Bypass and Insertion Algorithms for Exclusive Last-level Caches (LLC)

Abstract

Inclusive LLC 由于跨级复制而浪费了宝贵的硅空间。随着行业向内层更大的高速缓存层次结构发展，与 Exclusive LLC 相比，这种高速缓存空间的浪费会导致更大的性能损失。不过，Exclusive LLC 使替换策略的设计更具挑战性。在 Inclusive LLC 中，数据块可以收集经过过滤的访问历史记录，而在 Exclusive 设计中，由于数据块在命中时会从 LLC 中取消分配，因此不可能做到这一点。因此，流行的 LRU 替换策略及其近似方法就失效了，而且在 Exclsive 设计中，正确选择 cache block 的 insertion ages 变得更加重要。另一方面，也没有必要将每个块都填入 exclusive LLC。这被称为选择性 cache bypassing，不可能在 inclusive LLC 中实现，因为这违反了 inclusive 原则。

本文探讨了 Exclusive LLC 的 insertion 和 bypass 算法。我们详细的 execution-driven simulation 结果表明，在经过精心调试的 multi-stream 硬件预取器存在的情况下，在运行于 2 MB/16 路 Exclusive LLC 上的 97 个单线程动态指令跟踪中，与 baseline exclusive 设计相比，我们的最佳 insertion 和 bypass 策略的组合在 IPC 方面提高了最高 61.2% 和平均 3.4%，这些动态指令跟踪包含选定的 SPEC 2006 和服务器应用程序。使用 8 MB /16 路共享 exclusive LLC 运行的 35 个 4 路多程序 workloads 的吞吐量相应提高了 20.6%（最大值）和 2.5%（平均）。

1 Introduction

Inclusive LLC 简化了缓存一致性协议，LLC tag 查询足以决定 cache block 是否存在于 cache 中。然而，在 exclusive LLC 中，只有从上层 cache 中剔除时才会分配一个块，当上层 cache 调用该块时，会在命中时取消分配。因此，需要一个单独的一致性目录阵列（与 LLC tag 阵列分离）来有效地保持一致性。虽然一致性简化和可以从上层 cache 无声干净地剔除被认为是 inclusive LLC 的主要优势，但从定义上讲，这种设计会浪费硅空间，因为缓存数据会在层次结构的多个层级中复制。随着业界向具有相当大内层的三级或四级高速缓存层次结构发展，与 exclusive 设计相比，这种跨层复制开始损害 inclusive 设计的性能。这一现象已经

促使设计人员采用完全或部分 exclusive 的 LLC。

与相同的 inclusive 设计相比，exclusive 设计的性能提升通常来自两个因素。其一是 exclusive 设计所享有的总体容量优势。其二与 inclusive LLC 替换策略导致的上层 cache 块的过早驱逐有关。在没有来自 L1 和 L2 缓存的访问提示的情况下，inclusive 设计的最后一级可能最终做出错误的替换决定。而在 exclusive 设计中，则不存在因 LLC 替换而从 L1 和 L2 缓存中提前驱逐的风险。

图 1 显示了在代表不同区域的浮点 SPEC 2006 (FSPEC)、整数 SPEC 2006 (ISPEC) 和服务器 (SERVER) 应用程序的 97 个单线程动态指令跟踪中，exclusive LLC 相对于 inclusive LLC 的性能，并启用了经过良好调试的 multi-stream 硬件预取器。

