arXiv:2301.00996v2 [cs.DC] 7 Jan 2023

非阻塞数据结构的事务组合

Wentao Cai、Haosen Wen 和 Michael L. Scott
{wcai6,hwen5,scott}@ur.rochester.edu罗切斯特大学

2023年1月10日

摘要

本文介绍了一种新的方法 *nonblocking transaction composition* (NBTC)，用于对并发数据结构上的非阻塞操作进行原子组合。与以往的软件事务内存（STM）方法不同，NBTC 利用现有非阻塞数据结构的线性化性，将必须一起原子执行的内存访问次数减少到通常每个操作仅一次（这些访问通常是各组成操作的线性化指令）。

我们称为 *Medley* 的 NBTC 无阻塞实现，使得将大多数非阻塞数据结构转换为事务版本变得简单，同时保留其活性和高并发性。在我们的实验中，Medley 的性能比 Lock-Free Transactional Transform（LFTT）这一此前最快的竞争方法高出 40–170%。与依次执行独立操作相比，Medley 进行事务组合所带来的额外开销约为 2.2×。

对于 *persistent* 数据结构，我们观察到，借助基于纪元的 *periodic persistence*，事务的故障原子性几乎可以“零成本”实现。为此，我们将 Medley 与 *nbMontage*集成，后者是一个用于周期性持久化数据结构的通用系统。由此产生的*txMontage* 提供 ACID 事务，其吞吐量最高可达 OneFile 持久化软件事务内存系统的两个数量级。

1 引言

非阻塞并发数据结构最早于20世纪70年代开始受到研究，至今仍是一个活跃的研究课题。在这类数据结构中，不存在任何可达的系统状态会阻止单个操作取得进展。这种活性 属性对追求高可扩展性、且对资源占用线程遭遇不合时宜的抢占所导致的高尾延迟十分敏感的多线程程序而言，非常理想。

许多多线程系统，包括金融、旅行 [30], 仓库管理 [6], 以及一般数据库系统，都需要将操作组合成事务，并确保事务要么全部完成、要么完全不发生（即原子地执行）。然而，并发数据结构只保证单个操作的原子性；跨多个操作组合事务需要付出不小的编程工作量，还会引入很高的开销。要为每个事务都保持非阻塞活性，则更加困难。

一种潜在的解决方案存在于软件事务内存 （STM）系统中，这类系统能将几乎任意顺序代码转换为推测性事务。若干软件事务内存系统可提供非阻塞进展 [10, 19, 25, 26, 37]。大多数系统会对每次内存访问进行插桩，并安排重启在单个加载和存储层面发生冲突的操作。由此产生的编程模型很有吸引力，但插桩通常会带来 3–10× 的开销 [34, 第 9.2 节。3]

受软件事务内存启发，Spiegelman 等人 [36]提出了事务数据结构库 （TDSL），为某些经过手动修改的并发数据结构引入（阻塞式）事务。TDSL 注意到，只需跟踪关键节点上的读取，因为这些节点的更新可能表明存在语义冲突。通过这种方式，TDSL 缩小了读集，并且比通用软件事务内存系统具有更好的性能。

Herlihy 和 Koskinen [18] 提出了事务增强，这是一种（阻塞式）方法，允许软件事务内存系统将现有并发数据结构上的操作纳入其中。通过采用语义锁系统（例如，在映射中为每个键设置一把锁），事务可以并发执行，只要其增强操作在逻辑上相互独立，而不受底层冲突的影响。若事务因语义冲突（或增强代码之外的底层冲突）而重启，就会通过执行明确指定的逆操作，回滚所有已完成的增强操作。例如，insert(k,v) 操作可以通过执行 remove(k) 来回滚。事务增强利用了现有数据结构所具有的高并发潜力，但其本质上基于锁，且并非完全通用：例如，单链先进先出队列上的操作就没有明显的逆操作。

与 TDSL 同期开展的工作中，Zhang 等人 [43] 提出了无锁事务转换 （LFTT）这一组合非阻塞数据结构的非阻塞方法。该方法基于这样的观察：只有特定的节点——那些对事务语义至关重要的节点——才真正与冲突管理有关。LFTT 结构上的每个操作都会在每个关键节点上发布其所属事务的描述符，以便冲突事务能够相互查看并协助。例如，remove(7) 操作会在其结构中键为 7 的节点上发布其事务的描述符。最初，LFTT 仅支持静态事务，即其组成操作均已预先确定的事务。随后，LaBorde 等人 [23] 提出了动态事务转换 （DTT），将 LFTT 推广到动态事务（以 λ 表达式指定）。与此同时，Elizarov 等人 [8]提出了 LOFT。LOFT 与 LFTT 类似，但避免了错误地重复提供协助。

遗憾的是，与事务增强一样，识别关键

节点的需求往往会将 LFTT 和 DTT 限制在表示集合和映射的数据结构上。DTT 的发布与帮助机制还要求操作之间的“粘合”代码完全可重入（以允许帮助线程 [23] 并发执行），并且在发生冲突时可能导致冗余工作。更糟的是，对于读密集型工作负载，LFTT 和 DTT 要求读者对写者可见，从而引入元数据更新，显著增加缓存一致性协议中的竞争。

在我们的工作中，我们提出了非阻塞事务组合（NBTC），这是一种新方法，可为各种并发数据结构创建事务版本，同时保持非阻塞进展，且开销显著低于传统的软件事务内存（STM）。NBTC 的核心思想是：对于已有的非阻塞结构，只有关键内存访问——主要是线性化加载和比较并交换（CAS）指令——需要原子地执行；而大多数线性化前内存访问都可以在遇到时安全地执行，线性化后访问则可以推迟到事务提交之后。

与软件事务内存（STM）相比，NBTC 显著减少了需要插桩的内存访问次数，通常每个组成操作只需插桩一次。与事务增强和事务转换不同，NBTC 将关注点从语义重新转向底层内存访问，从而能够对现有结构进行机械变换，并支持几乎任意的抽象——远不止集合和映射。NBTC 还支持动态事务、不可见读者，以及事务各操作之间不可重入的“胶水”代码。兼容性唯一的要求是，组成操作的线性化点必须能够立即识别：每个操作都必须能够在运行时判断自身何时已线性化，而无需执行任何额外的共享内存访问。文献中大多数非阻塞结构似乎都满足这一要求。

为了评估 NBTC 的实用性，我们构建了一个无阻塞实现 $Medley$。该实现采用 Harris 等人的多字 CAS 变体 [16]，以原子地执行每个事务的关键内存访问，并在发现冲突事务时立即解决它们。借助 Medley，我们创建了 Michael 和 Scott 的队列 [29], 、Fraser 的跳表 [10],、Dick 等人的旋转跳表 [7], 、Michael 的链式哈希表 [28], 以及 Natarajan 和 Mittal 的二叉搜索树 [31]的 NBTC 版本。所有转换都十分直接。

用数据库事务的传统术语来说， [15], Medley 提供隔离性和一致性。在持久内存领域近期研究成果的基础上，我们还将 Medley 与 Cai 等人的 $nbMontage$ 系统集成， [2] 创建了一个名为 $txMontage$ 的系统，它还提供故障原子性和持久性——也就是说，支持完整的 ACID 事务。具体而言，我们利用 nbMontage 的 纪元 系统，将时间划分为较粗粒度的时间区间，并在发生故障时恢复到最近的纪元边界状态。txMontage 将纪元编号检查纳入其多字 CAS，从而确保同一事务的操作始终在线性化于同一个纪元，进而“几乎免费地”实现故障原子性和持久性。

贡献总结如下：

• (第2节) 我们介绍非阻塞事务组合 (NBTC)，这是一种用于组合非阻塞数据结构操作的新方法。

• (第3节) 基于 NBTC，我们实现了 Medley，这是一个用于事务性非阻塞结构的通用系统。Medley 易于使用的 API 和机械化转换使得将兼容的非阻塞结构转换为事务形式变得简单。

