Bent⁻o：优化持久内存程序

Sebastião Amaro、João Gonçalves、Miguel MatosIST、里斯本大学、
INESC-ID
{sebastiao.amaro,joao.tiago.goncalves,miguel.marques.matos}@tecnico.ulisboa.pt

arXiv:2603.01889v1 [cs.ET] 2 Mar 2026

摘要

持久内存（PM）是一种新型存储技术，具有高性能、字节寻址能力和持久性，且成本低于动态随机存取存储器（DRAM）。由于缓存易失性和存储操作重排序，开发人员必须使用显式指令（例如：刷新和内存栅栏），以确保应用程序状态在崩溃后仍保持一致。要正确做到这一点并不容易；事实上，已有多种工具被开发出来，用于检测持久内存程序中的错误。为克服这一难题，程序员往往采取过于保守的做法，例如强制施加不必要的顺序约束，从而部分抵消了使用持久内存所带来的性能优势。

本文研究了不同持久化指令组合对多个持久内存程序的影响，并发现某种特定组合在保留原有崩溃一致性语义的同时，能够提升性能。基于这些结果，我们开发了 Bent⁻o，这是一种自动化的黑盒二进制重写器，只需开发人员付出极少努力，就能将现有持久内存程序的性能提升多达 15%。

1 引言与相关工作

持久内存（PM）是一种新型存储技术，支持字节寻址，旨在以更低的每 GB 成本实现接近动态随机存取存储器的性能，同时确保程序崩溃和重启后数据仍然持久。持久内存已应用于许多程序中，例如持久内存专用索引、持久内存文件系统，以及 Microsoft SQL Server [23],memcached [13] 或 Redis [16]等数据库。

字节可寻址性与数据持久性相结合，为开发者设计程序带来了新的机遇。然而，由于现代 CPU 架构会缓冲并重新排序存储操作，且 CPU 缓存具有易失性，其缓存替换策略也难以预测，因此无法保证状态能够以崩溃一致的方式持久化[1]。因此，开发者必须显式使用刷新指令和内存栅栏，以确保状态按照适当的顺序持久化，从而在发生崩溃时，状态在重启后仍保持一致。此外，刷新指令和栅栏指令有多种变体，在性能与语义之间提供了不同的权衡。

[1]eADR [12] 可通过将持久性域扩展至 CPU 缓存来缓解这一问题，但：i）存储操作仍可能被重新排序（见 §2）；ii）其额外成本使其不太可能得到广泛应用，因此程序仍必须在没有 eADR 的情况下正常运行。

如日益增多的持久内存错误检测工具所示 [2, 4, 6–9, 17, 20, 21, 24, 25]，选择并使用能够最大限度提升性能且保证正确性的指令，是一项复杂且容易出错的任务。这种复杂性可能导致开发者采取过于保守的做法，例如使用比实际所需更多的刷新操作和内存栅栏，或采用语义更强的指令，或施加不必要的顺序限制，从而丧失使用持久内存原本能够带来的性能优势。

已有多项研究 [10, 14, 15, 19, 22, 26, 28–33]旨在分析持久内存的性能、理解其设计及其对性能的影响。Xiang 等人 [32] 对英特尔傲腾 DC 中的读写缓冲、读写放大、不同类型持久化屏障的开销，以及不同持久性模型的优缺点进行了深入研究。PerMA-BEnch [1] 是一种可配置的基准测试框架，用于评估可定制的数据库相关持久内存访问的带宽、延迟和每秒操作数。这项全面评估总结了八条影响持久内存性能的通用见解和实现相关见解。

简而言之，现有研究大多关注持久内存设备本身的读/写性能，而非不同持久内存指令在具体且具有代表性的工作负载中对程序性能和正确性的影响。因此，为了更好地理解这一问题，我们使用真实应用程序和代表性工作负载开展了一项研究，比较不同指令组合对程序性能和一致性的影响（§3）。我们发现，有一种通用的特定组合无需了解程序语义即可应用，最多可将性能提升15%，在最坏情况下也能保持原有性能。

受这些结果启发，我们开发了Bent⁻o，这是一种以自动且黑盒方式重写程序二进制文件的工具，它使用能带来最佳性能的指令，同时不改变现有的崩溃一致性保证（§4）。

最后，我们将讨论可能的未来研究方向，即寻找程序的最小崩溃一致性状态：也就是说，找出程序达到崩溃一致性所需的最少刷新和内存栅栏指令数量，从而在正确性与性能之间实现尽可能好的权衡（§5）。

