摘要

*警告：该PDF由GPT-Academic开源项目调用大语言模型+Latex翻译插件一键生成，版权归原文作者所有。翻译内容可靠性无保障，请仔细鉴别并以原文为准。项目Github地址 https://github.com/binary-husky/gpt_academic/。当前大语言模型：qwen-plus，当前语言模型温度设定：1。为了防止大语言模型的意外谬误产生扩散影响，禁止移除或修改此警告。

GPT-Academic程序无法找到本文的摘要部分。

A Better x86 Memory Model: x86-TSO

Scott Owens Susmit Sarkar Peter Sewell

剑桥大学

http://www.cl.cam.ac.uk/users/pes20/weakmemory

摘要. 实际的多核处理器并不提供大多数语义和验证工作所假设的顺序一致性内存。相反，它们具有宽松的记忆模型，这些模型通常用模糊的散文描述，导致广泛的理解混乱。这些都是机械化形式化的首要目标。在先前的工作中，我们生成了一个严格的$x86-CC$模型，形式化了当时的Intel和AMD架构规范，但这些模型被发现与实际硬件不一致，并且可能过于弱，无法在其上编程。本文讨论这些问题，并提出一个新的$x86-TSO$模型，该模型不存在上述问题，在HOL4中形式化。我们认为它在实际处理器中是有效的，更好地反映了供应商的意图，并且更适合编程。我们给出了两个等价的x86-TSO定义：一个基于本地写缓冲区的直观操作模型，以及一个类似于SPARCv8的公理总存储器排序模型。两者都适应于处理x86特有的特性。我们在memevents工具中实现了公理模型，该工具计算测试程序的所有有效执行集，并为了增加信心，直接验证这些执行的见证者，使用从第三个更算法化且等效的定义版本提取的代码。

1 Introduction

大多数关于并发程序语义和验证的先前研究假设了顺序一致性：多个线程对共享内存的访问发生在全局时间线性顺序中。然而，真实的多处理器系统却包含了多种性能优化。这些优化对于单线程程序通常是不可见的，但对于并发代码的行为则有可观察的后果。例如，在标准的 Intel 或 AMD x86 处理器上，给定两个内存位置 x 和 y（初始值为 0），如果两个处理器 proc: 0 和 proc: 1 分别将 1 写入 x 和 y，然后从 y 和 x 读取，如下程序所示，这两个处理器在同一执行过程中都可能读取到 0。

iwp2.3.a/amd4	proc:0	proc:1
poi:0	MOV [x] ← $1	MOV [y]←$1
poi:1	MOV EAX ← [y]	MOV EBX←[x]
Allow: $0 : \mathrm{EAX} = 0 \wedge 1 : \mathrm{EBX} = 0$		

可以将此视为写入缓冲的明显后果：每个处理器实际上有一个FIFO缓冲区，用于待处理的内存写入（以避免在写入完成时需要阻塞），因此对y和x的读取可以在写入从缓冲区传播到主内存之前发生。这种优化破坏了顺序一致性的幻觉，使得在这种抽象级别上，无法用一种直观的全局时间概念来进行推理。

为了描述程序员可以依赖的内容，处理器供应商会记录架构。这些是宽松的规范，声称覆盖了一系列过去和未来的实际处理器，应该揭示足够的信息以便有效编程，但不会过度限制未来处理器设计。然而，在实践中，它们是非正式的散文文档，例如Intel 64和IA-32架构软件开发手册[2]和AMD64架构程序员手册[1]。非正式散文是一种不适合描述微妙性质的松散规范的媒介，并且如我们将在2看到的那样，这样的文档通常是模糊的，有时是不完整的（不足以在此之上编程），有时甚至是不健全的（相对于实际处理器）。此外，不能对这种含糊的规范进行程序测试（只能在特定的实际处理器上运行程序），也不能将它们用作测试处理器实现的标准。

因此，架构规范是严格机械化形式化的主要目标。在以前的工作[19]中，我们引入了一个严格的x86-CC模型，在HOL4[11]中形式化，基于当时当前Intel和AMD文档中的非正式因果一致性描述。不幸的是，那些以及因此x86 – CC被证明是不健全的，禁止了实际处理器表现出的一些行为。

在本文中，我们描述了一个新的模型，x86-TSO，也在HOL4中进行了形式化。据我们所知，x86-TSO是健全的，足够强大以在此之上编程，并且总体上符合供应商的意图。我们在3.1中给出了模型的一个抽象机定义，在3.2中给出了一个公理版本。为了弥补形式化的主要缺点——即它可能会使规范变得不那么广泛可访问——我们对数学定义进行了广泛的注释。为了探索模型的后果，我们在我们的memevents工具中实现了手工编码，该工具可以探索上述类型的litmus测试示例的所有可能执行情况，为了更大的信心，我们从HOL4公理定义中提取了经过验证的执行检查器，详见4。我们在5中讨论了相关工作，并在6中做了总结。

2 Many Memory Models

我们首先回顾最近的Intel和AMD文档中非正式散文式的规范描述。这些规范有几个版本，有些差异很大；我们将它们相互对比，并与我们知道的实际处理器行为进行对比。

2.1 pre-IWP (before Aug. 2007)

早期的 Intel SDM 版本（例如，rev-22，2006年11月）提供了一个称为“处理器排序”的非正式散文模型，但没有给出任何示例支持。很难对这一描述给出精确的解释。

2.2 IWP/AMD64-3.14/x86-CC

2007年8月，英特尔白皮书[12]（IWP）提供了一个更为精确的模型，其中包含8个非正式文体的原理，并由10个示例（称为litmus测试）支持。这几乎未经修改地被纳入了后来的英特尔SDM修订版（包括第26-28版），而AMD则提供了类似的尽管不完全相同的描述和测试[1]。这些模型本质上是因果一致性模型[4]。它们允许独立的读取者以不同的顺序看到独立的写入操作（由不同处理器写入不同地址），如下所示（IRIW，另见[6]），但要求在某种意义上尊重因果关系：“P5. 在多处理器系统中，内存排序遵守因果关系（内存排序尊重传递可见性）”。

amd6	proc:0	proc:1	proc:2	proc:3
poi:0	MOV [x] ← $1	MOV [y]←$1	MOV EAX←[x]	MOV ECX←[y]
poi:1			MOV EBX←[y]	MOV EDX ← [x]
Final: $2{:}EAX{=}1 \wedge 2 : EBX = 0 \wedge 3 : ECX = 1 \wedge 3 : EDX = 0$				
cc : Allow; tso : Forbid				