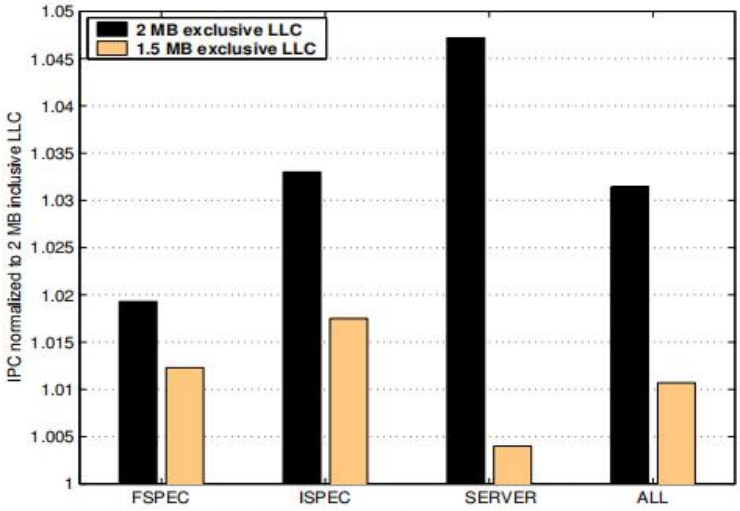


Figure 1: A comparison of IPC between exclusive and inclusive LLCs with a 512 KB L2 cache in each design.

在左侧 bar 中，包含型和独占型情况下模拟的三级高速缓存层次结构在每一级的容量和关联性上都是相同的。更具体地说，左侧条形图显示的是具有 512 KB/8 路二级缓存和 2 MB /16 路 LLC 的架构的模拟结果。右侧条形图显示了 exclusive LLC 相对于 inclusive 设计的性能，其中 exclusive LLC 的大小（1.5 MB/12 路）使得 exclusive 设计的有效容量优势被抵消。在这两种情况下，inclusive LLC 模拟的是 NRU 替换策略（每个 cache block 1bit 的 age），而 exclusive LLC 模拟的是 NRF 替换策略。NRU 策略会从有最小 id 的 way 中替换最近未使用的区块（age 为 0）。NRF 策略只在 fill 时更新 age array，其他方面和 NRU 相似。当一个 set 中的所有 block 的 age bits 都为 1 时，两种策略都将把这个 set 中的所有 block 的 age bit 重置为 0（除了最近访问/填充的块）。对于每个应用类别，图 1 中右侧的条形图

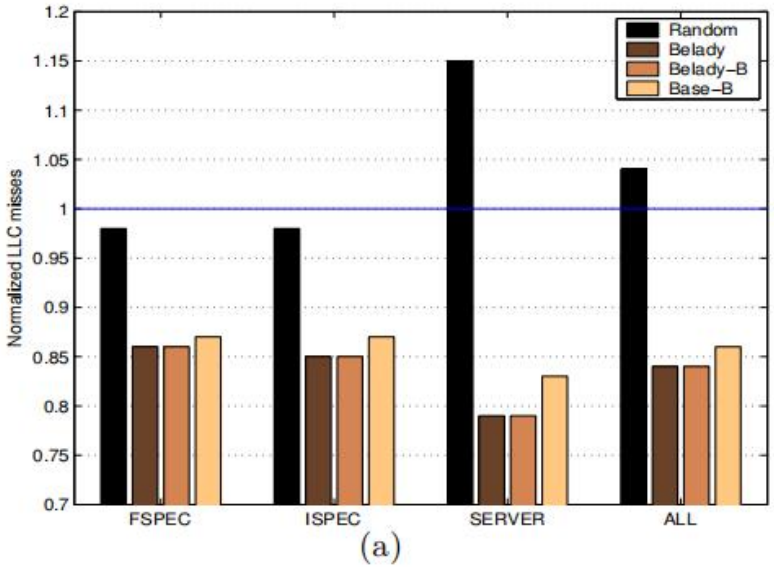
显示了在 inclusive 设计中，由于过早从 cache 的内层剔除 block 而造成的性能差异。左侧的 bar 进一步增加了 exclusive 设计所享有的容量优势。在 97 个 traces 中，exclusive 设计的 IPC 平均比 inclusive 设计高 3.1%。

虽然过早的 LLC 替换在 inclusive 设计中会导致性能下降，但在 LLC 中的 block 在其生命周期内仍然可以观察到 filtered version of access recency。通过来自上层 cache 或其他主动机制的 access hints，情况可以得到进一步的改善。这在 exclusive 设计中是不可能的。一个 block 存在于 exclusive LLC 中，从它被 L2 驱逐到它被 L2 调用或被 L3 驱逐。由于 exclusive LLC 缺乏任何访问信息，因此 LRU 及其衍生算法都失去了意义。因此，exclusive LLC 的替换算法需要一个新的思路。替换策略有 3 个组成部分，insertion age 算法，age update 算法和 victim selection 算法。这篇论文讨论 insertion age 算法。