2 背景

本节将介绍持久内存的持久化语义。为简便起见，我们聚焦于英特尔 x86 [27]，但 ARM 等其他架构也具有类似的持久化语义，并提供保证各异的指令。英特尔 x86 遵循宽松缓冲持久性模型。宽松这一特性意味着，写操作持久化的顺序不一定与其对其他线程可见的顺序相同。缓冲这一特性则与持久写入发生的时机有关：在采用缓冲的架构中，持久写入可以异步进行，写入也可以排队等待，同时程序继续执行。发生存储操作时，它首先进入存储缓冲区排队，最终到达 CPU 缓存。只有到达持久内存控制器中的写入待处理队列（WPM）后，该操作才会持久化。存储操作到达写入待处理队列的时机无法预测，因为缓存会在需要加载新数据时驱逐缓存行。显然，若仅依赖这种行为，发生崩溃时就会导致状态不一致。因此，开发者必须使用特殊指令来控制存储操作到达持久内存的顺序。这些指令可分为两类：刷新指令和栅栏指令。对于刷新指令，英特尔 x86 提供了 clflush、clflushopt 和 clwb。clflush 会驱逐并持久化单个缓存行，且不能与其他存储和刷新指令重新排序。clflushopt 会驱逐并持久化单个缓存行，但在出现栅栏指令之前可以重新排序。clwb 会持久化单个缓存行，但不会将其从缓存中驱逐，并且也可以重新排序。栅栏指令有两种。mfence 保证栅栏之前所有已缓冲的加载、存储和刷新操作都先于栅栏之后的操作执行，而 sfence 仅对存储和刷新操作提供这一保证。从某种意义上说，我们可以将使用这些指令与使用并发原语相提并论。一方面，完全不使用这些指令，或将其放在错误的位置，都会导致程序不正确。另一方面，使用超出严格必要数量的此类指令，或使用未经优化的对应指令，都会损害性能。

3 性能研究

在本节中，我们研究不同刷新与栅栏指令组合的开销，并讨论它们对崩溃一致性保证的影响。所有实验均在 128 核英特尔® 至强® Gold 6338N 处理器 @ 2.20GHz 上进行，配备 256 GB 随机存取存储器和 1 TB 英特尔傲腾持久内存

在 App Direct 模式下运行 Ubuntu 22.04 和内核 5.15。此外，对于所有实验结果，我们呈现五次独立执行的平均值和标准差。

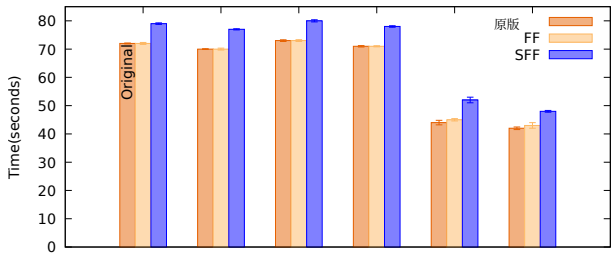

图1：在 100% 插入的 Level Hashing 工作负载中，不同刷新与栅栏指令组合的性能影响。

3.1 初步研究

我们首先研究 Level Hashing [34] 中每种刷新/栅栏组合对性能的影响。Level Hashing 是一种针对持久内存进行写优化的高性能哈希索引。在实现方面，Level Hashing 使用三条clflush指令，其中两条后面跟随 mfence。我们的目标是回答以下两个问题：i）采用简单、保守的方法实现崩溃一致性会带来怎样的性能影响？以及 ii）对于类似的崩溃一致性保证，哪种刷新指令和栅栏指令组合能够提供最佳性能？我们运行了包含 1 亿次插入（100% 写入）的工作负载，并测量完成该工作负载所需的总时间。结果如图1所示。我们考虑了 LevelHashing 的三种变体：i）原版，对应原始实现；ii）*FF*，一种每次刷新后都跟随栅栏的变体；以及 iii）*SFF*，一种每次存储操作后都跟随刷新和栅栏的变体。原版 变体代表开发者谨慎使用持久内存指令的场景，而 *FF* 和 *SFF* 则代表两种逐步趋于保守的场景，在这些场景中，开发者通过遵循简单的思路来简化问题。随后，我们通过实现刷新指令和栅栏指令的不同组合，手动调整了每种变体，总共得到 6 种组合（3 种刷新指令乘以 2 种栅栏指令）。

要回答第一个问题，我们可以得出结论：在每次刷新后添加 mfence 或 sfence（*FF* 变体）并不会显著影响性能。这是因为原始的 Level Hashing 已经在三次 clflush 操作中的两次之后添加了内存栅栏，这意味着我们只额外添加了一次内存栅栏。然而，在每次存储操作后进行刷新（*SFF* 变体）会显著影响

