用于乐观数据结构即时内存回收的高效硬件原语

Ajay Singh滑铁卢大学
ajay.singh1
@uwaterloo.ca

Trevor Brown滑铁卢大学
trevor.brown@uwaterloo.ca

Michael Spear利
哈伊大学
spear@lehigh.edu

arXiv:2302.12958v1 [cs.DC] 25 Feb 2023

摘要—**安全内存回收（SMR）算法对于防止乐观数据结构中的释放后使用错误至关重要。安全内存回收算法通常会延迟回收以确保安全，并以批次为单位回收对象以提高效率。要兼顾性能与空间效率并非易事。批次较小且频繁尝试回收会带来较高开销，而一次释放较大的批次则可能导致程序长时间中断和较高的内存占用。理想的安全内存回收算法应摒弃批处理，立即回收内存，同时避免产生高昂的回收开销。**

为此，我们提出条件访问**：一组硬件指令，可在乐观数据结构中实现即时回收并降低开销。**条件访问 **利用缓存一致性，使线程能够高效**检测**潜在的释放后使用错误，无需显式进行共享内存通信，也不会引入额外的一致性流量。**

我们在多核模拟器 Graphite 中实现并评估了条件访问。**实验表明，**条件访问**的性能可媲美经过高度优化和精心调优的安全内存回收算法，同时还能实现即时回收。因此，并发数据结构的内存占用与其顺序对应结构相近。**

索引词——**安全内存回收、乐观数据结构、共享内存数据结构**

I. 引言

当前许多乐观数据结构采用的安全内存回收(SMR) [1], [2], [3], [4],[5], [6], [7], [8], [9], [10], [11], [12], [13] 算法会延迟回收，并以批次释放节点，以牺牲空间换取高性能和安全性。当批次过小时，频繁回收带来的额外开销会降低数据结构的吞吐量。另一方面，当批次过大时，虽然回收频率降低可以摊销回收开销，但偶尔释放大批节点会导致程序长时间中断，并显著增加数据结构操作的尾延迟。

较大的批次大小也会增加应用程序的内存占用，使虚拟化环境中的内存利用和分配变得更具挑战性 [14]。例如，较大的批次大小会增加虚拟机 (VM) 或进程的内存占用，从而妨碍宿主机利用内存超分配；否则，可用的动态内存本可以在多个虚拟机实例（或其他进程）之间共享。

除了要求程序员找到可接受的批次大小（即回收频率）之外，大多数快速的基于纪元的安全内存回收算法 [1], [8], [9], [15]还必须确定全局时间戳的最佳递增频率（有时称为纪元频率）。这些参数的取值会共同影响安全内存回收算法的时间和空间效率。由于这些参数会因数据结构类型、工作负载和机器特性而异，因此选择它们的最佳取值颇具挑战性。

在这项工作中，我们彻底颠覆了传统的安全内存回收范式。安全内存回收算法通常会确保回收者延迟回收，直到读者无法再访问某个节点；而我们允许回收者立即释放内存，并由读者负责避免不一致。受 Alistarh 等人近期提出的内存标记方案 [16], 启发，我们提出了一种名为条件访问的新型硬件机制，使读者能够高效判断其试图访问的节点是否已被释放。该机制允许线程有条件地读取（cread）一个新位置，前提是一组由程序员定义的带标记位置自先前读取后未发生变化。类似地，线程可以通过验证所有带标记的位置均未发生变化，对一个带标记的位置进行条件写入（cwrite）。调用 cread 可为位置添加标记，调用 untagOne 或 untagAll 可手动取消标记。

条件访问非常适合用于实现这样一种数据结构：只要能证明一小组先前读取的位置自上次读取后未发生变化，就可以确定读取是安全的。例如，在一种链表中，删除节点前会先设置该节点的标记位；如果线程读取了一个未标记节点的后继指针，并且此后该节点的标记位和后继指针均未发生变化，那么解引用其后继指针仍然是安全的。在这种数据结构中，一旦节点被摘链并标记，就可以立即释放，因为这样做只会导致随后对该节点进行的条件读取或条件写入失败，从而触发重新启动（许多流行的安全内存回收方案都采用这种方法 [2], [7], [9], [17]）。

条件访问可以看作加载链接/条件存储（加载链接/条件存储）的一种推广：加载操作本身也是有条件的，而且存储操作可以依赖多个加载操作。加载链接操作实际上会给一个位置加上标签，而条件存储操作会移除该位置的标签；而在条件访问中，位置被写入时不会自动移除标签。因此，可以对同一组（或动态变化的一组）带标记位置执行多次条件读取和条件写入。

条件访问也与受限形式的事务内存有一些相似之处。然而，硬件

事务内存（硬件事务内存）由于安全方面的顾虑正越来越多地被禁用，而我们认为条件访问可以实现得更加安全。例如，线程只有在执行 cread 或 cwrite 并随后检查状态寄存器时，才会得知其已标记节点上的并发更新，因此我们可以避免一些时序攻击。这些攻击之所以可行，是因为在当前的硬件事务内存实现中，一旦其他线程进行冲突访问，就会立即中止。

高效实现条件访问所需的大部分信息，现代缓存一致性协议中已经具备。我们提出一种简单的扩展，无需更改一致性协议，即可在 L1 缓存层面实现标记。从宏观上看，每个 L1 缓存行都关联一个标签（一个比特）。对该缓存行中的任意位置执行 cread 时，都会设置此标签；对该缓存行中的某个位置执行 untagOne 或执行 untagAll 指令时，则会取消设置。每个核心都会跟踪针对其自身带标记位置的失效操作。（在 k 超线程共享核心的 SMT 架构中，每个超线程都会跟踪针对其自身带标记位置的失效操作）。

尽管微体系结构层面的详细实现超出了本文的讨论范围，但我们所需对缓存以及缓存与处理器流水线之间进行的扩展，严格来说只是实现硬件事务内存所需扩展的一部分。这有力地表明，条件访问的实现可以兼具实用性与高效性。

条件访问能够实现与顺序数据结构相近的内存占用。这对于现代数据中心很有价值，既可节省与内存超额分配相关的成本，也有助于实现内存超分配 [14]。此外，即时回收有助于避免此类攻击：攻击者利用延迟回收算法中已解除链接对象的延长生命周期，泄露私密数据。即时回收还有望防止拒绝服务攻击：线程诱导出一种调度，使成批未回收内存不断增长，最终导致内存不足错误。据报道，Linux 内核中的读-复制-更新实现曾出现过此类攻击 [18]。

本文作出以下贡献。(A) 我们引入了条件访问，这是一组能够实现高效即时内存回收的硬件指令。(B) 我们在开源多核模拟器 Graphite 上实现了条件访问的原型。(C) 我们实现了一个基准测试，其中包含多种最先进的内存回收技术和数据结构。(D) 我们展示了如何利用条件访问，为包括乐观两阶段锁定在内的多种数据结构设计模式避免释放后使用错误。

本文其余部分安排如下。第二节解释条件访问 的核心思想以及所提出指令的语义。第三节概述条件访问的一种直接硬件实现。第四节讨论如何将条件访问 用于乐观数据结构，并以栈和惰性链表为例，还简要讨论其正确性。第五节对条件访问进行基准测试，展示其效率和低内存占用。第六节介绍相关工作，随后第七节给出结论。

II. 条件访问

A. 关键思想

释放后使用 错误可视为读写数据竞争的一种特殊情况：某个共享内存位置在被另一个线程释放后仍被访问。在具有一致性缓存的现代系统中，释放后使用 错误之前总会发生缓存一致性协议中的事件。以传统的 MESI 协议为例：若要在当前处于共享状态的位置 X 存储一个值，核心 C 首先会向所有其他核心发送失效消息，使它们持有的 X 副本失效。核心收到此类消息后，会使该位置的副本失效，并回复一条确认消息。当 C 收到所有其他核心的确认后，它便获得对 X 的独占访问权。线程释放 X 后若要读取它，会先对这类失效消息作出响应，然后再读取 X。这表明，在缓存一致性协议层面，读者能够获知自己尝试访问的内存位置可能已被并发修改。此外，随后对 X 的读取必然以缓存未命中开始——如果能够利用并发修改的信息，这种开销本可避免。

在架构层面，读写数据竞争与释放后使用错误无法区分，这使得仅通过观察缓存一致性协议中的事件来识别释放后使用错误变得十分困难。如果我们愿意接受假阳性，就可以将每条失效消息解释为潜在释放后使用错误的迹象。为此，条件访问会监控更新线程与读取线程之间在架构层面的消息交换，并通过专用指令将这些交互暴露给程序，从而实现安全内存回收。

