摘要-硬件事务存储（HTM）；

ACID

Atomic,durable原子的，持久的-版本管理？ -内在intra-transaction；

isolation？ -并发控制； -inter transaction ；

Durability -日志记录/版本管理/内存持久性...;

在本文中，我们介绍了FlyHTM，一种快速的...; 同时优化两者;

\*即时进行日志/数据排序，明确的日志持久性指令和 sfence；

仅硬件？

软件支持？

处理器修改？

介绍了两种加速技术，用于日志记录和并发控制...；

# I.简介

NVM random access (more than 300 ns) is 10× slower than

DRAM [27]

==========;

HTM提案[15]；

在磁盘和闪存的持久性方面，几种NVRAM存储技术有望提供与DRAM相当的性能和字节寻址能力[21]，[29]。

非易失性存储器，称为持久性存储器...；

其中，英特尔[38]（Intel TSX）和IBM [16]都已经发布了包含HTM的商用处理器。

commercial商业HTM [10]；

== Kiln [39]和LAD [14]假定其他持久性组件更靠近CPU，即Kiln-LLC，LAD MC队列； 不归纳为traditional work； —需要对CPU体系结构及其cache一致性协议[cite-hoop]进行硬件修改。

HOOP [7]：类似于LAD和Kiln吗？

==在本文中，我们...; 总之，我们为缓解基于冲突的缓存定时攻击做出了以下贡献。

PiCL [27]：多撤消日志记录允许进行多个逻辑提交，同时仍维护单个中央撤消日志，从而无需同步缓存刷新。

通过对单个缓存行进行版本控制，异步缓存扫描进一步从关键路径中删除了缓存flush和异步执行检查点phase以最小化性能开销。

缓存驱动的日志记录。 通过抢先直接从片上缓存中获取撤消数据，现在可以在将这些条目写入NVM之前对其进行缓冲。

== ThyNVM [33]：ThyNVM是基于redo的WAL设计，其中内存转换表memory translation tables用于维护数据的提交版本和易失性执行版本。 它具有块大小和页面大小的混合检查点粒度，可以为具有较高空间局部性的工作负载提供良好的NVM行缓冲区使用率。 尽管ThyNVM仍在每个检查点都经受同步cache刷新停顿，但ThyNVM也将检查点阶段与执行阶段重叠，以最大程度地减少停顿，并且需要一个转换表, 这是一个无法扩展到大型多核系统的转换表。

Commit volatile 事务需要清除读/写集跟踪结构，并使推测状态对其他线程可见。 除上述步骤外，为了提交持久事务（使用重做日志记录），必须将重做日志条目写入持久性内存。

为什么重做日志的内存屏障效率低下：内存屏障会阻止后续的数据更新，直到事务完成先前的更新为止。但是，此写顺序控件可防止缓存通过合并和重新排序写操作来优化系统性能。 [28];

专注于HTM（例如[19]），有关软件TM的许多相关工作；

\*\*\*列出所有HTM和Persistent HTM相关论文；

其次，NVM具有接近DRAM的读取延迟，但与DRAM相比，其写入延迟约为10倍，例如，PCM的写入延迟为150〜1000ns，ReRAM为500ns，而DRAM的写入延迟仅为60ns [13]。

# 二。 问题

在本节中，我们回顾了持久性HTM的基础知识及其用于版本管理的构件-WAL预写日志记录。 它强制执行eager日志记录，该日志记录会立刻记录log 每条存储指令后在L1 cache，并且必须在数据之前保留该日志。 我们将这样的日志记录eager视为效率瓶颈，并研究它如何阻碍性能。

## A.持久HTM

新兴的非易失性内存（NVM）已将ACID应用于内存事务处理。 这主要归因于其类似磁盘的存储持久性和类似DRAM的访问延迟。 支持持久存储，NVM启用具有持久性的事务性存储解决方案

CPU不再需要使用缓慢的I / O进程将数据持久存储到磁盘中。 与磁盘不同，NVM提供了快速的字节寻址能力[21]，相当快地实现了内存中事务的ACID。 最新的技术将NVM与高效的HTM结合使用，通常被称为Persistent HTM [19]。

接下来，我们回顾一下传统的事务性内存Tx\_begin如何创建log（A）clwb log（A）sfence存储区（A’）clwb A’……Tx\_end实际上部分满足了ACID，以及持久性HTM如何弥补了这一差距。

