Load-Value axiom.

□

7.4 Completeness: GAM-I2E Axiomatic Model ⊆ GAM-I2E Operational Model

Theorem 7. *GAM-I2E axiomatic model ⊆ GAM-I2E operational model.*

Proof. The goal is that for any legal axiomatic relations $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ (which satisfy the GAM-I2E axioms), we can run the GAM-I2E operational model to simulate the same program behavior. In each step of the simulation, we first decide which rule to fire in the operational model based on the current state of the operational model and $<_{mo}$, and then we fire that rule. Here is the algorithm to determine which rule to fire in each simulation step:

1. If in the operational model there is a processor whose next instruction is not a load, we fire an Execute-Reg-Branch or Execute-Store-Fence rule to execute that instruction in the operational model.
2. If the above case does not apply, and in the operational model there is a fence that can be dequeued from the local buffer, then we fire the Dequeue-Fence rule to dequeue that fence in the operational model.
3. If neither of the above cases applies, and in the operational model there is a store S in the local buffer of a processor, and S can be dequeued from the local buffer (i.e., the guard for the Dequeue-Store rule is true), and all stores before S in $<_{mo}$ are already in $<_{mo\text{-}i2e}$, then we fire a Dequeue-Store rule to dequeue S in the operational model.
4. If none of the above cases applies, then in the operational model there must be a processor such that the next instruction of the processor is a load L, and L can be executed (i.e., the guard for the Execute-Load rule is true), and all stores before L in $<_{mo}$ are already in $<_{mo\text{-}i2e}$. We fire an Execute-Load rule to execute L in the operational model. In the Execute-Load rule of L, we insert L into $<_{mo\text{-}i2e}$ such that for any instruction I already in $<_{mo\text{-}i2e}$, if $I <_{mo} L$ then $I <_{mo\text{-}i2e} L$, otherwise $L <_{mo\text{-}i2e} I$.

After each step of the simulation, we keep the following invariants:

1. The execution order on each processor is a prefix of the $<_{po}$ of that processor.
2. The result of each executed instruction is the same as that in $<_{po}$.
3. The store read by each executed load is the same as that indicated by the $\rightarrow_{rf}$ edges.
4. The simulation cannot get stuck.
5. For two memory instruction I_1 and I_2, if $I_1 <_{mo\text{-}i2e} I_2$ in the operational model, then $I_1 <_{mo} I_2$ in the axiomatic relations.
6. The order of all stores in $<_{mo\text{-}i2e}$ is a prefix of the order of all stores in $<_{mo}$.

The first two induction invariants imply that *before* each simulation step, the following properties hold for each processor i (assuming the next instruction of the processor is I):

1. $<_{po\text{-}i2e}$ is a prefix of $<_{po}$ (of processor i) up to I (including I).
2. For any instructions $I_1 <_{ppo} I_2$ from processor i, if I_1 and I_2 are not ordered after I in $<_{po}$ (i.e., I_2 may be equal to I), then $I_1 <_{ppo\text{-}i2e} I_2$.
3. For any instructions I_1 and I_2, if $I_1 <_{ppo\text{-}i2e} I_2$, then $I_1 <_{ppo} I_2$.

The detailed proof for these invariants can be found in Appendix E. □

It should be noted that the above models and proofs of GAM-I2E do not rely on the specific forms of $<_{ppod}$ or $<_{ppomf}$. Therefore, GAM-I2E is fully parametrized by $<_{ppod}$ and $<_{ppomf}$.

8 Conclusion

For years, many of the leading industry memory models have been so complicated to understand and to analyze that the status quo was simply to live with an incomplete and underspecified memory model. Academics would attempt to build axiomatic and operational models and then to prove them equivalent,

but these models and proofs were subject to frequent breakage and refinement due to the thorniness of the issues at hand. Other models were simply never updated to modern standards, and were therefore left with definitions fence ordering, same-address ordering, and/or dependency ordering that are today well known to be insufficient. This has led to no shortage of confusion in the broader understanding of memory models in the field.

In response to the recently emerging trend back towards atomic memory models, we present GAM, a flexible operational *and* axiomatic memory model definition that is *parameterized* by the set of fences in the model. GAM corrects the preserved program order definition oversights present in memory models from past generations, and it reduces the definition of fence behavior into localized intra-thread ordering specifications that can be easily understood in isolation. GAM also comes with proofs of equivalence between its axiomatic and operational models, thereby overcoming the obstacle that many previous memory models have faced in being far too complicated to understand or to work with. The equivalence makes it much easier for architects, programmers, and theoreticians to each simply use the variant that they find easiest to work with.

Finally, GAM also makes it easy to understand the implications of tweaking a memory model's definition. It is easy to add new fences that trade off strength for performance, for example. It is also possible to remove behaviors; as we show, forbidding load-store reordering altogether allows GAM to be reduced to an even simpler I2E-based definition. We believe that all of these features will go a long way towards eliminating the worst of the subtleties and corner cases that have most of the memory models of past generations.

References

[1] The risc-v instruction set. `https://riscv.org/`.

[2] *Alpha Architecture Handbook, Version 4*. Compaq Computer Corporation, 1998.

[3] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[4] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

[5] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 13–24. ACM, 2009.

[6] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512–532. Springer, 2013.

[7] Jade Alglave and Luc Maranget. *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, chapter Stability in Weak Memory Models, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[8] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):7, 2014.

[9] ARM. Cortex-A9 MPCore^{TM}, programmer advice notice, read-after-read hazards. Technical report, 2011. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.uan0004a/UAN0004A_a9_read_read.pdf`.

[10] ARM. *ARM Architecture Reference Manual: ARMv8, for ARMv8-A architecture profile*. 2017.

[11] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 29–40. IEEE Computer Society, 2006.

[12] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling sc atomics in c11 and opencl. *SIGPLAN Not.*, 51(1):634–648, January 2016.

[13] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *ACM SIGPLAN Notices*, volume 46, pages 55–66. ACM, 2011.

[14] Colin Blundell, Milo MK Martin, and Thomas F Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 233–244. ACM, 2009.

[15] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[16] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 7:1–7:6, New York, NY, USA, 2014. ACM.

[17] Jason F Cantin, Mikko H Lipasti, and James E Smith. The complexity of verifying memory coherence. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 254–255. ACM, 2003.

[18] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *Programming Languages and Systems*, pages 331–346. Springer, 2007.

[19] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 278–289. ACM, 2007.

[20] Edsger W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[21] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 14, pages 434–442. IEEE Computer Society Press, 1986.

[22] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. Modelling the armv8 architecture, operationally: Concurrency and isa. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 608–621, New York, NY, USA, 2016. ACM.

[23] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[24] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26. ACM, 1990.

[25] Chris Gniady and Babak Falsafi. Speculative sequential consistency with little custom storage. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 179–188. IEEE, 2002.

[26] James R Goodman. *Cache consistency and sequential consistency.* University of Wisconsin-Madison, Computer Sciences Department, 1991.

[27] Dibakar Gope and Mikko H Lipasti. Atomic sc for simple in-order processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 404–415. IEEE, 2014.

[28] Chris Guiady, Babak Falsafi, and Terani N Vijaykumar. Is sc+ ilp= rc? In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 162–171. IEEE, 1999.

[29] IBM. *Power ISA, Version 2.07*. 2013.

[30] International Organization for Standardization (ISO). Information technology – programming languages – C, ISO/IEC 9899:2011. Technical report, December 2011.

[31] Daniel Jackson. Alloy: A lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 11, April 2002. URL: `http://alloy.mit.edu`.

[32] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal c memory model supporting integer-pointer casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 326–335, New York, NY, USA, 2015. ACM.

[33] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[34] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[35] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient sequential consistency via conflict ordering. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 273–286. ACM, 2012.

[36] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated generation of comprehensive memory model litmus test suites. *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[37] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo MK Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for power multiprocessors. In *Computer Aided Verification*, pages 495–512. Springer, 2012.

[38] Jan-Willem Maessen, Arvind, and Xiaowei Shen. Improving the java memory model using crf. *ACM SIGPLAN Notices*, 35(10):1–12, 2000.

[39] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[40] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models. `http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf`, 2012.

[41] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for c/c++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 111–128, New York, NY, USA, 2016. ACM.

[42] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. An operational semantics for c/c++11 concurrency. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 111–128, New York, NY, USA, 2016. ACM.

[43] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[44] Parthasarathy Ranganathan, Vijay S Pai, and Sarita V Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 199–210. ACM, 1997.

[45] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising c/c++ and power. In *ACM SIGPLAN Notices*, volume 47, pages 311–322. ACM, 2012.

[46] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *ACM SIGPLAN Notices*, volume 46, pages 175–186. ACM, 2011.

[47] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. *SIGPLAN Not.*, 44(1):379–391, January 2009.

[48] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[49] Xiaowei Shen, Arvind, and Larry Rudolph. Commit-reconcile and fences (crf): A new memory model for architects and compiler writers. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 150–161. IEEE, 1999.

[50] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 524–535. IEEE Computer Society, 2012.

[51] Richard Smith, editor. *Working Draft, Standard for Programming Language C++*. `http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4527.pdf`, May 2015.

[52] SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., 1992.

[53] David L Weaver and Tom Gremond. *The SPARC architecture manual (Version 9)*. PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[54] Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 266–277. ACM, 2007.

[55] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[56] Sizhuo Zhang, Muralidaran Vijayaraghavan, and Arvind. Weak memory models: Balancing definitional simplicity and implementation flexibility. In *Proceedings of the 2017 International Conference on Parallel Architectures and Compilation*, Portland, OR, USA, 2017.

