arXiv:2304.09411v1 [cs.AR] 19 Apr 2023

平等主义 ORAM：面向 ORAM 的磨损均衡

Yi Zheng[1], Aasheesh Kolli [1,2]，以及 Shaizeen Aga [3]*[1]宾夕法尼亚州立大学 [2]*
Google,Inc. [3]Advanced MicroDevices,Inc.[1]{ yfz5026, akolli } @psu.edu [3]
shaizeen.aga@amd.com

摘要

尽管非易失性存储器（*NVM*）具备多种理想特性，例如比动态随机存取存储器（*DRAM*）更高的存储密度和相当的能效、类似 *DRAM* 的性能以及类似磁盘的耐久性，但其有限的耐久性仍是这类存储器面临的一项挑战。事实上，非易失性存储器的耐久性限制可能导致原本常用于 *DRAM* 等主流存储器的解决方案无法原样移植到非易失性存储器上。具体而言，我们在本研究中发现，作为应对内存总线侧信道漏洞的最先进方案，不经意随机存取存储器（*ORAM*）原语虽已在 *DRAM* 上得到广泛研究，但若原样应用于非易失性存储器，实现起来尤其困难，因为它会严重影响非易失性存储器的耐久性。这是因为 *ORAM* 原语的固有特性会使写入流量增加一个数量级，而且会导致某些内存区域的写入频率远高于其他区域。*ORAM* 原语造成的这种非均匀写入流量会严重缩短非易失性存储器的使用寿命（降至基准水平的 *1%*，即不使用 *ORAM* 时的水平），甚至使得应对此安全漏洞变得不切实际。

为应对这一挑战，我们提出了平等主义 ***ORAM***（***E-ORAM***），探讨一种面向非易失性存储器的 *ORAM* 设计，同时使非易失性存储器的使用寿命接近理想水平。为此，我们发现，可以利用 *ORAM* 原语的内在特性来设计一种磨损均衡算法，使 *ORAM* 产生的额外写入流量更均匀地分布到整个非易失性存储器空间。我们证明，与现有的最先进的磨损均衡算法相比，后者在 *ORAM* 存在时只能达到 *2.8 %* 的使用寿命，而所提出的 *E-ORAM* 设计可达到约 *91%* 的使用寿命，使其在系统故障前能够多执行数个数量级的读写操作，同时引入的性能开销低于 *0.02%*。

1. 引言

存储系统的创新持续受到对更大内存容量和更高带宽需求的推动，这些需求始终存在且不断增长。一项此类创新是非易失性存储器（包括相变存储器 [22], 自旋转移力矩随机存取存储器 [20], 和电阻式随机存取存储器 [39]）。这类存储器具有多项理想特性，例如比动态随机存取存储器更高的存储密度和相当的能效、媲美动态随机存取存储器的性能，以及类似磁盘的耐久性。因此，非易失性存储器有望在未来的存储系统中发挥重要作用。

然而，广泛部署非易失性存储器并非没有挑战。与非易失性存储器相关的挑战之一是非易失性存储器所表现出的耐久性有限，即非易失性存储器中存储单元的物理属性决定了对存储单元的写入次数上限（通常介于 10^7 至 10^8[13] 之间）。超过这一上限后，存储单元可能失去改变状态的能力，从而导致数据错误。当足够多的存储行达到耐久性极限时，这反过来可能导致系统故障。当备用存储行的数量少于写入次数超过耐久性极限所规定次数的存储行数量时，就会发生这种情况。

我们在这项工作中发现，非易失性存储器的耐久性限制使得将为动态随机存取存储器等主流存储器广泛研究的解决方案直接移植到非易失性存储器上并不现实。具体而言，我们发现 ORAM 原语也存在这一问题；该原语通常用于解决动态随机存取存储器系统中的内存总线侧信道漏洞。为利用这一漏洞，攻击者会监听内存总线，以了解程序的内存地址轨迹。先前的工作 [40] 表明，即使进行了数据加密，攻击者也可以仅通过观察程序的内存地址轨迹来获取程序的敏感信息。为应对这一漏洞，ORAM 原语在每次内存访问时都会访问数量多一个数量级的内存位置，并在每次访问后进一步打乱内存。因此，在 ORAM 下，任意内存地址轨迹在计算上都无法与任何其他长度相同的地址轨迹区分开来。

然而，ORAM 原语的这一特性虽然能够有效应对内存总线侧信道，却严重影响非易失性存储器的耐久性。这是因为，在均匀的内存写入流量（写入流量均匀分布于整个内存空间）下，与非均匀内存流量相比，存储行更不容易达到耐久性极限；在非均匀内存流量下，少数存储行的写入频率高于其他存储行。目前最先进的 ORAM 实现 Path-ORAM [37], 然而，却呈现出指数级的写入分布，严重影响非易失性存储器的耐久性。这是因为 Path-ORAM 将内存组织为一棵二叉树，并在每次内存访问时读取和写入该树中的一条完整路径。图 1 展示了 Path ORAM 的内存布局。如图 1所示，Path ORAM 将内存排列成平衡二叉树结构，其中同一层级的每个节点均被等量写入。从根节点到叶节点，每个节点的预期写入次数呈指数级递减。这会导致指数级的写入分布，即内存树中距离根节点越近的节点，其写入频率远高于树中的叶节点。ORAM 原语所表现出的这种非均匀写入流量势必会严重影响非易失性存储器的使用寿命

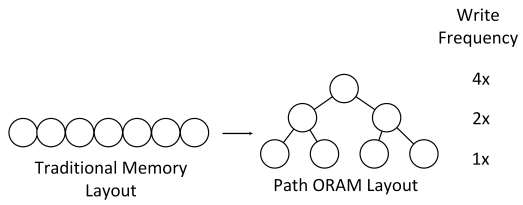


图 1：传统内存布局（左）与 Path ORAM 布局（右）的对比。Path ORAM 导致指数分布的内存访问模式。右侧的数字表示每个层级中每个节点的预期写入次数。

（在不使用 ORAM 时基准值的 1%），甚至使得在非易失性存储器中应对内存总线侧信道变得不切实际。

请注意，先前的工作 [28] 已经观察到，即使是典型的应用行为也可能加剧非易失性存储器的耐久性约束，因为这些行为会导致写入流量呈现不均匀性。因此，为了解决这一问题，先前的工作提出了若干磨损均衡算法 [7, 11, 28, 14, 5]。这些算法的核心是通过在必要时采用内存行重映射，尽可能均匀地将应用写入流量分散到非易失性存储器空间中。然而，正如我们在本文中所展示的，ORAM 原语会严重加剧这些已有方案的压力，因此，现有最先进的磨损均衡算法 [28] 在存在 ORAM 的情况下只能达到 2.8% 的使用寿命。

为了解决 NVM耐久性约束，同时利用 ORAM 原语应对内存总线侧信道漏洞，本文注意到，ORAM 原语的固有特性可以用于设计一种磨损均衡算法，直接应对 ORAM 原语带来的耐久性压力。具体而言，尽管 Path-ORAM [37] 实现的 ORAM 原语本身表现出指数写入分布，但这种分布是以确定性方式产生的。也就是说，对给定内存行的预期写入次数是静态可知的，具体取决于该内存行在 Path-ORAM树中的位置（例如，根节点的写入次数是其子节点的两倍）。我们利用 ORAM 原语固有特性所提供的这一信息，对最先进的磨损均衡算法 *Start-Gap* [28]进行增强，使其能够感知 ORAM 原语。我们将这种直接解决 NVM耐久性约束的面向 NVM 的 ORAM 设计称为**E-ORAM**。我们展示了所提出的设计如何实现 91% 的使用寿命，并在系统故障前执行数量级更多的读写操作，同时仅引入低于 0.02% 的性能开销。

本工作的贡献如下：

- 我们在本工作中发现，非易失性存储器（NVM）的耐久性限制使得在非易失性存储器上实现 Oblivious-RAM（ORAM）原语并不切实际。ORAM 是一种针对内存总线侧信道漏洞的最先进防御方案，但使用它会使系统寿命降至未采用 ORAM 的基准系统的 1%。
- 为应对这一挑战，我们观察到 ORAM 原语的固有特性可以被利用来设计一种磨损均衡算法，直接应对 ORAM 原语所面临的耐久性压力。
- 我们在为非易失性存储器设计 ORAM 时利用了上述发现，并提出了名为 **E-ORAM** 的方案。该方案将最先进的磨损均衡算法 *Start-Gap* [28]与 ORAM 原语感知能力相结合。
- 我们证明，采用 ORAM 时，最先进的磨损均衡算法 [28] 只能达到 2.8% 的使用寿命，而所提出的 E-ORAM 设计可达到 91% 的使用寿命，使其在系统故障前能够执行数量级更多的读写操作，同时引入的性能开销不到 0.02%。