• (第4节) 我们将 Medley 与 nbMontage 集成，创建了 txMontage，它不仅提供事务隔离和一致性，还提供故障原子性和持久性。

• (第5节) 我们论证，使用 NBTC 和 Medley，由非阻塞结构组成的事务是非阻塞且严格可串行化的。我们还论证，采用 txMontage 的事务可提供严格可串行化的一种持久化变体，类似于 Izraelevitz 等人提出的缓冲持久线性化 [21]。

• (第6节) 我们展示了性能结果，证实 Medley 带来的开销相对较小，并且能够扩展至大量线程。具体而言，Medley 的性能优于 LFTT $1.4\times$ 至 $2.7\times$，并且比 TDSL 和 OneFile 非阻塞软件事务内存 [33] 系统快一个数量级。在持久内存上，txMontage 的性能比非阻塞持久化软件事务内存高两个数量级。

2 非阻塞事务组合

非阻塞事务组合 （NBTC）是一种新方法，充分利用了非阻塞数据结构操作的线性化性。NBTC 通过仅原子地执行组合操作的关键内存访问来实现严格可串行化。它支持文献中大量的非阻塞数据结构（下文将对此进行更精确的界定），同时保留转换后数据结构的高并发性和非阻塞活性。

2.1 NBTC可组合性

NBTC可组合性的关键在于可立即识别的线性化点。具体而言：

定义1。 如果数据结构操作具有可立即识别的线-性化点，则：

1. 静态地，我们可以识别出每条可能充当操作线性化点的指令。这类指令必须是只读操作中的加载，或更新操作中的比较并交换（*CAS*）；*2.* 动态地，在执行一条可能实现线性化的指令后，我们可以判断它是否确实是线性化点。线性化加载必须

在操作返回前得到识别；线性化 *CAS* 必须在不执行任何额外共享内存访问的情况下得到识别。

一个操作的代码中可能有多个潜在线性化点，但在任何一次调用中，其中只有一个会成为线性化点。

定义 2. 如果一个非阻塞数据结构的每个操作都具有可立即识别的线性化点，那么它就是 NBTC-composable 。

虽然有可能放宽这一定义，但当前版本适用于大量现有的非阻塞结构。

2.2 方法论

众所周知，大多数非阻塞操作都包含一个“规划”阶段和一个“清理”阶段，两者由一条线性化指令 [12, 38] 分隔。执行规划阶段并不意味着该操作必定成功；如有需要，任何线程都可以执行清理。NBTC 的基本策略是先为当前事务中的所有组成操作执行规划，然后将这些操作一起原子地线性化，最后执行所有清理。我们对现有数据结构和组合模式的调查揭示了这一策略面临的两个主要难题。

第一个复杂之处涉及发布点的概念：操作可能已对其他线程可见，但尚未线性化。由于发布可能改变其他线程的行为，因此通常（与线性化点一样）在整个事务准备好提交之前，发布都必须保持推测状态。Natarajan 和 Mittal 的二叉搜索树 [31], 便是一个例子，其中更新操作 o 可能执行 CAS，以发布其即将线性化的意图，但此时尚未真正线性化。在这个发布点之后， o本身或任何遇到该发布通知的其他更新操作都可能尝试线性化 o （出于性能考虑，读取操作会忽略该通知）。值得注意的是，用于帮助其他（已线性化）操作、且不会透露当前操作性质的 CAS 指令，不必视为发布。

第二个复杂之处出现在事务 t 对同一数据结构执行两个或更多操作，且后续某个操作（称为 o_2）依赖于较早某个操作（称为 o_1）的结果时。此时，执行 t 的线程必须像 o_1 已经完成那样继续执行，但其他线程必须忽略它。如果 o_1 需要清理（NBTC 通常会将清理延迟到事务提交之后），那么 o_2 可能需要先帮助 o_1 ，才能继续执行；而其他事务甚至不应察觉到 o_1的存在。

这两种复杂情况都可以通过引入推测区间这一概念来处理：在该区间内，必须一并完成 CAS 指令，操作才能作为事务的一部分生效。这与*CAS* 执行器阶段类似，该阶段存在于规范化的非阻塞数据结构中 [38], ，但两者并不相同，主要是因为第二种复杂情况。对于一个变得可见的操作

在其线性化点之前，只需将发布点与线性化点之间（含两端）的所有 CAS 操作纳入推测区间即可。对于一个 o_2 需要看到同一事务中较早操作 o_1 的操作，只需跟踪该事务的写入，并且最迟从访问 o_1所写入位置的第一条指令开始 o_2的推测区间。

定义 3。更精确地说，我们称

• 若历史 H 中线程 t 的操作 o 中的 *CAS* 指令满足：不存在 H 的扩展 H'，使得 t 在 H' 中不再执行任何指令，但仍会在 H' 中线性化，那么该 *CAS* 指令就是无害的。

• o 中第一个非无害的 *CAS* 指令就是 o 的发布点（通常它也就是线性化点）。

• o 的推测区间始于发布点，或始于首次观察到同一事务中某个更早操作推测性写入的值的指令（以较早者为准），并延续至 o 的线性化点。

• 只读操作中的加载若是该操作可立即识别的线性化点，则为关键。更新操作中的 *CAS* 若位于推测区间内，则为关键操作。

不失一般性，我们假设对共享内存的所有更新（初始化其他线程尚不可见的对象除外）都通过 CAS 实现。

根据这些定义，NBTC 方法很直接：为了对 NBTC 可组合数据结构上的一组操作进行原子执行，我们对每个操作进行转换，使得：(1) 事务遇到推测区间之前的指令以及推测区间内的非关键指令时，立即执行这些指令；(2) 以推测方式执行关键指令，因此只有在事务提交时，这些指令才会原子地生效；(3) 将推测区间之后的指令推迟到提交之后执行。

3 Medley 系统

为了说明 NBTC，我们编写了一个名为 $Medley$ 的系统，它能够：(1) 插桩关键指令，以推测方式执行这些指令，并使用我们提出的 M-compare-N-swap 原子地提交这些指令；该方法是 Harris 等人 [16] 提出的多字 CAS 的一种变体；(2) 识别并及时解决事务冲突；(3) 将非关键清理操作延迟到事务提交时执行。

3.1 API

图 1 总结了 Medley 的 API。使用此 API，我们按如下方式将 NBTC-Composable 数据结构转换为事务结构：

```
template <class T> class CASObj { // Augmented atomic object
  T nbtcLoad();
  bool nbtcCAS(T expected, T desired, bool linPt, bool pubPt);
  /* Regular atomic methods: */
  T load(); void store(T desired); bool CAS(T expected, T desired);
};
class Composable { // Base class of all transactional objects
  template <class T> void addToReadSet(CASObj<T>*,T); // Register load
  void addToCleanups(function); // Register post-critical work
  template <class T> T* tNew(...); // Create a new block
  template <class T> void tDelete(T*); // Delete a block
  template <class T> void tRetire(T*); // Epoch-based safe retire
  TxManager* mgr; // Tx metadata shared among Composables
  struct OpStarter { OpStarter(TxManager*); } // RAII op starter
};
class TxManager { // Manager shared among composable objects
  void txBegin(); // Start a transaction
  void txEnd(); // Try to commit the transaction
  void txAbort(); // Explicitly abort the transaction
  void validateReads(); // Optional validation for opacity
};
struct TransactionAborted : public std::exception{ };
```

图 1：Medley 用于事务组合的 C++ API。

1. 将关键加载和比较并交换操作分别替换为 nbtcLoad 和 nbtcCAS。应使用 CASObj 模板声明进行此类访问的字段。2. 在读取操作中，为关键加载调用 addToReadSet，记录地址和加载的值。3. 通过 addToCleanups 注册每个操作在关键部分之后的工作。4. 将每个 new 和 delete 分别替换为 tNew 和 tDelete。将每个 retire（用于安全内存回收——SMR）替换为 tRetire。5. 在每个操作的开头声明一个 OpStarter 对象。