A GAM Operational model $\subseteq$ GAM Axiomatic Model

Lemma 5. *For any operational model state, the following properties hold:*

1. *If $I_1 <_{po\text{-}rob} I_2$, then whether I_1 is ordered before I_2 in any of $<_{ddep\text{-}rob}, <_{adep\text{-}rob}, <_{ntppo\text{-}rob}$ only depends on the states of I_1, I_2, and instructions between I_1 and I_2 in the ROB.*
2. *If we add a new instruction to the end of an ROB, then changes in $<_{ddep\text{-}rob}, <_{adep\text{-}rob}, <_{ntppo\text{-}rob}$ can only involve new edges pointing to the newly added instruction.*

3. *If we mark a not-done non-branch instruction as done in an ROB, then there is no change in* $<_{ddep\text{-}rob}$, $<_{adep\text{-}rob}$, $<_{ntppo\text{-}rob}$.
4. *If we mark a not-done branch instruction as done in an ROB, then the changes in* $<_{ddep\text{-}rob}$, $<_{adep\text{-}rob}$, $<_{ntppo\text{-}rob}$ *can only involve removing existing edges.*
5. *If we compute the store data of a store in an ROB, then there is no change in* $<_{ddep\text{-}rob}$, $<_{adep\text{-}rob}$, $<_{ntppo\text{-}rob}$.
6. *If we compute the address of a memory instruction in an ROB, then the changes in* $<_{ddep\text{-}rob}$, $<_{adep\text{-}rob}$ *can only involve removing existing edges.*
7. *If we compute the address of a load L to be a in an ROB, then the changes in* $<_{ntppo\text{-}rob}$ *can only involve:*
   (a) *new edges pointing to L, and*
   (b) *new edges from L, and*
   (c) *removal of existing edges.*
8. *If we compute the address of a store S to be a in an ROB, then the changes in* $<_{ntppo\text{-}rob}$ *can only involve:*
   (a) *new edges point to S, and*
   (b) *new edges starting from S, and*
   (c) *new edges across S, and*
   (d) *removal of existing edges.*

Proof. The cases in the lemma can be proved easily one by one. □

Lemma 6. *The following invariants hold during the execution of the operational model:*

1. *If* $I_1 <_{ntppo\text{-}rob} I_2$ *and* $I_2.\mathsf{doneTS} \neq \top$, *then* $I_1.\mathsf{doneTS} \neq \top$ *and* $I_1.\mathsf{doneTS} < I_2.\mathsf{doneTS}$.
2. *If* $I_1 <_{adep\text{-}rob} I_2$ *and* $I_2.\mathsf{addrTS} \neq \top$, *then* $I_1.\mathsf{doneTS} \neq \top$ *and* $I_1.\mathsf{doneTS} < I_2.\mathsf{addrTS}$.
3. *If* $I_1 <_{ddep\text{-}rob} I_2$, *and not* $I_1 <_{adep\text{-}rob} I_2$, *and* I_2 *is a store, and* $I_2.\mathsf{sdataTS} \neq \top$, *then* $I_1.\mathsf{doneTS} \neq \top$ *and* $I_1.\mathsf{doneTS} < I_2.\mathsf{sdataTS}$.
4. *If* $I_1 <_{po\text{-}rob} I_2$, *and* I_1 *is a memory instruction, and* I_2 *is a store, and* $I_2.\mathsf{doneTS} \neq \top$, *then* $I_1.\mathsf{addrTS} \neq \top$ *and* $I_1.\mathsf{addrTS} < I_2.\mathsf{doneTS}$.
5. *We never kill a done store.*
6. *For any address a, let S be the store with the maximum* doneTS *among all the done stores for address a. The monolithic memory value for a is equal to* $S.\mathsf{sdata}$.
7. *For any done load L, let* $S = L.\mathsf{from}$ *(i.e., S is the store read by L). All of the following properties are satisfied:*
   (a) *S still exists in an ROB (i.e., S is not killed).*
   (b) $S.\mathsf{addr} = L.\mathsf{addr}$ *and* $S.\mathsf{sdata} = L.\mathsf{ldval}$.
   (c) *If S is done, then there is no not-done store S' such that $S'.addr = a$ and* $S' <_{po\text{-}rob} L$.
   (d) *If S is done, then for any other done store S' with* $S'.\mathsf{addr} = L.\mathsf{addr}$, *if* $S' <_{po\text{-}rob} L$ *or* $S'.\mathsf{doneTS} < L.\mathsf{doneTS}$, *then* $S'.\mathsf{doneTS} < S.\mathsf{doneTS}$.
   (e) *If S is not done, then* $S <_{po\text{-}rob} L$, *and there is no store S' such that* $S'.\mathsf{addr} = L.\mathsf{addr}$ *and* $S <_{po\text{-}rob} S' <_{po\text{-}rob} L$.

Proof. We now prove the invariants inductively. That is, when rule R fires in the operational model, we assume that all the invariants hold before R fires, and try to prove that the invariants still hold after R fires. To avoid confusion, we add superscript 0 to model states and orderings before R fires, and add superscript 1 to model states and orderings after R fires. For example, $<^0_{ntppo\text{-}rob}$ denotes the non-transitive preserved program order before R fires, while $I.\mathsf{doneTS}^1$ denotes the done timestamp of instruction I after R fires. Consider the type of rule R:

1. Fetch: Assume R fetches a new instruction I into an ROB. According to Lemma 5, new edges in $<^1_{ddep\text{-}rob}$, $<^1_{adep\text{-}rob}$, $<^1_{ntppo\text{-}rob}$ must point to I. Now we consider each invariant:
   - Invariants 1, 2 and 3: These invariants may be affected by the new edges in $<^1_{ddep\text{-}rob}$, $<^1_{adep\text{-}rob}$, $<^1_{ntppo\text{-}rob}$. However, since $I.\mathsf{doneTS}^1$, $I.\mathsf{addrTS}^1$ and $I.\mathsf{sdataTS}^1$ are all $\top$, these invariants cannot be affected.

- Invariants 4, 5, 6, 7: These invariants cannot be affected.

2. Execute-Reg-to-Reg: Assume R executes a reg-to-reg instruction I and marks it as done. According to Lemma 5, $<^1_{ddep\text{-}rob}, <^1_{adep\text{-}rob}, <^1_{ntppo\text{-}rob}$ are the same as $<^0_{ddep\text{-}rob}, <^0_{adep\text{-}rob}, <^0_{ntppo\text{-}rob}$, respectively. $I.\mathsf{doneTS}$ changes from $\top$ to current global time. Now we consider each invariant:
   - Invariant 1: This invariant can be affected by the change in $I.\mathsf{doneTS}$. Consider any I_1 such that $I_1 <^1_{ntppo\text{-}rob} I$. Since I is a reg-to-reg instruction, it must be that $I_1 <^1_{ddep\text{-}rob} I$ according to the definition of $<_{ntppo\text{-}rob}$. Since $<^0_{ddep\text{-}rob}=<^1_{ddep\text{-}rob}$, $I_1 <^0_{ddep\text{-}rob} I$. The guard of R requires that I_1 is already done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < I.\mathsf{doneTS}^1$, i.e., the invariant still holds.
   - Invariants 2, 3, 4, 5, 6, 7: These invariants cannot be affected.
3. Execute-Branch: Assume R executes a branch instruction I and marks it as done. According to Lemma 5, $<^1_{ddep\text{-}rob}, <^1_{adep\text{-}rob}, <^1_{ntppo\text{-}rob}$ are contained by $<^0_{ddep\text{-}rob}, <^0_{adep\text{-}rob}, <^0_{ntppo\text{-}rob}$, respectively. $I.\mathsf{doneTS}$ changes from $\top$ to current global time, and instructions younger than I in the ROB may all be killed. Now we consider each invariant:
   (a) Invariant 1: Consider any I_1 such that $I_1 <^1_{ntppo\text{-}rob} I$. Since I is a branch, it must be that $I_1 <^1_{ddep\text{-}rob} I$. Since $<^1_{ddep\text{-}rob} \subseteq <^0_{ddep\text{-}rob}$, $I_1 <^0_{ddep\text{-}rob} I$. The guard of R ensures that I_1 must be already done right before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < I.\mathsf{doneTS}^1$, i.e., the invariant still holds.
   (b) Invariants 2, 3, 4: These invariants cannot be affected.
   (c) Invariant 5: This invariant may be affected if instructions are killed. We prove by contradiction, i.e., we assume a done store S is killed in rule R. Since S is killed, $I <^0_{po\text{-}rob} S \Rightarrow I <^0_{ntppo\text{-}rob} S$. According to invariant 1, $S.\mathsf{doneTS}^0 \neq \top \Rightarrow I.\mathsf{doneTS}^0 \neq \top$, i.e., I is done even before R fires. This contradicts with the guard of R.
   (d) Invariant 6: This invariant cannot be affected.
   (e) Invariant 7: We consider each case in this invariant:
      - Invariant 7a: This invariant can be affected by instruction kills. Assume a store S is killed by I in rule R ($I <^0_{po\text{-}rob} S$), and S is read by a load L (i.e., $S = L.\mathsf{from}$). We have shown that S cannot be done, so S is not done before R fires. Invariant 7e says that $S <^0_{po\text{-}rob} L$. Then L will also be killed by I, so this invariant still holds.
      - Invariants 7b, 7c, 7d, 7e: These invariants cannot be affected.
4. Execute-Fence: Assume R executes a fence instruction F and marks it as done. According to Lemma 5, $<^1_{ddep\text{-}rob}, <^1_{adep\text{-}rob}, <^1_{ntppo\text{-}rob}$ are the same as $<^0_{ddep\text{-}rob}, <^0_{adep\text{-}rob}, <^0_{ntppo\text{-}rob}$, respectively. $I.\mathsf{doneTS}$ changes from $\top$ to current global time. Now we consider each invariant:
   - Invariant 1: This invariant can be affected by the change in $I.\mathsf{doneTS}$. Consider any I_1 such that $I_1 <^1_{ntppo\text{-}rob} F$. Since $<^0_{ntppo\text{-}rob}=<^1_{ntppo\text{-}rob}$, $I_1 <^0_{ntppo\text{-}rob} F$. Since F is a fence, it must be that $\mathsf{ordered}_f(I_1, F)$ is true before R fires. Then the guard of R ensures that I_1 must be already done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < F.\mathsf{doneTS}^1$, i.e., the invariant still holds.
   - Invariants 2, 3, 4, 5, 6, 7: These invariants cannot be affected.
5. Compute-Store-Data: Assume R computes the data of a store S. According to Lemma 5, $<^1_{ddep\text{-}rob}, <^1_{adep\text{-}rob}, <^1_{ntppo\text{-}rob}$ are the same as $<^0_{ddep\text{-}rob}, <^0_{adep\text{-}rob}, <^0_{ntppo\text{-}rob}$, respectively. $S.\mathsf{sdataTS}$ changes from $\top$ to current global time. Now we consider each invariant:
   - Invariants 1, 2: These invariant cannot be affected.
   - Invariant 3: Consider any instruction I_1 such that $I_1 <^1_{ddep\text{-}rob} S$ but not $I_1 <^1_{adep\text{-}rob} S$. Note that $<^0_{ddep\text{-}rob}=<^1_{ddep\text{-}rob}$ and $<^0_{adep\text{-}rob}=<^1_{adep\text{-}rob}$. Therefore, the computation of the data of S uses the result of I_1 as a source operand. Then the guard of R ensures that I_1 must be already done before R fires. Therefore, we have $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{sdataTS}^1$, and the invariant still holds.
   - Invariants 4, 5, 6: These invariant cannot be affected.
   - Invariant 7: If there exists a done load L such that $L.\mathsf{from}^0 = S$, then $S.\mathsf{sdata}^0 \neq \top$ according to invariant 7b. That is, the store data of S is already computed before R fires, contradicting with the guard of R. Therefore, S is not read by any load yet, and this invariant is not affected.
6. Execute-Store: Assume R executes a store S and marks it as done. According to Lemma 5, $<^1_{ddep\text{-}rob}$