**事务性内存（TM）。** TM在多核系统中采用内存中事务处理进行并发控制[15]。 多核系统可以同时执行任务，因此比单核系统提供更好的性能。 性能的提高依赖于线程执行的并行性。 这需要并发控制来跨线程同步，以便它们生成与在单个内核上执行的结果相同的结果。 主要的复杂性来自协调访问相同数据的线程。 因此，它是在数据库事务之后由TM建模的，数据库事务应满足ACID属性。

但是，基于易失性DRAM的传统TM解决方案仅支持原子性，一致性和隔离性，而没有持久性durability。 原子性要求如果事务成功提交或完全不执行，则事务似乎已完全执行。 在后一种情况下，即使未提交的事务已对某些数据进行了修改，其修改也不会生效。 一致性要求已提交的事务不能成为串行执行不可能生成的状态[9]。 隔离要求事务的结果不能被并发事务篡改。 它不一定禁止事务访问相同的数据。 相反，只要不破坏一致性，就可以允许事务共享数据。 此要求也定义为可串行性，其后并发事务的结果与通过其串行执行获得的结果一致。

**Persistent HTM :**  NVM的出现增强了TM的耐用性。 最近的发展支持其有效的硬件实现[6]，[12]，[19]，[26]，[28]。 传统上，Software TM（STM）[20]，[23]要求程序员在程序[8]中手动检测内存引用. 如图1（a）所示,STM分别使用begin\_transaction和end\_transaction指令启动和提交事务。 在事务内部，应在就地更新之前记录旧的或新的数据状态。 这需要更多的指令来创建日志（例如，create），持久化日志（例如，clwb）以及在持久化日志（例如，内存屏障防护）的同时停止内存访问。 这些额外的指令不仅给程序员带来负担，而且给[11]，[23]，[26]，[35]带来沉重的负担。 相反，Hardware TM（HTM）[6]，[12]，[19]，[20]，[26]-[28]仅保留用于描绘事务的指令，而将所有其他用于将内存引用的指令编排为硬件（ 图1（b））。 因此，持久性HTM有望为内存中事务处理提供一种高效的方法[5]，[19]，[28]。问题：STM和HTM之间的区别.

为了有效地实现ACID属性，Persistent HTM需要制定两个正交的设计策略-版本管理和冲突管理[25]。 首先，版本管理可确保事务内规模上的崩溃一致性。 如果发生崩溃，则事务的恢复数据应保持一致，因为它们完全遵循修改后的版本或原始版本。 其次，冲突管理检测并解决事务间冲突，以确保隔离性。 当事务同时访问相同的数据并且其中至少有一个试图修改它时，可能会发生冲突。 版本管理是确保事务正确性的基础，而冲突管理对于探索事务并行性至关重要。 在本文中，我们专注于版本管理，旨在以高效率在硬件中实施它。

## B.版本管理：

预写日志记录版本管理通常使用预写日志记录（WAL）来确保崩溃一致性。 作为WAL的主要原则，日志记录在内存层次结构中维护原始（旧）和修改（新）数据版本。 它在NVM中分配了一个专用的日志区域，用于记录某些版本的数据。 WAL确保在提交事务之前将日志保存到NVM中。然后，可以在提交时就地更新新数据。 一旦系统在提交期间崩溃，我们就可以使用持久化的日志来恢复崩溃事务的整个数据集的旧版本或新版本。 这样的数据恢复保证了数据的一致性，因为恢复的数据集不会与新旧值混合在一起。 与其他替代方案相比，WAL由于其简单性（例如就地更新）而被广泛采用。 例如，影子分页shadow paging[7]，[20]直接使用日志区域来存储更新。 如果没有就地更新，则对新数据值的后续访问应重定向到日志区域。 这需要一个重定向表，该表将存储旧数据的地址映射到存储其新值的地址。 新值的进一步更新使地址重定向成为一项昂贵的costly递归操作。

根据日志区域是保留旧版本，新版本还是两个版本的修改数据，WAL可以分别分为撤消日志记录，重做日志记录和撤消+重做日志记录（图2）。

Undo logging :记录旧数据并支持在提交事务之前就地更新。 如图2（a）所示，日志和数据写入类似于管道[28]。 撤消日志（例如UL（D1））保留后，就可以对数据（例如W（D1'）进行相应的写入，以将D1更新为D1'）。 Cache eviction后，可以将修改后的数据直接写入NVM。 当事务即将提交时，其某些修改后的数据可能仍在cache中。 此类数据也需要在事务提交之前使用强制cache line回写命令（例如clwb）写入NVM。 在事务提交之前和期间，系统可能会崩溃。 如果发生这种情况，则可能某些修改后的数据已写入NVM，而另一些仍在缓存中。 我们可以使用撤消日志将NVM中的已修改数据恢复为它们的旧值，并保证一致性[20]，[26]，[27]，[34]。