性能，这是意料之中的，因为它消除了对存储操作进行批处理的可能性。

为回答第二个问题，我们比较了不同持久化指令组合的结果。可以看到，在任何变体中，将 clwb 与任意一种栅栏指令搭配使用，效果都远胜于对应的其他方案。clwb、clflush 和 clflushopt 的主要区别在于，前者会保留已写入内存的缓存行。这可能有助于后续操作，因为如果某个值已在缓存中，我们就可以避免应用程序访问持久内存。乍看之下，鉴于工作负载中的写操作占比为 100%，与使用 clwb 保留缓存行相比，使用 clflush 或 clflushopt 使缓存行失效似乎影响不大，甚至没有影响。然而，LevelHashing 在调整大小等操作中的元数据管理需要读取现有值，而缓存局部性有助于这些操作。基于这一发现，并考虑到与从缓存读取相比，从持久内存读取会造成显著开销，我们决定进一步研究这一观点：在持久内存应用中，使用 clwb 而非 clflush 和 clflushopt 能否获得更好的性能。

3.2 刷新性能影响

在本节中，我们将初步研究中获得的见解加以应用，并将其扩展到其他持久内存数据存储应用程序。除 Level Hashing 外，我们还分析了 FAST&FAIR [11]和 WOART [18],，它们分别是面向持久内存优化的 B+树和基数树实现。这三个应用程序都使用 clflush 来持久化存储，但正如我们在 §3.1 中所指出的，clwb 的性能要好得多。然而，其较弱的语义使得直接替换无法保持正确性保证。有鉴于此，我们研究了将原来的 clflush 替换为后跟 sfence 的 clwb 所带来的影响，这种替换能够有效保持原程序的崩溃一致性保证。我们针对每个应用程序手动进行了这一更改；在原代码中已经存在 clflush 后跟内存栅栏的情况下，我们仅将 clflush 替换为 clwb，以避免出现冗余的内存栅栏操作。

上一节的工作负载只有插入操作，仅作为初步研究。本节中，我们采用更贴近实际的工作负载，使用广为人知的 YCSB基准测试 [3]。我们使用以下工作负载：工作负载A，由 50% 的读取和 50% 的更新组成；工作负载B，由 95% 的读取和 5% 的更新组成；工作负载D，由 95% 的读取和 5% 的插入组成，但读取访问的是最近插入的键。所有工作负载均遵循*zipfian* 分布，执行 2500 万次操作，并在此之前进行加载阶段，将数据存储预先填入 2500 万个键。我们忽略了工作负载C，因为它完全由读取组成，因此不会执行刷新或内存栅栏；也忽略了工作负载E，因为数据存储不支持范围操作。对于工作负载F，我们获得了与工作负载A相似的结果（未展示）。结果见图2，其中标记为“原版”的柱状条表示未经修改的原始应用程序，而标记为“手动修改版本”的柱状条表示我们上述所讨论的修改。接下来，我们将讨论各个应用程序的结果。

3.2.1 Level Hashing. 对于工作负载A，将 clflush 替换为 clwb 加 sfence 使性能提升了 15%。Level Hashing 中的每次更新和插入操作都会有三次缓存行刷新，这些刷新在相应的存储执行之后按顺序进行。它分别刷新键、值和令牌（令牌用于指示桶是否为空）。这三个值使用 uint64_t（8 字节）实现，并且在内存中连续存放，因此它们可以放入单个缓存行（在64位系统中总大小为 64 字节）。clflush 会使它所作用的缓存行失效，从而导致驱逐。这意味着在刷新键之后，该缓存行（值和令牌也位于其中）会被驱逐。随后，要刷新值，必须先将其从持久内存重新加载回来，从而产生额外开销，令牌也是如此。将 clflush 改为 clwb 后，缓存行不再被驱逐，因此我们不会承担从持久内存进行两次额外读取的开销。我们认为这是性能提升的主要原因。鉴于一次刷新作用于整个缓存行，并且考虑到三元组（键、值、令牌）可以放入单个缓存行，实际上一次刷新就足以刷新整个三元组。为了确认这一点，我们对应用程序进行了插桩，以跟踪所有执行的刷新操作，并确实确认每次更新都执行了两次冗余刷新。由于这会影响性能，我们决定同时考虑一个优化版本，优化版层级哈希，在该版本中我们手动修复了这一问题。正如预期的那样，将 clflush 替换为后跟 sfence 的 clwb 所带来的潜在改进消失了，因为前面描述的行为不再发生，而且读取与写入之间没有相关性，因此我们无法从将值留在缓存中获益。