2. 背景与动机

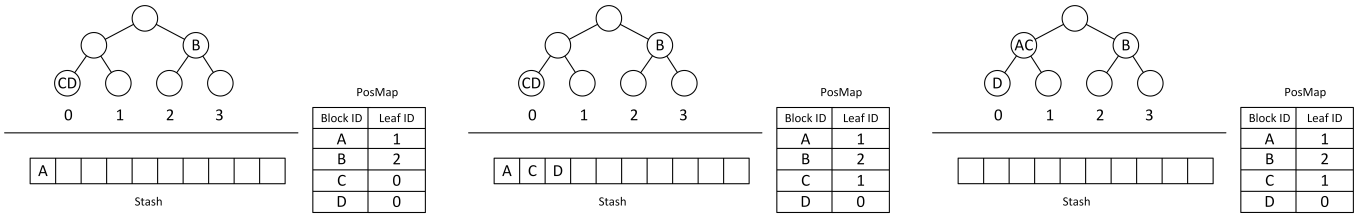
本节将介绍理解本文工作所需的背景知识。为此，我们首先讨论非易失性存储器（NVMs）及其所表现出的耐久性极限，然后讨论 Start-Gap [28], 这一先进的磨损均衡机制。接着，我们讨论本文所采用的威胁模型。之后，我们介绍目前最先进的 ORAM 原语实现 Path-ORAM [37]的背景知识。最后，我们讨论 Path-ORAM 算法如何对非易失性存储器的耐久性造成压力。

2.1. 非易失性存储器

新兴的非易失性存储器（NVMs），包括 PCM、STTRAM 和 ReRAM [22, 20, 39]，能够提供多种理想特性，例如高密度、非易失性以及接近动态随机存取存储器（DRAM）的访问延迟 [31]。此外，由于 NVMs 具备字节可寻址特性（提供加载/存储接口[43]）且具有非易失性，因此可以将可恢复数据结构存放在 NVMs 中，而无需从磁盘访问它们，从而避免更高的性能和能耗成本。

尽管 NVM 具有多项理想特性，使其有望应用于未来的存储系统，但仍面临一些挑战。其中一个挑战是写入耐久性有限。由于 NVM 存储单元的物理特性，典型的写入耐久性范围为 10^7至 10^8 [13]。一旦超过耐久性极限，就可能出现数据错误，因为存储单元可能无法再改变其状态；如果足够多的存储行达到耐久性极限，进而可能导致系统故障。当备用存储行的数量少于写入次数超过耐久性极限规定值的存储行数量时，就会发生这种情况。

已有多项工作被提出，用于解决非易失性存储器写入耐久性有限的问题。这些方法通常可分为写入减少技术和磨损均衡技术。写入减少技术专注于减少非易失性存储器的写入次数。例如，Qureshi 等人 [29]提出使用相变存储器作为动态随机存取存储器的扩展存储。Gogte 等人 [14] 提出使用软件管理系统，将频繁访问的页从非易失性存储器迁移到动态随机存取存储器。另一方面，磨损均衡技术关注于



(a) ORAM 树、暂存区和位置映射表的初始状态。应用程序发出对块 C 的读取请求。

(b) 从根节点到叶节点 0 的路径上所有节点完成读取访问后，ORAM 树、暂存区和位置映射表的状态。块 C 和块 D 被加载到暂存区中。随后，ORAM 将块 C 重新映射到叶节点 1。

(c) 写回操作后，ORAM 树、暂存区和位置映射表的状态。写回路径与读取路径相同。块 D 被映射到叶节点 0，因此被推送到叶节点 0。块 C 被映射到叶节点 1，因此不同于之前，它无法被放置在叶节点块中。块 A 可以从暂存区排出到树中。

图 2：Path ORAM 访问及其中涉及的重要硬件结构示意图。

将内存写入均匀地分散到整个非易失性存储器空间。我们将在第 8节讨论若干现有的磨损均衡技术，并将其与我们提出的工作进行比较。

2.2. 基线 Start-Gap

Start-Gap 将非易失性存储器划分为大小相等的组。每个组预留一个空的缓存行。中央处理器为每个组维护一个起始寄存器和一个间隙寄存器。每个内存地址都将根据起始寄存器和间隙寄存器的值进行转换。中央处理器为每个组维护一个计数器，用于记录截至目前对该组执行的写入次数。当计数器达到阈值时，间隙寄存器将被更新，从而将该组中的一个缓存行映射到新地址。当间隙寄存器完成对整个组的循环后，起始寄存器将被更新。该算法还采用地址随机化器，在使用起始寄存器和间隙寄存器将逻辑地址进一步转换为物理地址之前，先将逻辑地址映射到中间地址。

2.3. 威胁模型

我们假设存在一种安全处理器，应用程序可以利用隔离执行 [23]。我们假设应用程序的执行过程及其在处理器结构（寄存器、缓存、片上网络）中的数据都是安全的，并且与其他应用程序相互隔离。我们假设系统采用非易失性存储器（NVM）作为主存。我们假设非易失性存储器不受信任，因此，为了防御冷启动（重新启动）攻击 [17]，存储在内存中的数据均经过加密。此外，我们还假设攻击者能够物理访问计算机，因此可以探测片外内存总线 [10] ，以获取处理器与非易失性存储器之间传输的地址轨迹和数据。先前的研究 [40] 表明，即使数据经过加密，攻击者也可以仅通过观察应用程序的地址轨迹，推断出有关该应用程序的敏感秘密信息。我们假设已有解决方案可以缓解缺页侧信道攻击 [2, 8]。最后，先前的研究还针对其他重要漏洞提出了解决方案，例如功耗 [19]、热量 [27]、程序执行时间 [41] 侧信道攻击，以及通过网络上的通信模式泄露 [35, 26]确实存在，我们假设解决这些漏洞超出了本文的范围。

2.4. 路径无感知随机存取存储器

不经意随机存取存储器 [15]，是一种密码学原语，它使内存访问轨迹在计算上无法与长度相同的随机访问轨迹区分开来。迄今为止，尽管已有多种实现该原语的方案 [9, 16, 16, 21]，但这些方案的核心是 Path ORAM [37]，它是 ORAM 原语最实用的实现。本节将介绍 Path-ORAM 算法（下文简称 ORAM）的相关背景。

ORAM 将内存组织成一棵平衡二叉树，每个 ORAM 树节点都包含固定数量的槽位，每个槽位可存储一个数据块（通常是一个缓存块）。ORAM 树具有一个利用率因子，表示树中实际数据块占所有块的比例。非数据块由虚拟块填充。每个真实数据块都有对应的叶节点标识符，表明该数据块存储在从根节点到与此叶节点标识符对应的叶节点的路径上。此映射信息由 位置映射表(PosMap) 结构维护。所有块，无论是真实数据块还是虚拟块，在存入内存前都会进行加密。加密元数据（例如块标识符（每个节点中的槽位）、叶节点标识符以及加密计数器）也由 ORAM 树维护。ORAM 还使用一种片上结构暂存区来辅助其运行。

接下来，我们以图中的 3 层 ORAM 树为例，说明 ORAM 的工作原理。2 假设应用程序发出读取数据块 C 的请求。图2a 展示了 ORAM 树、暂存区和位置映射表的初始状态。收到读取请求后，算法首先查找位置映射表，找到数据块 C 对应的叶节点标识符，即 0（树中的叶节点标记为 0 到 3）。根据 ORAM 原语，每次程序内存访问都会转换为对从根节点到与

该数据块关联的叶节点的路径进行读取和写入。因此，算法会读取并写入通往叶节点 0 的路径，如图2a所示。为此，ORAM 会对从根节点到叶节点 0 路径上的所有节点执行读操作，并将真实数据块（C 和 D）载入暂存区。路径读取完成后，所需的数据块已在暂存区中，经过解密后提供给处理器。此时，ORAM 会将数据块 C 重新映射到新生成的随机叶节点 ID；本例中该 ID 为 1，如图2b所示。此重新映射确保下次访问数据块 C 时，会访问 ORAM 树中一条完全随机的路径。最后，ORAM 将数据块写回刚才读取的同一路径，并尽可能将暂存区中重新加密的数据块放置到更深的节点中。本例中，数据块 D 映射到叶节点 0，因此可以放入叶节点 0。数据块 A 和 C 映射到叶节点 1，因此它们能存放的最深位置是叶节点 0 和 1 的父节点。

每次访问数据块时，都将其重新映射到一个随机叶节点，因此 ORAM 能够确保每次访问同一数据块时，实际访问的都是一组全新的数据块。此外，每次访问都会对所有被访问的数据块重新加密，这有助于隐藏应用程序的地址轨迹。

2.5. 在非易失性存储器中实现 ORAM 的挑战

本节讨论 ORAM 原语的固有特性如何严重影响非易失性存储器的耐久性。原因在于，ORAM 会产生倾斜写入访问模式，如图 3所示。如第 2.4 节所述，采用 ORAM 时，内存被组织为二叉树，内存访问会转换为对树中路径的读写。这会形成图 3所示的情况：靠近根节点（节点 ID 较小）的节点，其写入频率远高于靠近叶节点（节点 ID 较大）的节点。根据 ORAM 的特性，从根节点出发，其两个子节点被访问的概率相同，因此每个子节点的访问频率是根节点的一半。所有节点都具有这一访问特性，因此，随着层级线性增加，每个层级中每个节点的写入次数会呈指数级递减。因此，叶节点的写入频率远低于根节点，但它们占据了树的一半。