Redo logging记录新数据并仅在其保留所有相应的重做日志后才修改数据。 如图2（b）所示，记录D1’和Dn’新值的所有重做日志（即RL（·））应该在写入数据（即W（·））之前保留。 换句话说，重做日志记录将持久保存日志并顺序写入[11]，[18]，[19]，[23]，[24]，[35]。 否则，系统崩溃时可能会破坏数据一致性。 例如，如果1）我们在其他日志和写入之前同时保留RL（D1'）和W（D1'），并且2）在事务提交之前系统崩溃，系统仅包含D1的新值while ：

* Only the new values for data whose logs and writes are also persisted as well；
* only the old values for data whose logs and writes are not persisted yet；
* both old and new values for data whose logs are persisted while writes are not.

在这种情况下，系统崩溃很容易使系统与无法同时恢复到旧值或新值的数据混合在一起。 重做日志记录比不重做日志记录更加复杂。 由于在日志创建期间不会在缓存或NVM中更新日期，因此在缓存或in place数据已过时的情况下，读取访问需要首先通过日志区域[14]。

尽管重做日志记录不会在提交之前强制执行cache行写回操作，但在提交之后可能需要这样做[28]。 此要求是为了保证log area limited时的一致性。 在这种情况下，某些日志需要在日志溢出时evict。 如果它们记录的新值没有及时写回NVM，则系统崩溃将使这些新值消失并导致不一致。

撤消+重做日志记录可收获两全其美。 如图2（c）所示，它记录了修改数据的旧版本和新版本[28]。 首先，通过撤消日志可以在事务提交之前适当地(in place)更新已修改的数据。 其次，重做日志在提交之前不用强制写回。 但是，作为重做日志记录的固有限制，撤消+重做日志记录仍然需要在日志区域溢出时强制执行cache行写回操作。 此外，同时记录新旧版本都需要更多的日志记录时间和更大的日志区域[18]。

## C.局限性：

日志记录急切性: 尽管WAL有所改进，我们仍将急切的日志记录视为其效率瓶颈。 具体来说，急切日志记录会在每条存储指令[18] – [20]，[28]上触发L1缓存处的立即日志。 根据WAL原则，需要在数据之前保留日志（图2）。 因此，以前的解决方案通常会在缓存中进行就地更新之前生成日志，并通过绕过缓存层次结构以确保日志比数据运行得更快来将日志写入内存控制器。 图3通过典型的undo log记录方案展示了eager日志记录。

步骤x：L1cache控制器从CPU内核接收写入请求以存储A’。 存储指令触发创建撤消日志，其中包括元数据信息和A的旧值。

•步骤y：核心收集元​​日志信息（例如，事务ID和地址）并将其写入日志缓冲区。 同时，它访问L1缓存以获取A的旧值。

•步骤z：将A的旧值直接写入日志缓冲区。 与步骤y中的元日志信息一起，它完成了日志条目。

•步骤{：准备好日志后，将A的新值更新到缓存行。

显然，急切的日志记录会在写入请求到达与相应的就地缓存更新发生之间不可避免地inevitable产生延迟。 日志记录的所有其他额外操作（例如，收集元日志信息，如果是撤消日志记录则获取旧值，如果是从缓存进行重做日志记录则获取新值并将其写入日志条目）会花费额外的时钟周期。

此外，当事务强制对同一数据块进行多次写入时，急切的日志记录可能会导致wasted efforts。 对于撤消日志记录，日志需要跟踪track旧值。 每个修改后的数据块的第一个日志足以suffices恢复。 诸如ATOM [20]和PiCL [27]之类的解决方案仅在每条高速缓存行添加指示器位，以跟踪其中的数据是否已记录。 但是，对于重做日志记录，修改后的数据块的最后更新占主导地位。 由于急切的日志记录会在写请求到达后立即创建日志，因此当再次修改记录的数据时，日志很容易过时。 数据块修改的频率越多，引起的开销就越大. 换句话说，急切的日志记录导致写入密集型事务的更多潜在低效率。

最后, 另一个inefficiency arising由于locality unfriendliness。 如图3所示，写入日志会绕过缓存层次结构，因此会忽略其中的时间和空间局部性。 为了locality，某些解决方案在缓存和内存控制器[19]，[27]，[28]之间临时缓存日志。 可以将缓冲的日志合并为更大的块，以提高带宽效率。 但是，这需要额外的缓冲区空间和复杂性来处理其中的日志。