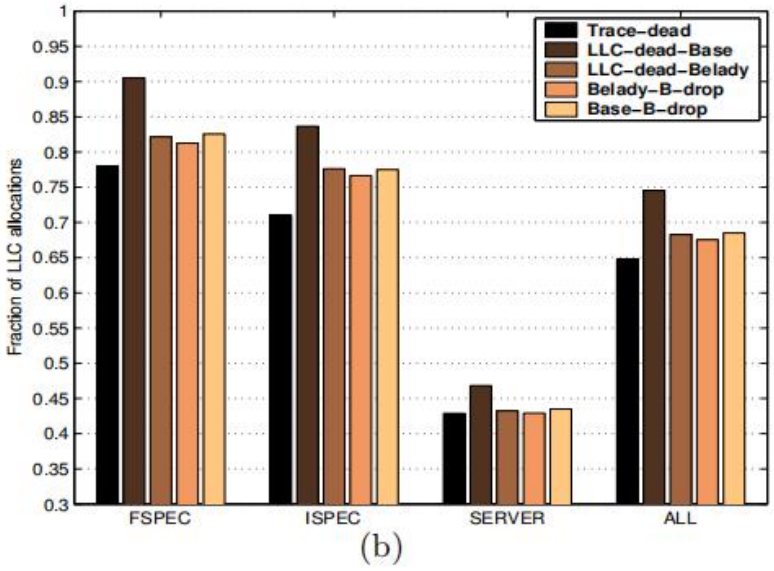
选择性 bypass 是一个可以被用于 exclusive design 的重要优化，因为每一个被 L2 驱逐的 block 都不需要填进 LLC。而在 inclusive design 中，所有从内存获取的 block 必须填入 LLC。本文探讨了针对 exclusive design 的 LLC bypass 算法，这种算法可以识别不需要填入 LLC 的 clean 或 dirty blocks。好的 LLC bypass 算法可以在两个方面提高性能，一方面是减小芯片互连和 LLC 控制器的带宽需求，另一方面是通过将 LLC 的容量只分配给重用距离相对较短的 blocks。

1.1 Motivation

图 2 激发了我们在本章中探讨的两个设计问题，即 exclusive LLC 中的 bypass 和 insertion 算法。在这个图中，我们考虑了一个具有 NRF 替换策略的 baseline exclusive LLC，并且没有 bypass（即所有的 L2 evictions 都在 LLC 中被分配）。我们关闭了预取器，以便更好地理解需要的请求行为（我们将在第 5 节中展示打开预取器的结果）。实验是在一个拥有 2MB/16 路 exclusive LLC 的单核系统上进行的。L2 为 512KB/8 路，L1I 为 16KB/4 路，L1D 为 32KB/8 路。L1 和 L2 利用伪 LRU 替换策略。



(a)



(b)

Figure 2: (a) Normalized LLC misses for random replacement and a number of oracle-assisted replacement policies. (b) Dead allocation and bypass analysis for a number of oracle-assisted bypass policies.

图 2(a)显示了一些策略的与 baseline NRF 策略归一化后的 LLC miss 数的对比，即：

（1）被用于 victim caches 的没有 bypass 的 random 替换策略（Random）；

（2）没有 bypass 的 Belady's optimal longest-forward-distance 替换策略（Belady）；

（3）被扩展的具有 bypass（drop 一个块，当它的 next forward use distance 在目标的 LLC set 中大于其他所有块）的 Belady's optimal 替换策略（Belady-B）；

（4）被扩展的具有 bypass（drop 一个块，当它的 next forward use distance 在目标的 LLC set 中比当前的 victim 更大）的 baseline 策略（Base-B）。

本篇论文中，我们不会 bypass instruction block。这些实验都是在一个离线 cache 模拟器上进行的，它可以访问整个 LLC 的 allocation/lookup trace。预热结束后，cache 状态从 checkpoint 加载。具有未知 forward distance（next potential use 超出了 trace length）的 block 之间的联系会被任意打断。