CASObj<T> 为每个可执行 CAS 操作的 64 位字（例如 atomic<Node*>）增加额外的元数据位，用于推测跟踪（详情见第 3.2 节）。它提供专用的加载和 CAS 操作，以及 atomic<T> 的常用方法。为了动态识别推测区间，nbtcCAS 接受两个额外参数 linPt 和 pubPt，用于指示该调用若成功，是否将构成其操作的线性化点和/或发布点。类似地，可以在 nbtcLoad 之后调用 addToReadSet，以指示（在检查返回值之后）这是否是（或很可能是）只读操作的线性化加载，并应在提交时进行验证跟踪。

Composable 是事务对象的基类。它提供了各种与 NBTC 相关的方法，包括支持安全内存回收 （SMR），用于确保只有在能够确定其他线程的私有变量中不再存在任何引用后，才回收节点。目前，我们的 SMR 实现采用基于纪元的回收 [10, 17, 27]。为增强通用性，Composable 还提供了用于事务增强的 API，可将基于锁的操作纳入 Medley 事务中（当然，代价是失去非阻塞进展的能力，

当然，这会牺牲非阻塞进展）。本文不再进一步讨论这一机制。

TxManager 类负责管理事务元数据，并提供启动、中止和完成事务的方法。所有计划在同一事务中使用的 Composable 实例共享同一个 TxManager 实例。在每次操作调用中，管理器都会通过 OpStarter() 判断当前执行是在事务内部还是外部。如果在事务外部，则省略所有事务插桩；如果在事务内部，则按照 NBTC 方法进行插桩。

鉴于非阻塞操作在系统的任何可达状态下都能安全执行，通常没有必要在冲突发生时立即停止注定要中止的事务——也就是说，保证 不透明性 [14]。在特殊情况下（例如，事务后续操作无法在某些参数组合下调用，或者中止的可能性足够大，以至于延迟中止可能影响性能），可以使用 validateReads 方法来判断先前的读取是否仍然正确。

为说明 Medley 的用法，图 2 标出了 Michael 的非阻塞哈希表 [28] 中必须为 NBTC 修改的代码行；图 3 则展示了一个修改两个哈希表的事务示例。在实际应用中，TransactionAborted 的 catch 块通常会循环回到事务代码的开头重试，并可能加入额外代码以避免活锁（例如采用退避策略，或向底层调度器提供提示）。与软件事务内存系统不同，Medley 不会对数据结构操作之间的事务内“粘合”代码进行插桩。这些代码始终作为普通代码在事务外部执行，并且应始终无数据竞争；如果它们产生副作用，那么程序员编写的、用于处理中止事务的 catch 块应在程序员决定重试或放弃之前对这些副作用进行补偿。

3.2 M-比较-N-交换

为了原子地执行每个事务的关键内存访问，我们采用一种软件模拟的 *M-compare-N-swap* (MCNS) 操作，该操作基于 Harris 等人的双重比较单次交换（RDCSS）和多字 CAS（CASN） [16]。每个事务都维护一个描述符，其中包含读集、写集以及由线程 ID、序列号和状态构成的 64 位三元组，如图 4 所示。描述符在 TxManager 实例中按线程预先分配，并在不同事务间重复使用。状态可以是 InPrep（初始状态）、InProg（准备提交）、Committed（InProg 时验证成功后）或 Aborted（在 InPrep 时被另一个线程显式中止，或因验证失败而中止）。

每个可能发生关键内存访问的原始 64 位字都会附加一个 64 位计数器，两者共同组成一个 128 位 CASObj。每次关键 CAS 都会在 CASObj 中安装指向其描述符的指针，并递增计数器；提交或中止时，则卸载该描述符并再次递增计数器。我们利用 x86 上的 128 位 CAS 指令，原子地同时更改原始字和计数器。当计数器为奇数时

```
class MHashTable {
struct Node { K key; V val; next; }; // from p,
find c >= k; may be used
bool find( * &p, Node* &c, Node* &n, K k); optional<V> get(K key) {
  prev = nullptr; Node *curr,
*next; optional<V> res = {}; if (find(prev,curr,next,key)) res = curr->val;
  return res; }
optional<V> put(K key, V val) { // insert or replace if key exists
  prev = nullptr; optional<V> res = {};
  if (find(prev, key)) { // update
Node *newNode {key, val), *curr, *next; while(true) {
    newNode->next.store(curr);
    if (curr->next. (next,mark(newNode)) {
      res = curr->val; {
        if (prev->CAS(curr,newNode)) (curr);
        else find(prev,curr,next,key); };
        // execute right away if not in tx break;
    } else { // key does not exist; insert
      newNode->next.store(curr);
    if (prev- (curr,newNode )) break; } } return res;
}};
```

图 2：Michael无锁哈希表示例（与 Medley 相关的部分突出显示 ）.

S

```
void doTx(MHashTable* ht1, MHashTable* ht2, V v, K a1, K a2) {
  TxManager* mgr=ht1->mgr; assert(mgr==ht2->mgr);
  try { // transfer ‘v’ from account ‘a1’ in ‘ht1’ to ‘a2’ in ‘ht2’
    mgr->txBegin();
    V v1 = ht1->get(a1); V v2 = ht2->get(a2);
    if (!v1.hasValue() or v1.value() < v) mgr->txAbort();
    ht1->put(a1, v1.value() - v); ht2->put(a2, v + v2.valueOr(0));
    mgr->txEnd();
  } catch (TransactionAborted) { /* transaction aborted */ }
}
```

图 3：Michael 的哈希表上的事务示例。

```
struct Desc {  map<CASObj* addr,{uint64 val,cnt}>* readSet;
  map<CASObj* addr {uint64 oldVal,cnt,newVal}>* writeSet;
  atomic<uint64> status;//63..50 tid 49..2 serialNumber 1..0 status
  enum STATUS { InPrep=0, InProg=1, Committed=2, Aborted=3 }; };
struct CASObj { atomic<uint128> val_cnt; };
```

图 4：描述符和 CASObj 结构。

CASObj 包含一个指向描述符的指针，即使它是一个真实值也是如此。

每个 MCNS 实例都会依次经历安装描述符、确定状态和卸载描述符这几个阶段。前两个阶段位于数据结构操作的关键路径上。新事务会在其描述符中初始化元数据（在 txBegin 时）：清空读集和写集，递增序列号，并将状态重置为 InPrep。随后，在事务执行过程中进入安装阶段：每次关键加载都会将其地址、计数器和值记录到读集中。每次关键 CAS 都会将其地址、旧计数器、旧值和期望的新值记录到写集中；随后，它会在 CASObj 中安装一个指向该描述符的指针。安装阶段的伪代码见图5。

为了免去程序员推理计数器的负担，nbtcLoad 会记录其 ⟨计数器、对象⟩ 对（图 5 第 15 行）；随后，addToReadSet 会将这一对（以及指定的 CASObj）添加到事务的读集（第 20 行）。

当线程遇到自己的描述符时，nbtcLoad 会从写集中返回推测值（第 11 行）。同样，nbtcCAS 会更新写入条目（第 34 行）。此类遇见会自动启动推测区间（第 10、30 和 32 行），并持续到当前操作的线性化点（第 38 行）。

如果某个操作遇到其他线程的描述符，就会调用 tryFinalize（图 6）将该描述符移开。若描述符处于 InPrep 状态，此方法会中止关联事务；若处于 InProg 状态，则会协助完成提交；无论哪种情况，都会从发现该描述符的 CASObj 中卸载它。当线程被迫中止，或到达事务末尾并尝试提交时，也会执行类似操作（第 39–58 行）。无论是在协助他人还是代表自己执行操作，执行 MCNS 的线程都必须验证：该描述符仍负责其被发现时所在的 CASObj（第 9 行）；并且（若要提交）读集中的值仍然有效（第 25 行）。将状态 CAS 为 Committed 或 Aborted 后，线程会从所有关联的 CASObj 中卸载该描述符，并将指向描述符的指针替换为相应的更新值（第 31 和 34 行）。卸载完成后，对于提交，所有者线程会调用清理例程（第 55 行）；对于中止，则会释放通过 tNew 分配的块（第 43 行）。