# 三 overview

在本节中，我们将介绍惰性日志记录，以解决现有持久性HTM设计中因急切日志记录而导致的低效率。 它直接利用LLC缓存的数据作为撤消日志，并在逐出缓存时将日志写入内存控制器。 这样，我们可以立即更新L1缓存的数据，而不会暂停写入操作，直到创建日志并将其写入内存控制器为止，就像在急切的日志记录中一样。 我们通过称为FlyHTM的更快的持久性HTM实现了懒惰日志记录。

## A.动机motivation

我们注意到修改缓存行时并不需要立即创建日志。 取而代之的是，在写回数据之前，日志是否可以在持久域中可用。 修改后的数据通常会保留在缓存中，并且可能会暂时不写回。 这在写入请求的到达与其触发的日志的写入之间提供了一个宽松relaxed的时间范围。 如果我们通过在缓存中进行更新来恢复宽松的时间范围，那么我们有望加快速度。 我们延迟日志创建和写入过程的时间越晚，完成事务的写入请求的速度就越快。 这促使我们探索将懒惰日志记录作为持久性HTM的基本加速技术。

## B.方法：懒惰日志记录

懒惰日志记录通过将日志创建与写入请求的到达分离开来，从而加速了持久HTM。 它直接利用LLC缓存的数据进行撤消日志记录。 在常用的inclusive缓存中，LLC缓存的数据是L1缓存的数据的超集。 当写请求tends to修改L1高速缓存的数据块时，我们不必将L1高速缓存的旧值写回到内存控制器即可撤消日志记录。 这是因为旧值也在LLC中。 即使我们立即在L1缓存中对其进行修改，我们也不会错过旧值。 简而言之，惰性日志记录可以在接收到写请求后执行缓存更新，并几乎同时准备撤消日志，而不会因创建日志而中断连续的写操作。 当必须从LLC撤出旧值时，会将旧值作为撤消日志写回到内存控制器。 这最大程度地减轻了eager日志记录的开销。 此外，延迟的日志创建还保证了更多的日志合并机会。

现在，我们将研究更多有关延迟日志记录如何加速存储指令并优化日志合并的信息。

**缩短store指令的执行时间。** 图4比较了传统热切日志记录和我们的惰性日志记录下存储指令实例的执行时间线。 设A表示旧值，A’表示要写入的新值。 在热切的日志记录下（图4（a）），存储指令与日志创建紧密结合。 日志创建会强制执行其他操作，例如将元日志信息（即，事务ID和A的地址）写入日志缓冲区，访问缓存以获取A的旧值，将A写入日志缓冲区以及将A与将元日志信息转换为日志条目。 只有完成所有这些额外的操作，我们才能使用新值A’修改缓存行。 相反，我们的延迟日志记录会在缓存内更新之前避免所有这些额外的操作。 存储指令立即将A从A修改为A’，就好像它是非事务性访问一样。 同时，我们通过缓存一致性目录（第IV-D节）跟踪元日志信息，无论如何应根据写入请求对其进行更新。 我们将元日志信息附加到目录中，而没有太多额外的开销。

**日志合并的固有支持。** 由于惰性日志记录利用LLC缓存的数据作为日志，因此直接受益于其中的固有位置。 在热切的日志记录中，先前的解决方案在写入请求到达后立即创建日志。 为了合并日志，它们通常会引入一个额外的buffer来临时保存日志，以便在将其写入内存控制器或内存之前可以合并为更大的块。 相反，我们的惰性日志记录使用LLC来保存日志，因此不需要额外的缓冲区。 此外，日志将保留在LLC中，直到必须清除它们所驻留的缓存行为止。 每条LLC高速缓存行本质上都包含地址相邻的数据。 这样可以将懒惰日志记录从大量额外开销中节省下来，以合并具有较高spatial locality空间局部性的事务的日志。

## C.体系结构FlyHTM：

**系统模型。** 我们使用惰性日志记录技术来开发更快的持久性HTM，称为FlyHTM。 与现有的HTM解决方案一样，FlyHTM的核心位于CPU芯片上。 图5说明了FlyHTM的体系结构。 在研究特定的日志记录框架之前，我们先定义系统模型。 我们考虑一种具有常用的三级缓存层次结构的多核系统。 每个内核都有一个专用的L1缓存和一个专用的L2缓存。 所有内核共享LastLevel缓存（LLC）。 缓存包含所有内容，并通过写分配进行写回。 为了保证缓存的一致性并检测事务冲突，我们使用基于MESI目录的协议[9]来跟踪私有缓存中的数据状态。