这些非正式的规范是我们x86-CC模型的基础，其中的关键问题是给出这种“因果关系”的合理解释。除此之外，这些非正式规范相对来说是明确的——但它们最终存在两个严重的缺陷。

首先，对程序员来说，这些规范可能相当弱。特别是，它们允许上述的IRIW行为，但是，在对最强的x86内存屏障MFENCE做出合理假设的情况下，添加MFENCE并不能恢复顺序一致性[19, 2.12]。在这里，规范似乎比实现的处理器的行为宽松得多：据我们所知，并经过一些测试，IRIW在实际中是不可观察的。这表明某些JVM实现依赖于这一事实，如果仅假设IWP/AMD64-3.14/x86-CC架构[9]，这些实现将是不正确的。

其次，更严重的是，它们相对于当前的处理器是不健全的。Paul Loewenstein [14]提供的以下n6示例显示了一种在x86-CC中不允许的行为，任何我们能理解的IWP和AMD64-3.14的解释也不允许这种行为（例如，在Intel Core 2 Duo上可以观察到）。

n6	proc:0	proc:1
poi:0	MOV [x] ← $1	MOV [y]←$2
poi:1	MOV EAX ← [x]	MOV [x] ← $2
poi:2	MOV EBX←[y]	
Final: $0:EAX = 1 \wedge 0:EBX = 0 \wedge [x] = 1$		
cc : Forbid; tso : Allow		

为了理解为什么具有FIFO写缓冲区的多处理器可能允许这种情况，假设首先是proc:1 写入 $[y] = 2$ 被缓冲，然后 proc:0 缓冲其写入 $[x] = 1$，从其自身的写缓冲区读取 $[x] = 1$，并从主存中读取 $[y] = 0$，随后 proc:1 缓冲其 $[x] = 2$ 写操作，并将其缓冲的 $[y] = 2$ 和 $[x] = 2$ 写操作刷新到内存，最后 proc:0 将其 $[x] = 1$ 写操作刷新到内存。

2.3 Intel SDM rev-29 (Nov. 2008)

最近的 ×86 供应商规范的更改是在 Intel SDM 的第 29 版（第 30 版基本上相同，我们被告知未来将有类似内容的 AMD 规范修订版）。这一版本仍然采用与先前版本类似的非正式散文风格，并通过 Litmus 测试支持，但与 IWP/AMD64-3.14/x86-CC 有显著不同。首先，上述 IRIW 最终状态被禁止 [示例 7-7, rev-29]，之前的相干性条件："P6. 在多处理器系统中，对同一位置的存储有一个全序"已经被替换为："P9. 任何两个存储操作对于除执行这些存储操作的处理器以外的其他处理器来说，都是以一致的顺序可见的"。

其次，现在包含了内存屏障指令，包括"P11. 读取操作不能超过 LFENCE 和 MFENCE 指令"和"P12. 写入操作不能超过 SFENCE 和 MFENCE 指令"。

第三，来自同一个处理器的写入现在被明确地排序（我们认为这在 IWP "P2. 存储操作不与其他存储操作重新排序"中是隐含的）："P10. 单个处理器的写入操作对所有处理器来说都以相同的顺序被观察到"。

该规范似乎解决了上述 n6 行为的不健全性，但不幸的是，它仍然有问题。第一个问题是，如何解释 P5 中使用的"因果关系"。第二个问题是弱点：新的 P9 没有提到两个处理器自身（或其中一个处理器和其他一个）对两个存储操作的观察。在缺乏此类保证的模型上进行编程将是困难的。以下 n5 和 n4 示例展示了潜在的问题。这些最终状态在 x86-CC 中是不允许的，如果它们被任何合理的实现允许，我们会感到惊讶（纯写缓冲区实现中不允许这些最终状态）。我们在实际处理器上没有观察到这些现象；然而，rev-29 似乎允许它们。

n5	proc:0	proc:1
poi:0	MOV [x] ← $1	MOV [x] ← $2
poi:1	MOV EAX ← [x]	MOV EBX←[x]
Forbid: $0:EAX = 2 \wedge 1:EBX = 1$		

n4	proc:0	proc:1
poi:0	MOV EAX←[x]	MOV ECX←[x]
poi:1	MOV [x] ← $1	MOV [x] ← $2
poi:2	MOV EBX←[x]	MOV EDX ← [x]
Forbid: 0:EAX=2 0̂:EBX=1		

总结关键的试金石差异，我们有：

	IWP/AMD64-3.14/x86-CC	rev-29	actual processors
IRIW	allowed	forbidden	not observed
n6	forbidden	allowed	observed
n4/n5	forbidden	allowed	not observed

还有许多非差异：在所有三种情况下行为都一致的测试。测试细节在此省略，但可以在扩展版本[16]或[19]中找到。它们包括其他9个IWP测试，说明除了iwp2.3.a/amd4（§1）所示的重排序之外，其他各种加载和存储重排序是不可能的；AMD的MFENCE测试amd5和amd10；以及其他几个测试。

3 The x86-TSO Model

鉴于这些非正式规范的问题，我们不能通过形式化它们所包含的“原则”来生成有用的严格模型（就像我们尝试对 x86-CC [19] 所做的那样）。相反，我们必须建立一个合理的模型，该模型与给定的试金石测试、观察到的处理器行为以及我们对程序员需求和供应商意图的了解相一致。

写缓冲区是可观察的（如 iwp2.3.a/amd4 和 n6）而 IRIW 不是，加上其他禁止许多其他重新排序的测试，这强烈表明，除了写缓冲区外，所有处理器都共享相同的内存视图（与 x86-CC 形成对比，在 x86-CC 中，每个处理器都有一个独立的视图顺序）。这大致类似于 SPARC 总线存储顺序 (TSO) 内存模型 [20, 21] ，后者基本上是对写缓冲多处理器行为的公理描述。此外，尽管没有使用“TSO”这一术语，非正式讨论表明这符合修订版 29 非正式规范背后的意图。因此，这里我们提出一个严格的 x86-TSO 模型，包括两个等价定义。