$, <^1_{adep\text{-}rob}, <^1_{ntppo\text{-}rob}$ are the same as $<^0_{ddep\text{-}rob}, <^0_{adep\text{-}rob}, <^0_{ntppo\text{-}rob}$, respectively. $S.\mathsf{doneTS}$ changes from $\top$ to current global time, and the monolithic memory is also updated. Now we consider each invariant:

- Invariant 1: This invariant can be affected by the change in $S.\mathsf{doneTS}$. Consider any I_1 such that $I_1 <^1_{ntppo\text{-}rob} S$. Since $<^0_{ntppo\text{-}rob}=<^1_{ntppo\text{-}rob}$, $I_1 <^0_{ntppo\text{-}rob} S$. Since S is a store, there can following cases to form $I_1 <^0_{ntppo\text{-}rob} S$:
  - $I_1 <^0_{ddep} S$: The guard of R ensures that $\max(S.\mathsf{addrTS}^0, S.\mathsf{sdataTS}^0) < S.\mathsf{doneTS}^1$. Invariants 2 and 3 says that $I_1.\mathsf{doneTS}^0 < \max(S.\mathsf{addrTS}^0, S.\mathsf{sdataTS}^0)$. Thus, we have $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
  - I_1 is a branch: The guard of R ensures that I is already done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
  - There exists a memory instruction I such that $I_1 <^0_{adep\text{-}rob} I <^0_{po\text{-}rob} S$: The guard of R ensures that the address of I has been computed before R fires, i.e., $I.\mathsf{addrTS}^0 < S.\mathsf{doneTS}^1$. Invariant 2 says that $I_1.\mathsf{doneTS}^0 < I.\mathsf{addrTS}^0$. Thus, $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
  - I_1 is a load whose address has been computed to the same as the address of S: The guard of R ensures that I is done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
  - I_1 is a store whose address has been computed to the same as the address of S: The guard of R ensures that I is done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
  - $\mathsf{ordered}(I_1, S)$ is true: The guard of R ensures that I is done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
- Invariants 2, 3: These invariants are not affected.
- Invariants 4: This invariant can be affected by the change in $S.\mathsf{doneTS}$. Consider any memory instruction I_1 such that $I_1 <^1_{po\text{-}rob} I_2$. Note that $<_{po\text{-}rob}$ cannot be changed by R, so $I_1 <^0_{po\text{-}rob} I_2$. The guard of R ensures that I_1 has computed its address before R fires, so $I_1.\mathsf{addrTS}^1 = I_1.\mathsf{addrTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
- Invariant 5: This invariant is not affected.
- Invariant 6: This invariant can be affected by making S done and updating the monolithic memory. We only need to focus on memory address $a = S.\mathsf{addr}^0$; other addresses are not affected. Note that $S.\mathsf{doneTS}^1$ is the maximum among the non-$\top$ doneTS of every instruction. Therefore, after R fires, S is the store with the maximum doneTS among all done stores for a. On the other hand, the monolithic memory location a is updated to $S.\mathsf{sdata}$ by rule R. Thus, the invariant still holds.
- Invariant 7: We assume $a = S.\mathsf{addr}$. We consider each case in this invariant:
  - Invariant 7a, 7b: These invariants are not affected.
  - Invariant 7c: This invariant can be affected when there exists a done load L such that $L.\mathsf{from}^0 = S$. We need to show that there is no not-done store S' such that $S'.\mathsf{addr}^1 = a$ and $S' <^1_{po\text{-}rob} L$. Since rule does not compute any address or change $<_{po\text{-}rob}$. It is equivalent to show that there is no not-done store S' such that $S'.\mathsf{addr}^0 = a$ and $S' <^0_{po\text{-}rob} L$. We prove by contradiction, i.e., we assume such S' exists before R fires. Since S is not done before R fires, according to invariant 7e, $S <^0_{po\text{-}rob} L$ and it cannot be that $S <^0_{po\text{-}rob} S' <^0_{po\text{-}rob} L$. Thus, it must be that $S' <_{po\text{-}rob} S$. However, the guard of R requires that S' to be done bofore R fires, contradicting with the assumption that S' is not done. Thus, the invariant holds.
  - Invariant 7d: This invariant can be affected in the following two ways:
    (a) There exists a done load L such that $L.\mathsf{from}^0 = S$: In this case, we need to show that for any other done store S' with $S'.\mathsf{addr}^1 = a$, if $S' <^1_{po\text{-}rob} L$ or $S'.\mathsf{doneTS}^1 < L.\mathsf{doneTS}^1$, then $S'.\mathsf{doneTS}^1 < S.\mathsf{doneTS}^1$. Since $S.\mathsf{doneTS}^1$ is the maximum doneTS among all done instructions, this invariant still holds.
    (b) There exists a done load L^* and a done store S^* such that $L^*.\mathsf{from}^1 = S^*$ and $L^*.\mathsf{addr}^1 =$

$S^*.\mathsf{addr}^1 = S^*$: In this case, we need to show that if $S <^1_{po\text{-}rob} L^*$ or $S.\mathsf{doneTS}^1 < L^*.\mathsf{doneTS}^1$, then $S.\mathsf{doneTS}^1 < S^*.\mathsf{doneTS}^1$. Since $S.\mathsf{doneTS}^1$ is the maximum, $S.\mathsf{doneTS}^1 < L^*.\mathsf{doneTS}^1$ must be false. We will now show that $S <^1_{po\text{-}rob} L^*$ is also impossible to prove the invariant holds. We prove it by contradiction, i.e., we assume $S <^1_{po\text{-}rob} L^*$. Since rule R does not change $<_{po\text{-}rob}$ or any store address or any state of L^* and S^*, we have $S <^0_{po\text{-}rob} L^*$, $S.\mathsf{addr}^0 = a = L^*.\mathsf{addr}^* = S^*.\mathsf{addr}^0$, $L^*.\mathsf{from}^0 = S^*$, and S^* is done before R fires. This contradicts with Invariant 7c. Therefore, the invariant still holds.

- Invariant 7e: This invariant is not affected.