受Proteus [34]和ATOM [20]的启发，FlyHTM采用了电池支持的持久性存储控制器。好处是将日志直接持久存储在内存控制器中，而不必将其写入速度慢得多的NVM。 此设计选择背后的另一个动机是，实际上大多数日志并不需要保留到NVM。 HTM解决方案可保持日志持久，以便在系统崩溃时将数据恢复到一致状态。 但是，系统崩溃很少发生，事务可以更频繁地成功提交。 因此，将不必要的日志写入NVM会浪费大量的内存带宽，并导致高开销。 当我们无法确定系统是否可能在正在进行的事务提交之前崩溃时，就会出现一个难题。 换句话说，我们几乎无法决定哪些日志可以省略。 使用持久性内存控制器有助于缓解这种困境，因为将日志持久存储到内存控制器中比将日志持久存储到NVM中要快得多。 仅当日志溢出时，内存控制器才需要将其写入NVM。 此外，一旦提交事务，便可以丢弃其日志以节省空间。

**日志记录框架。** 图5说明了FlyHTM如何逐步执行惰性日志记录。 在较高级别上，核心执行事务，并且在事务执行期间修改的数据保留在专用缓存中。 同时，撤消日志包含修改后的数据的旧值； 它们将保留在LLC中，直到替换后相应的缓存行被驱逐或事务提交后刷新为止。 我们按照以下步骤完成图5中的日志记录步骤。

•步骤x：内核向L1控制器发出写请求。 FlyHTM并没有像急切的日志记录一样强制L1控制器创建日志（图3），而是直接将LLC中已缓存的旧值用作撤消日志。 它可以轻松地使用write set中维护maintained的地址来track 跟踪LLC中的哪些数据用作日志，而无需对LLC缓存行进行任何额外的操作。 由于不需要额外的日志创建，因此写请求可以立即更新L1缓存中的数据。

•步骤y：在两种情况下，FlyHTM将撤消日志从LLC刷新到内存控制器。 第一种是由于替换而将包含撤消日志的高速缓存行逐出的情况。 第二个是事务在提交数据之前先提交并刷新非持久日志的时间。 在这两种情况下，旧值都从LLC逐出到内存控制器，而内核将事务ID和地址写入内存控制器。 所有这些项目构成一个完整的撤消日志条目。

•步骤z：为保持inclusive缓存的inclusion属性，逐出LLC高速缓存行会触发包含相同数据块的L2和L1高速缓存行上的向后无效。 具体来说，我们首先使L2和L1缓存中的两个缓存行均无效。 然后，我们将新值从L1高速缓存写入内存控制器，这会将数据进一步保存到NVM中。 这确保了日志在数据之前是持久的，这是WAL的要求。

# IV。 设计与实施

## A.日志结构

FlyHTM使用撤消日志，该撤消日志最初是LLC中的缓存数据，并在收回后保留在内存控制器的集中式日志缓冲区中。 在第II-B节中讨论的撤消日志记录和重做日志记录之间，我们选择撤消日志记录是因为其日志管理的简单性和就地更新的快速性。 首先，它记录旧值，在事务的整个生命周期中只需要记录一次。 无论数据块被写入多少次，它们都源自相同的初始旧值。 相反，每次写入数据块时，重做日志记录都会生成日志。 这不仅使日志管理复杂化，而且扩大了日志量。 与重做日志记录相比，撤消日志记录因此可以将日志量最小化，使其与更新的数据量成正比，而不是与更新时间成正比。 其次，撤消日志记录允许在关联的撤消日志持久保存之后就地更新。

与重做日志记录一样，这样一个快速的原地更新可保存undo log从重定向读取请求到记录的新值。 此外，与重做日志记录不同，撤消日志记录不必保留日志，直到事务的所有日志可用为止。 这大大缓解了对日志缓冲区的需求压力。 按照现有工作[20]，[28]，集中式日志缓冲区位于内存控制器中。

传统上，撤消日志记录具有数据块的粒度，每个撤消日志条目都包含四个字段-事务ID（TxID），核心ID，地址和旧值。 事务ID指定日志条目所属的事务。 核心ID代表执行事务并帮助区分不同线程的核心。 地址和旧值分别跟踪修改后的数据块的地址及其原始值。 当事务中止时，我们首先使用关联的TxID和核心ID查找其日志条目。 然后，我们找到就地更新的数据块并将其恢复到旧值。

但是，FlyHTM以高速缓存行粒度维护可变长度的撤消日志。 给定所有块...