对于工作负载B，我们观察到性能提升了 5%。这是因为读取不会导致写入持久内存，因此也不会触发刷新操作。我们仍然实现了性能提升，但鉴于该工作负载的特性，提升幅度较小。出于与工作负载A相同的原因，Level Hashing 优化版本无法从手动修改中获益。

最后，对于工作负载D，我们观察到性能提升了8%。该工作负载主要由读取操作（95%）组成，因此数据存储不会频繁出现前文所述的行为

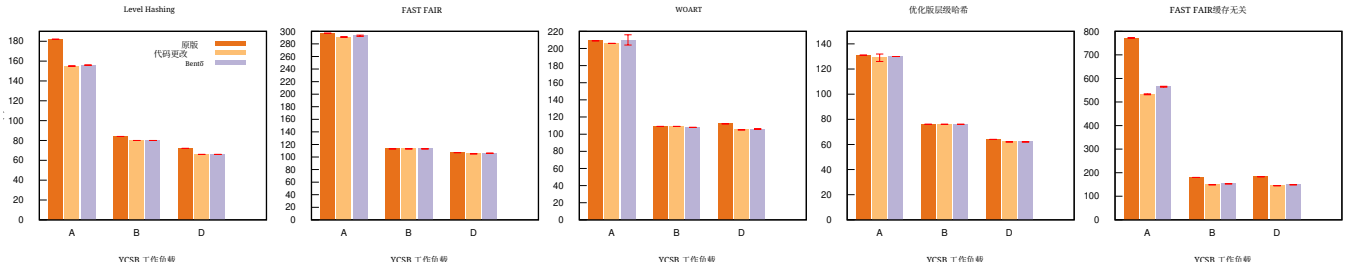

图2：不同刷新与内存栅栏组合对 YCSB 工作负载 A、B 和 D 的性能影响。原版柱状条表示未经修改的原始应用程序（使用 clflush），manual 表示我们在 §3.2 中介绍的手动修改版本（使用 clwb，之后执行 sfence），Bent-o 则表示 §4 中提出的自动重写。

。不过，由于这些工作负载中读取与插入存在相关性，即读取会访问最近插入的键，因此缓存行失效的影响仍然不可忽视。使用 clwb 持久化键的值后，缓存会保留该值，因此无需从持久内存中获取值，就能响应对同一键的读操作。这既适用于常规的 Level Hashing，也适用于我们的优化版本，不过由于前文讨论的原因，优化版本的提升幅度较小。

3.2.2 FAST&FAIR。故障原子移位（Failure-Atomic ShifT）和故障原子原位再平衡（Failure-Atomic In-place Rebalance）是一种针对持久内存优化的 B+-tree 实现。请注意，FAST&FAIR 不支持更新操作，因为它允许存在重复键。因此，我们将更新操作视为插入操作。对于工作负载 A，我们观察到性能略有提升。具体而言，B+-tree 通过移动相邻项来插入值。平均而言，每插入一个新值，就会移动一半的相邻项。由于每个值占 8 字节，每个指针也占 8 字节，共计 16 字节，因此一个缓存行中可容纳 4 个值。基于这一点，在移动过程中，FAST&FAIR 仅在跨越到下一个缓存行的边界时执行 clflush。我们指出，这项优化与 FAST&FAIR 的设计直接相关，无法轻易推导出来，也不适用于其他场景。为了模拟一位更保守，和/或对持久内存缓存行为不太熟悉的开发者，我们专门开发了一个保守版本，在键插入阶段每次存储后都执行刷新操作，称为 *FAST&FAIR缓存无关*。与 Level Hashing 类似，这会导致冗余刷新。正如前文所述，由于 clflush 会逐出缓存行，因此还会导致从持久内存额外读取三次数据。使用 clwb 时不会发生这些额外的持久内存读取，从而使性能提升 30%。

工作负载B在原始 FAST&FAIR 中表现出的性能差异可以忽略不计，我们认为原因与 Level Hashing 相同。对于不感知缓存的 FAST&FAIR，我们观察到性能提升了 17%，因为尽管写入所占比例很小（5%），这些插入操作的开销却很高。

最后，对于工作负载D，我们观察到 FAST&FAIR 的性能略有提升。这是因为 FAST&FAIR 中的 clflush指令很少，因此提升空间较小。对于不感知缓存的 FAST&FAIR，由于刷新操作的数量增加了三倍，我们观察到性能提升了 20%。这也可以用读取和插入操作的局部性来解释，因为较少的缓存驱逐进一步带来了好处。