第一个定义在 3.1 中，是一个具有显式写缓冲区的抽象机。第二个定义在 3.2 中，是一个公理模型，该模型根据内存顺序和读取来源映射定义有效执行。在这两种定义中，我们都处理具有多个内存访问的 x86 CISC 指令、带有锁定指令（CMPXCHG、LOCK；INC 等）、可能无限期计算以及通过寄存器传递依赖关系的情况。结合我们早期的指令语义，x86-TSO 因此定义了程序的完整语义。抽象机传达了 x86-TSO 在编程层面的操作直观，而公理模型支持基于约束的示例程序推理，例如，通过我们在 4 中的 memevents 工具进行推理。

x86-TSO 的预期范围，如同 x86-CC 模型一样，涵盖了典型的用户代码和大多数内核代码：使用一致写回内存的程序，不包括异常、未对齐或混合大小的访问、“非临时”操作（如 MOVNTI）、自修改代码或页表更改。

基本类型：动作、事件和事件结构如同我们早期的工作，程序（任何特定执行）的动作被抽象为一组事件（附带额外数据）称为事件结构。一个事件表示对特定内存地址或寄存器的特定值进行读取或写入，或者执行一个屏障。我们早期的工作包括了一个由汇编语言程序生成的事件结构集的定义。对于任何此类事件结构，内存模型（在此处为 x86-TSO，在其他地方为 x86-CC）定义了什么是有效执行。

详细而言，每条机器码指令可能有多个与其相关的事件：事件通过指令 ID iiid 来索引，该 ID 标识事件发生的处理器及其在指令流中的位置（即程序顺序索引，poi）。事件还具有事件 ID eiid 以识别指令内的不同事件（允许存在多个其他方面完全相同的事件）。事件结构指示指令的一个事件如何依赖于同一指令的另一个事件，这是一种部分顺序，称为内部因果关系。事件结构还记录了哪些事件同时发生在锁定指令中，包括一组原子性数据，即一组必须原子地一起发生的（不相交且非空）事件集。

用 HOL 表达这一点，我们通过类型 proc = num 对处理器进行索引，将地址和值类型定义为 32 位字，并将位置定义为内存地址或特定处理器的寄存器：

location = LOCATION_REG of proc ’reg

LOCATION_MEM of address

模型参数化为 x86 寄存器的类型 ’reg，可以理解为普通寄存器 EAX、EBX 等、指令指针 EIP 和状态标志的枚举。为了标识执行中的指令实例，我们需要指定其处理器及其程序顺序索引。

iiid = ⟨[proc : proc; | poi : num]⟩

动作要么是在某个位置读取或写入值，要么是一个屏障：

dirn = R | W

barrier = LFENCE | SFENCE | MFENCE

action = ACCESS of dirn (’reg location) value | BARRIER of barrier最终，一个事件具有指令实例ID、事件ID（类型为 eiid = num，每个 iiid 唯一）和一个动作：

event = ⟨ eiid : eiid; iiid : iiid; action : action

事件结构 E 包括一组处理器、一组事件、指令内的因果关系以及一个部分等价关系（PER），后者捕捉必须原子性发生的事件集。所有这些都需满足一些我们在这里省略的良构条件。

event_structure = { procs : proc set;

events : ('reg event)set;

intra_causality : ('reg event)reln;

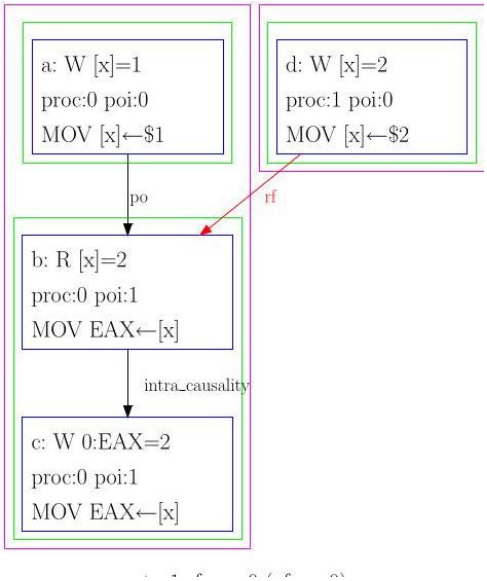
atomicity : ('reg event)set set(0)

示例我们下面展示了一个非常简单的事件结构，用于以下程序：

tso1	proc:0	proc:1
poi:0 poi:1	MOV [x] ← $1 MOV EAX ← [x]	MOV [x] ← $2

有四个事件——内部（在线版本中为蓝色）框。事件ID按字母顺序排列，例如a、b、c、d等。我们还显示了导致每个事件的汇编指令，例如MOV [x] ← $1 ，虽然这并不是事件结构的正式部分。

请注意，事件包含具体的值：在这个特定的事件结构中，有两次对 x 的写操作，值分别为1和2，一次读[x]的操作，值为2，以及一次将值2写入proc:0的EAX寄存器的操作。稍后我们将展示该程序的两种有效执行情况，一种对应于这个事件结构，另一种对应于另一个事件结构（注意某些事件结构可能没有有效的执行）。在图中，每个处理器的指令被聚类在一起，放入最外层（洋红色）的框中，并且它们之间有程序顺序（po）边，每个指令的事件也被聚类在一起，放入中间（绿色）的框中，根据需要添加内部因果关系边——在这里，MOV EAX ← [x] 中，EAX的写操作依赖于 x 的读操作。



3.1 The x86-TSO Abstract Machine Memory Model

为了理解我们的x86-TSO机器模型，考虑一个理想化的x86多处理器系统，该系统被分为两个部分：其内存和寄存器状态（所有处理器的总合），以及系统的其余部分（所有处理器核心的其他部分）。我们的抽象机是一个标记转换系统：一组状态，由 s 表示，以及一个转换关系 $s \xrightarrow{l} s'$ 。抽象机状态 s 建模了第一个部分的状态：多处理器系统的内存和寄存器状态。机器通过同步标签 l （抽象机的接口）与系统的其余部分交互，这些标签包括寄存器和内存读写。在图1中，状态 s 对应于虚线内的机器部分，而标签 l 对应于穿越虚线边界的数据通信。