在同一缓存行中;

然后是通用地址

根据空间容量，日志区域可以存储也可以不存储未记录的块。 理想情况下，额外的16位向量，但除了偏移量以外，省略重复的地址前缀（标记和索引）。 支持更紧凑的...日志从日志条目中保存公共地址...; 地址/偏移量...; 地址-位向量-基于事务ID和核心ID缓存行的顺序，一个以上的旧值，搭载多个日志应指定... ==从cpu到内存控制器的地址，然后控制器将它们合并为一个更紧凑的地址。 ..; ==合成的...：位向量-日志为1，注销为0 txid和核心ID缓存旧值； 在第IV-E节中进行了优化...;

**日志字段参数化。** txid coreid地址-传统的旧值-一条缓存行，在IV-E节中有更多讨论...; 然后，我们需要确定每个日志输入字段的长度。

对于事务ID，如果它太长，则会浪费存储空间； 如果太短，可能会为新事务分配与先前未提交事务相同的ID。 我们认为16位是合适的大小[28]。 至于Core ID，字段大小取决于Core的数量。 如果系统具有16个核心，则字段大小为4位。 然后，地址字段占用48位。 旧值的大小是LLC缓存行的长度。 在这里，我们应用64个字节。

要管理日志，有两个主要任务：日志创建和日志持久性。 日志创建侧重于收集日志条目字段并组织它们以形成完整的日志条目，而日志持久性侧重于使日志持久化并控制日志和数据的顺序。 简而言之，我们将日志创建与LLC高速缓存行逐出结合在一起，并直接利用inclusive的include属性来确保日志在数据之前持久存在。 接下来，我们将详细讨论FlyHTM如何管理这两项任务。

== creation讨论cpu和内存控制器如何协作...这些日志结构...;

## B.日志创建

日志创建集中于内容的两个部分：日志条目字段的源以及创建日志条目的过程。 FlyHTM的日志输入字段的来源类似于以前的设计，但是在此过程中，它会将日志创建延迟到缓存行逐出之前。

日志输入字段field的来源。 在FlyHTM中，日志条目由四个字段组成：undo value，地址，coreID和TxID。 首先，通过利用现有的缓存信息获得撤消值。 传统上有两种获取撤消值的方法：直接更新从缓存中获取撤消值 或者通过访问NVM [11]获得value的间接更新。 鉴于间接更新需要额外的NVM读取，这会浪费大量时间和带宽，因此FlyHTM选择直接更新。 对于L1缓存中的缓存行的每次更新，我们肯定可以发现它的原始值位于LLC中，因为它使用了inclusive cache。 FlyHTM直接利用此LLC信息作为撤消值。 其次，在写回高速缓存行时携带地址信息，这构成了日志条目的地址部分。 至于内核ID，可以从CPU ID寄存器中获得。 最后，对于事务ID，核心将为每个事务分配它，并添加一个特殊的寄存器来记录此信息[36]。

**创建日志条目的过程。** FlyHTM将日志创建与缓存行逐出结合在一起。 逐出高速缓存行后，它将从上述源中收集日志条目(entry)信息，并将它们写入内存控制器中的日志缓冲区，以形成完整的撤消日志条目。 以下显示了详细过程。

•通过清空事务的写集从LLC撤出缓存行时，将触发日志的创建。

•逐出后，undo value及其地址将写回到MC中的日志缓冲区。

•缓存行驱逐还将触发事务ID和核心ID发送到MC。 因此，创建了完整的日志条目。

## C.日志持久性

创建日志条目后，任务是持久化。 当我们使用预写日志记录来保持崩溃一致性时，一个关键问题是日志应该在数据之前持久保存。 FlyHTM并非使用绕过缓存层次结构的不经过cache日志记录来节省时间[19]，[28]，[34]，而是借助inclusive缓存实现了日志和数据之间的固有排序控制。

**日志和数据之间的固有顺序。** FlyHTM直接利用包含式缓存的包含属性来强制执行日志和数据的写入顺序。 对于包容式缓存，如果L1缓存中也存在要撤出的LLC缓存行中的数据，则为了保持包含属性，需要返回无效使L2和L1缓存中的缓存行无效。 因此，在FlyHTM中，首先将被视为撤消值的LLC高速缓存行写入MC，然后在较高的高速缓存层次结构中刷新已修改的值。 以下说明了具体过程。