7. Execute-Load: Assume R executes a load L. If L is not marked as done, the model state does not change, so all invariants still hold. Now we consider the case that L is marked as done. According to Lemma 5, $<^1_{ddep\text{-}rob}, <^1_{adep\text{-}rob}, <^1_{ntppo\text{-}rob}$ are the same as $<^0_{ddep\text{-}rob}, <^0_{adep\text{-}rob}, <^0_{ntppo\text{-}rob}$, respectively. $L.\mathsf{doneTS}$ changes from $\top$ to current global time. Now we consider each invariant:
   - Invariant 1: This invariant can be affected by the change in $L.\mathsf{doneTS}$. Assume $a = L.\mathsf{addr}^0$. Consider any I_1 such that $I_1 <^1_{ntppo\text{-}rob} L$. Since $<_{ntppo\text{-}rob}$ is not changed by R, we have $I_1 <^0_{ntppo\text{-}rob} L$. This ordering can be caused by the following cases:
     - $I_1 <^0_{ddep} L$: Since L only needs to compute the address from registers, $I_1 <^0_{adep} L$. Invariant 2 says that $I_1.\mathsf{doneTS}^0 < I_1.\mathsf{addrTS}^0$. The guard of R ensures that $L.\mathsf{addrTS}^0 < L.\mathsf{doneTS}^1$. Therefore $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < L.\mathsf{doneTS}^1$, i.e., the invariant still holds.
     - There exists a store S such that $S.\mathsf{addr}^1 = a$ and $I_1 <^1_{ddep\text{-}rob} S <^1_{po\text{-}rob} L$, and there is no store S' such that $S'.\mathsf{addr}^1 = a$ and $S <^1_{po\text{-}rob} S' <^1_{po\text{-}rob} L$: Since R does not change $<_{po\text{-}rob}$, $<_{ddep\text{-}rob}$ or any address, the above condition becomes: $S.\mathsf{addr}^0 = a$, and $I_1 <^0_{ddep\text{-}rob} S <_{po\text{-}rob} L$, and there is no store S' such that $S'.\mathsf{addr}^0 = a$ and $S <^0_{po\text{-}rob} S' <^0_{po\text{-}rob} L$. Before R fires, if there are not-done loads with computed addresses a between S and L in the ROB, then let L' be the youngest of them in ROB, and the ROB search conducted in R will stop at L'. This will make R not mark L as done, contradicting with our previous assumption. Therefore, right before R fires, any load with computed address a between S and L in the ROB must be done. Since $S.\mathsf{addr}^0 = a$, the ROB search in R will search through S. If $S.\mathsf{sdataTS}^0 = \top$, then the ROB search will stop at S and L cannot be marked as done, contradicting with our assumption. Therefore, $S.\mathsf{sdataTS} \neq \top \Rightarrow S.\mathsf{sdataTS}^0 < L.\mathsf{doneTS}^1$. Since address of S is computed before R fires, $S.\mathsf{addrTS}^0 < L.\mathsf{doneTS}^1$. Invariants 2 and 3 say that $I_1.\mathsf{doneTS}^0 < \max(S.\mathsf{addrTS}^0, S.\mathsf{sdataTS}^0)$, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < L.\mathsf{doneTS}^1$, i.e., the invariant still holds.
     - I_1 is a load with $I_1.\mathsf{addr}^1 = a$, and there is no store S such that $S.\mathsf{addr}^1 = a$ and $I_1 <^1_{po\text{-}rob} S <^1_{po\text{-}rob} L$: Since R does not change $<_{po\text{-}rob}$ or any address, the above condition becomes: $I_1.\mathsf{addr}^0 = a$, and there is no store S such that $S.\mathsf{addr}^0 = a$ and $I_1 <^0_{po\text{-}rob} S <^1_{po\text{-}rob} L$. Before R fires, if there are not-done loads with computed addresses a between I_1 and L in the ROB, then let L' be the youngest of them in ROB, and the ROB search conducted in R will stop at L'. This will make R not mark L as done, contradicting with our previous assumption. Therefore, right before R fires, any load with computed address a between I_1 and L in the ROB must be done. Since $I_1.\mathsf{addr} = a$, the ROB search in R will search through S. If I_1 is not done before R fires, then the ROB search will stop at L and L cannot be marked as done in R, contradicting with our previous assumption. Therefore I_1 is done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < L.\mathsf{doneTS}^1$, i.e., the invariant still holds.
     - $\mathsf{ordered}(I_1, L)$ is true: The guard of R ensures that I is done before R fires, so $I_1.\mathsf{doneTS}^1 = I_1.\mathsf{doneTS}^0 < S.\mathsf{doneTS}^1$, i.e., the invariant still holds.
   - Invariants 2, 4, 3, 5, 6: These invariants cannot be affected.
   - Invariant 7: This invariant can be affected because L becomes done. Let $S = L.\mathsf{from}^1$, and let $a = L.\mathsf{addr}^1$. We need to show that L and S satisfies all the sub-invariants. We consider each case of separately.
     - Invariant 7a: This invariant cannot be affected.
     - Invariant 7b: We need to show that $S.\mathsf{addr}^1 = L.\mathsf{addr}^1$ ad $S.\mathsf{sdata}^1 = L.\mathsf{ldval}^1$. Note that $S.\mathsf{addr}^0 =$

$S.\mathsf{addr}^0 = L.\mathsf{addr}^0 = L.\mathsf{addr}^1$. Also note that $S.\mathsf{sdata}^0 = S.\mathsf{sdata}^1$ and $L.\mathsf{ldval}$ is the value read in rule R. If L bypasses from local store in ROB, then S is the store being bypassed, and the invariant holds. Otherwise, L reads from monolithic memory, and invariant 6 ensures that invariant holds.

- Invariant 7c: Since S is done after R fires, S is also done before R fires. Then R reads the value from monolithic memory. The guard of R ensures that there is no store S' such that $S'.\mathsf{addr}^0 = a$ and $S' <^0_{po\text{-}rob} L$. Since R does not change address or $<_{po\text{-}rob}$, the invariant still holds.
- Invariant 7d: Using the same argument as above, R reads the value from monolithic memory. Right before R fires, invariant 6 says that for any other done store S' with $S'.\mathsf{addr}^0 = a$, $S'.\mathsf{doneTS} < S.\mathsf{doneTS}$. Since R does not change doneTS of stores or address, the invariant still holds.
- Invariant 7e: Since $S.\mathsf{doneTS}^1 = \top$, $S.\mathsf{doneTS}^0 = \top$. Then R reads the value by bypassing from S in the local ROB, i.e., the ROB search in R stops at S. We now prove by contradiction, i.e., we assume there exists S' with $S'.\mathsf{addr}^1 = a$ and $S <^1_{po\text{-}rob} S' <^1_{po\text{-}rob} L$. Since R does not change address or $<_{po\text{-}rob}$, $S'.\mathsf{addr}^0 = a$ and $S <^0_{po\text{-}rob} S' <^0_{po\text{-}rob} L$. Since $S <^0_{ntppo\text{-}rob} S'$, invariant 1 says that $S'.\mathsf{doneTS}^0 = \top$. Then the ROB search cannot stop at S, contradicting with our previous conclusion. Therefore the invariant still holds.

8. Compute-Mem-Addr for load L: R computes the address of load L to a. According to Lemma 5, edges in $<_{ddep\text{-}rob}$ and $<_{adep\text{-}rob}$ may reduce. Some edges in $<_{ntppo\text{-}rob}$ may be removed, but there can also be new $<_{ntppo\text{-}rob}$ edges. $L.\mathsf{addr}$ changes from $\top$ to a. We consider each invariant separately.
   - Invariant 1: Since L is not done, we only need to consider newly generated $<_{ntppo\text{-}rob}$ edges that start from L. Consider any I_2 such that $L <^1_{ntppo\text{-}rob} I_2$ but not $L <^0_{ntppo\text{-}rob} I_2$. We need to show that $I_2.\mathsf{doneTS}^1 = \top$. I_2 must be in the following cases:
     - I_2 is a store, $L <^1_{po\text{-}rob} I_2$, and $I_2.\mathsf{addr}^1 = a$: We prove by contradiction, i.e., assume $I_2.\mathsf{doneTS}^1 \neq \top$. This gives $I_2.\mathsf{doneTS}^0 \neq \top$. Since $L <^0_{po\text{-}rob} S$ and $L.\mathsf{addrTS}^0 = \top$, invariant 2 says that $I_2.\mathsf{doneTS}^0 = \top$, contradicting with previous assumption. Therefore the invariant still holds.
     - I_2 is a load, $L <^1_{po\text{-}rob} I_2$, $I_2.\mathsf{addr}^1 = a$, and there is no store S such that $S.\mathsf{addr}^1 = a$ and $L <^1_{po\text{-}rob} S <^1_{po\text{-}rob} I_2$: We prove by contradiction, i.e., assume $I_2.\mathsf{doneTS}^1 \neq \top$. This implies that I_2 is also done before R fires. Since the ROB search in R does not kill I_2, there must be a not-done load L' such that $L'.\mathsf{addr}^0 = a$ and $L <^0_{po\text{-}rob} L' <^0_{po\text{-}rob} I_2$. Now we have $L' <_{ntppo\text{-}rob} I_2$. Since $L'.\mathsf{doneTS} = \top$, this contradicts with invariant 1. Therefore the invariant still holds.
   - Invariant 2: The guard of R ensures that this invariant still holds.
   - Invariants 3, 4: These invariants cannot be affected.
   - Invariant 5: We prove by contradiction, i.e., we assume a done store S is killed. Note that $S.\mathsf{doneTS}^0 = S.\mathsf{doneTS}^1 \neq \top$ and $L <^0_{po\text{-}rob} S$. Invariant 4 says that $L.\mathsf{addrTS}^0 \neq \top$, contradicting with the guard of R. Therefore the invariant still holds.
   - Invariants 6 and 7: These invariants cannot be affected.