应该将这台机器视为与处理器内核（不包括它们的寄存器/内存子系统）并行运行，按照程序顺序执行指令流；后者的数据由事件结构提供。这种划分并不直接对应于任何现实的x86实现的微架构，在现实的x86实现中，内存和寄存器将由各种缓存等复杂机制分别实现。然而，它对于描述编程模型非常有用且足够，这是架构描述的正当业务。它还支持与我们公理性内存模型的精确对应。更详细地说，标签 l 是值的集合，这些值包括：

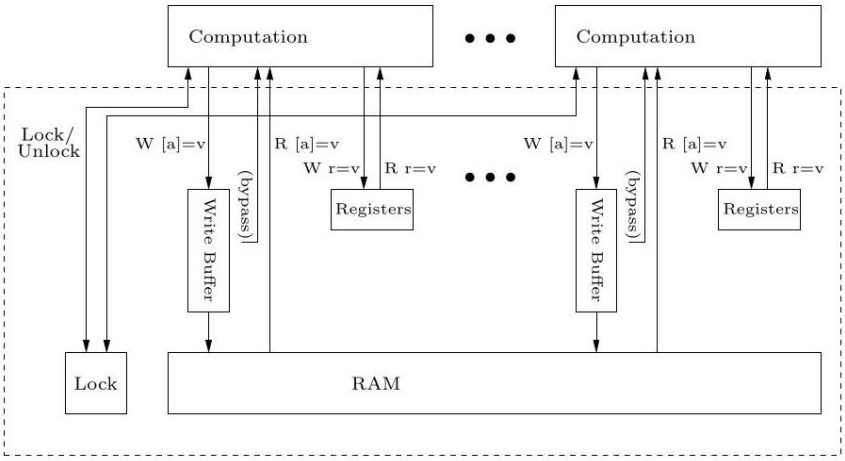


图1. 抽象机

HOL 类型：

标签 = TAU | EVT of 进程 ('reg 动作) | LOCK of 进程 | UNLOCK of 进程

- TAU，表示机器的内部动作；
- EVT pa ，其中 a 是由处理器 p 执行的动作（如上定义，可以是带有其值的内存或寄存器读写操作，或一个屏障）；
- LOCK p ，表示处理器 p 开始执行一条 LOCK'd 指令；或
- UNLOCK p ，表示 p 执行的 LOCK 指令结束。

（请注意，此接口中没有特定于任何特定内存模型的内容。）x86-TSO 机器的状态是记录，具有字段 R ，为每个处理器上的每个寄存器提供一个值；M ，为每个共享内存位置提供一个值；B，建模为每个处理器的写缓冲区，作为地址/值对的列表；以及 L，这是一个全局锁，如果 p 持有锁，则为 SOME p，否则为 NONE。HOL 类型如下。

machine_state = {R : proc → ' reg → value option; (* 每个处理器的寄存器 *)

M : address → value option; (* 主内存 *)

B : proc → (address#value)list; (* 每个处理器的写缓冲区 *)

L : proc option(* 哪个处理器持有锁 *)]

x86-TSO 机器的行为，即转换关系 $s \xrightarrow{l} s'$，由图2中的规则定义。这些规则使用两个辅助定义：如果处理器 p 在机器状态 s 中持有锁或没有处理器持有锁，则它不处于阻塞状态；如果有缓冲区 b 中没有待处理的写入到地址 a，即在 b 中没有 (a, v) 对。非正式地重述这些规则：

1. 如果 p 不被阻塞，没有缓冲到 a 的写入，并且内存确实包含 a 处的 v，则 p 可以从地址 a 的内存中读取 v；

从内存读取 not_blocked $sp \wedge (s.Ma =$ SOME $v) \wedge$ no_pending $(s.Bp)\, a$

S EVT p (ACCESS R (LOCATION_MEM a) v)

从写缓冲区读取

not_blocked $sp \wedge (\exists b\_1 b\_2.\, (s.Bp = b\_1 +\!+ [(a,v)) +\!+ b\_2) \wedge$ no_pending $b\_1 a)$

$s \xrightarrow{\text{EVT } p \text{ (Access } R \text{ (LOCATION\_MEM } a) v\ )} s$

从寄存器读取

($s.Rpr =$ SOME v)

Evt p (Access R (LOCATION _ REG pr) v)

向写缓冲区写入

T

$s \xrightarrow{\text{EVT } p \text{ (Access W (LOCATION\_MEM } a\ )\ v\ )}$

$s \oplus \langle [B := s \cdot B \oplus (p \mapsto [(a,v)] +\!+ (s \cdot Bp))] \rangle$

从写缓冲区向内存写入

not_blocked $sp \wedge (s.Bp = b +\!+ [(a,v)])$

$s \xrightarrow{\text{TAU}} s \oplus \{M := s \cdot M \oplus (a \mapsto \text{SOME } v)\,; B := s \cdot B \oplus (p \mapsto b)\} \rangle$

向寄存器写入**T**

EVT p (访问 W (位置注册 pr))

$s \oplus \{R := s \cdot R \oplus (p \mapsto ((s \cdot Rp) \oplus (r \mapsto \text{SOME } v)))\}$

障碍

$(b =$ MFENCE) $\Rightarrow (s.Bp = [])$

$s \xrightarrow{\text{EVT } p \text{ (BARRIER } b\ )} s$

锁

$(s.L =$ NONE) $\wedge (s.Bp = [])$

$s \xrightarrow{\text{LOCK } p} s \oplus \langle L := \text{SOME } p] \rangle$

解锁

图2. x86-TSO 机器行为

2. 如果处理器 p 未被阻塞并在其写缓冲区中有最新的写入值 v 到地址 a，则 p 可以从其写缓冲区中读取 v；

3. 处理器 p 随时可以从其寄存器 r 中读取存储的值 v；

4. 处理器 p 随时可以将值 v 写入其写缓冲区中的地址 a；

5. 如果处理器 p 未被阻塞，则它可以悄悄地将其写缓冲区中最旧的写操作出列到内存；

6. 处理器 p 随时可以将值 v 写入其任意一个寄存器 r；

7. 如果处理器 p 的写缓冲区为空，则可以执行 MFENCE（因此只有在所有写操作都已出列之后，MFENCE 才能继续，模拟缓冲区的刷新）；LFENCE 和 SFENCE 可以随时发生，因此它们是无操作指令；