我们假设程序员知道哪些内存访问可能 导致释放后使用 错误，并要求程序员使用我们的新指令执行这些访问。随后，硬件可以对相应的缓存行加标签，以向硬件表明此类缓存行的失效 是一个值得关注的事件。只有当带标签位置自被标记以来未被失效 时，对该位置后续的加载和存储操作才会完成。请注意，要使此技术奏效，回收者必须在释放节点前对某个带标签的位置执行存储操作，以确保触发缓存事件，从而撤销其他线程对该节点的访问权限。随后，回收者即可立即释放该节点：任何在线程释放该节点前已对其加标签、并随后尝试访问该带标签节点的线程，都会发现相应的缓存行已被失效，因此不会执行该访问。

由于内存访问是否受条件限制 取决于某个内存位置自先前被加标签以来是否已失效，我们将此技术称为条件访问。

B. 新状态与指令

额外存储：（A）每个核心都会标记一个它打算监控失效请求的地址。从抽象角度看，已标记地址的集合可以表示为一个标签集。（B）

此外，每个核心还维护一个访问撤销位，该位初始为清零状态；当它对其标签集中的任一地址的访问权限被撤销时，该位便会置位。为简洁起见，本节假设标签集的容量不受限制。如何高效地近似这一集合，将在第三节讨论。

远程事件：对于核心 C 的标签集中的每个条目，硬件都必须检测自 C 对该缓存行进行标记以来，是否有其他核心使该缓存行失效。如果另一个核心使此缓存行失效，硬件必须设置 C 的访问撤销位。

介绍了标签集 和访问撤销位后，下面介绍所提出的指令:

(1) `cread addr, dest`： 与加载指令类似，cread 会将寄存器 addr 所指地址处的值更新到寄存器 dest 中，但有两个关键区别：标记和条件访问[a]。具体来说，cread 会原子地检查 addr 是否位于标签集中；如果不在，就将其添加到标签集中。它还会检查访问撤销位 是否已设置；如果已设置，则跳过加载，并更新其他处理器状态（例如标志寄存器），以表明可能发生了释放后使用错误。在这种情况下，我们称 cread 执行失败。否则，它会将 addr 处的值加载到 dest 中，表明此次内存访问是安全的。在这种情况下，我们称 cread 执行成功。**(2) `cwrite addr, v`：** 与 cread 不同，cwrite 不会更新标签集。cwrite 会原子地检查访问撤销位 是否已设置，或者 addr 是否不在标签集中；如果符合其中任一条件，就会跳过存储，并设置处理器标志，以表明 cwrite 执行失败 （这意味着可能发生了释放后使用 错误）。否则，它会将 v 存储到 addr 处，我们称 cwrite 执行成功。

这里值得讨论的是，为什么条件写入在一个不属于标签集的地址上执行时会失败。这一设计决策排除了这样一种用法：程序员可能在调用 cread 之前调用条件写入（换言之，在首次为某个位置添加标签之前）。这有助于避免在条件写入期间进行标记；如果访问未命中 L1 缓存，这可能会造成显著延迟，也使得避免棘手的检查时至使用时（TOCTOU）问题变得更容易。具体而言，我们希望避免这样的情形：条件写入未命中 L1 缓存，等待数据并取得该缓存行的独占所有权，却在等待期间才发现访问撤销位已被设置，因而最终导致条件写入失败。通过要求先执行 cread，我们将该操作中高延迟的部分转移到共享模式访问中，从而有可能减少失效操作和一致性流量。

(3) untagOne 地址： untagOne 指令不访问内存。其目的是让程序员能够从标签集中移除一个地址。如果 addr 不在标签集中，untagOne 不会产生任何效果。一旦某个地址从一个核心的标签集中被移除，该地址后续的远程失效将不会 设置该核心的访问撤销位。

(4) 取消所有标记： 取消所有标记会清空标签集 并取消设置访问撤销位的值。它适用于两种情况：（1）cread 或条件写入失败时，此时需要重试数据结构操作；（2）成功完成数据结构操作后返回之前。

请注意，对于具有多个硬件线程的同步多线程架构，需要为每个硬件线程而非每个核心配备额外存储和远程事件。

[a]为便于说明，我们不按从内存读取的字节数对 cread 进行参数化，也不考虑不同的寻址模式。在实际系统中，需要为这些用途提供多个操作码。

III. 实现

条件访问可通过对现有缓存进行直接扩展来实现，只需在处理器与其一级缓存（例如 L1 数据缓存）之间引入修改。根据我们在多核模拟器上的原型验证，我们认为，这些修改严格属于实现硬件事务内存所需修改的子集，这意味着条件访问易于实现且效率高。

实现条件访问需要实现标签集和访问撤销位。所提出的指令使用这些数据结构来跟踪相关的失效消息，这些消息由底层缓存一致性协议生成。

(A) 可通过为核心的 L1 数据缓存中的每个缓存行增加一个标签位，来近似实现标签集。这类似于硬件事务内存对读写集的近似实现方式。(B) 用于跟踪线程标签集中各地址失效情况的访问撤销位，需要为每个核心增加一位。实现访问撤销位的一种方式，是将其加入宿主架构的条件码或标志寄存器中（例如 x86 上的 EFLAGS）。

注意，在 SMT 架构中，k 个超线程共享一个核心，每个超线程都会跟踪自身哪些缓存行带有标记，并跟踪其带标记位置的失效操作。例如，在双路 SMT 架构中，需要两个标签位和两个访问撤销位，每个硬件线程各一个。

进行这些更改后，我们现在可以利用缓存一致性协议来检测不安全的访问。当 cread 将一个地址添加到标签集时，它会将该缓存行加载到缓存中，并设置该行的标签位。之后，该缓存行可能通过两种方式离开缓存：远程失效或本地相联冲突。无论哪种情况，缓存都必须通知硬件线程设置其访问撤销位，以便后续的 cread 或 cwrite 操作失败。对于远程失效，此操作必须与确认远程请求保持原子性。对于相联冲突，此操作必须与从内存层次结构获取新数据保持原子性。untagOne 和 untagAll 的原子性要求较为简单：它们不能与同一硬件线程执行的加载和存储操作重新排序。此外，untagAll 必须清除访问撤销位，以供后续操作使用。

除了上述两种方式之外，在 SMT 架构中，线程的 访问撤销位 可以在另一个线程（在使用超线程时）写入共享缓存行时，或在发生上下文切换时被设置。在上下文切换时设置该位更易于实现，因为这样操作系统便无需代表

被切出的线程跟踪失效操作。这些特性为在多用户系统中使用条件访问提供了基础。

直观而言，条件访问中的标记机制有助于实现一种对共享内存位置的本地保护，且不会触发任何额外的一致性流量。这与诸如危险指针 [2] 或其他基于保留的 [9] 技术等流行范式相反，后者总会在线程之间触发全局缓存流量。

上述实现 标签集的方式意味着，标签集大小受缓存相联度限制，因此 标签集可能溢出。这将导致已标记地址（位于 标签集中）被驱逐，从而使 访问撤销位被设置。反过来，这可能导致后续条件读取或条件写入出现伪失败，进而阻碍进展。然而，在实践中这并不是问题，因为在大多数情况下，标签集都很小。我们的实验（第五节）表明，对于我们所考察的工作负载，相联度不会对进展产生显著影响。

IV. 将条件访问用于乐观数据结构

本节讨论如何利用条件访问，在链表 [19] 和外部二叉搜索树 [20]等乐观数据结构中实现安全内存回收。许多此类数据结构的操作都包含一个搜索阶段，由多次读取组成；在线程访问到其关注的节点集合、使操作最终生效之前，它会持续遍历节点的 next 字段。到达目标节点后，操作可能执行零次或多次写入。例如，在链表中，线程可能遍历多个链接，以查找操作应当生效的前驱节点和当前节点。

为便于说明，我们假设每个节点都能容纳在单个缓存行中，且一个缓存行只包含一个节点。因此，将一个节点添加到某个核心的标签集，意味着将包含该节点的缓存行添加到该核心的标签集中。

首先，我们提出以下高层指导原则，所有数据结构都必须遵循这些原则，才能正确使用条件访问。

(DI) 替换并分析： 替换：对可能被释放的节点进行的所有读写访问，都应替换为相应的 cread 和 cwrite 指令。这样，条件访问 便能标记节点、监控其是否发生并发修改，并通过更新标志寄存器通知程序员。分析：如果 cread 或 cwrite 失败，操作应立即取消所有标记并重试。指令失败意味着某个节点可能已被并发释放，因此之后的任何访问都不再安全。

(DII) 验证可达性： 节点首次执行 cread 时会被标记。为确保该标记节点有效，应事后验证它在数据结构中是否可达。

现在我们演示如何将这些指令应用于不同类别的乐观数据结构中的条件访问。根据数据结构的不同，DII 可以部分放宽，正如我们将在惰性链表示例中看到的那样或者，正如我们将在无锁栈示例中看到的那样，可能不需要 DII。惰性链表还需要一些其他规则，详见第四节B。