图中展示了 ORAM 的写入分布对非易失性存储器耐久性造成的影响。4 （有关我们方法的详细信息，请参见第7 节）。在图4 中，我们展示了实现 ORAM 的非易失性存储器的系统使用寿命。我们将系统寿命定义为：

$$\text{使用寿命} = \frac{\#\text{内存写入} \times \text{ORAM 树路径长度} \ \#\text{缓存行} \times \text{最大耐久次数}}{}$$

由于 ORAM 中的每次程序内存访问都会转换为对 ORAM 树中一条路径的读取和写入（如第2.4节所述），在上述公式中，分子计算了实现 ORAM 的非易失性存储器在系统故障前的总读取/写入次数。分母计算了在完美磨损均衡下内存可进行的理想读取/写入次数，其中最大耐久次数指缓存行的耐久性，例如，

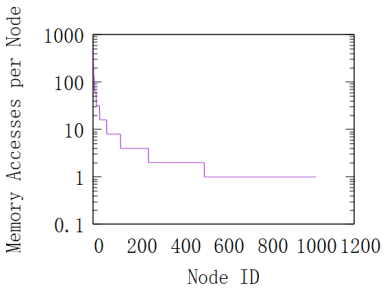


图 3：ORAM 的指数写入分布：x 轴表示节点ID。y 轴以对数刻度显示各节点的预期写入次数，并以叶节点的预期写入次数进行归一化。我们假设树有 10 个层级，且越靠近根节点的节点分配的 ID 越小。

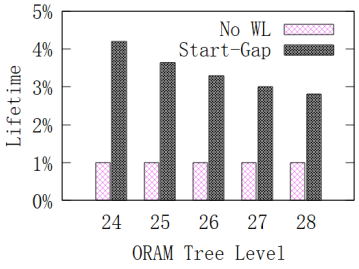


图 4：ORAM 会显著缩短非易失性存储器的使用寿命：该图展示了启用和未启用 Start-Gap [28] 的 ORAM 非易失性存储设备的使用寿命。我们假设采用基于区域的 Start-Gap，并使用随机分组和 256 个组，因为在我们的敏感性分析中，这种方案的磨损均衡效果最佳。

缓存行发生故障前的写入次数。我们将最大耐久次数 Wmax 设定为 10^7。当超过 1% 的缓存行达到耐久性极限时，我们认为非易失性存储器发生故障，并进而导致系统故障，这与 [14]中采用的机制类似。此外，我们展示了未采用磨损均衡以及采用最先进的磨损均衡解决方案 *Start-Gap* [28],的情况下的使用寿命。如图4 所示，这两种配置的使用寿命均不足 4%，而且随着 ORAM 树规模的增大，使用寿命进一步缩短（= 即非易失性存储器容量增大）。新兴的非易失性存储器部署规模已达到 TB 级 [1], ，这意味着需要构建规模非常庞大的树，例如 32 层的树。因此，预计其使用寿命将非常短。这使得在非易失性存储器中实现 ORAM 不切实际，进而也无法有效解决非易失性存储器中的内存总线侧信道问题。

3. 平等主义 ORAM——核心思路

本节将介绍 E-ORAM 的设计灵感。E-ORAM 是我们提出的一种磨损均衡算法，旨在解决 ORAM 原语给非易失性存储器带来的耐久性压力。

图 5 展示了一个包含31个节点的简单5层 ORAM 树。在此示例中，我们为每个节点分配了唯一的节点ID。

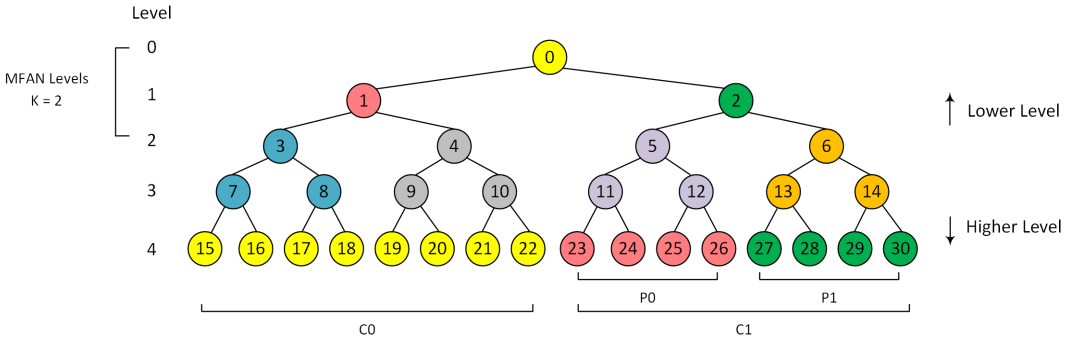


图 5：E-ORAM 中静态分组的示例：一棵包含 5 层和 31 个节点的 ORAM 树，被划分为 7 个组，每组用不同颜色标识。MFAN 层，K（在第 4.1 节中定义），为 2。

距离根节点越近的节点，其节点ID越小。对该树进行N次 ORAM访问，对于5个层级中的每一层，总共会产生$5*N$次节点读写。然而，由于第0层到第4层分别包含1、2、4、8、16个节点，因此这些层级中每个节点的读写次数分别为N、$N2$、$N4$、$N8$、$N16$。理想磨损均衡算法的目标是确保31个物理节点位置中的每一个都经历相同次数的写入，即 $^{5}*N31$。通过定期将逻辑节点重新映射到不同的物理节点，可以实现这一目标，从而使写入尽可能均匀地分布。为了尽量降低重新映射带来的性能开销，我们必须确保：(i) 重新映射次数仅达到必要程度；(ii) 将对逻辑节点的访问路由至相应物理节点所产生的间接寻址开销必须保持较低。

E-ORAM 基于以下观察：层级 0、1 和 2 中的节点访问频率高于理想值，而层级 3 和 4 中的其余节点访问频率低于理想值。如果我们将层级 0 中的根节点与层级 4 中的八个节点组成一个组，并假设组内实现理想磨损均衡，则可以达到每节点平均写入次数 $N6$，距离理想值仅差 3%（$\frac{5*N}{31}$）。类似地，由层级 1 中的一个节点和层级 4 中的四个节点组成的组，距离理想值仅差 7%。正如我们稍后将详细讨论的那样，持续将靠近根节点的层级中一个写入频率较高的节点（下文称为最频繁访问节点，即 MFAN）与相对写入频率较低的层级中的一些伙伴节点（下文称为伙伴节点，即 PNs）组成组，并在各个组内实施磨损均衡，可以使我们非常接近理想磨损均衡。我们将这种组内磨损均衡定义为 MFAN 移动。

请注意，先前最先进的磨损均衡方案 Start-Gap [28], 也提出将内存空间划分为多个组，并在每个组内进行磨损均衡。然而，我们以几种重要方式扩展了这一基本思路。在处理 ORAM 的指数写入分布时，由于 Start-Gap 选择将内存地址划分为随机的、固定大小的磨损均衡组，因此效果不佳。对于固定大小的组，如果某个组包含写入非常频繁的 ORAM 节点（例如根节点），那么该组很可能比只包含写入相对不频繁的节点（例如仅包含叶节点）的其他组更早磨损。

这种不均衡限制了 Start-Gap 在 ORAM 式访问模式下所能实现的使用寿命。不过，ORAM 式访问模式的一个优点是，鉴于其随机化特性和二叉树结构，只要知道 ORAM 树的大小，就能预先确定每个节点写入次数占总写入次数的比例。利用这一信息，可以对 ORAM 树进行静态分组，使各组中每个节点的平均写入次数大致相同。我们的核心思路是采用 ORAM 原语感知的方式，将节点划分为大小不一的组，而非固定大小的组（Start-Gap 则采用随机划分）。这种设计使写入频繁的节点（如根节点）能够与大量写入不频繁的节点分在同一组中，从而降低每个节点的平均写入次数。不过，采用大小不一的组也带来了挑战：如何跟踪节点所属的组，以及如何在每个组内进行磨损均衡移动。接下来我们将讨论应对这些挑战的方案。

4. 平等主义 ORAM

本节将讨论 E-ORAM 的两个组成部分：1）用于将 ORAM 树节点划分为组的分区算法；2）节点重映射算法，该算法可在同一组内高效地重映射节点。

4.1. 静态分组

E-ORAM 的第一部分是对 ORAM 树进行静态分区。我们将 ORAM 树划分为多个组，目标如下：（i）每组的平均每节点写入次数接近整棵树的平均每节点写入次数；（ii）每组中恰好有一个节点的写入频率高于所有其他节点（即第3节所定义的 MFAN），而其他所有节点的访问频率预计相同（即第 3 节所定义的伙伴节点）。这一目标旨在简化 E-ORAM 的硬件实现，具体说明见§4.2.1。