8. 如果锁未被持有，并且处理器 p 的写缓冲区为空，则可以开始锁定指令；

9. 如果处理器 p 持有锁，并且其写缓冲区为空，则可以结束锁定指令。

考虑通过机器执行的路径 $s\_0 \overset{l\_1}{\Rightarrow} s\_1 \overset{l\_2}{\Rightarrow} s\_2 \cdots$，这些路径由有限或无限的状态和标签序列组成。我们定义 okMpath 适用的条件是：路径从有效的初始状态开始（例如写缓冲区为空等），并且满足以下进展条件：路径中的每个内存写操作的相应 TAU 转移出现在后面的某个位置。这确保了没有任何写操作会永远停留在缓冲区中。（实际上，我们为下面描述的事件注释机器形式化了 okMpath。）我们强调，这是一台抽象机：我们关注的是它的扩展性行为，即它可以执行的（完成的、有限或无限）标记转移的轨迹（应包括实际实现的行为），而不是其内部状态和转移规则。该机器应该为程序员提供一个好的模型，但可能与实现内部结构几乎没有相似之处。确实，一个现实的设计肯定不会通过全局锁来实现 LOCK 指令，并会有很多其他优化——x86-TSO 模型的力量在于，所有这些优化对程序员来说都是不可见的，除了可能通过性能观察。该机器有几种不同锁定程度的变体，我们猜测它们在观察上是等价的。例如，当

一个处理器持有锁时，可以禁止其他处理器的所有活动，或者不要求在 LOCK 指令开始时刷新写缓冲区。

我们将这台机器与事件结构的关系分为两步，这里总结如下（完整的 HOL 细节可在在线找到 [16]）。首先，我们定义了一个更具有内涵性的事件机：我们用一个事件选项注释每个内存和寄存器位置，记录对该位置的最近一次写事件（如果有），将写缓冲区细化为记录事件列表而非简单的地址/值对，并用相关事件注释标签。其次，我们用谓词 okEpath 将带有注释标签的路径与事件结构关联起来，该谓词在路径为事件结构的合适线性化时成立：路径中的非 TAU/LOCK/UNLOCK 标签与 E 的事件之间存在一一对应关系，路径中标签的顺序与程序顺序和内部因果一致性相符，原子集由 LOCK/UNLOCK 对正确括起。因此，okMpath 描述了根据内存模型的合法路径，而 okEpath 描述了根据事件结构（封装了处理器语义的其他方面）的合法路径。

定理 1. 事件机的注记移除正是上面介绍的机器。[HOL 证明]

3.2 The x86-TSO Axiomatic Memory Model

我们的x86-TSO公理化内存模型基于SPARCv8内存模型规范[20, 21]，但已适应x86，并使用与我们先前的x86-CC模型相同的术语。不熟悉SPARCv8内存模型的读者可以安全地忽略本节中的SPARC特定评论。与SPARCv8 TSO规范相比，我们省略了指令获取（IF）、指令加载（IL）、刷新（F）和存储屏障（S）。前三者专门处理指令内存，而后者仅在SPARC PSO内存模型下有用。为了适应x86程序，我们添加了寄存器和屏障事件，泛化以支持产生多个事件的指令（这些事件由指令内部的因果关系部分排序），并将原子加载/存储对泛化为锁定指令。

如果存在一个执行见证X，对于其事件结构E，则该执行被我们的内存模型允许，且此执行见证是有效的。执行见证包含一个memory_order、一个rfmap以及一个initial_state；本节的其余部分定义了这些内容何时有效。

execution_witness =

([memory_order : ('reg event)reln;

rfmap : ('reg event)reln;

initial_state : (' reg location → value option)])

内存顺序是一个偏序关系，记录了内存事件的全局顺序。它必须是对内存写入的全序关系，并且对应于SPARCv8中的≤关系，受SPARCv8顺序条件的约束（在图中，我们使用mo_non-po_write_write标签表示此顺序中未强制的部分）。

partial_order $\left(<_{X\ .\mathrm{memory\_order}}\right)$ (mem_accesses E)

linear_order $\left(\left(<_{X\ .\mathrm{memory\_order}}\right)\big|_{(\ \mathrm{mem\_writes}\ E)}\right)$(mem_writes E)

初始状态是从位置到值的偏函数。每个读取事件的值必须来自初始状态或写入事件：rfmap（“读取从...”映射）记录了这一点，包含(ew, er)对，其中读取er从写入ew读取。reads_from_map_candidates谓词确保rfmap仅关联具有相同地址和值的对。（严格说来，rfmap是不必要的；涉及它的约束可以直接用内存顺序表述，如SPARCv8所做的那样。然而，我们发现这直观且有用。SPARCv8模型没有初始状态。）

$\forall\,(ew, er) \in$ rfmap. ($er \in$ reads E) $\wedge$ ($ew \in$ writes E) $\wedge$

(loc ew = loc er) $\wedge$ (value_of ew = value_of er)

我们将程序顺序从指令提升为对事件的关系po_iico E，这是指令程序顺序与指令内部因果关系的并集。这大致对应于SPARCv8中的;。然而，intra_causality可能不会关联某些指令中的事件对，因此我们的一般而言，po_iico E对于处理器的事件不是全序关系。

po_strict E =

$\{(e_1, e_2) \mid (e_1$ .iiid.proc $= e_2$ .iiid.proc) $\wedge\, e_1$ .iiid.poi $< e_2$ .iiid.poi $\wedge$

$e_1 \in$ E.events $\wedge e_2 \in$ E.events $\}$

$<_{(\text{ po\_iico } E)} =$ po_strict $E \cup$ E.intra_causality

下面的check_rfmap_written确保rfmap将读取与最近的先前写入相关联。对于寄存器读取，这是程序顺序中最近的写入。对于内存读取，这是在内存顺序或程序顺序中先于读取的所有内存写入中最近的写入（直观地，第一种情况是从已提交的写入中读取，第二种情况是从本地写缓冲区中读取）。check_rfmap_written和re词实现了上述rfmap见证数据的SPARCv8值一致性公理。check_rfmap_initial谓词扩展了这一点以处理初始状态，确保任何不在rfmap中的读取都从初始状态取值，并且该读取在内存顺序或程序顺序中不先于任何写入。

previous_writes E er $<_{\text{order}} =$

$\{ew' \mid ew' \in$ writes $E \wedge ew' <_{\text{order}} er \wedge (\text{loc}\, ew' = \text{loc}\, er)\}$

check_rfmap_written $EX = \forall\, (ew, er) \in (X.rfmap)$.