我们的设计采用不可见读者和急切冲突管理，以提高效率并保持简洁。急切冲突管理可能导致活锁——事务反复中止彼此——因此只能保证障碍自由。惰性（提交时）冲突管理配合某种描述符安装全序，或许能让我们预——

```
void TxManager::txBegin() {
  desc->readSet->clear(); desc->writeSet->clear();
  status.store((status.load() & ~3) + 4);
}
T CASObj::nbtcLoad() {
retry:
  {val,cnt} = val_cnt.load();
  if (cnt % 2) { // is descriptor
    if (val == desc) {
      startSpeculativeInterval();
      return desc->writeSet[this].newVal;
    } else val->tryFinalize(this, {val,cnt});
    goto retry; // until object has real value
  }
  ... /* Record 'this' and 'cnt' to be added to readSet */
  return val;
}
void Composable::addToReadSet(CASObj<T>* obj, T val) {
  ... /* Retrieve 'cnt' by 'obj' */
  mgr->readSet[obj] = {val,cnt};
}
bool CASObj::nbtcCAS(T expected,T desired,bool linPt,bool pubPt){
retry:
  {val,cnt} = val_cnt.load();
  if (cnt % 2) { // is descriptor
    if (val != desc) { // not own descriptor
      val->tryFinalize(this, {val,cnt});
      goto retry; // until object has real value
    }
    startSpeculativeInterval();
  } else if (val != expected) return false;
  if (pubPt) startSpeculativeInterval();
  if (inSpeculativeInterval()) { // Is critical CAS
    desc->writeSet[this] = {val,cnt,desired};
    bool ret = true;
    if (!(cnt % 2)) ret = this->CAS({val,cnt},{desc,cnt+1});
    if (!ret) desc->writeSet.remove(this);
    if (linPt and ret) endSpeculativeInterval();
    return ret;
  } else return CAS(expected, desired);
}
```

图5：MCNS 安装阶段的伪代码。

```
bool Desc::stsCAS(uint64 d, STATUS expected, STATUS desired) {
  d = d & ~3; return status.CAS(d + expected, d + desired);
}

bool Desc::setReady(){return stsCAS(status.load(),InPrep,InProg);}

bool Desc::commit(uint64 d){return stsCAS(d,InProg,Committed);}
bool Desc::abort(uint64 d){return stsCAS(d,d & 1,Aborted);}
void Desc::tryFinalize(CASObj* obj, uint128 var) {
  uint64 d = status.load();
  if (obj->val_cnt.load() != var) // ensure d indicates right tx
    return;
  if (d & 3 == InPrep) {
    abort(d);
    uint64 newd = status.load();
    if (newd & ~3 != d & ~3) return; // serial number mismatch
    d = newd;
  }
  if (d & 3 == InProg) {
    if (validateReads(d)) commit(d);
    else abort(d);
  }
  uninstall(status.load());
}
bool Desc::validateReads() {
  for (e:*readSet)
    if ({e.val,e.cnt} != e.addr->load()) return false;
  return true;
}
void Desc::uninstall(uint64 d) {
  if (d % 3 == Committed)
    for (e:*writeSet)
      e.addr->CAS({this,e.cnt+1}, {e.newVal,e.cnt+2});
  else // Aborted
    for (e:*writeSet)
      e.addr->CAS({this,e.cnt+1}, {e.oldVal,e.cnt+2});
}
struct TxManager {
  threadLocal vector<Function> cleanups, allocs;
  threadLocal Desc* desc;
  void txAbort() {
    desc->abort(d);
    for (f:allocs) f(); // undo tNew
    throw TransactionAborted();
  }
  void txEnd() {
    if (!desc->setReady()) txAbort();
    uint64 d = desc->status.load();
    if (!desc->validateReads()) desc->abort(d);
    else if (d & 3 == InProg) desc->commit(d);
    d = desc->status.load();
    if (d & 3 == Committed) {
      desc->uninstall(d);
      for (f:cleanups) f();
    } else txAbort();
  }
};
```

图6：最终确定事务的方法的伪代码。

为能够提供无锁保证的数据结构提供无锁性 [35], ，但这会显著增加跟踪和检索未提交更改的复杂性，也无法解决饥饿问题；在实际应用中，饥饿可能比活锁更严重。我们认为，这些实现选择与 NBTC 方法的有效性正交，因此留待未来工作研究。

4 持久内存

事务最初起源于数据库领域；事务内存（TM）则将其引入多线程程序中的内存中数据结构。低成本、低功耗的字节寻址非易失性内存（NVM）问世后，我们有机会将这两条历史脉络融合起来，理想情况下充分利用 NBTC。具体而言，TM 旨在将顺序代码转换为线程安全的并行代码，而 NBTC 则假设——如同数据库领域的情形——我们已经拥有高效的线程安全数据结构，并希望将它们的操作以原子且持久的方式组合起来。鉴于这一假设，如第 3.1 节末尾所述，让程序员负责各操作之间的“衔接”代码，并专注于组合操作的原子性和持久性，似乎是恰当的。

4.1 持久线性化

在采用易失性缓存的机器上，只有程序费心发出经过仔细选择的写回和栅栏指令，NVM 中的数据结构通常才能在崩溃后保持一致。为描述期望的行为，Izraelevitz 等人 [21] 将持久线性化引入为持久化数据结构的一种正确性判据。如果一个数据结构在无崩溃执行期间是线性化的，并且在省略崩溃事件后，其长期历史仍保持线性化，那么它就是持久线性化的。等价地说， [11], 每个操作都应在其调用与响应之间持久化，且持久化操作的顺序应与线性化顺序一致。

近年来，已经设计出许多具备持久线性化性的非阻塞数据结构 [3, 9, 11, 44]。多个研究团队还提出了相关方法，使现有的非阻塞结构能够实现持久线性化 [12,13, 21]。另一些团队开发了持久化软件事务内存系统，但其中大多数采用基于锁的方式 [4, 5, 24, 40]。据我们所知，OneFile [33]和 QSTM [1] 是仅有的非阻塞持久化软件事务内存系统。OneFile 使用全局序列号串行化事务，无需读集并提高了读取效率，但需要对数据结构进行侵入式修改，并使用 128 位宽 CAS。QSTM 使用全局持久化队列管理活跃事务，无需宽位 CAS，也无需进行侵入式结构性变更，但其执行仍然本质上是串行的。

4.2 降低持久性开销

遗憾的是，写回和栅栏指令往往具有较高的延迟。由于操作必须在返回前持久化，持久线性化似乎本质上代价高昂。对软件事务内存进行即时持久化会带来额外开销，因为用于事务并发控制的元数据也必须及时写回并执行栅栏操作。

为将高延迟指令移出应用程序的关键路径，Izrael-evitz 等人 [21] 提出了缓冲式持久线性化（BDL）的概念。BDL 允许在崩溃后恢复期间丢失崩溃前执行过程中的一小段后缀（只要整体历史仍保持线性化性），从而使写回和栅栏指令能够批量执行，移出应用程序的关键路径。需要在与外界通信之前确保数据持久化的应用程序，可以使用同步操作，这与传统文件系统和数据库中的此类操作类似。

周期性持久化最初是在 *Dalı* 持久化哈希表的背景下提出的，随后被 nbMontage [2], 这一通用系统采用，用于为现有非阻塞结构创建 BDL 版本。nbMontage 系统将挂钟时间划分为“纪元”，并在每个纪元结束时批量持久化操作。若在纪元 e期间发生崩溃，系统会将所有结构恢复到纪元 $e-2$结束时的状态。为了在没有崩溃的情况下最大限度地提高吞吐量，nbMontage 还区分了语义上重要的数据（即“有效载荷”）和仅用于提升性能的数据（例如索引）；后者可以保存在动态随机存取存储器中，并在恢复期间重建。例如，映射的有效载荷只是一组键值对；与之关联的哈希表、树或跳表则存放在易失性动态随机存取存储器中。队列的有效载荷是 ⟨序列号、项目⟩ 对。

为确保崩溃后恢复始终反映每个结构的一致状态，每个 nbMontage 操作都必须在线性化于其有效载荷所标记的纪元中。完成时间“过长”的操作可能会被迫中止并重新开始。整个 nbMontage 系统是无锁的；sync 实际上是无等待的。