A. 单次写入数据结构

更新阶段仅包含一次写入的数据结构包括一些基于链表的栈 [21]和队列 [22],，我们已经实现了这两种数据结构。为便于说明，我们将考虑一种基于链表的无界无锁栈。在这种栈中，入栈操作包括读取栈顶指针、为要压入的键值分配一个新节点，然后执行比较并交换（CAS）将该节点设为新的栈顶。同样，出栈操作包括读取栈顶，然后原子地将栈顶设置为其下一个节点。出栈后，如果并发线程仍可能访问已解除链接的节点，则不能释放该节点。

只需将每次读取替换为 cread，并将 CAS 替换为 cwrite（DI），即可升级栈的原始操作以启用 条件访问 。这样，出栈操作就可以立即释放已解除链接的节点，如算法1所示。注意，在我们的伪代码中，当 cread 或 cwrite 失败时，会设置 CAFAIL。这类似于更新标志寄存器。

升级后操作的线性化性。其正确性源于：使用 cread 读取 top 时，会将其添加到相应线程的标签集中，随后该线程便开始监控 top 是否发生任何后续修改。由于 top 本身永远不会被删除，因此可以保证在将其添加到标签集(DII)时，它始终存在于数据结构中。操作开始时，访问撤销位处于清零状态，只有当线程收到针对 top 的失效请求（即 top 在其他位置被修改时）才会将其置位。之后，线程尝试使用条件写入更改 top；条件写入会原子地检查访问撤销位 ，以检测是否存在干扰性内存访问。如果访问撤销位已置位，操作便会失败。这会使线程从其标签集中移除 top，使用取消所有标记清除访问撤销位 ，然后重试该操作。否则，线程会成功将 top 更改为另一个节点。压入和出栈操作分别在线程成功执行算法 1 第 9 行和第 17 行的条件写入时线性化。

请注意，第 18 行对 free 的调用是安全的，因为每当核心 C1 修改栈顶（无论是压入还是弹出）时，任何其他能够访问栈顶的核心 C2 都会因 C1 设置其访问撤销位而使标签失效，从而导致 C2 的条件写入失败。

条件访问 是 ABA安全的，尽管它允许立即重用已释放对象。假设线程 T1 为了插入一个新节点，从栈顶读取地址 A 并存入局部变量 t。就在它执行 CAS、将 top 设置为 t 的 next 之前，另一个线程 T2 将地址 B 处的节点设为新的栈顶，从而移除 A，释放 A，随后又在回收的地址 A 处压入一个新节点，使其成为新的栈顶。此时，T1 会成功执行 CAS（基于地址比较），因为 top 仍包含地址 A，与存储在其局部变量 t 中的预期地址一致。因此，它错误地执行成功，而本应失败——这是一种

算法 1 使用条件访问实现无界
无锁栈。标注为线性化点（LP）的行即为线性化点。

```
1:  类型 Node {键 key, Node *next}
2:  类 Stack {Node *top}
3:  #define CA_CHECK 如果CAFAIL 则取消所有标记(); goto 重试;

4:  过程压入(key)
5:      新顶部 =新建节点（键）;
6:      重试:
7:      t ← cread(top); CA_检查
8:      新顶部->next =t;
9:      cwrite(&top, newtop); CA_检查                ▷ 线性化点
10: 结束过程

11: 过程 POP( )
12:     重试:
13:     t ← cread(top); CA_CHECK
14:     如果NULL == t 则
15:         取消所有标记(); 并返回;
16:     结束 if
17:     条件写入(&top, t->next); CA_CHECK            ▷ 线性化点
18:     释放(t)
19: 结束过程
```

典型的 ABA 情况。条件访问 通过条件写入避免了这一错误；与 CAS 不同，条件写入并非基于比较两个值，而是依赖底层的缓存失效消息，检测某个位置自上次读取后是否已被修改。

B. 具有锁和多次写入的数据结构

另一类链式并发数据结构的更新操作中，线程会乐观地遍历一系列节点，随后在由锁保护的临界区内执行多次更新。惰性链表 [19] 就是一个例子。

具有此类设计模式的数据结构操作可按照以下通用指南进行改造，以启用所提出的安全内存回收：

1) 在搜索阶段，使用 DI 将所有读取替换为 cread。当先前遍历过的节点不再用于证明未来将访问的节点在其被标记时仍可从数据结构中到达时，使用 untagOne 将这些节点从当前线程的标签集 中移除。如果遍历期间 cread 失败，则执行 untagAll 并重试搜索。对于只读操作，这样就足够了。更新操作还需要执行以下步骤：

2) 使用基于 cread/cwrite 设计的尝试锁（算法 2），锁定搜索结束时确定的所有节点。这标志着临界区的开始，以便原子地执行更新。如果对任何节点的锁获取失败，则解锁先前已锁定的节点（如果有），执行 untagAll 并重试该操作。

3）在临界区内使用普通写入执行预期的更新。这是安全的，因为节点受到临界区保护，因此不会被并发更新或回收（对 DI 的部分放宽）。

4）如果更新是删除，则在解除节点链接之前对其进行标记。这满足禁止访问已解除链接节点规则。

5）最后，在退出操作之前，解锁所有已锁定的节点并执行取消所有标记。请注意，解锁可以使用普通存储而非条件写入，因为已锁定的节点无法被其他线程释放。

以惰性链表为例（参见算法 3），我们将演示如何轻松地将其升级为使用条件访问。

使用 D1，将搜索中的所有常规读取替换为条件读取，如算法 3 所示。如规范中所述，条件读取会原子地执行以下操作：将一个节点添加到当前线程的标签集中（如果该节点尚未在其中），检查访问撤销位 是否已清零，并在条件满足时完成一次普通读取。但是，如果条件读取失败（即 CAFAIL 已设置），则标签集 中的一部分节点可能自上次访问以来已被修改（甚至可能已删除），因此读取这些节点可能并不安全。在这种情况下，清空标签集 ，使用取消所有标记取消访问撤销位 的设置，然后重试搜索。否则，搜索将继续，直到找到所需的某些前驱节点和当前节点（算法 3 的第 22 缓存行）。

注意，如果我们在搜索期间不取消前一个节点的标签，那么由于条件读取会为节点添加标签，搜索路径中的所有节点都会被添加到线程的 标签集 中。当搜索路径中的任意节点（无论是否与当前访问相关）被修改时，都可能导致条件读取失败，从而迫使操作反复重试。换言之，某些更新将被串行化，就好像线程获取了全局锁一样，这会抑制并发。为解决这一问题，线程可以使用 untagOne 取消前一个节点的标签，并且在任意时刻只需保持两个连续节点带有标签；这等于执行更新所需的节点数量，类似于逐节点加锁。

此外，为了保证搜索是安全的，我们需要确保节点被添加标签时它仍可在链表中访问（DII）。为说明原因，假设线程使用 cread 访问任意节点的内容。在此次 cread 期间，原子地执行以下操作：为包含该节点的缓存行添加标签，然后加载其内容。现在，如果该节点在被 cread 添加标签之前就已被标记，那么后续的 cread 仍会成功，即使该节点已被标记（逻辑删除），这并不安全，因为该节点可能已被回收（释放后使用 错误）。解决方法是在为节点添加标签的 cread 操作之后立即验证该节点未被标记。如果发现节点已被标记，则验证失败，并且相应操作会取消所有节点的标签并重试。例如，在惰性链表中，节点会在算法 3 第 5 行的 cread 期间首次被添加标签，这是由于第 11、13 或 19 行调用了 validate()。如果 validate() 因节点已被标记而返回假，则操作会取消所有节点的标签并重试。这样即可满足 DII。

还可以进一步追问：如果节点已被标记（即已逻辑删除），并且在打上标签之前就被释放了，会怎样？在这种情况下，后续的 cread 可能会成功，因为节点打上标签后不会再发生更新，所以它的访问撤销位不会被置位。这可能导致释放后使用错误。不过，这种情况

算法 2 条件访问 式锁。前置条件:
包含锁字段的节点应先进行 cread，以便将其
标记。

```
1: 过程 TRYLOCK(bool *lock)
2:     lockVal ← cread(lock);
3:     若 CAFAIL or 1 ==lockVal 则 返回 假;

4:     条件写入(锁, 1);如果 CAFAIL 则 返回 假;
5:     返回 真;
6: 结束过程

7: 过程 UNLOCK(布尔值 *锁)
8:     *锁 ← 0;                ▷安全，因为节点只能由所有者修改。
9: 结束过程
```

不可能发生，因为要释放该节点，回收者必须通过修改其前驱节点的 next 字段将其从链表中摘除，而该前驱节点已位于线程的标签集中。因此，如果前驱节点被修改，线程的访问撤销位就会被置位，cread 也会失败，从而防止不安全访问。在搜索过程中，这一不变式始终成立；搜索最终得到的前驱节点和当前节点在打上标签时仍可从链表中访问。