鉴于这两项原则，我们可以设计一种算法来

划分我们的 ORAM 树。对于具有 L 层级的 ORAM 树（其中层级 0 为根节点），我们将选择一个阈值层 t，使得属于层级 0 至（$t-1$）的所有节点都与层级（$L-1$）分组，该层级为叶层。我们将在本节稍后介绍具体如何选择t 的值。然后，将叶层中的节点进一步划分为t个大小相等的块，并为其分配块 ID，定义为C_i，其中 $0 \le i \le t$。C_i中的节点与层级 i中的节点配对

以图 5 为例，我们首先选择 $t = floor(5/2) = 2$，然后将第 4 层的节点划分为 2 个块，C_0 和C_1。 C_0 中的节点（第 4 层的黄色节点）与根节点（也是黄色）分为一组，而第 1 层的两个节点（节点 1 和节点 2，分别为红色和绿色）则分别与C_1 中一半的节点分为一组（分别为红色和绿色）。我们将C_1 中的每个节点子集称为 Part，并为其分配 Part ID。至此，我们已为第 0、1 层和第 4 层的节点分组，剩下第 2、3 层的节点待分组。如图 5所示，第 2、3 层的节点可视为四棵彼此独立、规模较小的 ORAM 树（分别为蓝色、灰色、紫色和橙色）。我们可以将分区策略递归地应用于这些中间的较小 ORAM 树，直至整棵较大的树完成分区。我们的算法所形成的组有一个重要特征：每组包含一个 MFAN（最频繁访问节点），其余伙伴节点均属于同一个访问频率较低的层级。根据这一特征，我们将包含 MFAN 节点的最高层级定义为 K，并称之为 MFAN 层。在本例中，$K = 2$。讨论第 4.2.2 节中如何触发 MFAN 移动，以及第 5.1 节中的性能开销时，这一数值至关重要。

我们尚未讨论的最后一个方面是如何选择合适的阈值层t。假设组内磨损均衡完美，我们对t的不同取值进行了实验。我们发现，阈值层取值为$t2$时，最易受损组中的每节点写入次数最低，因此这是我们算法中的最优阈值层取值。

4.2. 节点重映射

虽然组是静态确定的，但在运行时，磨损均衡算法必须执行以下操作：(i) 决定何时对每个组执行 MFAN 移动；(ii) 准确维护逻辑节点 ID 与对应物理位置之间的映射，并将已执行的所有 MFAN 移动考虑在内。除了确保功能正确外，我们还致力于高效执行上述两项操作，同时降低存储开销。接下来，我们将介绍如何执行这两项操作。

4.2.1. MFAN移动算法 本节将讨论如何处理 MFAN 移动。注意，对于组 g，发生 MFAN 移动时，我们需要在 MFAN 与其某个伙伴节点之间执行数据交换。我们将以图 5为例说明 MFAN移动算法。

考虑 在红色组中，总共有五个节点： the MFAN（第 1 层的节点 1）和四个伙伴节点（P1、P2、P3、P4，分别是第 4 层的节点 23、24、25、26）。为便于说明，我们可以将这五个节点按 P1、P2、P3、P4、MFAN 的顺序排列成一个数组，如图所示。6a。在对该组执行达到阈值的写入次数后，我们将 MFAN 与数组中其左侧的节点交换（必要时循环至数组另一端）。图 6b展示了交换 P4 和 MFAN 后数组的状态。交换前，所有对 MFAN 的写入都由物理节点 1 处理，所有对 P4 的写入都由物理节点 26 处理。交换后，所有对 MFAN 的写入都将由物理节点 26 处理，反之亦然。因此，这次交换有效地在物理节点 1 和 26 之间实现了磨损均衡。同样，在新的排列中达到写入次数阈值后，MFAN（此时位于物理节点 26）将与其左侧的伙伴节点 P3（此时位于物理节点 25）交换，如图所示。6c。准备进行下一次交换时，物理节点 25、26 和 1 的写入次数都将大致相同，也就是说，它们已经实现了磨损均衡。我们会继续在组内移动 MFAN，以实现有效的组内磨损均衡。

关于这种组内磨损均衡方法，有几点重要事项需要注意：

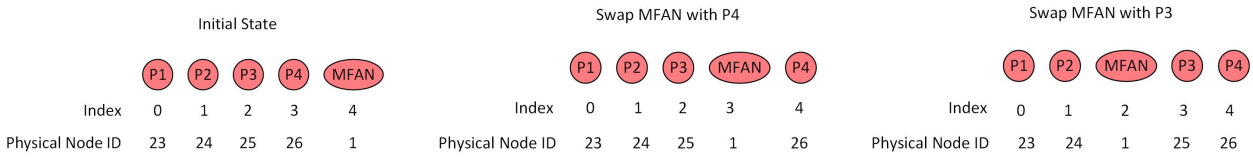
简单且理想的组内磨损均衡： 根据我们构建组的方式，每个组包含一个 MFAN 和多个伙伴节点，并且预计每个伙伴节点都会产生相同次数的写入。从写入频率的角度来看，所有伙伴节点都是等价的，因此只需移动 MFAN，无需移动其他节点（除非它们要与 MFAN 进行交换），即可实现理想的组内磨损均衡。

确定性位置： 基于交换的磨损均衡方法的一项要求，是能够在进行一系列交换后准确定位节点。一种跟踪逻辑节点物理位置的方法，是维护用于间接寻址的映射表。然而，鉴于我们按照确定性模式并以固定间隔（每达到写入次数阈值时）移动 MFAN，只需跟踪对该组的写入次数，就足以准确确定 MFAN 当前映射到的物理位置。此外，由于只有 MFAN 会被交换，因此也可以仅根据该组产生的写入次数，通过公式确定伙伴节点的位置。具体而言，下面的两个公式将使我们能够根据对该组的写入次数，准确确定 MFAN 及每个伙伴节点的确切物理位置。

对于 MFAN 的新数组索引，我们可以通过以下公式计算：

$$MFAN = (N_g - 1) - s_g \% N_g \quad (1)$$

其中 N_g 是该组的大小，s_g 表示该组中发生的交换次数。



(a) 红色组的初始状态。

(b) 交换后的红色组状态 MFAN 与 P4。

(c) 交换后红色组的状态 搭配 P3 的 MFAN。

图 6：一个包含一个 MFAN 和四个伙伴节点的组中 MFAN 移动的示意图。

对于初始数组索引为 y 的任意伙伴节点，其新数组索引 y' 可通过以下方式计算：

$$y' = \left(y + floor(\frac{s_g + y}{N_g - 1}) \right) \%N_g \ (\mathrm{y} \neq N_g - 1) \qquad (2)$$

请注意，对于公式 1 和 2，输入值和输出值都是数组索引。例如，如果我们想计算图 6a中所示 MFAN 的位置，可以将 $N_g = 5$ 和 $s_g = 0$ 代入公式 1，得到 4。由于 4 是数组的最后一个元素，并且该元素映射到节点 1，因此需要将值 4 映射到物理节点 1。类似地，执行 1 次交换操作后，公式1 会告诉我们 MFAN 位于数组索引 3。由于元素 3 不是数组中的最后一个元素，我们知道该值必定映射到组中的某个伙伴节点。将 3 与一个 偏移量相加即可找到其物理位置，其中偏移量 表示数组索引 0 所映射到的节点 ID。在此示例中，偏移量 $= 23$，因此 MFAN 现在位于物理节点 26。

不同组的偏移量值各不相同。鉴于组的数量极其庞大，将所有偏移量值存储在片上是不可接受的。幸运的是，我们不难证明，任意组的偏移量值都可以通过以下信息计算得出：1）所有伙伴节点所在的层级，记为 l_P，在本例中为 4；2）MFAN ID，由 (i, j) 指定，表示第 i^{th} 层中的第 j^{th} 个节点；3）与同一伙伴层级共享的层级数量，记为 l_s。我们提出将这些信息存储在片上查找表中，如图 7 所示。在图 7 中，'IsMFAN'列包含一个比特，用于指示某一层级中的节点是否为其所属组的 MFAN。'PartnerLevel'列表示包含伙伴节点的层级，即 l_P。'LevelFrom'列表示与 l_P配对的起始层级，'LevelTo'列表示与 l_P配对的结束层级。本质上，对于给定的层级ID，可以通过将 'LevelTo' 减去 'LevelFrom' 来计算变量 l_s。随后即可轻松计算偏移量：

$$OFFSET = 2^{l_P} - 1 + ChunkID \times ChunkSize + PartID \times PartSize \qquad (3)$$

其中，术语 $2^{l_P} - 1$ 是层级 l_P 中的第一个节点；ChunkID 可以通过值 i 计算得出，在我们的示例中为 1；ChunkSize 可以通过 l_s 的值获得，方法是将数量除以

Level ID	IsMFAN	Level_From	Level_To	Partner_Level
0	1	0	1	4
1	1	0	1	4
2	1	2	2	3
3	0	2	2	3
4	0	0	1	4