3.2.3 WOART。写优化自适应基数树是一种持久内存优化基数树。与 FAST&FAIR 一样，它没有更新操作，因此我们改用插入操作。基数树的一个关键特性是，查找特定键时无需进行键比较。因此，与 B树相比，它造成的缓存污染更少，缓存未命中也更少。此外，与 FAST&FAIR 不同，WOART 不具备缓存行感知能力，这意味着它可能会执行冗余的刷新操作。关于实验结果，在工作负载A下，我们没有看到显著改善，因为读取与插入无关，因此避免缓存被逐出并无优势。

对于工作负载B，出于与 Level Hashing 和 FAST&FAIR 相同的原因，我们也没有观察到显著变化。

然而，在工作负载D中，我们观察到性能提升了6%，这归因于我们会读取最近插入的值，因此能够利用缓存。有人可能会问，为什么 WOART 会有这样的表现，而 FAST&FAIR 却不会。我们认为有两个原因。首先，由于 FAST&FAIR 具备缓存感知能力，它执行的缓存行刷新较少，因此将 clflush 指令替换为 clwb 后接 sfence 所带来的提升空间也较小。其次，由于 WOART 的缓存污染较少，clwb 会让值在缓存中保留更长时间。这一点尤为重要，因为我们知道，在此工作负载中，近期插入的值很快就会被读操作访问。

上一节的研究结果表明，在某些工作负载中，应用程序的性能最高可提升15%，且不会在对抗性工作负载中造成性能损失。实现这一点的方法是将 clflush 指令替换为 clwb 后接 sfence。刷新时不驱逐缓存，使应用程序能够利用缓存局部性，同时保留原有的崩溃一致性保证。受此启发，我们构建了工具 Bent¯o，以黑盒方式自动执行这种替换。我们的目标是让开发者能够使用 Bent¯o，而无需深入了解每条持久内存指令的底层工作细节，也无需修改源代码。这样一来，开发者便能采取更稳妥的编程方式，并有望构建出崩溃一致性保证更强、同时仍能保持良好性能并利用底层硬件特性的持久内存程序。

我们分析了多种实现这一目标的方案，最终选择了二进制重写，而非编译器遍等方案，因为这种方式是黑盒（不依赖源代码）且便于使用，也就是说，它不会影响目标应用程序的常规构建流程。Bent¯o 依赖 e9patch [5], ，这是一种能够静态重写 x86_64二进制文件而无需了解控制流信息的工具。它结合运用指令双关、填充和驱逐，在无需移动其他指令的情况下插入跳转至跳板的指令，并在 99% 的情况下避免执行开销高昂的跳转目标重计算。简而言之，Bent¯o 接收一个二进制文件作为输入，并输出一个修补后的二进制文件，其中所有 clflush 指令都被替换为 clwb，后接一条 sfence。由于原始二进制文件可能已经包含内存栅栏指令，因此在每条 clwb 后一律添加 sfence 可能会造成不必要的开销。有鉴于此，Bent¯o 的修补规则还会检查 clflush 后面的指令是否为 sfence 或 mfence。如果是，我们只将 clflush 替换为 clwb，而不添加额外的 sfence，以避免增加开销。需要注意的是，这种方法并未涵盖所有情况，因为刷新指令和内存栅栏指令之间可能隔着任意数量的其他指令，而我们的二进制重写工具的匹配规则无法识别这些情况。未来可以通过执行静态二进制分析来解决这一问题。

Bent¯o 的结果如图2所示。可以看出，尽管存在上述限制，Bent¯o 仍取得了几乎与上一节讨论的手动方法相同的结果。因此，我们认为，开发者只需付出很少的努力，就能使用 Bent¯o 提升其持久内存应用的性能，同时保留原有的崩溃一致性保证。

5 讨论

持久内存兼具接近动态随机存取存储器的访问延迟和类似磁盘的持久性，且成本更低。不过，要充分利用硬件特性，开发者必须谨慎使用内存栅栏和刷新指令。在不恰当的位置使用这些指令，或完全不使用它们，都会导致程序出错；而过于保守、使用次数超过必要限度，或使用未经优化的替代指令，则会损害性能。

我们的研究表明，通过选择恰当的刷新操作和内存栅栏组合，更具体地说，采用 clwb 后接 sfence，可以在某些应用程序和工作负载中将性能提升最多 15%。有趣的是，即使在不利用缓存的对抗性工作负载中，这种做法也不会降低性能，同时还能保持原有的崩溃一致性保证。需要注意的是，我们假设原始应用程序是正确的，也就是说，它能够确保崩溃一致性。如果情况并非如此，则意味着应用程序缺少更多刷新操作和栅栏指令，而这也为进一步提升性能创造了机会。