之后，在开始更新之前，会获取前驱节点和当前节点上的锁（对于 insert()，在第 33 行和第 36 行；对于 delete()，在第 49 行和第 52 行）。然而，搜索返回这些节点后、获取锁之前，可能会有某个线程删除这些节点。在这种情况下，如果使用普通的读取和写入来访问锁，线程可能会尝试获取已释放节点上的锁，从而导致未定义行为：与条件读取不同，常规读取无法检查节点是否已被修改。因此，为了解决这个问题，我们提供基于 cread/cwrite 的尝试锁，只有在节点未被并发修改（删除）时才会获取该节点上的锁。

算法 2 展示了该锁的实现。其前提条件是，包含锁字段的节点必须先通过 cread 访问过，从而将其添加到该线程的标签集中，使 cread/cwrite 能够通过访问撤销位 验证该节点此后是否已被修改。具体来说，线程会对锁变量执行 cread。如果它设置了 CAFAIL，则锁所属的节点可能已被删除；如果返回 1，则表示锁正被占用。这两种情况下，锁获取都会失败。否则，线程会通过 cwrite 将锁字段设为 1，以继续获取锁；该操作也会再次检查包含锁字段的节点是否已被修改（可能已被删除）。如果检查成功，它会将 1 写入锁字段并返回真，表示锁获取成功。否则，如果 cwrite 失败（设置了 CAFAIL），则返回假，表示锁获取失败；调用该锁的操作会取消所有节点的标签并重试。

插入操作（第 29 行，算法 3）首先执行定位，返回带标记的前驱节点和当前节点，以及当前键（当前节点的键字段）。如果要插入的键已存在于链表中，则操作返回假。否则，该键不存在，需要将其插入。为

算法 3 使用条件访问的惰性链表 [19]。行
标注为线性化点的行即为线性化点。

```
1: 类型 节点 {键 key，锁，标记，节点 *next}
2: 类 惰性链表 {节点 *头节点}
3: #define CA_CHECK 如果CAFAIL 则取消所有标记(); goto 重试;

4: 过程VALIDATE(Node *node)
5:     isMarked ← cread(node->mark);如果CAFAIL 则return 假;
6:     如果isMarked 则return 假; 否则return 真;
7: 结束过程

8: 过程LOCATE(键 key)
9:     重试:
10:    pred ← cread(头节点); CA_CHECK
11:    验证(pred); 如果 False 则 取消所有标记(); goto 重试;
12:    当前节点 ← cread(前驱节点->next); CA_CHECK
13:    验证(curr); 如果 False 则 取消所有标记(); goto 重试;
14:    当前键 ← cread(当前节点->键); CA_CHECK
15:    当currkey < key 时
16:        untagOne(pred);
17:        前驱节点 ← 读取当前节点； CA_检查
18:        当前节点 ← cread(curr->next); CA_CHECK
19:        验证(curr); 若 False 则untagAll();goto 重试;
20:        当前键 ← cread(curr->key); CA_CHECK
21:    结束 while
22:    返回 ⟨前驱节点、当前节点、当前键 ⟩;
23: 结束过程

24: 过程CONTAIN(键)
25: ⟨前驱节点、当前节点、当前键 ⟩ ← 定位(键)； ▷ 线性化点：读取当前键时。
26:    取消所有标记();
27:    返回 (当前键 == 键);
28: 结束过程

29: 过程插入(key)
30:    重试:
31:    ⟨前驱节点, 当前节点, 当前键 ⟩ ← 定位(key);          ▷ LP 当插入失败时。
32:    如果当前键 == 键 则 取消所有标记; & 返回 假;
33:    如果假 == 尝试加锁(&前驱节点->锁)则        ▷尝试锁定前驱节点。
34:        取消所有标记并重试;
35:    结束如果
36:    如果假 == 尝试加锁(&当前节点->锁)则        ▷尝试锁定当前节点。
37:        解锁(&pred->锁);
38:        取消所有标记 & 重试;
39:    结束 if
40:    节点 ← 新建 节点(键, 当前节点);
41:    前驱节点->下一个 ← 节点;                    ▷ LP 当插入成功时。
42:    全部解锁() 并取消所有标记;
43:    返回 真;
44: 结束过程

45: 过程删除(key)
46:    重试:
47:    ⟨ 前驱节点, 当前节点, 当前键 ⟩ ←定位(key);          ▷ LP 删除失败时。
48:    如果当前键 != 键 那么 取消所有标记； & 返回 假;
49:    如果 假 == 尝试加锁(&前驱节点->锁)那么        ▷尝试锁定前驱节点。
50:        取消所有标记； & 重试;
51:    结束如果
52:    如果 假 == 尝试加锁(&当前节点->锁)则        ▷尝试锁定当前节点。
53:        解锁(&前驱节点->锁);
54:        取消所有标记; & 重试;
55:    结束如果
56:    当前节点->标记 ← 真;                        ▷ LP 当删除成功时。
57:    前驱节点->next ← 当前节点->next;
58:    全部解锁(); & 取消所有标记;
59:    释放(当前节点);
60:    返回 真;
61: 结束过程
```

插入键时，首先获取基于条件访问 的前驱节点和当前节点的尝试锁，然后创建一个新节点并将其

插入前驱节点和当前节点之间。随后释放所有锁，取消节点标签，然后操作返回真。

注意，由于前驱节点和当前节点已被锁定，因此其他线程若不先获取锁，就无法修改它们。因此，无需验证该节点是否已被并发释放。这样，我们便可在临界区内使用普通读写操作，而不是 cread/cwrite。

删除操作（算法 3，第 45 行）会调用定位操作，后者返回带标签的前驱节点和当前节点，以及当前键。如果要删除的键不在链表中，该操作就会移除节点的标签并返回假。否则，与插入操作类似，它会尝试锁定这两个节点，写入当前节点以设置其标记字段，解除当前节点的链接，解锁并移除两个节点的标签，释放当前节点，然后返回真。

1) 正确性：如果遵循上述所有规则，在惰性链表中启用条件访问，那么该链表就是线性化的，且其中的所有访问都是安全的。包含操作，以及行为类似于包含操作的插入失败和删除失败操作，都可以在线程读取当前节点的键时进行线性化。而插入成功和删除成功操作，则分别可以在线性表中链接新节点时（第 41 行）或标记节点时（第 56 行）进行线性化。此外，由于条件读取绝不会解引用不可达（已解除链接）的节点，因此不会发生释放后使用错误。因此，启用条件访问的惰性链表是安全的。下一节将详细讨论其正确性。

假设 1. 列表操作按照给定的指南实现，以支持基于条件访问的即时回收。

1) 遵循替换与分析（*DI*）规则。*2)* 遵循验证可达性（*DII*）。为确保只有可达且未标记的节点被标记，在节点被标记后验证其标记字段。*3)* 在更新中，用基于条件访问的尝试锁替换普通锁。*4)* 在线程释放节点之前，必须先对该节点执行写入（即更新标记字段）。

为确保基于条件访问 的程序正确，所有数据结构操作都应使用 cread 和条件写入来读写共享内存；线程在释放节点之前必须先对其执行写入；并且节点被标记时应当可达且未标记。此外，为使**DI** 正常工作，我们假设底层硬件提供缓存一致性机制，例如 MSI、MESI 或其他等效机制。

性质 2. 节点在其内容首次被 *cread* 时即被标记，并一直保持标记状态，直到通过 *untagOne* 或 *untagAll* 显式取消标记。

性质 3。 当节点的任意字段被写入时，底层缓存一致性机制会使其他线程中该位置的所有缓存副本失效，方法是将它们的访问撤销位置为真；除非通过取消所有标记将其取消设置，否则它会一直保持为真。

断言 4。 对节点执行条件访问，即对节点执行 *cread* 或 *cwrite*，只有在所有已标记节点自被标记以来均未失效时才会成功。

引理 5。 节点的前驱在该节点被标记时未被标记，且其 *next* 字段指向该节点。

这意味着线程永远不会从已标记节点读取 next 字段（或发现新节点）。

证明。 我们按照基于条件访问 的惰性链表中的节点序列，通过归纳法证明该引理。

基本情形： 搜索开始时，首先对头节点执行 cread，从而将头节点的地址加载到局部指针 pred 中。随后，对头节点的标记字段执行 cread，该操作会原子地：标记头节点，并加载其标记字段的值。由于头指针从未改变，且头节点从未被标记，因此头节点在被标记时是有效的。此后，如果头节点的任一字段发生改变，对其字段执行的任何 cread 都将失败。之后，对头节点的 next 字段执行 cread，并将其保存到局部指针 curr 中，接着对当前节点的标记字段执行 cread。后一个 cread 会标记当前节点，然后加载其标记字段的值。