图 7：查找表示意图：芯片上存储的查找表示例，用于计算偏移量。

层级 l_P 中的节点数量为 l_s，在我们的示例中为 8；PartID 可通过 j 的值确定，在我们的示例中为 1；最后，PartSize 可通过将 ChunkSize 除以层级 i中的节点数量来计算，在本例中为 4。将所有数值代入公式 3，得到 偏移量 $= 23$。

这种确定性的移动模式意味着，我们不需要为每个组配置成本高昂的映射表，只需在 ORAM 控制器中实现必要的电路，即可计算出每个节点的确切位置。

4.2.2. 触发 MFAN 移动为了触发间隙移动，Start-Gap [28] 为 NVM 中的每个组保留一个片上计数器。每个计数器存储对其对应组的写入次数。当计数器达到阈值 X 时，Start-Gap 将触发对该组的间隙移动。Start-Gap 将X 定义为磨损均衡频率 [28]。我们将在此处采用相同的定义。在我们的情况下，我们在设计中采用了类似的机制。理想情况下，我们可以为所有组存储片上计数器，但由于分区算法生成的组数量极其庞大，这是不现实的。幸运的是，我们观察到各组的写入次数之间存在很强的相关性。回顾一下，ORAM 假设随机访问叶节点。假设这个随机数服从均匀分布。因此，给定静态 ORAM 树分区，我们可以计算每个节点以及每个组的预期写入次数。然后，如果我们保留一个存储截至目前 ORAM访问总次数的全局计数器，就可以计算任意组的预期内存写入次数。此外，我们发现，对于 MFAN 位于层级i中的组，记为$g(i)$（$g(i)$ 可能指 2^i 组），对组$g(i+1)$的预期写入次数约为$g(i)$的一半。这是因为，$g(i+1)$中的 MFAN 节点接收的写入次数是$g(i)$中 MFAN 节点的一半，并且$g(i+1)$中的伙伴节点数量也是$g(i)$中伙伴节点数量的一半。如果$g(i)$中的伙伴节点与$g(i+1)$中的伙伴节点是

位于同一层级，则对组 $g(i+1)$ 的预期写入次数恰好是 $g(i)$ 的一半。

由于对一个组的预期写入次数由其 MFAN 决定，因此我们只需预留一个片上计数器 Ctr，用于存储对非易失性存储器的总写入次数。对于 g(0)，我们以 X 为周期执行 MFAN 移动；对于 g(1)，其 MFAN 移动频率为 g(0) 的一半，以此类推。等价地，对于每 X 次 ORAM 访问，所有 g(0) 都将执行 MFAN 移动；对于每 2X 次 ORAM 访问，所有 g(0) 和 g(1) 都将执行 MFAN 移动，以此类推。因此，MFAN 移动触发算法可以表述为算法 1 所示：

算法 1： MFAN 移动触发算法-

算法：

数据： 计数器片上计数器，MFAN 层级 K，磨损均衡频率 X

结果： 一个包含层级标识符的集合，S。所有组其 MFAN 位于这些层级的需要执行 MFAN 移动

在一次 ORAM 请求中：

如果 $Ctr \% X = 0$ **那么**

  $k \leftarrow 0$；

  当 $k < K \& Ctr \% 2^{k+1}X \neq 0$ **执行**

    $k \leftarrow k+1$；

  end

end

$S \leftarrow \{0, 1, ..., k\}$；

片上计数器 $\leftarrow$ 片上计数器 $+1$；

return (S,片上计数器)；

在算法 1中，片上计数器 指片上计数器； K 指 MFAN 层。在图 5中，K 的值为 2。返回值 S本质上表示需要对 $g(0)$、$g(1)$.....$g(k)$执行 MFAN 移动。

算法1 的一个缺点是，当计数器达到 $2^K X$ 时，所有组都需要执行 MFAN 移动。同时执行大量 MFAN 移动，可能会挤占实际 ORAM 访问所需的资源。例如，对于一棵 32 层 ORAM 树，有$2^{28}-1$ 个组。为避免这一问题，我们改进了 MFAN移动触发算法，使 MFAN 移动错峰执行。下面通过一个示例说明问题及我们的解决方案。为简单起见，我们将每当片上计数器$\%X = 0$ 满足条件时定义为一个检查点。首次片上计数器 达到 X时，我们将其定义为检查点 0。图8 展示了这样一个示例。假设我们有一棵 MFAN 层级为 $K=2$的 ORAM 树。那么在检查点 0，只有 MFAN 节点为根节点的组需要执行 MFAN 移动。在检查点 1，3 个黄色组需要执行 MFAN 移动。在检查点 2，只有根组需要执行 MFAN 移动。这是因为

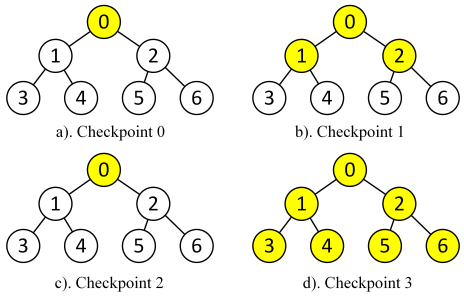


图 8：朴素 MFAN移动触发算法示意图：对于不同的检查点，需要执行 MFAN 移动的组数量存在显著不平衡。

片上计数器 $=3X$，因此 k 仍为 0。在检查点 3，由于片上计数器 $=4X$，因此 k 现在为 2，我们需要对全部 7 个黄色组执行 MFAN 移动。在每个检查点执行的 MFAN 移动数量不均衡，可能导致 ORAM 请求长时间被阻塞。另一种方法是，我们观察到，只需重新安排这些 MFAN 移动的执行顺序即可解决该问题，如图 9所示。我们注意到，可以在每个检查点恰好执行三次 MFAN 移动。在检查点 0，我们对 MFAN 节点是每个层级中的第一个节点的组执行 MFAN 移动。在检查点 1，我们对 MFAN 节点是每个层级中的第二个节点的组执行 MFAN 移动。实质上，需要执行 MFAN 移动的 MFAN 节点 ID 可以通过将检查点对每个 MFAN 层级中的节点数量取模来计算。这样，对于检查点 2，我们得到 MFAN (0, 0, 2)，分别对应于节点 (0, 1, 5)；对于检查点 3，我们得到 (0, 1, 3)，分别对应于节点 (0, 2, 6)。

虽然这看起来已经足够好，但我们仍然观察到类似的缺点。也就是说，在每个检查点，对应的 ORAM 请求相比普通 ORAM 请求会多执行$K+1$ 次 MFAN 移动。我们可以通过将$K+1$ 次 MFAN 移动分摊到X个 ORAM 请求中来利用这一点。也就是说，每当片上计数器 达到$xK+1$时，我们就为一个组恰好执行一次 MFAN 移动。这样，每个 ORAM 请求最多只会额外执行一次 MFAN 移动操作。

最后，我们需要选择合适的 MFAN 移动频率 X。较小的 X 会导致 MFAN 移动次数过多，并因产生额外的内存访问而增加磨损均衡算法的开销；而过大的X 则会导致 MFAN 移动次数过少，妨碍高效的磨损均衡。通过对 X 的不同取值进行实验，我们发现，当最大耐久次数为 10^7 [14],时，频率设为 10000 会使内存访问量增加约 0.1%，同时也有助于显著延长使用寿命。关于 X 的详细性能影响将在 5.1 中讨论。

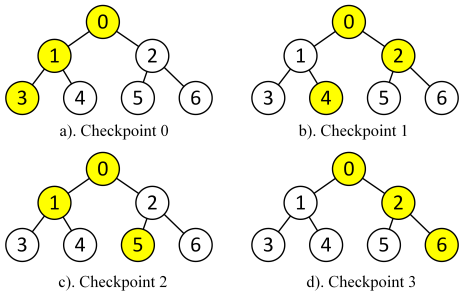


图9：均衡 MFAN 移动触发算法示意图：在新算法中，每个检查点恰好有 3 个组需要执行 MFAN 移动。

5. 算法开销

本节讨论 E-ORAM 带来的性能和存储开销。

5.1. 性能开销

E-ORAM 产生的性能开销可以分为两个部分：(i) MFAN 移动所产生的额外内存读写开销；(ii) ORAM 控制器中用于逻辑节点ID到物理节点ID转换等操作的额外逻辑所产生的开销。

MFAN移动开销： 采用 ORAM 的系统通常受内存限制，因为每次 ORAM访问都需要执行大量内存访问。因此，计算因 E-ORAM 而执行的额外内存访问次数，可以合理近似其产生的性能开销。