如果 $ew \in$ mem_accesses E 那么

ew $\in$ maximal_elements (previous_writes E er $\left(<_{X\ .\text{memory\_order}}\right) \cup$

previous_writes E er $\left(<_{(\text{ po\_iico } E)}\right)$)

$(< X$ .memory_order)

否则 (* ew $\in$ reg_accesses E *)

ew $\in$ maximal_elements (previous_writes E er $\left(<_{(\text{ po\_iico } E)}\right)$) $\left(<_{(\text{ po\_iico } E)}\right)$ check_rfmap_initial $EX =$

$\forall er \in$ (reads $E \setminus$ range $X.rfmap$).

$(\exists l$.(loc er $=$ SOME $l) \wedge$ (value_of er $= X$ .initial_state $l)) \wedge$

(previous_writes E er $\left(<_{X\ .\text{memory\_order}}\right) \cup$

previous_writes E er $\left(<_{(\text{ po\_iico } E)}\right) = \{\})$

我们现在进一步约束内存顺序，以确保其遵守程序顺序的相关部分，并且确保 LOCK 指令的内存访问确实原子性地发生。

- 程序顺序包含在内存顺序中，对于内存读取操作之前的一个内存访问（在图中标记为 mo_po_read_access）（SPAR的 LoadOp）：

  $\forall er \in (\text{ mem\_reads } E)\,.\forall e \in (\text{ mem\_accesses } E)\,.$

  $er <_{(\text{ po\_iico } E)} e \Rightarrow er <_{X\ .\text{memory\_order}} e$

  这段文本的中文翻译如下：

  $\forall er \in (\text{ mem\_reads } E)\,.\forall e \in (\text{ mem\_accesses } E)\,.$

  $er <_{(\text{ po\_iico } E)} e \Rightarrow er <_{X\ .\text{memory\_order}} e$

  （对于所有 $er \in$ mem_reads E 和所有 $e \in$ mem_accesses E，如果 er 在 po_iico E 中早于 e，则 er 在 X.memory_order 中也早于 e。）

- 程序顺序包含在内存顺序中，对于一个内存写操作之前的另一个内存写操作（mo_po_write_write）（SPARCv8存储存储）：

  $\forall ew_1 ew_2 \in (\text{ mem\_writes } E)\,.$

  $ew_1 <_{(\text{ po\_iico } E)} ew_2 \Rightarrow ew_1 <_{X\ .\text{memory\_order}} ew_2$

- 程序顺序包含在内存顺序中，对于一个先于内存读取的内存写入，如果它们之间有一个 MFENCE（mo_po_mfence）。（不必将围栏事件本身包含在内存排序中。）

  $\forall ew \in (\text{ mem\_writes } E)\,.\forall er \in (\text{ mem\_reads } E)\,.\forall ef \in (\text{ mfences } E)\,.$

  $\left(ew <_{(\text{ po\_iico } E)} ef \wedge ef <_{(\text{ po\_iico } E)} er\right) \Rightarrow ew <_{X\ .\text{memory\_order}} er$

- 程序顺序包含在内存顺序中，对于任何两个内存访问，其中至少一个是来自锁指令（mo_po_access/lock）：

$\forall e_1 e_2 \in (\text{ mem\_accesses } E).\forall es \in (\text{ E. atomicity }).$
$\big((e_1 \in es \vee e_2 \in es) \wedge e_1 <_{(\text{ po\_iico } E)} e_2\big) \Rightarrow e_1 <_X .\text{memory\_order } e_2$

- LOCK指令的内存访问在内存顺序（mo_atomicity）中以原子方式发生，即，不得有中间的内存事件。此外，锁定的内存访问与其他内存访问之间的所有程序顺序关系都包含在内存顺序中（这是SPARCv8原子性公理的一般化）：

$\forall es \in (\text{ E.原子性 }).\forall e \in (\text{ mem\_accesses } E \setminus es)$.
$(\forall e' \in (es \cap \text{ mem\_accesses } E).e <\_X.$ 内存顺序 $e') \vee$
$(\forall e' \in (es \cap \text{ mem\_accesses } E).e' <\_X.$ 内存顺序 $e)$

为了妥善处理无限执行，我们还要求内存顺序的前缀都是有限的，确保没有极限点，并且，为了确保每个写入最终都能全局生效，不能有一个无限的读取集与任何特定的写入无关，都是在同一个内存位置上进行（这正式化了SPARCv8终止公理）。

finite_prefixes ($<\_X$.内存顺序) (mem_accesses E)
$\forall ew \in (\text{ mem\_writes } E)$.
finite $\{er \mid er \in \text{E.events} \wedge (\text{loc}\, er = \text{loc}\, ew) \wedge$
er $\not<\_X$.内存顺序 ew $\wedge$ ew $\not<\_X$.内存顺序 er $\}$

一个有效执行的最终状态包含了每个内存位置的最后一次写入以及每个寄存器在程序顺序中的最大写入（如果没有这样的写入，则为初始状态）。假设没有指令对同一寄存器进行多个不相关的写入，这是唯一的定义——对于x86指令来说，这是一个合理的属性。

有效_执行 EX 包含上述条件的定义等价于要求 $<\_X$.内存顺序 是线性顺序而不是部分顺序的定义（再次，全部细节在线可查）：

定理 2.