这些结果表明，相比 baseline 策略，random 的 LLC miss 率高 4%。但 Belady，Belady-B，Base-B 展现出了很大的提升潜力。平均来说，这 3 种方案分别降低了 16%，16%和 14%的 LLC miss 率。有趣的是，Belady 和 Belady-B 同样有效，意味着在一个已经很好的替换策略中加入 bypass 并不会带来对 LLC 命中率的额外提升。在这种情况下，即使 bypass 策略不能提升 LLC 的命中数量，如果可以在 L2 边界上实现 bypass 方案，可以有效节省片上互连带宽。然而，值得鼓舞的是，使用带前瞻性 bypass 的 NRF 策略（Base-B）也能减少 14%的 LLC miss 率。正如预期的那样，随着替换策略变差，好的 bypass 策略对命中率提升的潜力会增加。然而，我们注意到 Base-B 惊人地接近 Belady 和 Belady-B，这个差距在 server workloads 中被最大化。我们将通过图 2(b)来解释这一点。

图 2(b)显示了 exclusive 设计中与 bypass 潜力有关的附加数据。对于每个 trace 类别，图 2(b)显示了 5 个不同的统计数据：

（1）在 LLC 中分配但在剩余的 trace 中没有被再次使用的 block 的数量占所有分配的比例（Trace-dead）；

（2）在 LLC 中分配但在被 baseline LLC 替换策略驱逐之前都没有被 L2 调用的块的比例（LLC-dead-Base）；

（3）当采用不带 bypass 的 optimal 替换策略时，在 LLC 中分配但在被 baseline LLC 驱逐之前都没有被 L2 调用的块的比例（LLC-dead-Belady）；

（4）由 Belady-B 进行 bypass 的 L2 驱逐的 cache 块比例（(Belady-B-drop）；

（5）由 Base-B 进行 bypass 的 L2 驱逐的 cache 块比例（(Base-B-drop）。

其中，第一个 bar 显示了在一个无限容量的 LLC 中，无用分配所占的比例。虽然这部分是和 trace 长度直接相关的函数（更长的 traces 的话可能会占更小的比例），但是这一数据强调了 L1 和 L2 很好地吸收了所有短期重用的 blocks，而

且大多数重用群都离得非常远。第二个 bar 显示了在 baseline LLC 中，无用的分配所占的比例，平均占 75%。这一比例是 bypass 潜力的真实表现。Bypass 算法应该尝试在 LLC 中腾出更大的空间，以便保留导致前两个 bar 有差异的 blocks 的子集。第三个 bar 显示，即使是最优的替换策略，在 LLC 中也会有 68%的无用块。这个结果表明了一个好的 bypass 策略的重要性，即使替换策略是最优的。第四个 bar 更进一步证明了这一结果，它显示了最优替换策略和最优 bypass 策略所观察到的等量 bypass。这一结果强调了在 L2 缓存边界上运行的 LLC bypass 策略可以实现的节省模上互连带宽的巨大潜力。最后，最右边的 bar 表明，就被 bypass 的 blocks 比例而言，加上 forward-looking bypass 策略的 baseline 替换策略和共同运行的 optimal bypass and replacement 同样有效。我们已经注意到，对于 server workloads 而言，Base-B 和 Belady-B 的性能差距最大。这是意料之中的，因为在这些 workloads 中，bypass 所占的比例是最小的，这将导致调用 Base-B 的次优 NRF 替换策略。

考虑到替换和 bypass 算法的巨大潜力，本论文将系统地推导和实现一些这样的算法（Sections 2 and 3）。在本研究中，我们只讨论替换算法的 insertion component。即使我们的 bypass 算法可以和 L2 无缝集成，但我们在 LLC 控制器中实现它，只研究这些算法的容量增益。我们详细的执行驱动的仿真结果（Sections 4 and 5）表明，与启用了激进的 multi-stream 硬件预取器的 baseline exclusive design 相比，我们的 best insertion and bypass 算法，可以将 2 MB/16 way exclusive LLC 的 97 个单线程 traces 的 IPC 提高最高 61.2%，平均 3.4%。相应的，在 35 个 4-way 多编程的 workloads 上的吞吐率最大提高 20.6%，平均提高 2.5%。