1）FlyHTM驱逐LLC中的高速缓存行，该行被视为对MC中的日志缓冲区的撤消值。

2）FlyHTM将无效信号发送回专用高速缓存。

3）专用缓存将相应的dirty缓存行（新值）写回到MC。

## D.缓存刷新

当事务完成执行后，为了使所有更新持久化，需要使用缓存刷新将刷新后的数据刷新回持久存储中。 根据以前的研究,根据更新时的标准在NVM中，刷新可分为同步刷新[20]，[34]，[35]和异步刷新[18]，[27]，[28]。 在同步刷新下，应在提交之前回写修改后的数据，而对于异步刷新，可以延迟NVM中的更新。 同步刷新和异步刷新的选择是性能和复杂性之间的折衷。 异步刷新摆脱了等待在关键路径中刷新所有已修改数据的时间，从而可以更好地利用带宽和性能。 但是，它在缓存修改，隔离级别和复杂机制方面做出了让步。 FlyHTM可以采用同步刷新和异步刷新两种版本。

实现异步刷新并不难。 首先，它需要允许事务提交后存在非持久性修改数据，并允许其他事务访问此类数据。 就像上面的技术一样，我们可以用事务ID标记每条缓存行。

===冲洗程序FlyHTM采用同步冲洗。

我们还可以轻松实现异步刷新。 ??

我们的刷新目标是修改的缓存行及其对应的日志。 因为使用了包含性缓存，所以我们只需要刷新LLC中的日志。 在我们的设计中，日志和数据共享同一地址，LLC中日志的逐出将直接导致L1中的修改数据被写回。 这就带来了另一个问题：如何在LLC中定位修改后的数据地址。 在L1中冲洗很容易。 我们可以根据其dirty位识别出修改后的缓存行。 但是在LLC中，要刷新日志，我们没有这些明显的信息。 出现此问题的主要原因是我们没有修改后的数据的地址信息。 因此，我们需要获取事务的写入集。 以前的研究使用布隆过滤器来解决这个问题[3]。 还有一些疑问。 我们建议直接利用目录缓存一致性协议。 LLC中的目录维护所有高速缓存行状态的信息。 根据缓存行的状态，我们可以推断出写入集信息。 当事务倾向于提交时，LLC控制器将开始刷新。 根据写集提供的地址信息，LLC将相应的日志刷新回MC。

L1和L2，从一个级别回写，只是将另一个级别中的设置无效位？

## E.日志区域和日志溢出

在事务中，日志负责故障恢复。 将日志写回到NVM会花费大量时间和带宽。 但是，偶尔会发生故障。 大多数情况下，事务将成功提交，并且不会使用日志。 牺牲性能以防止意外crash听起来并不是一件好事。 为了避免这种成本，FlyHTM将MC添加到了持久域中。 数据到达MC的那一刻，它将变得持久。 我们在MC中引入了一个日志缓冲区，作为Proteus [34]的一个单独的写等待队列。 避免将日志条目写入NVM。

我们希望尽可能延迟日志的回写过程。 日志只需要驻留在日志缓冲区中，而无需强制写回到NVM。 结构如图??所示。

添加circular array的介绍/图；

如果MC中的日志缓冲区是无限的，则可以解决该问题。 但是它受到大小的限制，这是硬件方法的常见缺陷. 当处理稍大的事务时，日志溢出是不可避免的。 因此，我们需要向NVM寻求帮助。 我们的原则是：如果日志缓冲区没有溢出，则日志将完全保留在MC的日志缓冲区中； 如果日志缓冲区有溢出风险，则必须将导致溢出的事务日志完全写入NVM。 超过80％的缓冲区占用是信号,表明缓冲区有溢出风险仅应是硬件。 溢出将触发异常，请求操作系统帮助。 根据事务ID，操作系统将在NVM中分配一个单独的空间来存储日志。 该空间仍被组织成圆形结构。 然后，通知日志缓冲区将具有溢出风险的事务日志写回NVM。

## F.效率分析

根据事务时间分析效率...；

在这里，我们进行时间分析。 我们可以将整个过程分为两个主要部分：执行和刷新。 为了方便起见，我们仅在执行时考虑存储指令。 假设一个事务存储了A，B，C，D四个变量？为什么基于图6进行此假设？ 重画并解释；