对于每次 MFAN 移动，总共需要执行两次节点读取和两次节点写入。根据第 4.2 节中的节点重映射算法可知，每经过 $XK+1$ 次 ORAM访问，就会发生一次 MFAN 移动，其中 X 是磨损均衡频率，K 是 MFAN层级。对于每 $XK+1$ 次 ORAM 请求所执行的节点读取（或写入）总数（不计由磨损均衡导致的额外读取或写入次数）为 $XLK+1$，而我们的算法会额外引入两次读取（或写入），因此开销为：

$$MFAN_{overhead} = \frac{2}{\frac{XL}{K+1}} = \frac{2(K+1)}{XL} \tag{4}$$

对于一棵 32 层 ORAM 树，其 MFAN 层为 27，而 E-ORAM 会使读写次数增加 0.0175%，开销可以忽略不计。

ORAM 控制器开销： 另一部分性能开销来自 ORAM 控制器执行的额外操作，例如查找表重构、逻辑节点ID到物理节点ID的转换等。我们对包含组合逻辑的此类硬件进行了 RTL级综合，结果表明，该开销很小，并且可以在 ORAM访问的关键路径上与其他 ORAM 控制器操作并行执行。因此，这部分性能开销同样可以忽略不计。

5.2. 存储开销

我们的算法产生的存储开销完全位于片上，用于：(i) 跟踪已执行的 ORAM 访问次数的计数器，以及 (ii) 存储静态分区信息的查找表，该信息用于计算 4.2 中的节点重映射。对于计数器而言，64 位计数器足以跟踪 2^{64} 次 ORAM 访问；即使最大耐久次数为 10^8，也远超 32 层 ORAM 树的使用寿命。

其他存储开销是查找表，其中每个条目包含四个元素：一个布尔变量“MFAN 层”，以及用于存储层级信息的 3 个整数。该布尔变量每个条目占用 1 位。对于这三个整数，对于最多包含 64 个层级的树，每个整数使用 6 位就足够了。因此，查找表开销为

$$LookupTable_{overhead} = (6\times 3+1)\times L = 19L(\text{bits}) \tag{5}$$

对于一个 32 层 ORAM 树，查找表开销仅为 76 字节，导致总开销为 84 字节。这表明 E-ORAM 产生的存储开销极小。

6. 讨论

安全影响： 虽然我们的算法显著延长了非易失性存储器的使用寿命，但它偏离了 ORAM 算法：在 ORAM 中，每次程序内存访问都需要转换为一次 ORAM访问。在 E-ORAM 中，当触发 MFAN 移动时，我们不会进一步将其转换为 ORAM 访问。我们认为，这一变化既不会破坏 ORAM 的不可观测性，也不会引入任何新的安全漏洞。这是因为，攻击者现在仅额外知道某次 MFAN 移动已被触发，以及自上次 MFAN 移动以来已执行了 $XK+1$ 次 ORAM 访问。这不会泄露有关应用程序特有行为的信息，除了已执行的 ORAM 访问次数之外，而这一信息攻击者本来就可以获知。

现在我们将讨论如何将 E-ORAM 潜在地集成到旨在提升 ORAM 性能的现有 ORAM 变体中。

Ring ORAM： Ring ORAM [32]通过每次路径读取仅获取一个块，而不是整个路径，并且每进行A 次路径读取就写回一条确定性路径，从而节省 ORAM 带宽。每次路径读取操作都会导致该块元数据更新。这显著节省了 ORAM 带宽，并将 NVM寿命（以天为单位）延长A 倍。然而，Ring ORAM 仍会产生偏斜的访问模式，而 E-ORAM 可以很好地与其结合使用。如果将 E-ORAM 与 Ring ORAM 结合，还可以进行进一步优化，但这超出了本文的范围。

动态随机存取存储器+非易失性存储器 [29]： Qureshi 等人 [29] 提出了一种动态随机存取存储器与非易失性存储器混合内存系统。如果我们在这个混合内存系统上应用 ORAM，我们可以：1）在动态随机存取存储器上较小的树中缓存一些块（称为 ρ），并将所有块存储在非易失性存储器上。（这种架构已在 [25] 中提出。）或者 2）拆分

DRAM 和 NVM 中的地址空间，使得写入频率更高的顶部节点映射到 DRAM。对于第一种情形，NVM 的使用寿命可能会因写入频率降低而延长，但由于 NVM 的树大小没有变化，且对 NVM 的访问模式没有变化，因此 NVM 的使用寿命（使用寿命百分比）不会改变。对于第二种情形，DRAM 充当更大的树顶缓存 [24]。如果 DRAM 大小为 NVM 大小的 3% [29], , 且 NVM 中的 ORAM 树有 32 层（NVM 树），那么 DRAM 中的 ORAM 树（DRAM 树）有 27 层。这个极大的树顶缓存使得对 NVM 的访问模式变为 2^{27} 每棵有五层的较小 ORAM 树。诚然，如果我们对这些 2^{27} 5层 ORAM 树应用基线起始间隔策略，即使不使用 E-ORAM，使用寿命也能显著延长。然而，我们认为方案 1）的性能优于方案 2）。如果将 DRAM 访问延迟定义为 T_D，并将 NVM 访问延迟定义为 T_N，那么我们可以计算两种方案的平均内存访问时间（AMAT）：

$$\begin{aligned} AMAT_1 &= 27T_D + r_m \times 32T_N \\ AMAT_2 &= 27T_D + 5T_N \end{aligned}$$

其中r_m 表示 ρ 未命中率。显而易见，只要r_m 小于 $\frac{5}{32} = 15.6\%$，$AMAT_1$ 就会小于$AMAT_2$。

由于 Setup1) 能为用户提供更好的性能，且其非易失性存储器拥有更大的树，因此按使用寿命衡量，E-ORAM 肯定优于 Start-Gap。

处理器	
ISA	UltraSPARC III 指令集架构
片上多处理器规模和核心频率	1 核，2.6 GHz
重排序缓冲区	128 个条目
非易失性存储器参数	
非易失性存储器设备参数	每通道 2 个秩，每芯片 8 个银行，每银行行数：32768
NVM 总线频率	2666 MHz
NVM 写队列大小	64 个条目
NVM 缓存行耐久度	10^7 次写入
ORAM 配置	
块大小	64B
桶大小	4 行缓存行
ORAM 树级别	24
PLB 大小	64KB
加密/解密延迟	21 个周期
递归位置映射数量	4
暂存区大小	200 个条目

表 1：模拟系统配置

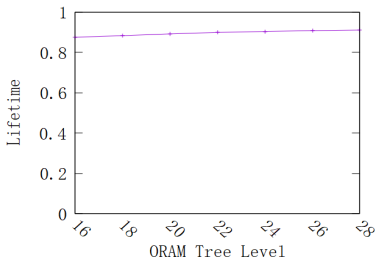


图 10：不同 ORAM 树大小下使用 E-ORAM 模拟的 NVM 使用寿命。与使用 Start-Gap 时仅达到 4% 相比，对于不同大小的 ORAM 树，E-ORAM 均能达到理想使用寿命的 90% 或更高 [28]。

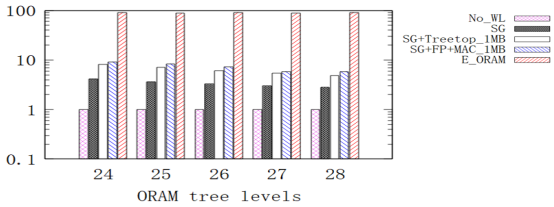


图 11：E-ORAM 与现有最先进方案的对比：该图显示了在各种现有最先进的 ORAM 和磨损均衡配置及 ORAM 树下，设备故障前预期的 ORAM 访问次数，并将其归一化为无磨损均衡基准情况。结果以对数刻度绘制，其中 100 表示理想值。E-ORAM 达到接近理想的使用寿命，而现有最先进方案至少差 10 倍。

7. 评估

7.1. 方法

我们通过应用基于跟踪、周期精确的内存模拟器 USIMM 来评估 E-ORAM 的性能开销[6]。我们在 USIMM 上实现了 Freecursive ORAM [12] 以及我们的磨损均衡引擎。在后端，我们采用了 FR-FCFS 内存调度策略 [34]。对于 ORAM 树，由于更大树的模拟时间过长，我们模拟了一棵相对较小的 24 层树。表 1 展示了我们的模拟器和 ORAM 树的配置。对于跟踪，我们采用了类似的机制 [38]。我们从现有最先进的基准测试中选择了 15 个工作负载，并从 SPEC CPU 中选择了 15 个工作负载 ®2017[4] [1] [2]。选择这些工作负载是因为它们属于内存密集型工作负载[36, 38]。在工作负载经过快速前进以预热末级缓存后，我们收集了 500 万次内存读写的跟踪。

最先进技术比较： 我们将 E-ORAM 与五种 ORAM 设置/配置进行比较：1）不采用任何磨损均衡算法或其变体的 ORAM 算法

[1]有关 SPEC CPU® 2017 的更多信息可从 https://www.spec.org/cpu2017 获取。[2]SPEC CPU® 是标准性能评估公司（Standard Performance Evaluation Corporation）的注册商标。