9. Compute-Mem-Addr for store S: R computes the address of S to be a. According to Lemma 5, edges in $<_{ddep\text{-}rob}$ and $<_{adep\text{-}rob}$ may reduce. Some edges in $<_{ntppo\text{-}rob}$ may be removed, but there can also be new $<_{ntppo\text{-}rob}$ edges. $S.\mathsf{addr}$ changes from $\top$ to a. We consider each invariant separately.
   - Invariant 1: Since S is not done, we do not need to consider new $<_{ntppo\text{-}rob}$ edges ending at S. We only need to consider new $<_{ntppo\text{-}rob}$ edges starting from S or across S. There are the following two cases:
     - There is store S' such that $S'.\mathsf{addr}^1 = a$ and $S <^1_{po\text{-}rob} S'$: We need to show that $S'.\mathsf{doneTS}^1 = \top$. We prove by contradiction, i.e., we assume $S'.\mathsf{doneTS}^1 \neq \top$. This implies $S'.\mathsf{doneTS}^0 \neq \top$ and $S <^0_{po\text{-}rob} S'$. According to invariant 4, $S.\mathsf{addr}^0 \neq \top$, contradicting with the guard of R.
     - There is instruction I_1 and load L such that $L.\mathsf{addr}^1 = a$ and $I_1 <^1_{ddep\text{-}rob} S <^1_{po\text{-}rob} L$, and there is no store S' such that $S'.\mathsf{addr}^1 = a$ and $S <^1_{po\text{-}rob} S' <^1_{po\text{-}rob} L$: The above statement becomes: $L.\mathsf{addr}^0 = a$, $I_1 <^0_{ddep\text{-}rob} S <^0_{po\text{-}rob} L$, and there is not store S' such that $S'.\mathsf{addr}^0 = a$ and $S <^0_{po\text{-}rob} S' <^0_{po\text{-}rob} L$. We can show that $L.\mathsf{doneTS}^1 = \top$, so the invariant still holds. We prove by contradiction, i.e., $L.\mathsf{doneTS}^1 \neq \top$. This implies $L.\mathsf{doneTS}^0 \neq \top$. Since S is not killed by the ROB search in R,

there must be L' such that $L'.\mathsf{addr}^0 = a$, $L'.\mathsf{doneTS}^0 = \top$ and $S <^0_{po\text{-}rob} L' <^0_{po\text{-}rob} L$. This gives $L' <^0_{ntppo\text{-}rob} L$. Since $L.\mathsf{doneTS}^0 \neq \top$, this contradicts invariant 1.

- Invariant 2: The guard of R ensures that this invariant still holds.
- Invariants 3, 4: These invariant cannot be affected.
- Invariant 5: We prove by contradiction, i.e., we assume a done store S' is killed. That is, $S'.\mathsf{doneTS}^0 \neq \top$ and $S <^0_{po\text{-}rob} S'$. Invariant 4 says that $S.\mathsf{addr}^0 \neq \top$, contradicting with the guard of R.
- Invariant 6: This invariant is not affected.
- Invariant 7: For any done load L^* for address a, assume $S^* = L^*.\mathsf{from}^1$. The address computation of S may prevent L^* and S^* from satisfying the invariants here. Note that $L^*.\mathsf{addr}^0 = a$ and $S^* = L^*.\mathsf{from}^0$. We consider each case of this invariant:
  - Invariant 7a: We prove by contradiction, i.e., S^* is killed but L^* is not. We have proved that S^* cannot be done, so S^* is not done before R fires. Invariant 7e says that $S^* <^0_{po\text{-}rob} L^*$. Then L^* is also killed, contradicting with our assumption.
  - Invariant 7b: This invariant cannot be affected.
  - Invariant 7c: We need to show that if S^* is done, then it is impossible that $S <^1_{po\text{-}rob} L^*$. We prove by contradiction, i.e., assume $S^*.\mathsf{doneTS}^0 \neq \top$ and $S <^1_{po\text{-}rob} L^*$. This implies that $S <^0_{po\text{-}rob} L^*$.. Invariant 7c says that there is no store S' such that $S'.\mathsf{addr}^0 = a$ and $S' <^0_{po\text{-}rob} L^*$. Since L^* is not killed in the ROB search of rule R, there must be load L' such that $L'.\mathsf{addr}^0 = a$ and $L'.\mathsf{doneTS}^0 = \top$ and $L' <^0_{po\text{-}rob} L^*$. This gives $L' <^0_{ntppo\text{-}rob} L^* \Rightarrow L'.\mathsf{doneTS}^0 \neq \top$, contradicting with previous conclusion. Therefore the invariant still holds.
  - Invariant 7d: This invariant is not affected.
  - Invariant 7e: We need to show that if S^* is not done, then it is impossible that $S^* <^1_{po\text{-}rob} S <^1_{po\text{-}rob} L^*$. We prove by contradiction, i.e., we assume $S^*.\mathsf{doneTS}^0 \neq \top$ and $S^* <^1_{po\text{-}rob} S <^1_{po\text{-}rob} L^*$. This implies $S^* <^0_{po\text{-}rob} S <^0_{po\text{-}rob} L^*$. Invariant 7e says that there is no store S' such that $S'.\mathsf{addr}^0 = a$ and $S^* <^0_{po\text{-}rob} S' <^0_{po\text{-}rob} L^*$. Since L^* is not killed by the ROB search in rule R, there must be load L' such that $L'.\mathsf{addr}^0 = a$ and $L'.\mathsf{doneTS}^0 = \top$ and $S <^0_{po\text{-}rob} L' <^0_{po\text{-}rob} L^*$. This gives $L' <^0_{ntppo\text{-}rob} L^* \Rightarrow L'.\mathsf{doneTS}^0 \neq \top$, contradicting with previous conclusion. Therefore the invariant still holds.

□

B GAM Axiomatic model ⊆ GAM Operational Model

Theorem 8. *GAM axiomatic model ⊆ GAM operational model.*

Proof. The goal is that for any legal axiomatic relations $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ (which satisfy the GAM axioms), we can run the operational model to give the same program behavior. The strategy to run the operational model consists of two major phases. In the first phase, we only fire Fetch rules to fetch all instructions into all ROBs according to $<_{po}$. During the second phase, in each step we fire a rule that either marks an instruction as done or computes the address or data of a memory instruction. Which rule to fire in a step depends on the current state of the operational model and $<_{mo}$. Here we give the detailed algorithm that determines which rule to fire in each step:

1. If in the operational model there is a not-done reg-to-reg or branch instruction whose source registers are all ready, then we fire an Execute-Reg-to-Reg or Execute-Branch rule to execute that instruction.
2. If the above case does not apply, and in the operational model there is a memory instruction, whose address is not computed but the source registers for the address computation are all ready, then we fire a Compute-Mem-Addr rule to compute the address of that instruction.
3. If neither of the above cases applies, and in the operational model there is a store instruction, whose store data is not computed but the source registers for the data computation are all ready, then we fire a Compute-Store-Data rule to compute the store data of that instruction.

4. If none of the above cases applies, and in the operational model there is a fence instruction and the guard of the Execute-Fence rule for this fence is ready, then we fire the Execute-Fence rule to execute that fence.
5. If none of the above cases applies, then we find the oldest instruction in $<_{mo}$, which is not-done in the operational model, and we fire an Execute-Load or Execute-Store rule to execute that instruction.

Before giving the invariants, we give a definition related to the ordering of stores for the same address. For each address a, all stores for a are totally ordered by $<_{mo}$, and we refer to this total order of stores for a as $<^a_{co}$.

Now we show the invariants. After each step, we maintain the following invariants:

1. The order of instructions in each ROB in the operational model is the same as the $<_{po}$ of that processor in the axiomatic relations.
2. The results of all the instructions that have been marked as done so far in the operational model are the same as those in the axiomatic relations.
3. All the load/store addresses that have been computed so far in the operational model are the same as those in the axiomatic relations.
4. All the store data that have been computed so far in the operational model are the same as those in the axiomatic relations.
5. No kill has ever happened in the operational model.
6. For the rule fired in each step that we have performed so far, the guard of the rule is satisfied the at that step (i.e., the rule can fire).
7. In each step that we have performed so far, if we fire a rule to execute an instruction (especially a load) in that step, the instruction must be marked as done by the rule.
8. For each address a, the order of all the store updates on monolithic memory address a that have happened so far in the operational model is a prefix of $<^a_{co}$.

We now examine each option that we may choose in each step of phase 2, and verify that all invariants hold.

1. We execute a reg-to-reg or branch instruction: trivial.
2. We compute the address of a load or store instruction I: we only need to verify invariant 5, i.e., this address computation will not kill any done load. Assume the address of I is a, and I is in processor i. We prove by contradiction, i.e., we assume the address computation of I kills a done load L in the ROB of processor i. We search $<_{po}$ of processor i from L towards the oldest instruction (excluding L). We stop the search when we find a load or store instruction for address a (note that all addresses in $<_{po}$ are known). We refer to the instruction found as I_a. Since I has address a, either $I = I_a$ or $I <_{po} I_a$. If $I = I_a$, then I_a is not-done and its address is not computed before the kill. In case of $I <_{po} I_a$, I_a cannnot be a done store (because the address of I is just computed). In this case, I_a must be not-done and its address must not be computed, otherwise L will not be killed. In either case, I_a must be not-done and the address of I_a is not computed before the kill. We also have the following observation:
   - L can only become done via option 5 in a previous step, so for any not-done memory instruction I' in the operational model, we have $L <_{mo} I'$ in the axiomatic relations.