**执行时间**。 在传统的硬件撤消日志记录设计中，日志生成与存储指令结合在一起。 创建日志条目需要额外的时间。 为了完成存储，我们需要花一些时间判断是否应该生成日志，并访问L1缓存以检索旧值。

TimePrev Store = TimeStore + TimeJudge + TimeAccess在FlyHTM设计中，我们将日志创建与存储指令分离。 日志创建被延迟，直到要从缓存中逐出缓存行为止。 因此，在处理存储指令时不需要简单的步骤。 显然，我们可以得出以下结论：

TimePrevstore> TimeFlyHTMstore

**Flushing time**。传统方法只需要将dirty的缓存行刷新回NVM，因为日志与写修改结合在一起并且已经被持久化了。 但是FlyHTM需要刷新日志和数据。 我们认为这不会影响性能，因为写入日志和写入数据可以并行运行。 有人可能会怀疑冲洗L1比冲洗LLC快，因为LLC L1吗？ 具有较短的访问延迟。 我们认为并非如此。 因为LLC实际上会合并并更多地利用空间局部性。 尽管一次冲洗需要更多时间，但冲洗次数要少得多。 对于具有局域性的事务，FlyHTM的刷新时间将更短。

TimeTraditonal Flush ≈ TimeFlyHTM Flush

Numstore ×TimeFlyHTM Store +TimeFlyHTM Flush < Numstore ×TimeTraditional Store +TimeTraditonal

经过时间分析，我们的FlyHTM设计比传统的撤消日志记录方法运行得更快。

G.延迟提交

延迟提交直到几个事务...； 不一定刷新并提交事务...; 由于其修改后的数据可能会在提交后不久被后续事务使用和修改；

# V.实施

A.软件支持日志支持。

B.非易失性区域扩展内存控制器扩展。 持久性MC被引入FlyHTM。 在内存中，我们添加了一个日志缓冲区来存储日志条目。 日志缓冲区被组织成一个循环队列，如图?? 显示。 我们跟踪

C.挥发性区域扩展添加的寄存器。 为了支持事务，我们添加了一些特殊的寄存器。

D.硬件开销分析

# VI。 评估

**Implementation and configuration** 我们使用ChampSim [1]（基于跟踪的微体系结构模拟器）实现PhantomCache。 与纯硬件随机映射解决方案一样（例如，NewCache [22]，[37]和CEASER [31]），我们对启用了包含性的ChampSim的修改仅涉及LLC模块。 我们采用LRU作为替换政策。 table ？？ 显示了架构配置。

**工作量。** 我们通过运行SPEC CPU 2017基准测试包[2]中的工作负载来评估PhantomCache性能。 具体来说，我们使用SPECspeed 2017 Integer和SPECspeed 2017 Floating Point套件中的所有20个基准测试。 每个工作负载至少运行20亿条指令。 前十亿条指令用于预热缓存，而其他十亿条或更多指令用于收集性能统计信息。

**指标。** 我们使用三个性能指标评估PhantomCache，这三个性能指标是：LLC的每周期指令（IPC），每1000条指令的未命中率（MPKI）和LLC的失效率。 为了评估PhantomCache对缓存性能的影响，我们使用PhantomCache指标与基准缓存的比率（未经修改）将所有这些指标归一化。 较高的归一化IPC表示性能较好，如果PhantomCache优于基准，则性能会超过100％。另外, 较低的归一化MPKI或归一化未命中率表明性能更好。 为了衡量总体性能，我们进一步报告了归一化IPC的几何平均值，归一化MPKI和归一化未命中率的平均值。 结果。 根据表??中的配置，结果表明，为了保护LLC免受强大的O（| E |）攻击，如果制造的LLC能够满足其r集并行搜索要求，则PhantomCache平均只会降低0.3％的速度 （部分??-部分??）。 此外，为了确保8个存储区的16 MB 16路LLC，由于牺牲了某些并行性，PhantomCache平均只会使平均速度降低约0.5％（第??节）。

# 七。 结论

我们研究了针对基于冲突的缓存定时攻击利用局部随机化的思想。 它被证明具有与全局随机化对策相同的强大防御作用，并且避免了先前的全局随机化设计中效率低下的机制，例如随机替换和动态重新映射。 我们通过PhantomCache实现局部随机化。 对安全性的分析表明，当PhantomCache的程度设置为8时，攻击者无法在100年内成功发起基于冲突的缓存定时攻击。最后，我们使用ChampSim实现了PhantomCache，评估表明，PhantomCache仅带来了0.5％ 性能下降和负担得起的硬件开销。