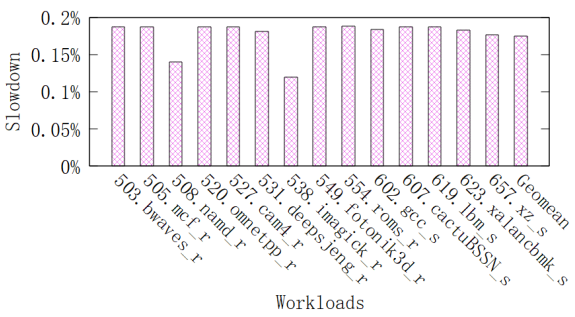


图 12：E-ORAM 的性能开销：与不采用磨损均衡相比，在各种内存密集型 SPEC CPU® 2017工作负载下，E-ORAM 仅引入可忽略的性能下降（≤ 0.2%）。

ORAM，这是我们的基线；2）采用 Start-Gap 的 ORAM 算法 [28]；3）采用 Start-Gap 和 1MB 树顶缓存的 ORAM 算法 [24]；以及 4）ForkPath 算法 [42]，并辅以 Start-Gap 和 1MB 合并感知缓存（MAC）。对于 Start-Gap，我们使用 256 个大小相等的组，以最大限度地延长使用寿命。

7.2. 结果

不同 ORAM 树大小下的 NVM 使用寿命： 图 10展示了不同 ORAM 树大小下模拟得到的使用寿命。如图 10所示，对于 16 层 ORAM 树，E-ORAM 可实现 87.45% 的使用寿命；随着 ORAM 树规模扩大，使用寿命逐渐增加。对于 28 层树，使用寿命可达到 91.04%。由于模拟时间过长，我们无法模拟更大的 ORAM 树。由此可以推断，随着非易失性存储器容量增大，使用寿命会持续提升。

与最先进方案的比较： 图 11 展示了将 E-ORAM 与 §7.1 中提到的另外四种对比设计进行比较时的仿真结果。我们使用不同的 ORAM 树大小对这五种配置进行仿真，并收集设备发生故障前完成的 ORAM 访问次数。纵轴的计算方式是：将某一配置收集到的 ORAM 访问总次数除以基线配置收集到的次数并进行归一化。由于我们观察到 E-ORAM 带来了巨大的使用寿命增益，因此使用对数刻度绘制结果。如图 11 所示，基线配置的数值为 1。Start gap 的性能约为基线的 4 倍。在 1MB 树顶缓存的帮助下，Start gap 可进一步将 ORAM 访问次数提升至约 8 倍。如果在 Start gap 之上应用 Fork Path 并采用 MAC，由于 Fork Path 能够合并冗余的节点访问，因此 ORAM 访问次数最多可进一步提升至 9 倍。与所有这些配置相比，E-ORAM 可将 ORAM 访问次数显著提升至 90 倍，甚至达到配置 4（Fork Path 配置）的 10 倍。E-ORAM 效果如此出色的原因在于，它会根据预期的每节点写入次数动态调整分组。这一信息源于 ORAM 的自身特性，而以往的磨损均衡算法并未加以利用。

性能影响： 图 12 展示了 E-ORAM 对 SPEC CPU® 2017基准测试造成的减速。如图 12 所示，E-ORAM 带来的性能开销可以忽略不计，范围从 0.119%（538.imagick_r）到 0.187%（554.roms_r）。我们在 GAP 基准测试中也观察到了类似结果 [3]。我们还测得，E-ORAM 产生的额外内存访问约为 0.22%。

8. 相关工作

ORAM： 一些工作 [33, 12, 24, 42, 25] 构建并改进了 Path ORAM [37],，后者被认为是最先进的 ORAM 原语实现之一。我们的工作与这些方案正交且互补。

磨损均衡与写入减少： 应对非易失性存储器耐久性问题的既有方案，要么减少对非易失性存储器的写入流量（写入减少），要么采用磨损均衡来平衡写入流量。我们提出的 E-ORAM 与 LEO [30] 等通过减少块加密次数来降低非易失性存储器写入量的写入减少技术正交。

Park 等人 [7] 提出了一种基于年龄的自适应交换与移位磨损均衡算法，该算法会跟踪写入次数。根据所跟踪的访问模式，他们会交换页并在页内移动行。然而，与 E-ORAM 相比，该设计引入了相对较大的片上存储开销。Ferreira 等人 [11] 提出了一种随机选择两个页进行交换的磨损均衡算法。尽管该设计简单，但在 ORAM 中无法实现均匀的写入模式，因为写入最频繁的节点（根节点）由于 ORAM 树中的节点数量众多而不太可能被选中。Hu 等人 [18] 提出了一种基于软件的磨损均衡算法，用于在相变存储器中均匀分布数据。然而，在存在 ORAM 的情况下无法应用该技术，因为 ORAM 控制器仍会产生呈指数分布的内存访问模式。Chang 等人 [5] 提出了一种使用动态窗口大小的滑动窗口磨损均衡技术。该技术会跟踪窗口内地址的内存写入，并根据观察到的写入次数交换存储行。

9. 结论

ORAM 会产生呈指数分布的内存写入模式，由于非易失性存储器的写入耐久性有限，这会显著缩短其使用寿命。我们提出 E-ORAM，利用这种偏斜但确定性的内存访问模式来缓解非易失性存储器使用寿命的缩短问题。E-ORAM 使用预计算静态分区分组技术来确定磨损均衡粒度，并采用高效的组内磨损均衡算法。最后，我们表明，与不进行磨损均衡相比，E-ORAM 带来的性能开销低于 0.2%，存储开销可忽略不计（对于 32 层 ORAM 树，仅需 84 字节），且使用寿命提升 90 倍。