   We consider the following two possibilities:
   (a) I_a is a load: In this case, the above observation says that $L <_{mo} I_a$. However, since there is no other memory instruction for address a between I_a and L in processor i, $I_a <_{pposa} L \Rightarrow I_a <_{mo} L$. Thus this case is impossible.
   (b) I_a is a store: In this case, we consider which store is read by L.
      i. L bypasses from a store S in processor i. S must be older than I_a in ROB (because address of I_a is not computed at the time when L is executed), so $S <_{po} I_a \Rightarrow S <_{mo} I_a$. This contradicts with the Load-Value axiom.
      ii. L reads the value of a store S from the monolithic memory. Since all the done stores for a form the prefix of $<^a_{co}$ and I_a is not-done, $S <^a_{co} I_a \Rightarrow S <_{mo} I_a$. This also contradicts with the Load-Value axiom.

   Therefore this address computation cannot cause any kill.
3. We compute a store data: trivial.

4. We execute a fence instruction: trivial.
5. We execute a memory instruction I from $<_{mo}$: First note that for any memory instruction I' such that $I' <_{mo} I$, I' must be done in the operational model (because of the way we pick I). We now prove according to the type of I.
   - I is a store for address a: we first show that all the guards are satisfied:
     (a) Address and data of I have been computed: We prove by contradiction, i.e., the address or data of I has not been computed. We backtrack the dependency chain on I. The only reason for not being able to compute the address or data of I is that an older instruction I_1 is not-done, and that $I_1 <_{ddep} I$. I_1 can only be a not-done reg-to-reg instruction or a not-done load. If I_1 is a reg-to-reg instruction, it cannot be computed because of a not-done instruction $I_2 <_{ddep} I_1$. We trace this dependency chain until we encounter a load, i.e., $I_k <_{ddep} I_{k-1} <_{ddep} \ldots <_{ddep} I_1 <_{ddep} I$, where I_k is a not-done load and $I_1 \ldots I_{k-1}$ are all not-done reg-to-reg instructions. Now we have $I_k <_{ppo} I \Rightarrow I_k <_{mo} I$, contradicting with the way we pick I in option 5.
     (b) All older memory or fence instructions that are ordered before I by ordered are done: We prove by contradiction, i.e., assume there is a not-done memory or fence instruction I_1 that is older than I in ROB and satisfies $\mathsf{ordered}(I_1, I)$. This implies $I_1 <_{po} I \Rightarrow I_1 <_{ppomf} I$. If I_1 is a not-done fence, the guard to execute it in the operational model must be false according to our algorithm. Therefore, there must be a not-done memory or fence instruction I_2 that is older than I_1 in ROB and satisfies $\mathsf{ordered}(I_2, I_1)$. This implies $I_2 <_{ppomf} I$. We keep backtracking if I_2 is also a not-done fence. We stop backtracking until I_k is a not-done memory instruction. That is, we have $I_k <_{ppomf} I_{k-1} <_{ppomf} \cdots <_{ppomf} I_1 <_{ppomf} I \Rightarrow I_k <_{mo} I$. Since I_k is a not-done memory instruction, this contradicts with the way we pick I in option 5. Therefore, such I_1 does not exist.
     (c) All older branches are done: We prove by contradiction, i.e., an older branch B in the ROB of I is not-done. We backtrack the dependency chain on B, and get $I_k <_{ddep} I_{k-1} <_{ddep} \ldots <_{ddep} I_1 <_{ddep} B$, where I_k is a not-done load and $I_1 \ldots I_{k-1}$ are all not-done reg-to-reg instructions. Since $B <_{po} I$, we have $I_k <_{ppo} B <_{ppod} I \Rightarrow I_k <_{ppo} I \Rightarrow I_k <_{mo} I$, contradicting with the way we pick I in option 5.
     (d) All older loads and stores have computed their addresses: We prove by contradiction, i.e., an older load or store I' in the ROB of I has not computed its address. We backtrack the dependency chain on the address of I', and get $I_k <_{ddep} I_{k-1} <_{ddep} \ldots <_{ddep} I_1 <_{adep} I'$, where I_k is a not-done load and $I_1 \ldots I_{k-1}$ are all not-done reg-to-reg instructions. Since $I' <_{po} I$, we have $I_k <_{ppo} I_1 <_{ppod} I \Rightarrow I_k <_{ppo} I \Rightarrow I_k <_{mo} I$, contradicting with the way we pick I in option 5.
     (e) All older loads and stores for address a are done: For any store S for address a that is older than I in the ROB of I, we have $S <_{pposa} I \Rightarrow S <_{mo} I$. Therefore, S must be done. For any load L for address a that is older than I in the ROB of i, we have $L <_{pposa} I \Rightarrow L <_{mo} I$. Therefore, L must be done.

     We now only need to verify invariant 8. This is trivial, because all stores for a that is older than I in $<_{mo}$ are done (i.e., have updated $m[a]$).
   - I is a load for address a: we first show that all the guards are satisfied:
     (a) Address of I has been computed: same argument as store case.
     (b) All older memory or fence instructions that are ordered before I by ordered are done: same argument as store case.

     We now need to verify invariants 2 and 7. To do this, we consider the three possible outcomes of the ROB search in the Execute-Load rule that executes I:
     (a) The search finds a not-done load L: We prove that this case is impossible (for invariant 7) by contradiction. If there are intervening stores for a between L and I in the ROB, none of those stores can be done, because the not-done load L will make the guards of Execute-Store rules fail. Let S be the youngest store among these stores. S must not have computed its address, because otherwise the search will stop at S. Now we backtrack the dependency chain on the address of S, and get $I_k <_{ddep} I_{k-1} <_{ddep} \ldots <_{ddep} I_1 <_{ddep} S$, where I_k is a not-done load and $I_1 \ldots I_{k-1}$

are all not-done reg-to-reg instructions. Since there is no store for a between S and I, we have $I_k <_{ppo} I_1 <_{ppod} I \Rightarrow I_k <_{ppo} I \Rightarrow I_k <_{mo} I$. Since I_k is not done, this contradicts with the way we pick I in option 5. Therefore, there is no store for a between L and I in the ROB. Then we have $L <_{pposa} I \Rightarrow L <_{mo} I$. Since L is not done, this contradicts with the way we pick I in option 5.

(b) The search finds a not-done store S: Using the same argument as above, there cannot be any store for a between I and S in ROB. We now prove that the data of the S must have been computed (for invariant 7). We prove by contradiction, i.e., we assume the data of S is not yet computed. We backtrack the dependency chain on the data of S, and get $I_k <_{ddep} I_{k-1} <_{ddep} \ldots <_{ddep} I_1 <_{ddep} S$, where I_k is a not-done load and $I_1 \ldots I_{k-1}$ are all not-done reg-to-reg instructions. Since there is no store for a between S and I, we have $I_k <_{ppo} I_1 <_{ppod} I \Rightarrow I_k <_{mo} i$, contradicting with the way we pick i in option 5 (j_n is a not-done load).
Since the data of S has been computed, I reads from S. We now need to verify invariant 2. Since S is not-done, we have $I <_{mo} S$, i.e., the Load-Value axiom can only select from stores $<_{po} I$. Since there is no other store for a between S and I in the ROB, the Load-Value axiom also agrees on $S \rightarrow_{rf} I$.

(c) The search finds nothing: In this case, I reads from $m[a]$, and we need to verify invariant 2. We first show that all stores for a older than I in ROB are done. If there are not-done stores for a older than I in the ROB, then let S be the youngest one among them. There cannot be any done store for a between S and I, because the guard of the Execute-Store rule that marks the store as done cannot be satisfied. The address of S cannot be computed (otherwise the search will stop at S). Now we backtrack the dependency chain on the address of S as we do in the first case, and can show a contradiction.
Assume $m[a]$ is last written by store S^* before this rule fires. Thus, for any done store S' for a when this rule fires, either $S' = S^*$ or $S' <_{mo} S^*$. Since loads and stores can only be marked as done via option 5 in the operational model and S^* is already done, we have $S^* <_{mo} I$. For any store $S_1 <_{mo} I$, S_1 must be done, so either $S_1 = S^*$ or $S_1 <_{mo} S^*$. For any store $S_2 <_{po} I$, S_2 is also done, so either $S_2 = S^*$ or $S_2 <_{mo} S^*$. Therefore, the Load-Value axiom also agrees on $S^* \rightarrow_{rf} I$.

□

C Equivalence of COM and GAM

We first define one more derived relation:

- Reads-from internal ($\rightarrow_{rfi}$), which is the subset of $\rightarrow_{rf}$ for which both the read and the write are in the same thread

C.1 GAM $\subseteq$ COM

Lemma 7. *All of $\rightarrow_{rfe}$, $<_{co}$, $\rightarrow_{fr}$, and $<_{ppo}$ are contained in $<_{mo}$.*

Proof. Two of the four cases are easy: $<_{co}$ is contained in $<_{mo}$ by construction, and $<_{ppo}$ is contained in $<_{mo}$ by the Inst-Order axiom.

By the Load-Value axiom, if for any write w and read r, if $w \rightarrow_{rf} r$, then w precedes r either in $<_{po}$ or in $<_{mo}$. The former is ruled out in the definition of $\rightarrow_{rfe}$, and hence w must precede r in $<_{mo}$.