1. 如果 linear_valid_execution EX 那么 valid_execution EX。

2. 如果 valid_execution EX，那么存在一个 $\widehat{X}$，其内存顺序的线性化使得 linear_valid_execution $E\widehat{X}$。[HOL 证明]

解释“不重新排序”或许令人惊讶的是，上述定义并不要求程序顺序包含在内存顺序中，当一个内存写入后跟着对该地址的读取。该定义确实意味着任何这样的读取不能在写入之前推测（通过 check_rfmap_written 来检查，因为这同时考虑了 $<\_($ po_iico $E)$ 和 $<\_X$.内存顺序 ）。然而，如果加入了一个内存顺序边，可能是对修订版29 “P4. 可以将对不同位置的较旧写入重新排序为读取，但不能与同一位置的较旧写入重新排序”的天真的解释，那么模型将会变得更严格：下面的 n7 示例将被禁止，而它在 x86-TSO 上是允许的。我们推测，这将对应于一个（相当奇怪的）机器，即图2中的规则，但没有从写缓冲区读取的规则，在这种情况下，任何处理器在可以从中读取之前都必须将其写缓冲区刷新至（包括）本地写入。

n7	proc:0	proc:1	proc:2
poi:0	MOV [x] ← $1	MOV [y] ← $1	MOV ECX←[y]
poi:1 poi:2	MOV EAX ← [x] MOV EBX←[y]		MOV EDX ← [x]
Allow: 0:EAX=1 ∧ 0:EBX=0 ∧ 2:ECX=1 ∧ 2:EDX=0			

示例我们在图3中展示了前一个示例程序的两种有效执行。在这两种执行中，内存顺序中的事件 proc: 0 Wx = 1 在事件 proc: 1 Wx = 2 之前（加粗的 mo_non-po_write_write_write 边）。在左边的第一种执行中，proc:0 对 x 的读取从内存顺序中的最新写入中读取（加粗的 mo_non-po_write_write 边和 mo_rf 边的组合），即 proc: 1 Wx = 2 。在右边的第二种执行中，proc:0 对 x 的读取从程序顺序中的最新写入中读取，即 proc: 0 Wx = 1 。此示例还说明了一些寄存器事件：指令 MOV EAX ← [x] 产生对 x 的内存读取，随后（在指令内部因果关系中）对 EAX 进行寄存器写入。

3.3 The Machine and Axiomatic x86-TSO Models are Equivalent

为了证明抽象机仅接受有效的执行，我们定义了一个从事件注解路径到线性执行见证的函数 path_to_X，该函数通过按顺序使用 TAU 和内存读取标签中的事件来构建。因此，执行见证中的内存排序与抽象机中读取和写入内存的事件顺序相对应。

定理 3. 对于任何良好的事件结构 E 和事件机路径 path，如果 (okEpath E path) 并且 (okMpath path)，那么 (path_to_X path) 是 E 的一个有效执行。[HOL 证明]

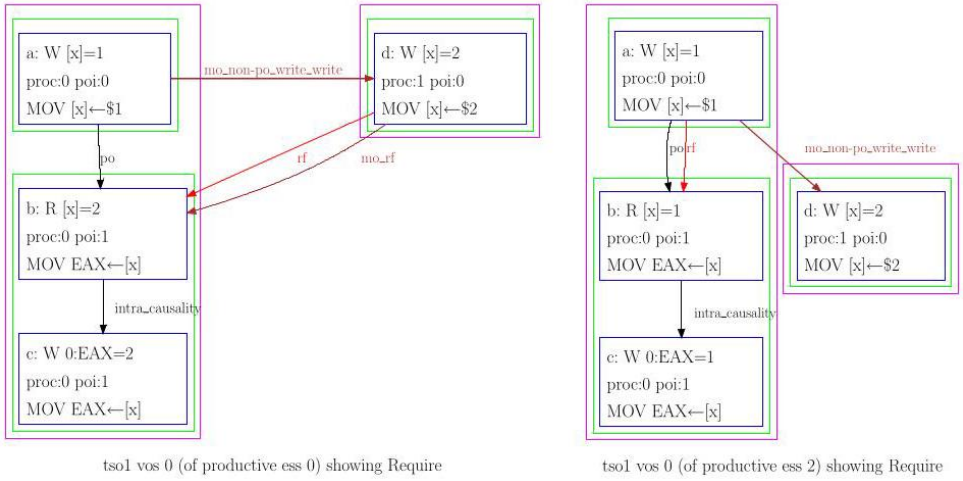


图3. 示例有效执行见证（针对两种不同的事件结构）

为了证明抽象机接受每一个有效的执行，我们首先在HOL中证明了一个引理，该引理展示了任何有效的执行都可以转换成一个满足若干条件的标签的流式线性顺序（在HOL源代码中为label_order），描述了okMpath中的标签。然后我们有：

定理4. 对于任何格式良好的事件结构E 和有效的执行X 对于E，存在某条事件机路径，使得okEpath E 路径和 okMpath 路径，在其中内存读取和写缓冲区刷新都尊重$<_{X.\ \text{memory}\ -\ \text{order}}$。[手工证明，依赖于前面的引理]

4 Verified Checker and Results

为了探讨x86-TSO的影响，我们在我们的memevents工具中实现了公理模型，该工具全面探索候选的执行见证。为增加信心，我们添加了一个验证过的见证检查器：我们定义了事件结构和执行见证的变体，使用列表而不是集合，编写了well_formed_event_structure和valid_execution的算法版本，证明这些在有限情况下等同于我们的其他定义，从HOL中提取出OCaml代码，并将其集成到memevents中。（显然，这仅对正测试提供保证，即那些具有允许最终状态的测试。）

memevents的结果与我们在实际处理器上的观察结果以及供应商的规范相符，对于10个IWP测试、（否定的）IRIW测试、两个MFENCE测试（amd5和amd10）、我们的n2 – n6以及rwc-fenced测试。其余测试（amd3、n1、n7、n8和rwc-unfenced）是“允许”测试，在实践中我们尚未观察到所指定的最终状态。