参考文献

[1] 英特尔傲腾持久内存和 *SAP HANA* 平台配置。英特尔和 SAP。[2] Shaizeen Aga 和 Satish Narayanasamy。不经意按需分页：面向安全飞地。在 第*46*届国际计算机体系结构研讨会论文集中，ISCA '19，第 372–384 页，美国纽约州纽约市，2019年。计算机协会。[3] Scott Beamer、Krste Asanovi´c 和 David Patterson。GAP 基准测试套件，2017年。[4] James Bucek、Klaus-Dieter Lange 和 Jóakim v. Kistowski。SPEC CPU2017：下一代计算基准测试。在 *2018*年 *ACM/SPEC* 国际性能工程会议伴随论文集中，ICPE '18，第 41–42 页，美国纽约州纽约市，2018年。计算机协会。[5] Hung-Sheng Chang、Yuan-Hao Chang、Pi-Cheng Hsiu、Tei-Wei Kuo 和 Hsiang-Pang Li。面向基于相变存储器的存储系统的基于步进的磨损均衡。*ACM* 设计自动化与电子系统汇刊，20(2)，2015年3月。[6] Niladrish Chatterjee、Rajeev Balasubramonian、Manjunath Shevgoor、Seth H. Pugsley、Aniruddha N. Udipi、Ali Shafiee、Kshitij Sudan、Manu Awasthi 和 Zeshan Chishti。USIMM：犹他州模拟内存模块——JWAC 内存调度锦标赛的仿真基础设施，2012年。[7] C. Chen、P. Hsiu、T. Kuo、C. Yang 和 C. M. Wang。具有近乎零搜索开销的基于老化的相变存储器磨损均衡。在 *2012*年设计自动化会议中，第 453–458 页，2012年。[8] Victor Costan、Ilia Lebedev 和 Srinivas Devadas。Sanctum：用于强软件隔离的最小硬件扩展。在 第*25*届 *USENIX* 安全研讨会（*USENIX Security 16*）中，第 857–874 页，美国得克萨斯州奥斯汀，2016年8月。USENIX 协会。[9] D. Mazieres、D. Boneh 和 R. A. Popa。远程不经意存储：让不经意随机存取存储器变得实用。手稿，2011年。[10] John Demme、Robert Martin、Adam Waksman 和 Simha Sethumad-havan。侧信道脆弱性因子：衡量信息泄露的指标。在 第*39*届年度国际计算机体系结构研讨会论文集中，ISCA '12，第 106–117 页，美国，2012年。IEEE计算机学会。[11] Alexandre P. Ferreira、Miao Zhou、Santiago Bock、Bruce Childers、Rami Melhem 和 Daniel Mossé。延长相变存储器主存的使用寿命。在 欧洲设计、自动化与测试会议论文集中，DATE '10，第 914–919 页，比利时鲁汶，2010年。欧洲设计与自动化协会。[12] Christopher Fletcher、Ling Ren、Albert Kwon、Marten van Dijk 和 Srinivas Devadas。Freecursive ORAM： [几乎] 免费的递归与基于位置的不经意随机存取存储器完整性验证。*ACM SIGPLAN* 通知，50：103–116，2015年5月。[13] R. F. Freitas 和 W. W. Wilcke。存储级内存：下一代存储系统技术。*IBM*研究与开发期刊，52(4.5)：439–447，2008年。[14] Vaibhav Gogte、William Wang、Stephan Diestelhorst、Aasheesh Kolli、Peter M. Chen、Satish Narayanasamy 和 Thomas F. Wenisch。持久内存的软件磨损管理。在 第*17*届 *USENIX* 文件与存储技术会议论文集中，FAST'19，第 45–63 页，美国，2019年。USENIX 协会。[15] O. Goldreich。通过不经意随机存取存储器实现软件保护与仿真理论。在 第*19*届 *ACM* 年度计算理论研讨会论文集中，STOC '87，第 182–194 页，美国纽约州纽约市，1987年。计算机协会。[16] Michael T. Goodrich 和 Michael Mitzenmacher。通过不经意随机存取存储器仿真实现外包数据的隐私保护访问。在 Luca Aceto、Monika Henzinger 和 Ji˘rí Sgall（编）所著的 自动机、语言与编程中，第 576–587 页，柏林、海德堡，2011年。施普林格·柏林·海德堡。[17] J. Alex Halderman、Seth D. Schoen、Nadia Heninger、William Clark-son、William Paul、Joseph A. Calandrino、Ariel J. Feldman、Jacob Appelbaum 和 Edward W. Felten。以免我们忘记：针对加密密钥的冷启动攻击。*ACM* 通讯，52(5)：91–98，2009年5月。[18] J. Hu、M. Xie、C. Pan、C. J. Xue、Q. Zhuge 和 E. H. . Sha。嵌入式系统中混合相变存储器/动态随机存取存储器主存的低开销软件磨损均衡。*IEEE* 超大规模集成（*VLSI*）系统汇刊，23(4)：654–663，2015年。[19] Paul Kocher、Joshua Jaffe、Benjamin Jun 和 Pankaj Rohatgi。差分功耗分析简介。密码工程期刊，1：5–27，2011年4月。[20] Emre Kultursay、Mahmut Kandemir、Anand Sivasubramaniam 和 Onur Mutlu。将 STT-RAM 作为高能效主存替代方案进行评估。在 *ISPASS 2013 - IEEE* 系统与软件性能分析国际研讨会，ISPASS 2013 - IEEE 系统与软件性能分析国际研讨会，第 256–267 页，2013年。版权：版权所有 2013 Elsevier B.V.，保留所有权利；2013 IEEE 系统与软件性能分析国际研讨会，ISPASS 2013；会议日期：2013年4月21日至2013年4月23日。[21] Eyal Kushilevitz、Steve Lu 和 Rafail Ostrovsky。基于哈希的不经意随机存取存储器的（不）安全性与一种新的平衡方案，第 143–156 页。[22] Benjamin Lee、Engin Ipek、Onur Mutlu 和 Doug Burger。将相变存储器构建为可扩展的 DRAM 替代方案。第 37 卷，第 2–13 页，2009年6月。[23] Dayeol Lee、David Kohlbrenner、Shweta Shinde、Krste Asanovi´c 和 Dawn Song。Keystone：用于构建可信执行环境的开放框架。在 第*15*届欧洲计算机系统会议论文集，EuroSys '20，美国纽约州纽约市，2020年。计算机协会。[24] Martin Maas、Eric Love、Emil Stefanov、Mohit Tiwari、Elaine Shi、Krste Asanovic、John Kubiatowicz 和 Dawn Song。Phantom：安全处理器中的实用不经意计算。在 *2013*年 *ACM SIGSAC*计算机与通信安全会议论文集，CCS '13，第 311–324 页，美国纽约州纽约市，2013年。计算机协会。[25] Chandrasekhar Nagarajan、Ali Shafiee、Rajeev Balasubramonian 和 Mohit Tiwari。 ρ：宽松分层随机访问机。在 第*24*届编程语言与操作系统的体系结构支持国际会议论文集，ASPLOS '19，第 659–671 页，美国纽约州纽约市，2019年。计算机协会。[26] Olga Ohrimenko、Manuel Costa、Cédric Fournet、Christos Gkantsidis、Markulf Kohlweiss 和 Divya Sharma。观察并防止 MapReduce 中的信息泄漏。在 第*22*届 *ACM SIGSAC*计算机与通信安全会议论文集，CCS '15，第 1570–1581 页，美国纽约州纽约市，2015年。计算机协会。[27] Jean-Jacques Quisquater 和 David Samyde。侧信道密码分析。在 互联网通信安全研讨会（*SECI*），2002年。[28] Moinuddin K. Qureshi、John Karidis、Michele Franceschini、Vijayalakshmi Srinivasan、Luis Lastras 和 Bulent Abali。通过起始间隙磨损均衡提升基于相变存储器的主存的使用寿命和安全性。在 第*42*届 *IEEE/ACM* 微体系结构国际研讨会论文集，MICRO 42，第 14–23 页，美国纽约州纽约市，2009年。计算机协会。[29] Moinuddin K. Qureshi、Vijayalakshmi Srinivasan 和 Jude A. Rivers。使用相变存储器技术的可扩展高性能主存系统。在 第*36*届年度国际计算机体系结构研讨会论文集，ISCA '09，第 24–33 页，美国纽约州纽约市，2009年。计算机协会。[30] Joydeep Rakshit 和 Kartik Mohanram。LEO：面向非易失性存储器的低开销加密 ORAM。*IEEE* 计算机体系结构快报，PP:1–1，2018年1月。[31] S. Raoux、G. W. Burr、M. J. Breitwisch、C. T. Rettner、Y. . Chen、R. M. Shelby、M. Salinga、D. Krebs、S. . Chen、H. . Lung 和 C. H. Lam。相变随机存取存储器：一种可扩展技术。*IBM*研究与开发期刊，52(4.5):465–479，2008年。[32] Ling Ren、Christopher Fletcher、Albert Kwon、Emil Stefanov、Elaine Shi、Marten van Dijk 和 Srinivas Devadas。常数很重要：对不经意随机存取存储器的实用改进。在 第*24*届 *USENIX*安全研讨会（*USENIX Security 15*），第 415–430 页，华盛顿特区，2015年8月。USENIX 协会。[33] Ling Ren、Xiangyao Yu、Christopher W. Fletcher、Marten van Dijk 和 Srinivas Devadas。安全处理器中路径无感知随机存取存储器的设计空间探索与优化。在 第*40*届年度国际计算机体系结构研讨会论文集，ISCA '13，第 571–582 页，美国纽约州纽约市，2013年。计算机协会。[34] Scott Rixner、William J. Dally、Ujval J. Kapasi、Peter Mattson 和 John D. Owens。内存访问调度。在 第*27*届年度国际计算机体系结构研讨会论文集，ISCA '00，第 128–138 页，美国纽约州纽约市，2000年。计算机协会。[35] Felix Schuster、Manuel Costa、Cédric Fournet、Christos Gkantsidis、Marcus Peinado、Gloria Mainar-Ruiz 和 Mark Russinovich。VC3：云中的可信数据分析。技术报告 MSR-TR-2014-39，2014年2月。

[36] Sarabjeet Singh 和 Manu Awasthi。SPEC CPU2017 套件的以内存为中心的特征描述与分析。在 *2019*年 *ACM/SPEC* 国际性能工程会议论文集，ICPE '19，第 285–292 页，美国纽约州纽约市，2019年。计算机协会。[37] Emil Stefanov、Marten van Dijk、Elaine Shi、Christopher Fletcher、Ling Ren、Xiangyao Yu 和 Srinivas Devadas。Path ORAM：一种极其简单的不经意随机存取存储器协议。在 *2013*年 *ACM SIGSAC*计算机与通信安全会议论文集，CCS '13，第 299–310 页，美国纽约州纽约市，2013年。计算机协会。[38] Meysam Taassori、Rajeev Balasubramonian、Siddhartha Chhabra、Alaa R. Alameldeen、Manjula Peddireddy、Rajat Agarwal 和 Ryan Stutsman。面向完整性与可靠性的紧凑型无泄漏支持。在 *ACM/IEEE* 第*47*届年度计算机体系结构国际研讨会论文集，ISCA '20，第 735–748 页。IEEE 出版社，2020年。[39] C. Xu、D. Niu、N. Muralimanohar、R. Balasubramonian、T. Zhang、S. Yu 和 Y. Xie。克服交叉开关电阻式存储器架构的挑战。在 *2015*年第*21*届 *IEEE* 国际高性能计算机体系结构研讨会（*HPCA*），第 476–488 页，2015年。[40] Y. Xu、W. Cui 和 M. Peinado。受控通道攻击：针对不可信操作系统的确定性侧信道。在 *2015*年 *IEEE* 安全与隐私研讨会，第 640–656 页，2015年。[41] Danfeng Zhang、Aslan Askarov 和 Andrew C. Myers。交互式系统中时间侧信道的预测性缓解。在 第*18*届 *ACM* 计算机与通信安全会议论文集，CCS '11，第 563–574 页，美国纽约州纽约市，2011年。计算机协会。[42] Xian Zhang、Guangyu Sun、Chao Zhang、Weiqi Zhang、Yun Liang、Tao Wang、Yiran Chen 和 Jia Di。Fork Path：通过移除冗余内存访问提升 ORAM 效率。在 第*48*届国际微体系结构研讨会论文集，MICRO-48，第 102–114 页，美国纽约州纽约市，2015年。计算机协会。[43] Y. Zhang 和 S. Swanson。使用非易失性主存时的应用程序性能研究。在 *2015*年第*31*届大容量存储系统与技术研讨会（*MSST*），第 1–10 页，2015年5月。