The proof for $\rightarrow_{fr}$ proceeds by contradiction. Suppose there is some write w and some read r such that $r \rightarrow_{fr} w$ and $w <_{mo} r$. Then by definition of $\rightarrow_{fr}$, there is some other write w' such that $w' \rightarrow_{rf} r$ and $w' <_{co} w$. Furthermore, since $<_{co} \subseteq <_{mo}$, we have $w' <_{mo} w <_{mo} r$. This, however, contradicts the Load-value axiom, as w' is not the $<_{mo}$-maximal candidate write. □

The SC-per-Location axiom will take a bit more work to prove. To start, define $<_{eco}$ as the union of the following relations:

- $<_{co}$ (Write to Write)
- $\rightarrow_{fr}$ (Read to Write)
- $<_{co}{}^*;\rightarrow_{rf}$ (Write to Read)
- $\rightarrow_{rf^{-1}};<_{co}{}^*;\rightarrow_{rf}$ (Read to Read)

Lemma 8. *For all pairs i_1, i_2 of memory accesses to the same address, either $i_1 <_{eco} i_2$ or $i_2 <_{eco} i_1$.*

Proof. By construction. All pairs of same-address writes are ordered in $<_{co}$ by definition. For any read r and write w, let w' be the write such that $w' \rightarrow_{rf} r$. Then either:

- $w = w'$, so $w \rightarrow_{rf} r$, and hence $w<_{co}{}^*;\rightarrow_{rf} r$,
- $w <_{co} w'$, so $w <_{co};\rightarrow_{rf} r$, and hence $w<_{co}{}^*;\rightarrow_{rf} r$, or
- $w' <_{co} w$, so $r \rightarrow_{rf^{-1}};<_{co} w$, and $r \rightarrow_{fr} w$.

Likewise, for any two reads r_1 and r_2, let w_1 and w_2 be the writes such that $w_1 \rightarrow_{rf} r_1$ and $w_2 \rightarrow_{rf} r_2$. Then either:

- $w_1 = w_2$, so $r_1 \rightarrow_{rf^{-1}};\rightarrow_{rf} r_2$, and hence $r_1 \rightarrow_{rf^{-1}};<_{co}{}^*;\rightarrow_{rf} r_2$,
- $w_1 <_{co} w_2$, so $r_1 \rightarrow_{rf^{-1}};<_{co};\rightarrow_{rf} r_2$, and hence $r_1 \rightarrow_{rf^{-1}};<_{co}{}^*;\rightarrow_{rf} r_2$,
- $w_2 <_{co} w_1$, so $r_2 \rightarrow_{rf^{-1}};<_{co};\rightarrow_{rf} r_1$, and hence $r_2 \rightarrow_{rf^{-1}};<_{co}{}^*;\rightarrow_{rf} r_1$.

□

If i_1 and i_2 are related in program order, then the $<_{eco}$ direction must match:

Lemma 9. *If $i_1 <_{poloc} i_2$, then $i_1 <_{eco} i_2$.*

Proof. By Lemma 8, either $i_1 <_{eco} i_2$ or $i_2 <_{eco} i_1$. We show that the latter always results in a contradiction (except for one case in which it overlaps with the former).

- If i_1 and i_2 are both writes, then $i_1 <_{poloc} i_2 <_{co} i_1$, so $i_1 <_{poloc} i_2 <_{mo} i_1$, which contradicts Definition 8.2.
- If i_1 is a read and i_2 is a write, then suppose $i_2<_{co}{}^* i \rightarrow_{rf} i_1$ for some i. If $i_2 = i$, then by the LoadValue axiom, either $i_2 <_{poloc} i_1$, which contradicts the hypothesis, or $i_2 <_{mo} i_1$, which contradicts Definition 8.1. Therefore, suppose $i_2 <_{co} i \rightarrow_{rf} i_1$. If $i \rightarrow_{rfe} i_1$, then $i_1 <_{ppo} i_2 <_{co} i \rightarrow_{rfe} i_1$ by Definition 8.1, which contradicts Causality. If $i \rightarrow_{rfi} i_1$, then $i <_{poloc} i_1$; otherwise, by the LoadValue axiom, $i_1 <_{mo} i$, which contradicts Definition 8.1. However, this means $i <_{poloc} i_2 <_{co} i$, which again contradicts Definition 8.2.
- If i_1 is a write and i_2 is a read, then suppose $i_2 \rightarrow_{fr} i_1$, and let i be the write such that $i \rightarrow_{rf} i_2$ and $i <_{co} i_1$. Since $i <_{mo} i_1$, i is not the $<_{mo}$-maximal store from which i_2 should read, and the LoadValue axiom is violated.
- If i_1 and i_2 are both reads, then suppose $i_2 \rightarrow_{rf^{-1}} i_3 <_{co} i_4 \rightarrow_{rf} i_1$ for some i_3 and i_4. (The case $i_2 \rightarrow_{rf^{-1}};\rightarrow_{rf} i_1$ implies $i_1 <_{eco} i_2$.) Then $i_2 \rightarrow_{fr} i_4$. If $i_4 \rightarrow_{rfi} i_1$, then $i_4 <_{poloc} i_2 \rightarrow_{fr} i_4$, which as we have already seen in the previous case is forbidden. If $i_4 \rightarrow_{rfe} i_1$, then either there is some write i_5 such that $i_1 <_{poloc} i_5 <_{poloc} i_2$, or there is no such write. If i_5 exists, then $i_4 <_{co} i_5$, $i_3 <_{co} i_5$, and $i_2 \rightarrow_{fr} i_5 <_{poloc} i_2$, which as we have already seen is forbidden. If i_5 does not exist, then $i_1 <_{ppo} i_2 \rightarrow_{fr} i_4 \rightarrow_{rfe} i_1$, which contradicts Causality.

□

Theorem 9. *The SC-per-Location axiom is satisfied.*

Proof. First, by Lemma 9, all $<_{poloc}$ edges involving at least one write can be converted into sequences containing only $\rightarrow_{rf}$, $<_{co}$, and $\rightarrow_{fr}$. So we consider only cycles with $\rightarrow_{rf}$, $<_{co}$, $\rightarrow_{fr}$, and read-to-read $<_{poloc}$ edges. Among such cycles, first consider cycles with no $<_{co}$ or $\rightarrow_{fr}$ edges. Such cycles cannot contain $\rightarrow_{rf}$ either, because neither $\rightarrow_{rf}$ nor read-read $<_{poloc}$ edges can end at a write node, and so there cannot be a source for $\rightarrow_{rf}$ relations. This leaves a cycle consisting only of $<_{poloc}$, which is a contradiction.

Now, consider cycles with at least one $<_{co}$ or $\rightarrow_{fr}$ edge. Replace every instance of read-read $<_{poloc}$ in the cycle with $\rightarrow_{rf^{-1}};<_{co}{}^*;\rightarrow_{rf}$ per Lemma 9. Now, because $<_{co}$ and $\rightarrow_{fr}$ both target writes, every appearance of $\rightarrow_{rf^{-1}}$ must be preceded either by $\rightarrow_{rf}$ or by $\rightarrow_{rf^{-1}};<_{co}{}^*;\rightarrow_{rf}$. In particular, every appearance of $\rightarrow_{rf^{-1}}$ must be preceded directly by $\rightarrow_{rf}$. Since $\rightarrow_{rf};\rightarrow_{rf^{-1}}$ is the identity function, all appearances of $\rightarrow_{rf^{-1}}$ in

the cycle can be eliminated by simply removing each $\rightarrow_{rf}; \rightarrow_{rf^{-1}}$ pair in the cycle. This leaves a cycle with only $\rightarrow_{rf}$, $<_{co}$, and $\rightarrow_{fr}$. If there are any reads in such a cycle, then by similar logic as above, the incoming relation must be $\rightarrow_{rf}$ and the outgoing relation must be $\rightarrow_{fr}$, but this pair is equivalent to a $<_{co}$ edge between writes alone. Repeating such a transformation produces a cycle consisting of only $<_{co}$. Since by hypothesis there is at least one such $<_{co}$ edge, this is a contradiction. □

D Alloy Model for Empirical Validation

Figure 8 shows the Alloy model used for validation.

```
// Model of memory
sig Address {}
abstract sig Event {
  po: lone Event, ppo: set Event, mo: set Event, address: one Address }
sig Read extends Event {}
sig Write extends Event { rf: set Read }
fun po_loc : Event->Event { ^po & address.~address }
fact { acyclic[po] }
fact { rf.~rf in iden }
fact { total[mo, Event] } // definition of total omitted for space
fact { (Write <: po_loc :> Write) + (Read <: po_loc :> Write)
  + (Read <: (po_loc - (po_loc.po_loc)) :> Read) in ppo }

// GAM axioms
fun candidates[r: Read] : set Write {
  (r.~mo & Write & r.address.~address) // writes preceding r in <mo
  + (r.^~po & Write & r.address.~address)} // writes preceding r in <po
pred InstOrder { ppo in mo }
pred LoadValue { all w: Write | all r: Read |
  w->r in rf <=> w in (let c = candidates[r] | c - c.~mo)} // i.e., max_<mo
pred GAM { InstOrder and LoadValue }

// COM axioms
fun rfe : Write->Read { rf - (^po + ^~po) }
fun co : Write->Write { Write <: ((address.~address) & mo) :> Write }
fun fr : Read->Write { ~rf.co + ((Read - Write.rf) <: address.~address :> Write) }
pred SC_per_Location  { acyclic[rf + co + fr + po_loc] }
pred Causality { acyclic[rfe + co + fr + ppo] }  // def. of acyclic omitted for space
pred COM { SC_per_Location and Causality }

// Equivalence Checks
check gam_com { GAM => COM } for 7
check com_gam { rfe + co + fr + ppo in mo => COM => GAM } for 7
```