1.2 Related Work

在这个 section，我们将简略地回顾和我们的工作最相关的研究。[29]已经研究了 Exclusive LLC 相对于 Inclusive LLC 的容量优势，[14]中提出了几种在 inclusive 结构中避免过早驱逐 cache 块的方法。[24]强调了在 exclusive LLC 中处理共享 cache 块的重要性。[7]研究了分布式磁盘缓存上下文中的多级 exclusive 缓存。

Exclusive cache 在功能上等价于一个大型的 victim cache[18]。选择性 victim cache（类似于一个带 bypass 的 exclusive LLC）已经被研究过，在与 L1 cache work

well 的 small victim cache 中[5,11]。一个基于 miss 频率的带有选择性 insertion 的 large victim cache 的设计在[2]中实现，并且在 inclusive LLC 中 work well。一项最近的工作在 inclusive LLC 中利用 dead blocks 配置“embedded”victim cache[21]。

Dead block 预测方案[11,19,21,22,23,25]与我们的 bypass proposal 密切相关。大多数现存的 dead block 预测方案在 cache set 中选择一个 dead block，要么用于替换，要么在这个块在 cache 中停留一段时间后，作为预取的目标。这些方案通常将指令地址和/或数据地址与 cache 块的死亡联系起来。最近的一项 proposal 表明了怎样设计一个不含地址的 DBP（dead block predictor），它利用了 reuse probabilities 来在 inclusive LLC 中 improve replacement decisions[4]。另一方面，我们的 bypass 算法在填充时将会在 LLC 中识别出一个 dead-on-fill 的块。虽然指令或数据地址可以提高 bypass 的质量，但我们的 bypass 算法并不依赖于任何这样的信息。

一些最近的 proposals 研究了为 LLC 设计的 bypass 算法。一个 proposal[6]记住 bypass 块（如果进来的这个块是 bypass 的）或者 victimized 块（如果进来的这个块不是 bypass 的）的 tag，来观察对 bypassed/allocated block 和 saved/victimized block 的下一次使用的情况，并据此了解 bypass 是否是一个好的 decision。然而，这个 proposal 基于一个 bypass probability 随机选择要进来的 blocks 用于 bypass，这个 bypass probability 根据 bypass 的效果进行动态调整。另一个 proposal[19]表明了怎样利用一个基于 PC 的在 LLC 中学习一些 sample sets 的 caching 行为的 skewed dead block predictor 去识别 dead-on-fill 的块，并将这样的块 bypass 掉。[22]中探讨了访问 counter-based LLC bypass 算法，它帮助了由 PC 的哈希函数索引的预测表。我们的 bypass 算法不需要单独的预测表，也不需要任何 PC 相关信息，但利用一个块在 L2 中看到的 reuse 频率，以及它在 L2 和 exclusive LLC 之间移动的次数。需要注意的是，bypass 算法也在小数据 cache 的上下文中进行了研究。这些 proposals 需要收集位置信息的 profile 传递[17]，或者构建一个 PC 和数据地址相关的位置预测器[8]，或者对 bypass[28]的 data cache 的潜力进行基于指令的描述，或者将 data cache miss 的情况分为 capacity 或 conflict，用于驱动 bypass decision[5]。我们的 proposal 不需要任何 profile 运行或任何 PC/address/instruction 信息或 miss 分类。

最近有几项关于 LLC 的 insertion age selection 研究。其中一些需要一个待填充块的源指令的 PC 信息[9,13]。此外，一些研究通常利用 LRU/MRU 或其他在一个 set 中的 access recency positions 去确定 insertion age[9,13,16,20,27]。因此，这些 proposals 与 access recency order（访问时间顺序？）的概念相关联，这个在 exclusive 缓存中是不存在的（在没有任何额外信息的情况下，exclusive cache set 中的唯一有用顺序是填充顺序）。最近的一项研究在 inclusive LLC 中基于 re-reference 间隔预测来分配 insertion age 和在命中时更新 predicted age[15]。即使这样的 age 更新选项在 exclusive LLC 中是不存在的，我们将展示如何为 exclusive LLCs 设计类似的策略。在[20]中探索了一种基于决策树的与访问顺序有关的选择 insertion age 的技术。我们的 insertion 策略必须独立于访问顺序（access recency order）。