因此，在此时，当当前节点被标记时，可以保证其前驱节点 pred 未被标记，因为头节点永远不会被标记；同时也可以保证其 next 字段未发生变化（仍指向当前节点），否则，根据性质 2、性质 3 和断言 4，对当前节点标记字段的 cread 就会失败。还要注意，已验证当前节点的标记字段未被标记。因此，在基本情形中，pred 和当前节点都未被标记，且 pred 指向当前节点。

归纳假设： 假设对于某个线程搜索路径中的任意前驱节点 pred 和当前节点，都满足这一条件。也就是说，pred 和当前节点都未被标记，且 pred 指向当前节点。现在我们将证明：当当前节点之后的节点（设为后继节点 succ ）被标记时，当前节点仍未被标记，且指向 succ。

归纳步骤： 为了访问 succ 并对其进行标记，线程首先对其前驱节点（即当前节点）执行 acread，以加载其 next 字段。如果 cread 成功，则意味着当前节点自被标记以来未发生变化。也就是说，它既未被标记，其 next 字段也未发生变化。因此，当对 succ 的标记字段执行 cread 以进行标记时，其前驱节点未被标记，且指向 succ。否则，根据性质 2、性质 3 和断言 4，cread 就会失败。

因此，该引理对任意节点都成立。 □

定理 6 (条件访问 是安全的)。 在线程条件访问基于列表的结构中，线程无法访问已回收节点。

证明。 反设，即某个线程可能访问已回收节点。这种情况可能发生在基于条件访问 的惰性链表中，具体有以下两种情形：

1）线程回收某个节点时，没有将其标记字段设为真。在这种情况下，已为该节点添加标签的读者不会收到失效消息，因此其访问撤销位 不会被置位。结果，读者未察觉该节点已被删除，后续 cread 将成功，这可能导致释放后使用 错误。但这不可能发生，因为我们假设回收者规则得到了遵守。

2) 线程会在节点被标记后为其添加标签，这样后续的 cread 或 cwrite 就不会收到针对该节点的任何失效请求。因此，其 访问撤销位 也不会被置位，进而导致条件访问成功。但根据引理5，这种情况同样不可能发生。

因此，这两种情形都与我们的假设矛盾，这意味着在惰性链表中使用条件访问 是安全的。 □

定理7 （条件访问无 ABA 问题）。 条件访问不会受到 *ABA* 问题的影响。

证明。 假设基于条件访问 的惰性链表存在 ABA 问题。因此，存在这样一种情形：线程 T_i (A)对内存位置m 进行条件访问，随后另一个线程 T_j (B)修改m ，然后 T_i (A)再次对m进行条件内存访问，且第二次内存访问成功，却未能识别m 自首次访问以来已发生更改。

由于内存位置首次被 cread 时总会添加标签，因此 T_i 在首次对m执行 cread 时会为m 添加标签。之后，如果 T_j 修改m ，缓存一致性机制会使m的所有远程副本失效，并在此过程中使标签失效（性质 3）。因此，当 T_i 再次对m 执行 cread 时，会发现m 已失效，从而导致操作失败（断言 4）。由于 cread 不可能同时失败和成功，因此我们最初的假设——即使另一个线程在 T_i 首次访问m 之后对其进行修改，T_i 第二次对其执行 cread 仍会成功——是错误的。因此，不会出现 ABA 问题。

□

2) 进展： 如果以硬件方式实现条件访问 指令，使其能够消除由组相联驱逐或中断导致的伪失败，并且条件访问仅会因真实的数据冲突而失败，那么基于条件访问 的数据结构搜索就可以保证无锁。因为，条件访问失败意味着某个线程已成功执行了对冲突地址的更新，因此至少有一个线程能够保证取得进展。

另一方面，如果所提出的条件访问 语义的实现无法保证不会发生伪失败，则可以使用一种回退技术。

促进进展如前所述，条件访问容易因硬件资源受限而发生伪失败，例如组相联驱逐导致带标记的缓存行溢出。为降低此类伪失败的发生概率，我们提出了 untagOne，它通过取消先前位置的标记来帮助仅保留最小集合的带标记引用；这些位置对于确保搜索的正确性并非必需。

V. 实验

我们使用 Graphite 多核模拟器 [23] 对条件访问 （CA）进行了原型实现。我们的修改仅限于 L1 数据缓存级别；我们没有更改缓存一致性协议。Graphite 配置为使用基于目录的 MSI 缓存一致性协议，配备私有 32K L1 缓存和共享包含式 256K L2 缓存。每个缓存行大小为 64 字节，每个线程在专用的模拟核心上运行，并配备基本的分支预测机制和乱序内存子系统。

我们使用对惰性链表和外部二叉搜索树（extbst）进行压力测试的微基准测试，评估 CA 的可扩展性和内存效率。我们还使用栈和哈希表微基准测试，在不同竞争程度下评估 CA。惰性链表、栈和哈希表中的键范围为 0 到 1K；extbst 中的键范围为 0 到 10K。哈希表有 128 个桶，每个桶都是一个惰性链表。

这些数据结构均采用以下安全内存回收技术：有泄漏的实现（**none**：）、条件访问 （**ca**）、IBR 的 2geibr 变体（**ibr**）、**rcu**、基于静止状态的回收（**qsbr**）、危险指针（**hp**）和危险时期（**he**）。CA 会立即回收每个已删除节点，且无需其他参数。其他回收方案设置为每成功执行 30 次删除操作后尝试回收一次（回收频率）。对于基于纪元的方案（ibr、rcu、qsbr 和 he），纪元设置为每分配 150 次后变更一次（纪元频率）。这些数值是 IBR 基准测试[9]中的默认设置。

每次实验中的每项试验都会先将其数据结构预填充至 50% 的容量，然后每个线程执行 3K 次操作。线程数在 1 至 32 之间变化。每当线程调用一次数据结构操作时，都会随机选择一个操作以及一个随机键。在我们的实验中，线程以相同的概率选择插入或删除，该概率分别为 0%、5% 或 50%，因此我们可以分别运行 0%（只读）、10% 和 100% 更新的实验。由于在所有工作负载中插入和删除的概率相等，数据结构的大小基本保持不变，存储键范围内一半的元素。对于每种工作负载配置，我们报告三次运行的平均值。各次运行之间没有显著差异。

在图 1 和图 2 的整个实验过程中，hp、he 和 ibr 通常比其他算法更慢。这主要可归因于较高的每次读取开销，因为这些算法分别需要使用读写栅栏来访问或更新保留记录和纪元。此外，这些算法还存在回收开销，需要扫描保留记录，以确定哪些记录可以安全释放。总体而言，这会导致较差的缓存行为和较高的操作延迟。

另一方面，读-复制-更新和 QSBR 不会产生每次读取的开销。它们的主要开销来自回收事件，在这些事件中，

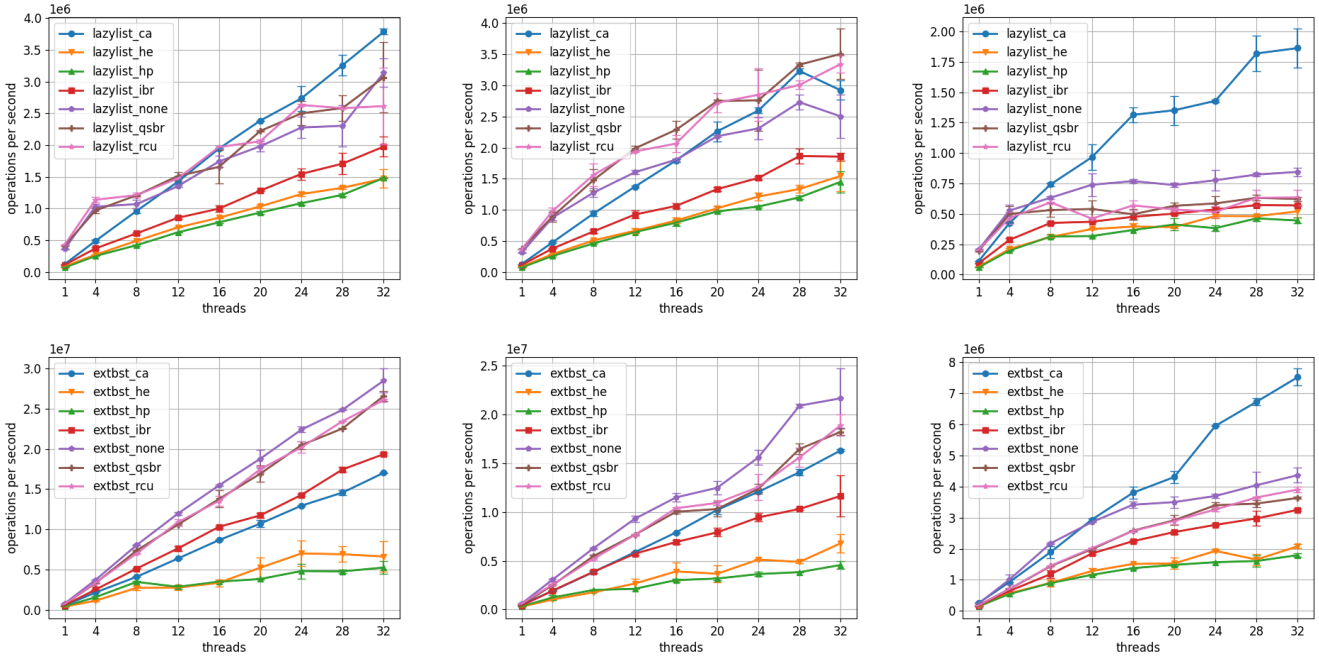

图 1：吞吐量评估。　　纵轴：吞吐量。横轴：线程数。左：0i-0d。中：5i-5d。右：50i-50d。（上排）惰性链表，大小：1K。（下排）外部二叉搜索树，大小：10K。