受研究实验结果的鼓舞，我们开发了 Bent¯o，这是一款能够自动重写应用程序二进制文件的工具，其性能提升几乎与我们手动重写源代码所取得的效果相同。Bent¯o 可自动运行，且是黑盒工具——它通过重写应用程序二进制文件来工作，无需访问源代码。这使开发人员能够轻松使用 Bent¯o 提升其持久内存应用的性能；据我们所知，此举没有任何缺点。

我们现在讨论这项工作所开启的未来研究方向。Bent¯o 通过利用一个简单的想法带来了良好的性能提升。自然地，那些比必要程度执行更多刷新和内存栅栏的更保守方法，能从使用 Bent¯o 中获益更多，而已经优化的方法则留下较少的改进空间。目前，我们正在探索在不牺牲正确性的前提下进一步提升持久内存应用性能的机制。为此，我们计划研究并发展最小崩溃一致性状态（MCCS）这一概念，其定义为仍能保证崩溃一致性所需的最小持久性约束集合。要确定 MCCS，我们需要一个能够判定目标应用程序是否具有崩溃一致性的判定器。作为第一步，我们可以依赖现有的持久内存错误检测工具 [9, 17, 20, 21, 24]作为该判定器的一个不完美实例。给定这样的判定器，并基于对现有内存栅栏和刷新指令所提供的性能与语义权衡的了解，我们可以遵循一个智能迭代过程，对二进制文件进行连续变换，直至达到 MCCS。

致谢

本工作得到科学与技术基金会（FCT）的支持，资助包括多年期资助 UIDB/50021/2020、博士奖学金 2021.07401.BD，以及研究项目资助 PTDC/CCI-COM/4485/2021（Ainur）。