Figure 8: Comparing GAM and COM in Alloy

E Completeness: GAM-I2E Axiomatic Model ⊆ GAM-I2E Operational Model

Theorem 10. *GAM-I2E axiomatic model ⊆ GAM-I2E operational model.*

Proof. The goal is that for any legal axiomatic relations $\langle <_{po}, <_{mo}, \rightarrow_{rf} \rangle$ (which satisfy the GAM-I2E axioms), we can run the GAM-I2E operational model to simulate the same program behavior. In each step of the simulation, we first decide which rule to fire in the operational model based on the current state of the operational model and $<_{mo}$, and then we fire that rule. Here is the algorithm to determine which rule to fire in each simulation step:

1. If in the operational model there is a processor whose next instruction is not a load, we fire an Execute-Reg-Branch or Execute-Store-Fence rule to execute that instruction in the operational model.
2. If the above case does not apply, and in the operational model there is a fence that can be dequeued from the local buffer, then we fire the Dequeue-Fence rule to dequeue that fence in the operational model.
3. If neither of the above cases applies, and in the operational model there is a store S in the local buffer of a processor, and S can be dequeued from the local buffer (i.e., the guard for the Dequeue-Store rule is true), and all stores before S in $<_{mo}$ are already in $<_{mo\text{-}i2e}$, then we fire a Dequeue-Store rule to dequeue S in the operational model.
4. If none of the above cases applies, then in the operational model there must be a processor such that the next instruction of the processor is a load L, and L can be executed (i.e., the guard for the Execute-Load rule is true), and all stores before L in $<_{mo}$ are already in $<_{mo\text{-}i2e}$. We fire an Execute-Load rule to execute L in the operational model. In the Execute-Load rule of L, we insert L into $<_{mo\text{-}i2e}$ such that for any instruction I already in $<_{mo\text{-}i2e}$, if $I <_{mo} L$ then $I <_{mo\text{-}i2e} L$, otherwise $L <_{mo\text{-}i2e} I$.

After each step of the simulation, we keep the following invariants:

1. The execution order on each processor is a prefix of the $<_{po}$ of that processor.
2. The result of each executed instruction is the same as that in $<_{po}$.
3. The store read by each executed load is the same as that indicated by the $\rightarrow_{rf}$ edges.
4. The simulation cannot get stuck.
5. For two memory instruction I_1 and I_2, if $I_1 <_{mo\text{-}i2e} I_2$ in the operational model, then $I_1 <_{mo} I_2$ in the axiomatic relations.
6. The order of all stores in $<_{mo\text{-}i2e}$ is a prefix of the order of all stores in $<_{mo}$.

The first two induction invariants imply that *before* each simulation step, the following properties hold for each processor i (assuming the next instruction of the processor is I):

1. $<_{po\text{-}i2e}$ is a prefix of $<_{po}$ (of processor i) up to I (including I).
2. For any instructions $I_1 <_{ppo} I_2$ from processor i, if I_1 and I_2 are not ordered after I in $<_{po}$ (i.e., I_2 may be equal to I), then $I_1 <_{ppo\text{-}i2e} I_2$.
3. For any instructions I_1 and I_2, if $I_1 <_{ppo\text{-}i2e} I_2$, then $I_1 <_{ppo} I_2$.

Now we examine each case in the simulation algorithm and prove that all invariants hold:

1. We execute a non-load instruction: trivial.
2. We dequeue a fence from the local buffer: trivial.
3. We dequeue a store S from the local buffer: In this case, we need to verify invariants 5 and 6. Invariant 6 is trivial, because all stores older than S in $<_{mo}$ are already in $<_{mo\text{-}i2e}$ (as required by case 3 in the algorithm). We now consider invariant 5. For each instruction I already in $<_{mo\text{-}i2e}$ at the dequeue time, I must be added to $<_{mo\text{-}i2e}$ by case 3 or 4 in the simulation algorithm. Since these two cases require that every store older than I in $<_{mo}$ to be present in $<_{mo\text{-}i2e}$, S cannot be older than I in $<_{mo}$, i.e., $I <_{mo} S$.
4. We execute a load L: We first need to verify invariant 4, i.e., we are able to find such an L that satisfies the requirements in case 4 of the simulation algorithm. We prove this by contradiction, i.e., such L cannot be found. In this case, the next instruction of every processor is a load. We examine why the next instruction L_1 (which is a load) of processor 1 does not satisfy the requirements of case 4 of the simulation algorithm. There are two possibilities:
   (a) There is a store $S_2 <_{mo} L_1$ but S_2 is not yet in $<_{mo\text{-}i2e}$.
   (b) The guard of the Execute-Load rule for L_1 is false. We backtrack which instruction is stalling L_1. There must exist an instruction I_1 in the local buffer of processor 1 which is ordered before L_1 in $<_{ppo\text{-}i2e}$. If I_1 is a fence, then I_1 cannot be dequeued because there is another instruction $I_2 <_{mo\text{-}i2e} I_1$ in the local buffer. We keep doing this until we find a store, i.e., $I_k <_{ppo\text{-}i2e} I_{k-1} <_{ppo\text{-}i2e} \cdots <_{ppo\text{-}i2e} I_1 <_{ppo\text{-}i2e} L_1$, where $I_1 \cdot I_k$ are all in the local buffer of processor 1, $I_1 \cdots I_{k-1}$ are fences, and I_k is a store. According to property 3, $I_k <_{ppo\text{-}i2e} L_1 \Rightarrow I_k <_{ppo} L_1 \Rightarrow I_k <_{mo} L_1$.

   In either case, we find a store $S_2 <_{mo} L_1$, and S_2 is not in $<_{mo\text{-}i2e}$. Now we consider why S_2 is not in $<_{mo\text{-}i2e}$, and there are two possibilities:
   (a) S_2 is not executed yet: Assume S_2 is in processor i in $<_{po}$. The next instruction to execute in the

processor of S_2 in the operational model must be a load L_3. According to invariant 1, since S_2 is not in the prefix of $<_{po}$ of processor i up to L_3, we have $L_3 <_{po} S_2 \Rightarrow L_3 <_{ppomf} S_2 \Rightarrow L_3 <_{mo} S_2$. Following the previous argument, L_3 cannot be executed because of a store S_3 which is before L_3 in $<_{mo}$ but is not in $<_{mo\text{-}i2e}$. That is, $S_3 <_{mo} S_2$ and S_3 is not in $<_{mo\text{-}i2e}$.

(b) S_2 is the local buffer of processor i: There are two possible reasons that stops S_2 from being dequeued:
   i. There is a store $S_3 <_{mo} S_2$ and S_3 is not in $<_{mo\text{-}i2e}$.
   ii. The guard of the Dequeue-Store rule is false. Using the previous argument, there must be a store $S_3 <_{mo} S_2$, and S_3 is in the local buffer.

In all cases, we can find a store $S_3 <_{mo} S_2$, and S_3 is not in $<_{mo\text{-}i2e}$. Since the simulation algorithm is assumed to get stuck, we can keep doing this, and find $S_k <_{mo} S_{k-1} <_{mo} \cdots <_{mo} S_1 <_{mo} L_1$, where $S_1 \ldots S_k$ are all stores that are not in $<_{mo\text{-}i2e}$, and k can be infinitely large. However, there can only be finite number of stores before L_1 in $<_{mo}$. Therefore, we must be able to find a load L that satisfies the requirements of case 4 of the simulation algorithm.

We also need to verify that L can indeed be inserted into $<_{mo\text{-}i2e}$ as instructed in case 4 of the simulation algorithm. Since both $<_{mo}$ and $<_{mo\text{-}i2e}$ are total orders, invariant 5 ensures that we can cut $<_{mo\text{-}i2e}$ into two parts, i.e., one part is before L in $<_{mo}$ and the other part is after L in $<_{mo}$. Then we can simply place L at the cutting point of $<_{mo\text{-}i2e}$. This also ensures that invariant 5 will still hold after this step of simulation.

Finally we need to show that invariant 3 still holds. Assume L is from processor i, loads address a, and reads from store S in the Execute-Load rule. Consider where S resides when we fire the Execute-Load rule:

(a) S is in the local buffer of processor i: Since all stores $<_{mo} L$ are already in $<_{mo\text{-}i2e}$, S does not precede L in $<_{mo}$, i.e., $L <_{mo} S$. Invariant 1 implies that $S <_{po} L$. Therefore the Load-Value axiom can only select stores $<_{po} L$ as the source for the load result. Since all stores for the same address in the same processor are ordered by $<_{pposa}$ and thus $<_{mo}$, the Load-Value axiom will pick the youngest store in $<_{po}$ among all stores that is before L in $<_{po}$. Since S is the most recently executed store for a in processor i, invariant 1 ensures that S is the store picked by the Load-Value axiom.

(b) S is already in $<_{mo\text{-}i2e}$: In this case, the local buffer of processor i cannot have any store for address a. Invariant 1 says that for any store S' for a which is ordered before L in $<_{po}$, S' must have been executed in processor i in the operational model. Therefore, S' must be already in $<_{mo\text{-}i2e}$. The way we find L ensures that for any store S'' for a that is ordered before L in $<_{mo}$, S'' must be in $<_{mo\text{-}i2e}$. Thus, all stores that are visible to L according to the Load-Value axiom are all in $<_{mo\text{-}i2e}$ now. Invariants 5 and refi2e:mo-prefix both say that the orderings between all such S' and S'' are the same in $<_{mo\text{-}i2e}$ and $<_{mo}$. Since the Execute-Load rule uses the same way as the Load-Value axiom to determine the load value, invariant 3 must hold.

□