在扫描所有进程的纪元后，会释放成批的已退役对象。这一开销会分摊到多次操作中。因此，在各种工作负载和数据结构中，这些算法速度更快，性能与基线 none 相近。

在只读工作负载中，CA 的速度比 RCU、QSBR 和 none 慢一些。这是因为延迟增加了：每次条件读取后都要检查 访问撤销位 ，导致指令计数增加。由于没有冲突，这些检查是多余的。然而，在包含更新的工作负载中，CA 的表现接近甚至优于 RCU、QSBR 和 none。在竞争激烈的场景中（即更新频繁且线程数较多时），CA 甚至胜过这些算法。这是因为 CA 避免了读者和回收者双方的读写栅栏。CA 还带来了其他优势。即时回收提升了缓存和 TLB 局部性，尤其是相较于 none；它能比其他算法更早发现失败，从而无需浪费太多工作即可重新启动；并且能够避免部分缓存未命中延迟。这些因素都有助于降低延迟、提高吞吐量。缓存未命中的代价较低，是因为与常规读取不同，条件读取中的缓存未命中影响仅限于其所在核心 [16]。我们将在下文解释这一点。

在使用常规读取和写入的数据结构操作中，所有共享内存位置的线程都会因缓存未命中而产生延迟。以惰性链表为例，假设线程 T1 即将获取其前驱节点和当前节点的锁。再假设另一个线程 T2 已读取前驱节点，即将再次读取它。此时，在缓存层面，T1 和 T2 都持有共享状态中与这些地址对应的缓存行副本。当 T1 获取前驱节点的锁时，会执行一次写入。这会触发一致性流量：所有其他前驱节点的读者都必须使其缓存行副本失效（此处为 T2）。当 T2 再次读取前驱节点时，由于其缓存行副本已失效，将遭遇缓存未命中。为处理这次缓存未命中：

- T2 触发缓存级事务，以获取该缓存行的最新副本，并等待响应。
- 拥有该缓存行 M 状态副本的 T1 可能会被迫将其副本写回内存层次结构，同时也将其提供给 T2。

这浪费了 T2 的计算时间，因为 T2 最终会发现缓存行已发生变化，因此必须重新启动其操作。如果它没有等待，就可以已经重新启动并执行多条指令。此外，由于 T1 获取了前驱节点的锁，它很可能会再次写入前驱节点。T2 的请求使 T1 缓存中的缓存行从 M 状态降级为 S 状态，因此 T1 随后的写入必须先请求所有权，这会导致 T2 再次使该缓存行失效。频繁降级到共享状态并升级到修改状态，会削弱写缓冲带来的收益，也使隐藏缓存延迟变得困难。共享位置上的竞争越激烈，这些开销就越大。

另一方面，在使用 cread 和条件写入设计的数据结构中，T2 的第二次 cread 会因检测到缓存行已不再存在而验证失败，并重试其操作，而不会请求该缓存行的新副本。与上述常规读写的问题不同，CA 允许 T2 跳过请求前驱节点的值，从而避免产生全局缓存流量。这既避免了 T2 的读取延迟，也有助于 T1 避免缓存状态升级事务。换言之，与基于常规读写的数据结构不同，

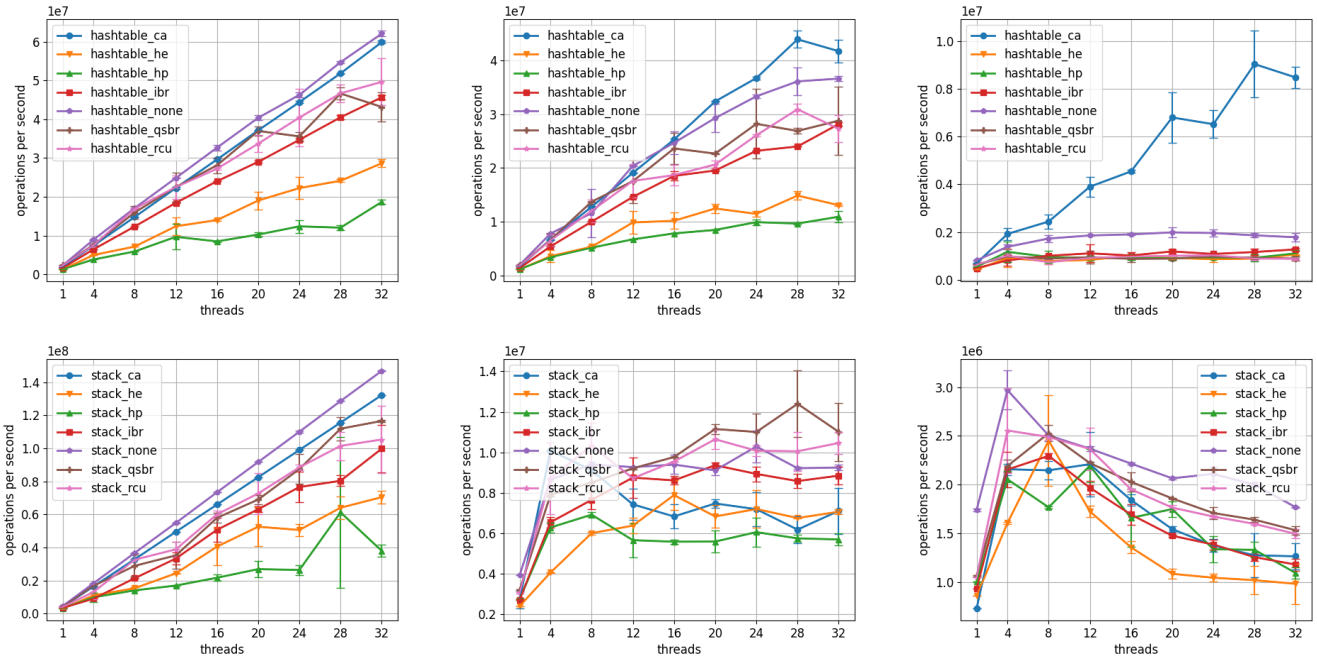

图 2：吞吐量评估。纵轴：吞吐量。横轴：线程数。左：0i-0d。中：5i-5d。右：50i-50d。（上排）链式哈希表，桶数：128，最大大小 128K。（下排）栈。

在采用 CA 的数据结构中，访问缓存行失败的影响仍局限于本地核心 [16]。

因此，在所有数据结构实现中，上述特性带来的较低 L1 数据缓存延迟，使 CA 在竞争激烈时能够与其他算法表现相当，甚至更快。在只读工作负载中，由于指令计数较高所带来的开销，CA 的表现比基线（无）和其他快速算法（QSBR 和读-复制-更新）更慢或相当。

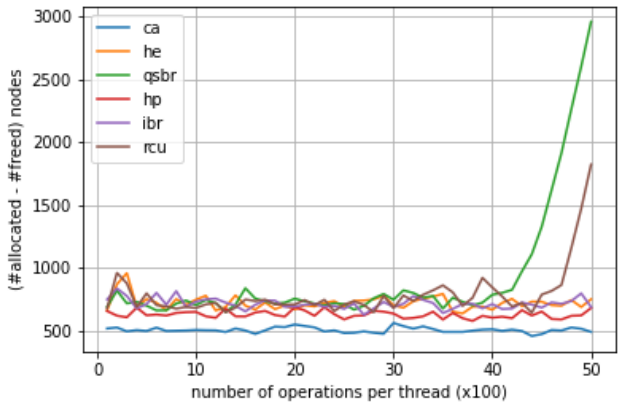

图 3：内存消耗：显示大小为 ~500 的链表中已分配但尚未释放的节点数。插入和删除比例各为 50%。针对 16 个线程。

图 3 展示了内存开销：对于每种回收方案，我们测量惰性链表数据结构在执行过程中每进行 1000 次操作后，已分配但尚未释放的节点数量（Y 轴）（X 轴）。该测试揭示了采用不同回收方案时，数据结构的内存占用会增加多少。本实验使用一个值范围为 0 到 1000 的惰性链表，初始时预填充了 500 个节点。实验中有 16 个线程对链表进行操作。在实验测量阶段，所有线程以相同的 50% 概率执行插入和删除操作（100% 更新工作负载）。每个线程运行 5000 次操作。