4.3 持久严格可串行化

当然，线性化性并不适用于事务，因为事务必须保持推测状态，直到所有操作都能一并对外可见。软件事务内存系统通常提供严格可串行化：在无崩溃历史中，事务看起来按某种顺序发生，且该顺序遵循实时顺序（如果 A 在B 开始之前提交，那么 A 必须先于 B串行化） [34, 第 3.1 节。2]。对于 NBTC 的持久化版本，我们需要考虑崩溃。

与 Izraelevitz 等人 [21], 一样，我们假设采用全系统崩溃故障模型：崩溃后数据结构仍然存在，但只能由新线程访问——旧线程会消失。在此模型下：

定义 4. 若执行历史 H 在省略崩溃事件后满足严格可串行化，则称其满足持久严格可串行化*(DSS)*。

与持久线性化一样，这一定义要求崩溃前完成的所有工作在崩溃后都可见。缓冲版本的定义类似：

定义 5. 如果存在每个崩溃间隔的一个符合先发生关系的一致切割，使得在省略崩溃事件以及每个崩溃间隔中切割后的后缀后，执行历史 H 在 H 上具有严格可串行化特性，那么该执行历史 H 就展现了缓冲持久严格可串行化（*BDSS*）。

4.4 将 Medley 与 nbMontage 合并

nbMontage 的纪元系统提供了一种自然机制，可为 Medley 事务提供故障原子性和持久性：如果同一事务的操作始终发生在同一纪元中，那么在崩溃后，它们将一同恢复（或一同丢失）。基于这一观察，我们将这两个系统合并，创建了$txMontage$。给定事务中的所有操作的有效载荷都标记为相同的纪元编号。随后，在 MCNS 提交期间，该编号会与读集的其余部分一起验证，以确保事务在预期的纪元中提交。尽管 nbMontage 本身相当复杂，但只需这一项小小的改动，就足以将它（以及它所转换的所有持久化数据结构）接入 Medley：持久性几乎可以“免费”融入事务。

5 正确性

本节将论证：由格式良好的 Medley 事务组成的历史记录是严格可串行化的；Medley 是无阻塞的；而 txMontage 提供缓冲持久严格可串行化。

定义 6. 如果满足以下条件，则 *Medley* 事务是格式良好的：

1. 以txBegin 开始，并以txEnd结束，中间可以选择包含txAbort ； *2.* 包含 *NBTC* 转换的数据结构的操作；*3.* 所有其他事务内代码均无阻塞，且不产生任何未由处理程序管理的副作用，以处理TransactionAborted 异常。

5.1 严格可串行化

引理 1. 在实现层面（对构成系统内存的字数组进行操作），nbtcLoad、nbtcCAS、tryFinalize、txAbort 和txEnd*(MCNS)* 都是线性化操作。

证明（概要）。这直接由 Harris 等人 [16] 的结果得出。他们的 RDCSS 只比较（而不更改）单个位置，而他们的 CASN 支持更新所有 被触及的字，但证明可以直接调整。具体而言，与 RDCSS 一样，未成功的 tryFinalize 或 txEnd 可以在线性化于一次（失败的）验证读，或其状态字的一次失败 CAS。其

状态 CAS 成功的 tryFinalize 或 txEnd 会“追溯”到过去，在其第一次验证读时线性化。（具有讽刺意味的是，这意味着 MCNS 在实现层面并没有一个可立即识别的线性化点。）
□

引理 2. 在任何历史中，若事务 t 在 CASObj o上执行 nbtcLoad 或nbtcCAS 操作 x，且 t 的 txEnd 操作 y 成功，则在 x 与 y之间，不会有修改 o 的其他事务的 tryFinalize 或txEnd 操作成功。

证明（概要）。假设结论不成立，并将执行冲突的 tryFinalize 或 txEnd 的事务称为 u。如果 u 对 o 执行的 nbtcCAS 发生在 x 和 y之间，它将中止并卸载 t的描述符，或者导致 y中的读验证失败，这与 t的 txEnd 成功这一假设矛盾。如果 u 对 o 执行的 nbtcCAS 发生在 x之前，那么 x 将中止并卸载 u的描述符，这与 u的 tryFinalize 或 txEnd 在 x之后成功这一假设矛盾。 □

定理 3. 由良构 *Medley* 事务组成的历史是严格可串行化的。

证明（概要）。在经过 NBTC 转换的数据结构中，所有关键内存访问都将通过 nbtcLoad 或 nbtcCAS 执行。之后的某个时刻会调用 txEnd。根据引理 2，若该调用成功，则期间不会有冲突的 tryFinalize 或 txEnd 调用成功。这又意味着，我们的 Medley 历史等价于一个顺序历史，其中每个操作都在与 NBTC 转换之前原始数据结构操作的线性化点相对应的 nbtcLoad 或 nbtcCAS 处生效。此外，同一事务的所有操作在该顺序历史中都是连续的——也就是说，我们的 Medley 历史是严格可串行化的。
□

5.2 障碍自由

定理 4. 用于构建中止时会重试的格式良好事务时，*Medley* 是无阻塞的。

证明（概要）。 在任何可达的系统状态中，如果一个线程继续执行，而其他线程均暂停，那么每个遇到冲突的 nbtcLoad 或 nbtcCAS 都会先完成（提交或中止）所遇到的描述符，将其卸载，并安装自己的描述符。如果线程遇到的是自己的描述符，nbtcLoad 将返回推测值，而 nbtcCAS 会在参数与写集中先前的新值匹配时更新写集。无论哪种情况，MCNS 都会取得进展。如果它最终中止，则可以重新执行一轮全新的 MCNS；由于没有新引入的竞争，该轮必定会提交。
□

5.3 缓冲持久严格可串行化

定理 5. 由格式良好的 *txMontage* 事务组成的历史具有缓冲耐久严格可串行化性质。

证明（概要）。每个事务都会在 txBegin 中读取当前纪元 e。然后，它会在 MCNS 提交期间验证该纪元编号。根据引理 1，此 MCNS 必须在线性化点 e 内线性化。由于 nbMontage 为同一纪元中的所有操作提供故障原子性，该定理显然成立。

□

6 性能结果

如第 1 节所述，我们使用 Medley 创建了 Michael 和 Scott 的队列 [29], Fraser 的跳表 [10], Dick 等人的旋转跳表 [7],Michael 的链式哈希表 [28], 以及 Natarajan 和 Mittal 的二叉搜索树 [31]的 NBTC 版本。所有转换都很直接。本节将报告 Medley 和 txMontage 哈希表及跳表的性能，并将其与文献中的各种替代方案进行比较。