参考文献

[1] Lawrence Benson、Leon Papke 和 Tilmann Rabl。2022年。PerMA-Bench：持久内存访问基准测试。《*VLDB* 会议录》15，11（2022 2463 2476 10 14778 3551793 3551807 年7月），–。[2] https://doi.org/。/。Zhangyu Chen、Yu Hua、Yongle Zhang 和 Luochangqi Ding。2022年。高效检测持久内存程序中的并发错误。载于第*27*届 *ACM* 编程语言与操作系统架构支持国际会议论文集（*ASPLOS'22* 873 887 10 1145 3503222 3507755）。[3] ACM，–。https://doi.org/。/。Brian F. Cooper、Adam Silberstein、Erwin Tam、Raghu Ramakrishnan 和 Russell Sears。2010年。使用 YCSB 对云服务系统进行基准测试。载于第*1*届 *ACM* 云计算研讨会论文集（美国印第安纳州印第安纳波利斯）（*SoCC '10*）。ACM，美国纽约州纽约，143 154 10 1145 1807128 1807152 –。https://doi.org/。/。[4] Bang Di、Jiawen Liu、Hao Chen 和 Dong Li。2021。。，面向持久内存程序的快速、灵活且全面的错误检测。载于第*26*届 *ACM* 编程语言与操作系统架构支持国际会议论文集（*ASPLOS'21*）。503 516 10 1145 3445814 3446744 –。https://doi.org/。/。[5] Gregory J. Duck、Xiang Gao 和 Abhik Roychoudhury。2020年。无需控制流恢复的二进制重写。载于第*41*届 *ACM SIGPLAN* 编程语言设计与实现会议论文集（英国伦敦）（*PLDI 2020*）。美国计算机协会，美国纽约州纽约，151–163。https://doi.org/10.1145/3385412.3385972[6] Xinwei Fu、Wook-Hee Kim、Ajay Paddayuru Shreepathi、Mohannad Ismail、Sunny Wadkar、Dongyoon Lee 和 Changwoo Min。2021年。Witcher：针对非易失性内存键值存储的系统化崩溃一致性测试。载于*ACM SIGOPS* 第*28*届操作系统原理研讨会论文集（*SOSP'21*）。ACM，100–115。https://doi.org/ 10 1145 3477132 3483556 。/。Xinwei Fu、Dongyoon Lee 和 Changwoo Min。2022年。DURINN：通过对抗性内存与线程交错检测持久线性化错误。载于第*16*届 *USENIX* 操作系统设计与实现研讨会（*OSDI'22*）。USENIX 协会，195–211。https://www.usenix.org/conference/osdi22/presentation/fu [8] João Gonçalves、Miguel Matos 和 Rodrigo Rodrigues。2023年。Mumak：针对持久内存的高效黑盒错误检测。载于第*18*届欧洲计算机系统会议论文集（*EuroSys '23*）。美国计算机协会，美国纽约州纽约，734 750 10 1145 3552326 3587447 –。https://doi.org/。/。[9] Hamed Gorjiara、Guoqing Harry Xu 和 Brian Demsky。2021年。Jaaru：高效地对持久内存程序进行模型检测。载于第*26*届 *ACM* 编程语言与操作系统架构支持国际会议论文集（虚拟，美国）（*ASPLOS '21*）。美国计算机协会，美国纽约州纽约，415 428 10 1145 3445814 3446735 –。https://doi.org/。/。[10] Shashank Gugnani、Arjun Kashyap 和 Xiaoyi Lu。2020年。理解真实持久内存的特性。《*VLDB* 会议录》14, 4 2020 626 639 10 14778 3436905 3436921 （12月），–。[11] https://doi.org/。/。Deukyeon Hwang、Wook-Hee Kim、Youjip Won 和 Beomseok Nam。2018 年。字节寻址持久性 B+- 树中的可持续瞬态不一致。载于第*16*届 *USENIX* 文件与存储技术会议（*FAST 18*）。USENIX 协会，奥克兰，加利福尼亚州，187–200。https://www.usenix.org/conference/fast18/presentation/hwang [12] 英特尔。2021年。eADR：持久内存应用的新机遇。https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html 访问日期：2023年3月28日。[13] Danga Interactive。2003年。Memcached。https://github.com/lenovo/ memcached-pmem 访问日期：2023年3月9日。[14] Joseph Izraelevitz、Jian Yang、Lu Zhang、Juno Kim、Xiao Liu、Amir Saman Memaripour、Yun Joon Soh、Zixuan Wang、Yi Xu、Sub- ramanya R. Dulloor、Jishen Zhao 和 Steven Swanson。2019年。英特尔傲腾 DC 持久内存模块的基本性能测量。*CoRR* abs/1903.05714（2019年），18、25。arXiv:1903.05714 http://arxiv.org/abs/ 1903 05714 。Wook-Hee Kim、R. Madhava Krishnan、Xinwei Fu、Sanidhya Kashyap 和 Changwoo Min。2021年。PACTree：采用 PAC 指南的高性能持久化范围索引。载于美国计算机协会 *SIGOPS* 第*28*届操作系统原理研讨会论文集（德国，虚拟会议）*(SOSP '21)*。美国计算机协会，纽约，纽约州，美国，424 439 10 1145 3477132 3483589 – 。2009 3 [16] https://doi.org/ 。/ 。Redis Labs。 。Redis。https://github.com/pmem/redis/tree/ 。2-nvml 访问日期：2023年3月9日。[17] Philip Lantz、Subramanya Dulloor、Sanjay Kumar、Rajesh Sankaran 和 Jeff Jackson。2014年。Yat：持久内存软件验证框架。载于*2014*年 *USENIX* 年度技术会议论文集（宾夕法尼亚州费城）*(USENIX ATC' 14)*。USENIX 协会，美国，433–438。[18] Se Kwon Lee、K. Hyun Lim、Hyunsub Song、Beomseok Nam 和 Sam H. Noh。2017年。WORT：面向持久内存存储系统的写入最优基数树。载于第*15*届 *USENIX* 文件与存储技术会议（*FAST 17*）。USENIX 协会，加利福尼亚州圣克拉拉，257–270。https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon[19] Lucas Lersch、Xiangpeng Hao、Ismail Oukid、Tianzheng Wang 和 Thomas Willhalm。2019年。持久内存范围索引评估。《*VLDB* 会议录》13，4（2019年12月），574–587。https://doi.org/10.14778/ 3372716.3372728[20] Sihang Liu、Korakit Seemakhupt、Yizhou Wei、Thomas Wenisch、Aasheesh Kolli 和 Samira Khan。2020年。持久内存程序中的跨故障错误检测。载于第二十五届编程语言与操作系统架构支持国际会议论文集（瑞士洛桑）*(ASPLOS '20)*。美国计算机协会，纽约，纽约州，美国，1187–1202。https://doi.org/ 10 1145 3373376 3378452 。/ 。Sihang Liu、Yizhou Wei、Jishen Zhao、Aasheesh Kolli 和 Samira Khan。2019年。PMTest：用于持久内存程序的快速、灵活测试框架。载于第二十四届编程语言与操作系统架构支持国际会议论文集（美国罗德岛州普罗维登斯）*(ASPLOS '19)*。美国计算机协会，纽约，纽约州，美国，411–425。https://doi.org/10.1145/3297858.3304015[22] Tony Mason、Thaleia Dimitra Doudali、Margo Seltzer 和 Ada Gavrilovska。2020年。英特尔®傲腾TM DC 持久内存的意外性能表现。《*IEEE* 计算机体系结构快报》19，1（2020年），55 58 10 1109 2020 2987303 – 。微软。 。为 SQL Server 2023 配置持久内存（PMEM）。https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/configure-persistent-memory?view=sql-server-ver16 访问日期：2023年3月28日。[24] Ian Neal、Ben Reeves、Ben Stoler、Andrew Quinn、Youngjin Kwon、Simon Peter 和 Baris Kasikci。2020年。AGAMOTTO：你的持久内存应用程序有多持久？ 载于第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*）。USENIX