2 Characterization of Dead and Live LLC Blocks

利用 recency 和利用 frequency 是两种传统的判断一个块是 dead/live 的特性。Exclusive LLC 使这两种特性的有效利用变得很有挑战性，因为当上层 cache 第一次 recall 这个块时，它就会被从 LLC 中 de-allocated。因此，在 LLC set 的块中，唯一有意义的顺序是填充顺序。不幸的是，LLC set 中的 blocks 的填充顺序和 use recency order 几乎无关，因为从多个 L2 sets 中被逐出的块可能在同一个 LLC set 中被分配。尽管 L2 实现了基于使用近因顺序的替换算法（例如，在我们的例子中是伪 LRU），但 LLC set 中的填充顺序是来自不同 core 或同一个 core 的多个 L2 set 的使用近因顺序的任意交错。在 LLC set 中重建正确的使用近因顺序是代价高昂的，因为它需要在跨多个核的 L2 中建立一个全局的使用近因顺序。在本文中，我们设计了基于估算 LLC 块的平均 recall 距离及其在 L2 中的使用次数的 bypass 和 insertion 方案。一个 LLC block（B）的平均 recall 距离被定义为在 LLC 中 B 的分配和从 L2 中对 B 的 recall 的 LLC 分配的平均数量。

2.1 Estimate of Recall Distance

在具有 exclusive LLC 的三级缓存层次结构中，当第一次从 DRAM 引入块时，块被填充到 L2 中。当 L2 驱逐这个块时，它第一次访问 LLC（first trip）。如果它

在被 L2 驱逐之前就被 LLC recall，当它被 L2 再次驱逐时，最终将第二次访问 LLC（second trip）。这些 trips 一直持续到 LLC 驱逐这个块。Trip count 高的块被认为有低的平均 recall distance。Exclusive LLC 中，一个块的 trip count 翻译为在 inclusive LLC 中的 use count。

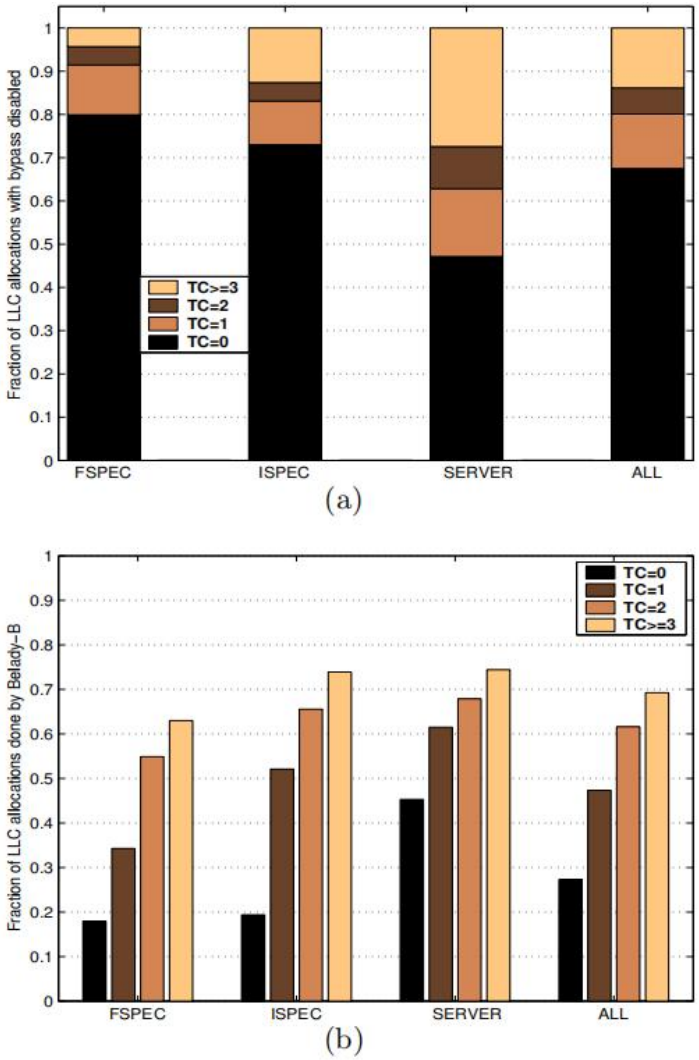


Figure 3: (a) Distribution of LLC allocations with trip count in the presence of optimal replacement with no bypass (Belady) in the LLC. (b) Fraction of LLC allocations in each trip count bin in the presence of optimal replacement with optimal bypass (Belady-B) in the LLC.