具体而言，我们测试了以下易失性系统：

Medley – 如前所述（哈希表和跳表）

OneFile – Ramalhete 等人 [33] 的无锁软件事务内存的易失性版本（哈希表和跳表）

TDSL – Spiegelman 等人 [36] 的事务数据结构库（仅作者的跳表）

LFTT – Zhang 等人的无锁事务转换 [43] （仅作者的跳表）

我们还测试了以下持久化系统：

txMontage – Medley + nbMontage（哈希表和跳表）

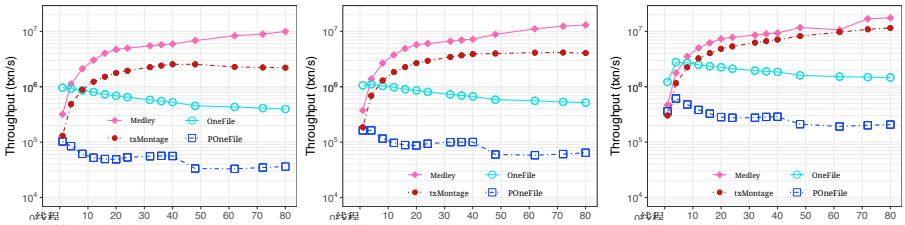
POneFile – OneFile 的持久化版本 [33] （哈希表和跳表）

6.1 实验设置

我们报告哈希表和跳表微基准测试的吞吐量，以及用于运行部分 TPC-C [6] 的跳表的吞吐量。我们还测量跳表的延迟。所有代码都将在会议论文发表前公开。

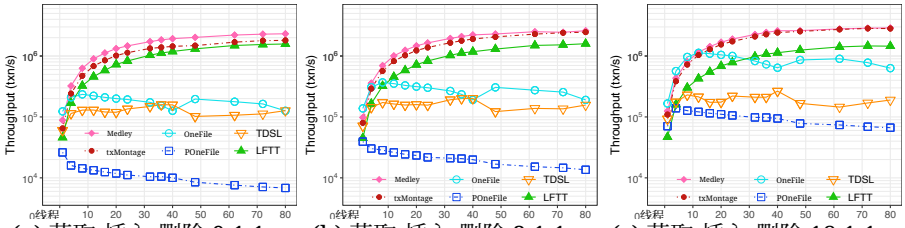
所有测试均在一台运行 Linux 5.3.7 (Fedora 30) 的服务器上进行，该服务器配备两颗英特尔至强 Gold 6230 处理器。每个处理器插槽有 20 个物理核心和 40 个超线程，共计 80 个超线程。所有实验中的线程首先按每个核心一个的方式固定到处理器插槽 0 上，然后分配到该插槽的额外超线程，最后分配到处理器插槽 1 上。每个处理器插槽有 6 个通道，每个通道配备 32 GB 动态随机存取存储器和 128 GB Optane 双列直插式内存模块。我们将每个处理器插槽上的非易失性内存分别挂载为独立的 ext4 文件系统。在所有实验中，动态随机存取存储器均按照 Linux 默认策略分配到两个处理器插槽；对于持久化数据结构，仅使用处理器插槽 0 上处于直接访问（DAX）模式的非易失性内存。所有情况下，我们报告三次试验的平均值，每次试验运行 30 秒。

我们的吞吐量和延迟微基准测试首先将 0.5 M 个键值对预加载到数据结构中，这些键值对取自包含 1 M 个键的键空间。键和值均为 8 字节整数。在基准测试阶段，每个线程



(a) 获取:插入:删除 0:1:1　(b) 获取:插入:删除 2:1:1　(c) 获取:插入:删除 18:1:1

图 7：事务性哈希表的吞吐量（对数 Y 轴）。



(a) 获取:插入:删除 0:1:1　(b) 获取:插入:删除 2:1:1　(c) 获取:插入:删除 18:1:1

图8：事务性跳表的吞吐量（log Y 轴）。

会编排并执行事务，每个事务包含 1 到 10 个操作。操作（针对均匀随机选取的键）从获取、插入和删除中选择，其比例由参数指定（在我们的实验中为 0:1:1、2:1:1 或 18:1:1）。

在 OneFile 中，我们使用通过软件事务内存并行化的顺序链式哈希表。在 Medley 中，我们使用经过 NBTC 转换的 Michael无锁哈希表 [28]。每个哈希表都有 1 M 个桶。在 OneFile 和 TDSL 中，跳表源自 Fraser 的基于软件事务内存的跳表 [10]。在 LFTT 和 Medley 中，跳表源自 Fraser 的基于比较并交换的非阻塞跳表 [10]。每个跳表最多有 20 层。

对于 TPC-C，我们受到 Fraser 的跳表不支持范围查询这一事实的限制。借鉴 Yu 等人 [42] 在使用 DBx1000[42], 进行实验时的做法，我们将实验限制在 TPC-C 的 newOrder 和付款事务上，并以 1:1 的比例执行。这些是该基准测试中占主导地位的事务；两者都不执行范围查询。

6.2 吞吐量（易失性）

哈希表和跳表微基准测试的吞吐量结果分别见图 7 和图 8。实线表示针对易失性数据结构的事务；虚线表示持久化事务。暂且只考虑易失性情形，在线程数不再微不足道时，无论是哈希表还是跳表，Medley 的表现始终比 OneFile 的易失性版本高出一个数量级以上。工作负载中写入所占比例越高，差距就越大。尽管缺乏可扩展性，OneFile 在线程数较少时表现良好，尤其是在以读为主的工作负载下。我们将这一点归因于它的串行化事务设计，

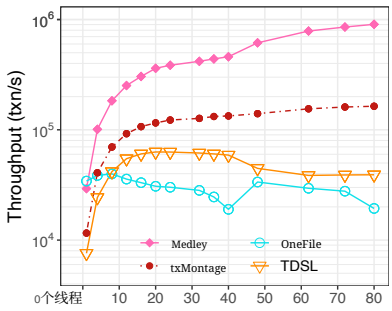


图9：TPC-C 跳表吞吐量（对数 Y 轴）。

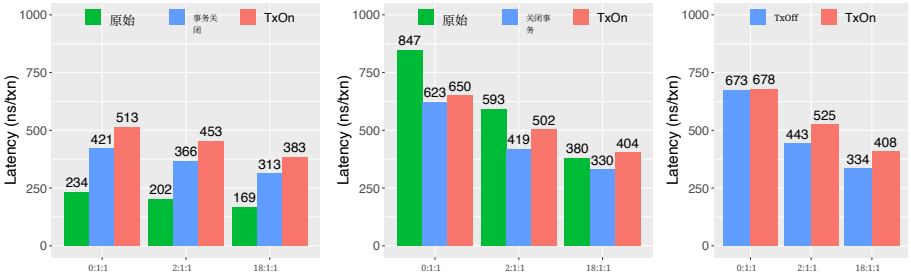
该设计消除了对读集的需求。

如第 1 节所述，TDSL 在各种专门构造的数据结构上提供（阻塞式）事务。虽然写入仍会发生冲突，但读集仅限于语义上关键的节点；作者报告称，与通用软件事务内存相比，吞吐量显著提升。然而，如图8所示，TDSL 和 OneFile 一样，可扩展性有限，且性能远不及 Medley。令我们有些意外的是，在这个微基准测试中，TDSL 的表现也不及 OneFile，这大概是因为后者消除了读集。

在各种跳表竞争方案中，LFTT 最接近 Medley，但即使在表现最佳的（仅写）情况下，仍落后 1.4–2× 倍。LFTT 需要重新执行整个事务，因此会产生大量冗余工作，尤其是规划工作。在 Medley 能够受益于不可见读者的读密集型工作负载下，LFTT 则落后 2–2.7× 倍。

作为一项稍具现实性的基准测试，我们针对 TPC-C 的 newOrder 和 payment 事务，重新比较了 Medley、OneFile 和 TDSL。由于 LFTT 仅支持静态事务（即事先已知数据结构操作集合的事务），因此我们无法将其纳入这些测试；我们也无法集成其动态变体 DTT [23]，因为现有代码版本不支持任意的键和值类型。不过，LaBorde 等人 [23]报告称，在简单事务上，DTT 的性能与 LFTT 相近。鉴于 DTT 必须在其所有关键节点上以 λ 表达式的形式发布整个事务，我们预计 DTT 在 TPC-C 的大型事务上的性能即便不更差，也会略差一些；而在微基准测试中，LFTT 的速度本就比 Medley 慢约 2× 。

Medley、（易失性）OneFile 和 TDSL 在 TPC-C 上的吞吐量如图 9 所示。由于 TPC-C 上的事务规模较大，OneFile 受到的影响很严重。通过仅保证关键访问的原子性，Medley 在大量线程下仍能扩展，性能最高可比竞争方案高出 45×。



(a) 在动态随机存取存储器上 (b) 在非易失性内存上易失 (c) 在非易失性内存上持久化

图10：40 个线程下跳表的平均延迟。X 标签表示获取:
插入:删除的比例。

6.3 延迟（易失性）

为了评估事务组合的边际成本，我们在 Fraser 的原始跳表（**原始版本**——无事务）、经 NBTC 转换但不含事务的跳表（**关闭事务**——不调用 txBegin 或 txEnd），以及经 NBTC 转换且含事务的跳表（**开启事务**——如图 8 所示）上重新运行了微基准测试。

图10a报告了存放在动态随机存取存储器中的数据结构的延迟。无事务时，转换后的跳表比原始跳表慢 1.8× 。开启事务后，速度约慢 2.2× 。这些结果表明，CAS 操作（安装和卸载描述符）的成本增加了一倍多，约占 Medley 开销的 $^2/_3$ 。