协会，虚拟会议，1047–1064。https://www.usenix.org/conference/osdi20/presentation/neal [25] João Oliveira、João Gonçalves 和 Miguel Matos。2025年。HawkSet：自动化、与应用程序无关且高效的并发持久内存错误检测。载于第二十届欧洲计算机系统会议论文集（荷兰鹿特丹）*(EuroSys '25)*。美国计算机协会，纽约，纽约州，美国，1092–1108。https: //doi.org/ 10 1145 3689031 3717477 。/ 。Ivy B. Peng、Maya B. Gokhale [26] 和 Eric W. Green。2019年。英特尔傲腾字节寻址非易失性存储器的系统评估。载于国际存储系统研讨会论文集（美国哥伦比亚特区华盛顿）*(MEMSYS '19)*。美国计算机协会，纽约，纽约州，美国，304 315 10 1145 3357526 3357568 – 。2019 [27] https://doi.org/ 。/ 。Azalea Raad、John Wickerson、Gil Neiger 和 Viktor Vafeiadis。。英特尔 x86 架构的持久性语义。《*ACM* 编程语言会议录》4，POPL，文章 11（2019年12月），31 页。https: //doi.org/10.1145/3371079[28] Alexander van Renen、Lukas Vogel、Viktor Leis、Thomas Neumann 和 Alfons Kemper。2019年。持久内存 I/O 原语。载于第*15*届新型硬件数据管理国际研讨会论文集（荷兰阿姆斯特丹）*(DaMoN'19)*。美国计算机协会，纽约，纽约州，美国，文章 12，7 页。https://doi.org/10.1145/3329785.3329930[29] Zixuan Wang、Xiao Liu、Jian Yang、Theodore Michailidis、Steven Swanson 和 Jishen Zhao。2021年。非易失性内存系统的特征分析与建模。*IEEE Micro* 41，3（2021年），63–70。https: //doi.org/10.1109/MM.2021.3065305[30] Michèle Weiland、Holger Brunst、Tiago Quintino、Nick Johnson、Olivier Iffrig、Simon Smart、Christian Herold、Antonino Bonanni、Adrian Jack- son 和 Mark Parsons。2019年。对英特尔傲腾 DC 的早期评估持久内存模块及其对高性能科学应用程序的影响。载于高性能计算、网络、存储与分析国际会议论文集（丹佛，科罗拉多州）（*SC '19*）。美国计算机协会，纽约，纽约州，美国，文章 76, 19 10 1145 3295500 3356159 页。[31] https://doi.org/ 。/ 。Yinjun Wu、Kwanghyun Park、Rathijit Sen、Brian Kroth 和 Jaeyoung Do。2020年。从英特尔傲腾 DC 持久内存在数据库管理系统中的早期性能评估中吸取的经验教训。载于第*16*届新型硬件数据管理国际研讨会论文集（波特兰，俄勒冈州）（*DaMoN '20*）。美国计算机协会，纽约，纽约州，美国，文章 14，3 页。https: //doi.org/ 10 1145 3399666 3399898 。/ 。Lingfeng Xiang、Xingsheng Zhao、Jia Rao、Song Jiang 和 Hong Jiang。2022年。英特尔傲腾持久内存性能特征：深入考察其 DIMM 上缓冲。载于第十七届欧洲计算机系统会议论文集（雷恩，法国）（*EuroSys '22*）。美国计算机协会，纽约，纽约州，美国，488 505 10 1145 3492321 3519556 – 。2021 [33] https://doi.org/ 。/ 。Jialiang Zhang、Nicholas Beckwith 和 Jing Jane Li。。GORDON：在 FPGA 上对傲腾 DC 持久内存模块进行基准测试。载于*2021*年 *IEEE* 第*29*届年度现场可编程定制计算机国际研讨会（*FCCM*）。IEEE，奥兰多，佛罗里达州，美国，97–105。https://doi.org/ 10 1109 51124 2021 00019 。 /FCCM 。。Pengfei Zuo、Yu Hua 和 Jie Wu。。面向写入优化的持久内存高性能哈希索引方案。载于第*13*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 18*）。USENIX 协会，卡尔斯巴德，加利福尼亚州，461–476。https://www.usenix.org/conference/osdi18/presentation/zuo 2018 [34]