5 Related Work

关于松弛内存模型的文献十分丰富，但大多数文献都没有涉及x86架构，我们也没有发现任何以前的模型处理了2 中的问题。在此，我们将简要介绍一些最为相关的工作。

存在多个关于弱内存模型的综述，包括Adve和Gharachorloo[3]以及Higham等人[13]的综述；后者用操作风格和公理化风格的形式化了一系列模型，包括TSO模型，并证明了等价性结果。他们的公理化TSO模型与我们的更接近于操作风格，且两者都是理想化的而不是x86-特定的。Burckhardt和Musuvathi[8, 附录A]也给出了TSO模型的操作定义和公理定义，并证明了等价性，但仅限于有限执行。他们的模型处理了内存读写和屏障事件，但缺乏寄存器事件和包含多个事件的原子锁定指令。Hangel等人[10]描述了Sun公司的TSO工具，该工具通过TSO模型检查伪随机生成程序的观察行为。Roy等人[17]描述了一种高效的算法，用于检查执行是否属于TSO模型的一个近似值，该算法被用于Intel的随机指令测试（RIT）生成器。Boudol和Petri[7]提供了一个具有层次写缓冲区的操作模型（从而允许IRIW行为），并证明了无数据竞赛（DRF）程序的顺序一致性。Loewenstein等人[15]描述了一个“金色内存模型”，用于SPARC TSO，这个模型相比于我们在3.1中给出的抽象机更接近于特定实现的微架构，他们使用这个模型进行实现的测试。他们认为，增加的意图细节增加了基于模拟验证的有效性。Saraswat等人[18]也根据局部重新排序定义了内存模型，并证明了DRF定理，但主要关注高级语言。多个研究团队使用了证明工具来驯服这些模型的复杂性，包括Yang等人[22]，他们使用Prolog和SAT求解器探索了公理化的Itanium模型，以及Aspinall和Ševčík[5]，他们使用Isabelle/HOL形式化并发现了Java内存模型中的问题。

6 Conclusion

我们描述了x86-TSO，这是一种针对x86处理器的内存模型，不会受到早期模型的模糊性、弱点或不健全性的影响。它的抽象机定义应该对程序员来说是直观的，其等效的公理定义支持memevents的穷尽搜索，并允许与相关模型进行轻松比较；与SPARCv8的相似性表明x86-TSO足够强大，可以用于编程。在HOL4中的机械化揭示了一些细节上的细微之处，包括我们依赖的一些良好事件结构条件（例如，指令没有内部数据竞争）。我们希望这将澄清x86架构的语义。

致谢我们感谢Luc Maranget在memevents方面的工作，感谢David Christie、Dave Dice、Doug Lea、Paul Loewenstein、Gil Neiger和Francesco Zappa Nardelli的有益建议。我们感谢来自EPSRC资助EP/F036345的资金支持。

References

1. AMD64 架构程序员手册（共3册）。Advanced Micro Devices，2007年9月。修订版 3.14。

2. Intel 64 和 IA-32 架构软件开发者手册（共5册）。Intel Corporation，2008年11月。修订版 29。

3. S. Adve 和 K. Gharachorloo。共享内存一致性模型：教程。IEEE Computer, 29(12):66-76, 1996年12月。

4. M. Ahamad, G. Neiger, J. Burns, P. Kohli, 和 P. Hutto。因果内存：定义、实现和编程。Distributed Computing, 9(1):37-49, 1995。

5. D. Aspinall 和 J. Ševčík。形式化 Java 的无数据竞争保证。Proc. TPHOLs, LNCS, 2007。

6. H.-J. Boehm 和 S. Adve。C++ 并发内存模型的基础。Proc. PLDI, 2008。

7. G. Boudol 和 G. Petri。放松的内存模型：一种操作方法。Proc. POPL, 页码 392-403, 2009。

8. S. Burckhardt 和 M. Musuvathi。松弛内存模型中的有效程序验证。技术报告 MSR-TR-2008-12, Microsoft Research, 2008。会议版本见 Proc. CAV 2008, LNCS 5123。

9. D. Dice。Intel 和 AMD 系统上的 Java 内存模型问题。http://blogs.sun.com/dave/entry/java_memory_model_co 2008年1月。

10. S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, 和 S. Narayanan。TSOtool：一个使用内存一致性模型验证内存系统的程序。Proc. ISCA, 页码 114-123, 2004。

11. HOL 4 系统。http://hol.sourceforge.net/。

12. Intel。Intel 64 架构内存排序白皮书，2007。SKU 318147-001。

13. L.Higham, J.Kawash, 和 N. Verwaal。定义和比较内存一致性模型。PDCS, 1997。完整版本为 TR #98/612/03, U. Calgary。

14. P. Loewenstein。私人通信，2008年11月。

15. P. N. Loewenstein, S. Chaudhry, R. Cypher, 和 C. Manovit。多处理器内存模型验证。Proc. AFM (Automated Formal Methods), 2006年8月。FLoC 工作坊。http://fm.csl.sri.com/AFM06/。

16. S. Owens, S. Sarkar, 和 P. Sewell。更好的 x86 内存模型：x86-TSO（扩展版）。技术报告 UCAM-CL-TR-745, Univ. of Cambridge, 2009。支持材料位于 www. cl. cam. ac. uk/users/pes20/weakmemory/。

17. A. Roy, S. Zeisset, C. J. Fleckenstein, 和 J. C. Huang。快速且通用的多项式时间内存一致性验证。CAV, 页码 503-516, 2006。

18. V. Saraswat, R. Jagadeesan, M. Michael, 和 C. von Praun。内存模型理论。Proc. PPoPP, 2007。

19. S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, 和 J. Alglave。x86-CC 多处理器机器代码的语义。Proc. POPL 2009, 2009年1月。

20. P. S. Sindhu, J.-M. Frailong, 和 M. Cekleov。内存模型的形式化规范。Scalable Shared Memory Multiprocessors, 页码 25-42。Kluwer, 1991。

21. SPARC International, Inc。SPARC 架构手册，第 8 版。修订版 SAV080SI9308。http://www.sparc.org/standards 1992。

22. Y. Yang, G. Gopalakrishnan, G. Lindstrom, 和 K. Slind。Nemos：一个用于内存一致性模型公理和可执行规范的框架。IPDPS, 2004。