6.4 持久性

为了评估故障原子性和持久性对 txMontage 吞吐量的影响，我们可以回看图7、图8和图9中的虚线。

吞吐量

在微基准测试中，采用严格持久化和急切缓存行写回时，持久化 OneFile 比其瞬态版本慢一个数量级。然而，采用周期性持久化时，在仅写工作负载下，txMontage 哈希表在 40 个线程时的吞吐量达到 Medley 的一半——比 POneFile 快近两个数量级。对于哈希表上的以读为主的工作负载，或跳表上的任何工作负载（其整体并发性较低），txMontage 的速度几乎与 Medley 一样快。在极端写密集型情况下（80 个线程运行 0:1:1 哈希表工作负载），我们将 txMontage 约 4× 的性能下降归因于非易失性内存的写入瓶颈 [22]——尤其是写放大 [20, 41]这一现象。

TPC-C 中的结果也类似（图 9）。这里的事务规模较大，且写入操作繁重；在非易失性内存上分配有效载荷，使 txMontage 的吞吐量仅约为 Medley 的五分之一，但仍比易失性 OneFile 快约 4× 倍。

至于 POneFile，在 TPC-C 的预热阶段耗时太久，我们失去了耐心，便终止了测试。

延迟

图10b展示了当 txMontage 的有效载荷分配在非易失性内存中（而索引位于动态随机存取存储器中），但关闭持久性时（不使用纪元或显式缓存行回写），跳表事务的延迟。作为对比，我们还展示了原始非事务跳表的延迟，其所有数据均存放在非易失性内存中。图10c展示了 txMontage 完全运行时对应的延迟。

对比图10a和图10b可见，在非易失性内存上运行时，事务的边际开销更低。这或许表明，硬件写入瓶颈降低了整体吞吐量，因而减少了竞争。

在仅写工作负载下（最左侧的柱状组），将有效载荷移至非易失性内存会带来近 50% 的开销（图10a 与图10b 对比）。在以读为主的工作负载下（最右侧的柱状组），这一开销降至 5%。同样，我们将这一现象归因于非易失性内存的写入瓶颈。原始跳表完全分配在非易失性内存中时延迟很高（图10b 中的绿色柱状图），这似乎印证了这一假设。

对比图 10b 和图 10c 可知，相较于在非易失性内存上运行的 Medley，txMontage 为实现故障原子性和持久性所付出的代价不到 5%。

7 结论

我们提出了非阻塞事务组合（NBTC），这是一种新方法，可在构建动态事务时利用现有非阻塞数据结构的线性化性。作为具体实现，我们引入了用于瞬态结构的$Medley$ 系统，以及用于（缓冲式）持久化结构的$txMontage$ 系统。Medley 事务具有隔离性和一致性；txMontage 事务还具备故障原子性和持久性。这两个系统都相当快速：过去即使是最优秀的软件事务内存系统，性能也通常会下降 3–$10\times$ 倍，而 Medley 的性能下降幅度更接近 $2.2\times$ 倍；txMontage 则只比 nbMontage 增加 5–20% 的开销，因此其性能比现有非阻塞持久化软件事务内存系统快近两个数量级。

由于采用急切竞争管理，Medley 和 txMontage 能够确保非阻塞结构上的事务具备障碍自由性。在未来的工作中，我们计划探索惰性竞争管理，即推迟安装描述符，直到事务准备好提交。通过按规范顺序对描述符排序并进行安装，生成的系统将保持无锁性。惰性竞争管理还有助于协助，因为任何已安装描述符的状态都会是 ==InProg，其他任何线程都可以推动其完成。

按照 NBTC 目前的定义，推测区间 很容易识别，但可能会不必要地对 publication 与线性化之间某些无害的协助指令进行插桩。

我们目前正在努力制定一个更精确但仍切实可行的协助定义，以减少每个事务中必须原子地执行的“关键”内存访问次数。

参考文献

[1] H. Alan Beadle、Wentao Cai、Haosen Wen 和 Michael L. Scott。非阻塞持久化软件事务内存。载于第*27*届国际高性能计算、数据与分析会议（*HiPC*），第283–293页，线上会议，2020年12月。[2] Wentao Cai、Haosen Wen、Vladimir Maksimovski、Mingzhe Du、Rafaello Sanna、Shreif Abdallah 和 Michael L. Scott。面向并发数据结构的快速非阻塞持久化。载于第*35*届国际分布式计算研讨会（*DISC*），第14:1–14:20页，德国弗赖堡，2021年10月。[3] Zhangyu Chen、Yu Huang、Bo Ding 和 Pengfei Zuo。面向持久内存的无锁并发分层哈希。载于*Usenix* 年度技术会议（*ATC*），第799–812页，线上会议，2020年7月。[4] Joel Coburn、Adrian M. Caulfield、Ameen Akel、Laura M. Grupp、Rajesh K. Gupta、Ranjit Jhala 和 Steven Swanson。NV-Heaps：利用新一代非易失性存储器实现快速、安全的持久化对象。载于第*16*届国际编程语言与操作系统体系结构支持会议（*ASPLOS*），第105–118页，加利福尼亚州纽波特比奇，2011年3月。[5] Andreia Correia、Pascal Felber 和 Pedro Ramalhete。Romulus：持久化事务内存的高效算法。载于第*30*届美国计算机协会并行算法与体系结构研讨会（*SPAA*），第271–282页，奥地利维也纳，2018年7月。[6] 事务处理委员会。TPC-C 基准测试（修订版 5.11.0）。http://www.tpc.org/tpcc/，2010年2月。[7] Ian Dick、Alan Fekete 和 Vincent Gramoli。面向多核的跳表。《并发与计算：实践与经验》，29(4)，2016年5月。[8] Avner Elizarov、Guy Golan-Gueta 和 Erez Petrank。LOFT：无锁事务数据结构。载于第*24*届美国计算机协会 *SIGPLAN* 并行编程原理与实践研讨会（*PPoPP*），第425–426页，华盛顿特区，2019年。

[9] Panagiota Fatourou、Elias Papavasileiou 和 Eric Ruppert。支持无等待范围查询的持久化非阻塞二叉搜索树。载于第*31*届 *ACM* 算法与体系结构并行性研讨会（*SPAA*），第275–286页，亚利桑那州菲尼克斯，2019年6月。

[10] Keir Fraser。《实用无锁》。博士论文，剑桥大学国王学院，2003年9月。作为剑桥大学计算机实验室技术报告 #579 发表，2004年2月。www.cl.cam.ac. uk/techreports/UCAM-CL-TR-579.pdf。[11] Michal Friedman、Maurice Herlihy、Virendra Marathe 和 Erez Petrank。面向非易失性内存的持久化无锁队列。载于第*23*届 *ACM SIGPLAN*并行编程原理与实践研讨会（*PPoPP*），第28–40页，奥地利维也纳，2018年2月。[12] Michal Friedman、Naama Ben-David、Yuanhao Wei、Guy E. Blelloch 和 Erez Petrank。NVTraverse：在NVRAM数据结构中，目的地比旅程更重要。载于第*41*届 *ACM*编程语言设计与实现会议（*PLDI*），第377–392页，线上会议，2020年6月。[13] Michal Friedman、Erez Petrank 和 Pedro Ramalhete。Mirror：让无锁数据结构实现持久化。载于第*42*届 *ACM*编程语言设计与实现会议（*PLDI*），第1218–1232页，线上会议，2021年6月。[14] Rachid Guerraoui 和 Michal Kapalka。论事务内存的正确性。载于第*13*届 *ACM SIGPLAN*并行编程原理与实践研讨会（*PPoPP*），第175–184页，犹他州盐湖城，2008年2月。[15]Theo Haerder 和 Andreas Reuter。面向事务的数据库恢复原理。*ACM*计算调查，15(4):287–317，1983年12月。[16] Timothy L. Harris、Keir Fraser 和 Ian A. Pratt。一种实用的多字比较并交换操作。载于第*16*届国际分布式计算研讨会（*DISC*），第265–279页，法国图卢兹，2002年10月。[17] Thomas E. Hart、Paul E. McKenney、Angela Demke Brown 和 Jonathan Walpole。无锁同步的内存回收性能。并行与分布式计算杂志（*JPDC*），67(12):1270–1285，2007年12月。[18] Maurice Herlihy 和 Eric Koskinen。事务增强：一种用于高并发事务对象的方法。载于第*13*届 *ACM SIGPLAN*并行编程原理与实践研讨会（*PPoPP*），第207–216页，犹他州盐湖城，2008年2月。[19] Maurice Herlihy、Victor Luchangco、Mark Moir 和 William N. Scherer III。适用于动态大小数据结构的软件事务内存。载于第*22*届 *ACM*分布式计算原理研讨会（*PODC*），第92–101页，马萨诸塞州波士顿，2003年7月。