理想情况下，由于工作负载的特性，实验期间列表大小应始终大致为 500，而已删除但尚未释放的节点数应为零。CA 方案始终显示约有 500 个节点已分配但尚未释放；这些节点仍可从列表中访问。这证实我们实现了即时回收，并将内存占用保持在较低水平。由于其他回收方案会将已删除节点收集到本地退役列表中，从而延迟内存回收，因此尚未回收的节点数会增加，进而导致内存占用上升。图表证实了这一判断：hp、he、ibr、rcu 和 qsbr 报告的未回收节点数都更高。值得一提的是，在 qsbr 和 rcu 中，一个延迟线程可能会阻止所有线程进行回收，因此未回收节点数可能会无限增长。如果实验运行时间更长，那么一旦有线程发生上下文切换，这些方案的未回收节点数预计就会迅速膨胀。

VI. 相关工作

A. 回收技术讨论

许多现有的安全内存回收技术会延迟回收已解除链接的节点，大致可分为基于纪元的回收（EBR）、基于危险指针的回收（HBR）、基于引用计数的回收（RCBR）和混合回收（HYR）：后者结合了先前的技术或专用硬件支持。EBR [11], [24],[25], [1] 方案速度快，但未回收节点的数量可能没有上限。HBR [2], [26], [27]和 RCBR [10], [26], [28], [8] 能够限制未回收节点的数量，但由于每次读取的开销较高或需要对节点进行检测，通常速度较慢。HYR 技术在兼顾速度和限制未回收节点数量方面取得了一定成效，但需要依赖有关专用硬件、操作系统和内存分配器的假设 [8], [29], [9], [12], [13], [4], [5], [3], [30], [6], [31], [17], [7], [32], [33], [15], [34], [35], [36]。尽管如此，这些技术仍倾向于通过批次延迟回收，以降低回收算法的开销，但这会增加内存占用。有关这些基于批次的回收技术的综述见 [1], [13], [33]。本节重点介绍能够实现即时回收 [37], [33]、因而与条件访问最为相关的技术。

Zhou 等人[37] 利用一系列短硬件事务，以接力方式执行，从而设计出保留即时内存回收特性的并发数据结构。该技术依赖于为数据结构添加一张元数据表，而这可能导致伪冲突。因此，它似乎不如条件访问通用。此外，我们发现，为只读操作频繁启动和提交事务会引入显著延迟。

VBR [33] 为并发数据结构中每个节点的每个可变字段附加元数据。它还要求使用类型保持型分配器，在这种分配器中，已解除链接的节点永远不会被归还给操作系统。线程可以通过每个字段的元数据检测释放后使用错误；这些元数据会与相应字段原子地更新。虽然 VBR 支持即时回收，但只有等到积累了一批节点后再通过单次操作进行回收，效率才最高。

另一方面，条件访问 无需任何元数据即可实现安全回收，并且只利用底层缓存一致性协议的隐式记账机制。此外，由于它不对系统中的线程数量作任何假设，因此能够完全适应 [38]。此外，硬件事务内存可以利用线程在发生内存冲突时立即中止这一特性，加速基于时序的攻击 [39], ；特别是，事务回滚可能导致数据泄露 [40],。相比之下，我们认为条件访问 风险较低，因为线程必须轮询才能获知远程一致性事件。

B. 关于相似同步技术的讨论

条件访问受到 Alistarh 等人的内存标记提案启发，但又与之大不相同 [16]。或许最显著的区别在于，条件访问 解决了安全内存回收问题（而且还支持即时回收），同时提供了有助于设计并发数据结构的同步原语。相比之下，内存标记并未解决内存回收问题，而且要求数据结构设计者依赖单独的安全内存回收算法，而这些算法各有取舍。

从编程接口的角度来看，条件访问 提供了 cread，这对我们的即时内存回收技术至关重要。内存标记中没有与 cread 指令等效的指令，而且目前尚不清楚如何利用内存标记实现 cread。我们还通过将标记操作整合到 cread 中简化了标记流程，而内存标记则要求程序员在读取前显式使用*AddTag*指令。

从实现角度来看，条件访问不需要修改底层的一致性协议，而内存标记的失效与交换（IAS）指令则需要，因为这条指令可以使许多（可能不连续的）远程缓存行失效（这些缓存行可能跨越许多页面）。此外，条件访问 每个缓存行只需要 1 位（双路超线程时每个缓存行需要 2 位），而内存标记还需要维护一个地址集合，以供 IAS 使其失效。

值得注意的是，乍看之下，条件访问 可能与带提前释放（ER）的硬件事务内存（HTM） [41], [42]相似。或许可以利用带提前释放的 HTM 实现我们工作的许多特性。然而，这样做会带来各种缺点。HTM 默认会将所有读取和写入都纳入读写集合，其中包括栈、分配器、库代码等。在数据结构中，许多读取和写入都需要释放，这会显著增加指令计数。这可能会降低性能，而且接口可能不如条件访问方便。

实际上，一些商用硬件事务内存支持基于区域（而非逐次访问）的禁用跟踪功能，例如英特尔的新型 TSXLDTRK [43], 以及 IBM 的 TSUSPEND/TRESUME [44],。但这并不会释放对已读取位置的加载跟踪，只会阻止跟踪未来的访问。这与条件访问提出的 untag 指令不同，后者允许释放任何先前访问过的位置。在提前释放的提案中，我们没有发现任何能够释放写入的方案，不过 AMD 于 2008 年提出的 ASF 方案允许逐次决定是否进行跟踪 [45]。然而，ASF 至今仍未实现。此外，我们尚未试验 TSXLDTRK，但 TSUSPEND 的开销相对较高。

与硬件事务内存不同，条件访问 完全不需要写集，因此硬件实现更简单，冲突跟踪与解决也更简单。我们认为，条件访问 解决了乐观数据

结构中的一个重要问题，而且所需硬件更少（如第三节所示）。我们希望，针对条件访问 的软硬件协同设计方法能够促使并发数据结构领域发现解决现有并发问题的新颖而高效的方法。

VII. 结论

本文介绍了条件访问，这是一种硬件扩展，使并发数据结构能够立即回收内存，而无需引入新的线程间协调。条件访问速度很快。与其他方案不同，它无需调优即可实现高性能，并且专为现代乐观数据结构的需求而设计。

迄今为止，我们已将条件访问 用于简单的非阻塞数据结构以及基于乐观锁的数据结构。未来，探究条件访问 能否也用于更复杂的无锁数据结构，将会很有意义。我们还认为，条件访问 与非易失性主存技术的结合蕴藏着令人振奋的机遇。

致谢

本研究得到了以下资助：加拿大自然科学与工程研究委员会（NSERC）探索计划项目（RGPIN-2019-04227）；加拿大创新基金会约翰·R·埃文斯领袖基金（获安大略研究基金加拿大创新基金会领袖机会基金等额资助，项目编号：38512）；加拿大自然科学与工程研究委员会探索启动补助金（DGECR-2019-00048）；美国国家科学基金会资助（项目编号：CNS-CSR-1814974）；以及滑铁卢大学。本文所表达的研究结果和观点仅代表作者本人，并不一定反映资助机构的意见。我们还感谢匿名审稿人帮助我们改进本文。

参考文献

[1] T. A. Brown，“为无锁数据结构回收内存：必定有更好的方法”，载于*2015*年*ACM*分布式计算原理研讨会论文集，2015年，第261–270页。[2] M. M. Michael，“危险指针：无锁对象的安全内存回收”，*IEEE*并行与分布式系统汇刊，第15卷，第6期，第491–504页，2004年。[3] D. Alistarh、P. Eugster、M. Herlihy、A. Matveev和N. Shavit，“Stacktrack：一种用于并发内存回收的自动化事务方法”，载于第九届欧洲计算机系统会议论文集，2014年，第1–14页。[4] D. Alistarh、W. Leiserson、A. Matveev和N. Shavit，“Forkscan：面向现代操作系统的保守内存回收”，载于第十二届欧洲计算机系统会议论文集，2017年，第483–498页。[5] “Threadscan：自动化且可扩展的内存回收”，D. Alistarh、W. Leiserson、A. Matveev和N. Shavit编，第4卷，第4期。ACM，纽约，纽约州，美国，2018年，第1–18页。[6] O. Balmau、R. Guerraoui、M. Herlihy和I. Zablotchi，“面向并发数据结构的快速而健壮的内存回收”，载于第*28*届*ACM*算法与体系结构中的并行性研讨会论文集，2016年，第349–359页。[7] N. Cohen，“每种数据结构都值得拥有无锁内存回收”，*ACM*编程语言汇刊，第2卷，OOPSLA专刊，第1–24页，2018年。[8] R. Nikolaev和B. Ravindran，“Hyaline：快速且透明的无锁内存回收”，载于*2019*年*ACM*分布式计算原理研讨会论文集，2019年，第419–421页。

[9] H. Wen、J. Izraelevitz、W. Cai、H. A. Beadle 和 M. L. Scott，《基于区间的内存回收》，ACM SIGPLAN Notices，第 53 卷，第 1 期，第 1–13 页，2018 年。[10] D. L. Detlefs、P. A. Martin、M. Moir 和 G. L. Steele Jr，《无锁引用计数》，分布式计算，第 15 卷，271，2002 4，第 255 页。[11] T. E. Hart、P. E. McKenney、A. D. Brown 和 J. Walpole，《无锁同步的内存回收性能》，《并行与分布式计算杂志》，第 67 卷，第 12 期，第 1270–1285 页，2007 年。[12] R. Nikolaev 和 B. Ravindran，《通用无等待内存回收》，载于第 25 届 ACM SIGPLAN 并行编程原理与实践研讨会论文集，2020 年，第 130–143 页。[13] A. Singh、T. Brown 和 A. Mashtizadeh，《NBR：基于中和的回收》。美国纽约州纽约：计算机协会，2021 年，第 175–190 页。 [在线]。获取地址：https://doi.org/10. 1145/3437801.3441625[14] E. V Mware，《了解 VMware ESX 4.1 中的内存资源管理》。[15] R. Nikolaev 和 B. Ravindran，《Crystalline：快速且节省内存的无等待回收》，arXiv 预印本 arXiv:2108.02763，2021 年。[16] D. Alistarh、T. Brown 和 N. Singhal，《内存标记：面向可扩展并发数据结构的极简同步》，载于第 32 届 ACM算法与体系结构并行性研讨会论文集，2020 年，第 37–49 页。[17]《无锁数据结构的自动内存回收》，N. Cohen 和 E. Petrank 编，第 50 卷，第 10 期。ACM，美国纽约州纽约，2015 年，第 260–279 页。[18] P. E. McKenney、D. Sarma、I. Molnar 和 S. Bhattacharya，《扩展 RCU 以支持实时和嵌入式工作负载》，载于渥太华 Linux 研讨会，页码 v2，2006 年，第 123–138 页。[19] S. Heller、M. Herlihy、V. Luchangco、M. Moir、W. N. Scherer 和 N. Shavit，《一种惰性并发链表集合算法》，载于国际分布式系统原理会议。施普林格，2005 年，第 3–16 页。[20] F. Ellen、P. Fatourou、E. Ruppert 和 F. van Breugel，《非阻塞二叉搜索树》，载于第 29 届 ACM SIGACT-SIGOPS 分布式计算原理研讨会论文集，2010 年，第 131–140 页。[21] R. K. Treiber，《系统编程：应对并行性》。国际商业机器公司，Thomas J. Watson研究中心，1986 年。[22] M. M. Michael 和 M. L. Scott，《简单、快速且实用的非阻塞与阻塞并发队列算法》，载于第十五届 ACM 分布式计算原理年度研讨会论文集，1996 年，第 267–275 页。[23] J. E. Miller、H. Kasture、G. Kurian、C. Gruenwald、N. Beckmann、C. Celio、J. Eastep 和 A. Agarwal，《Graphite：用于多核处理器的分布式并行模拟器》，载于 HPCA-16 2010 第十六届国际高性能计算机体系结构研讨会。IEEE，2010 年，第 1–12 页。[24] K. Fraser，《实用无锁性》，剑桥大学计算机实验室，技术报告，2004 年。[25] P. E. McKenney 和 J. D. Slingwine，《读-复制-更新：利用执行历史解决并发问题》，载于《并行与分布式计算及系统》，第 509518 卷，1998 年。[26] M. Herlihy、V. Luchangco、P. Martin 和 M. Moir，《动态大小数据结构的非阻塞内存管理支持》，《ACM计算机系统汇刊》（TOCS），第 23 卷，196，2005 2，第 146 页。[27] D. Dice、M. Herlihy 和 A. Kogan，《面向高度并发数据结构的快速非侵入式内存回收》，载于 2016 年 ACM SIGPLAN 内存管理国际研讨会论文集，2016 年，第 36–45 页。[28] G. E. Blelloch 和 Y. Wei，《并发引用计数与资源管理：具有常数时间开销且无等待》，arXiv 预印本 arXiv:2002 07053，2020 年。[29] P. Ramalhete 和 A. Correia，《简短公告：危险时期——非阻塞内存回收》，载于第 29 届 ACM算法与体系结构并行性研讨会论文集，2017 年。[30] A. Dragojevi´c、M. Herlihy、Y. Lev 和 M. Moir，《硬件事务内存简化内存管理的能力》，载于第 30 届 ACM SIGACT-SIGOPS 分布式计算原理研讨会论文集，2011 年，第 99–108 页。[31] N. Cohen 和 E. Petrank，《通过乐观访问实现无锁数据结构的高效内存管理》，载于第 27 届 ACM

算法与体系结构中的并行性研讨会，2015 年，第 254–263 页。[32] R. Nikolaev 和 B. Ravindran，《面向无锁数据结构的无快照、透明且健壮的内存回收》，载于第 *42* 届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集，2021 年，第 987–1002 页。[33] G. Sheffi、M. Herlihy 和 E. Petrank，《VBR：基于版本的回收》，载于第 *33* 届 *ACM*算法与体系结构并行性研讨会论文集，2021 年，第 443–445 页。[34] A. Correia、P. Ramalhete 和 P. Felber，《Orcgc：自动无锁内存回收》，载于第 *26* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会论文集，2021 年。[35] D. Anderson、G. E. Blelloch 和 Y. Wei，《具有常数时间开销的并发延迟引用计数》，载于第 *42* 届 *ACM SIGPLAN* 编程语言设计与实现国际会议论文集，2021 年，第 526–541 页。[36] N. Ben-David、G. E. Blelloch、P. Fatourou、E. Ruppert、Y. Sun 和 Y. Wei，《空间和时间有界的多版本垃圾回收》，*arXiv* 预印本 *arXiv:2108.02775*，2021 年。[37] T. Zhou、V. Luchangco 和 M. Spear，《具有精确内存回收的逐步交接事务》，载于第 *29* 届 *ACM*算法与体系结构并行性研讨会论文集，2017 年，第 255–264 页。[38] M. Herlihy、V. Luchangco 和 M. Moir，《空间和时间自适应非阻塞算法》，《理论计算机科学电子札记》，第 78 卷，第 260–280 页，2003 年。[39] M. Lipp、M. Schwarz、D. Gruss、T. Prescher、W. Haas、A. Fogh、J. Horn、S. Mangard、P. Kocher、D. Genkin、Y. Yarom 和 M. Hamburg，《熔断漏洞：从用户空间读取内核内存》，载于第 *27* 届 *USENIX* 安全研讨会（*USENIX Security 18*），2018 年。[40] 英特尔，《英特尔事务同步扩展（Intel TSX）异步中止》，技术报告 CVE-2019-11135，2019 年。[41] N. Sönmez、C. Perfumo、S. Stipic、A. Cristal、O. S. Unsal 和 M. Valero，《unreadtvar：扩展 Haskell 软件事务内存以提升性能》。《函数式编程趋势》，第 8 卷，114，2007 年，第 89 页。[42] T. Skare 和 C. Kozyrakis，《提前释放：利还是弊》，载于事务内存工作负载研讨会，第 77 卷，2006 年。[43] 英特尔，《Intel®64 和 IA-32 架构软件开发人员手册》，技术报告，文档编号：252046-070，2022 年 12 月（在线）。[在线]。获取地址：https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html[44] H. Q. Le、G. L. Guthrie、D. E. Williams、M. M. Michael、B. G. Frey、W. J. Starke、C. May、R. Odaira 和 T. Nakaike，《IBM Power8 处理器中的事务内存支持》，《*IBM* 研究与开发杂志》，第 59 卷，第 1 期，第 8–1 页，2015 年。[45] D. Christie、J.-W. Chung、S. Diestelhorst、M. Hohmuth、M. Pohlack、C. Fetzer、M. Nowack、T. Riegel、P. Felber、P. Marlier 等，《在完整事务内存栈中评估 AMD 的高级同步设施》，载于第 *5* 届欧洲计算机系统会议论文集，2010 年，第 27–40 页。