[20] Daokun Hu、Zhiwen Chen、Wenkui Che、Jianhua Sun 和 Hao Chen。Halo：一种支持快速恢复的混合型持久内存/动态随机存取存储器持久化哈希索引。发表于数据管理国际会议（*SIGMOD*），第 1049–1063 页，费城，宾夕法尼亚州，2022年6月。[21] Joseph Izraelevitz、Hammurabi Mendes 和 Michael L. Scott。全系统崩溃故障模型下持久内存对象的线性化。发表于第 *30* 届国际分布式计算研讨会（*DISC*），第 313–327 页，法国巴黎，2016年9月。[22] Joseph Izraelevitz、Jian Yang、Lu Zhang、Juno Kim、Xiao Liu、Amirsaman Memaripour、Yun Joon Soh、Zixuan Wang、Yi Xu、Subramanya R. Dulloor、Jishen Zhao 和 Steven Swanson。英特尔傲腾 DC 持久内存模块的基本性能测量，2019年8月。arXiv 预印本 arXiv:1903.05714v3。[23]Pierre LaBorde、Lance Lebanoff、Christina Peterson、Deli Zhang 和 Damian Dechev。面向链式数据结构的无等待动态事务。发表于第 *10* 届多核与众核编程模型及应用国际研讨会（*PMAM*），第 41–50 页，华盛顿特区，2019年2月。[24] Mengxing Liu、Mingxing Zhang、Kang Chen、Xuehai Qian、Yongwei Wu、Weimin Zheng 和 Jinglei Ren。DudeTM：通过解耦为持久内存构建持久事务。发表于第 *22* 届编程语言与操作系统架构支持国际会议（*ASPLOS*），第 329–343 页，中国西安，2017年4月。[25] Virendra J. Marathe、Michael F. Spear、Christopher Heriot、Athul Acharya、David Eisenstat、William N. Scherer III 和 Michael L. Scott。降低软件事务内存的开销。发表于第 *1* 届 *ACM SIGPLAN* 事务计算研讨会（*TRANSACT*），加拿大安大略省渥太华，2006年6月。[26] Virendra Jayant Marathe 和 Mark Moir。迈向高性能非阻塞软件事务内存。发表于第 *13* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会（*PPoPP*），第 227–236 页，犹他州盐湖城，2008年2月。[27] Paul E. McKenney、Dipankar Sarma、Andrea Arcangeli、Andi Kleen、Orran Krieger 和 Rusty Russell。读-复制-更新。发表于渥太华 *Linux* 研讨会，第 338–367 页，加拿大安大略省渥太华，2001年7月。[28] Maged M. Michael。高性能动态无锁哈希表与基于链表的集合。发表于第 *14* 届 *ACM* 算法与体系结构并行性研讨会（*SPAA*），第 73–82 页，加拿大马尼托巴省温尼伯，2002年8月。[29] Maged M. Michael 和 Michael L. Scott。简单、快速且实用的非阻塞与阻塞并发队列算法。发表于第 *15* 届 *ACM* 算法与体系结构中的并行性研讨会

分布式计算原理（*PODC*），第 267–275 页，费城，宾夕法尼亚州，1996年5月。[30] Chi Cao Minh、JaeWoong Chung、Christos Kozyrakis 和 Kunle Olukotun。Stamp：斯坦福多处理事务应用。发表于*IEEE*国际工作负载特征研讨会（*IISWC*），第 35–46 页，西雅图，华盛顿州，2008年9月。[31] Aravind Natarajan 和 Neeraj Mittal。快速并发无锁二叉搜索树。发表于第 *19* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会（*PPoPP*），第 317–328 页，奥兰多，佛罗里达州，2014年2月。[32] Faisal Nawab、Joseph Izraelevitz、Terence Kelly、Charles B. Morrey III、Dhruva R. Chakrabarti 和 Michael L. Scott。Dal´ı：一种周期性持久化哈希映射。发表于第 *31* 届国际分布式计算研讨会（*DISC*），第 37:1–37:16 页，奥地利维也纳，2017年10月。[33] Pedro Ramalhete、Andreia Correia、Pascal Felber 和 Nachshon Cohen。OneFile：一种无等待持久化事务内存。发表于第 *49* 届 *IEEE/IFIP* 国际可靠系统与网络会议（*DSN*），第 151–163 页，波特兰，俄勒冈州，2019年6月。[34] Michael L. Scott。共享内存同步。Morgan & Claypool出版社，圣拉斐尔，加利福尼亚州，2013年。[35] Michael F. Spear、Luke Dalessandro、Virendra J. Marathe 和 Michael L. Scott。软件事务内存的全面竞争管理策略。发表于第 *14* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会（*PPoPP*），第 141–150 页，罗利，北卡罗来纳州，2009年2月。[36] Alexander Spiegelman、Guy Golan-Gueta 和 Idit Keidar。事务数据结构库。发表于第 *37* 届 *ACM* 编程语言设计与实现会议（*PLDI*），第 682–696 页，圣巴巴拉，加利福尼亚州，2016年6月。[37] Fuad Tabba、Mark Moir、James R. Goodman、Andrew W. Hay 和 Cong Wang。NZTM：非阻塞零间接寻址事务内存。发表于第 *21* 届 *ACM* 算法与体系结构并行性研讨会（*SPAA*），第 204–213 页，加拿大阿尔伯塔省卡尔加里，2009年8月。[38] Shahar Timnat 和 Erez Petrank。无锁数据结构的实用无等待模拟。发表于第 *19* 届 *ACM SIGPLAN* 并行编程原理与实践研讨会（*PPoPP*），第 357—368 页，奥兰多，佛罗里达州，2014年2月。[39] Stephen Tu、Wenting Zheng、Eddie Kohler、Barbara Liskov 和 Samuel Madden。多核内存数据库中的高速事务。发表于

第*24*届美国计算机协会操作系统原理研讨会（*SOSP*），第18–32页，宾夕法尼亚州法明顿，2013年11月。[40]Haris Volos、Andres Jaan Tack 和 Michael M. Swift。Mnemosyne：轻量级持久内存。载于第*16*届编程语言与操作系统体系结构支持国际会议（*ASPLOS*），第91–104页，加利福尼亚州纽波特比奇，2011年3月。[41] Zixuan Wang、Xiao Liu、Jian Yang、Theodore Michailidis、Steven Swanson 和 Jishen Zhao。非易失性内存系统的特征分析与建模。载于第*53*届微体系结构国际研讨会（*MICRO*），第496–508页，线上会议，2020年10月。[42]Xiangyao Yu、George Bezerra、Andrew Pavlo、Srinivas Devadas 和 Michael Stonebraker。直面深渊：对千核并发控制的评估。*VLDB*基金会会刊，8（3）：209–220，2014年11月。[43] Deli Zhang 和 Damian Dechev。适用于链式数据结构的无回滚无锁事务。载于第*28*届美国计算机协会算法与体系结构并行性研讨会（*SPAA*），第325–336页，加利福尼亚州太平洋丛林，2016年7月。[44] Yoav Zuriel、Michal Friedman、Gali Sheffi、Nachshon Cohen 和 Erez Petrank。高效的无锁持久化集合。*ACM*编程语言会刊，3(OOPSLA)：128:1–128:26，2019年10月。