为了理解当最优 LLC 替换存在时的 trip count 行为，图 3(a)展示了 4 个 trip count（TC）的 LLC allocations 分布（从 L2 到 LLC 的 first trip 被设置为 TC=0）。因为每一个进入 cache hierarchy 的块在 LLC 中被第一次分配时 TC=0，所以只有其中的一小部分能够存活下来，经历 TC=1 的分配。TC=0 和 TC=1 分配之间的差值显示了在 TC=0 处发生的无用分配的比例。总体而言，约占 LLC 分配的 55%。我们

注意到，这构成了 68%的总体无用分配的很大部分（参见图 2(b)中的 LLC-dead-Belady）。将 TC=0 的块称为 TC_0，将其他的块称为 $TC_{\geqslant 1}$。

图3(b)进一步显示了在LLC中的最佳替换策略（Belady-B）之上启用最佳bypass时，在每个 TC bin 中发生的分配的比例。特定分区的这一比例是根据从该分区分配的 LLC 数量与属于该分区的进入区块数量之比计算得出的。注意，属于 TC=k bin 的传入块必然是从 TC=k-1bin 分配的块的子集（剩余的子集在下一次 trip 之前从 LLC 驱逐）。这些数据表明，总的来说，只有 27%的 TC_0 块被分配在 LLC 中，其余的 73%的 TC_0 块被 bypass。随着一个块移动到更高的 TC bin，分配百分比逐渐增加。这些数据清楚地表明了这样一个事实，即 LLC block 是 live 的概率随着其 TC 值的增加（有上限）而增加。我们从这些数据中得出了三个主要的结论。第一，TC=0 bin 是理想的 bypass 目标，$TC_{\geqslant 1}$ 块应该主要分配在 LLC 中。然而，把死的 TC_0 块从活的 TC_0 块中分离出来是很重要的（这些活的 TC_0 块最终将成为 $TC_{\geqslant 1}$ 块）。在下一个 section 中，我们将探讨 L2 use count 作为实现这种分类的可能特征。第二，由于 $TC_{\geqslant 1}$ 块大多是 live 的，如果它们不被 bypass，它们可以被分配一个较高的 insertion age。然而，我们需要更多的非 bypass 的 TC_0 块的属性来适当地对它们的 insertion age 进行分级。出于这一原因，我们将在下一节中探讨 L2 use count。第三，2 个 TC bin，即 TC_0 和 $TC_{\geqslant 1}$，已经足够获得大部分收益了。因此，每个 L2 block 需要一个 bit，LLC block 不储存 trip count（一个被 LLC recall 的 block 总是在 L2 中被分类为 $TC_{\geqslant 1}$）。更多的 TC bits 可能有助于 server traces，但是这些 bits 的总体利用率会低。

2.2 Use Count and Synergy with Trip Count

在上一个 section 中，我们已经发现，trip count 是一个很好的在 exclusive LLC 中用于识别大部分 bypass candidates（TC_0 块的主要部分）和 live blocks 的 starting point。接下来，我们将探讨一种可能性，即利用一个块在 L2 停留期间的 use count 来更进一步地调整 dead 和 live blocks 的分类。每次当一个 block 被 demand 请求填入 L2（从 DRAM 或 LLC），它的 use count 被设置为 1。由 prefetch 请求填入 L2 的块的 use count 被设置为 0。只有 L2 中的 demand hit 才会使 use count 增加。虽然 trip count 与近期重用的块的 clusters 之间的平均距离松散相关，但 L2 use

count 捕获一个块看到的最后一个集群的大小。在图 4(a)中，我们首先探索 LLC 分配在三个 L2 use count（L2UC） bin 中的分布。LLC 使用 Belady's optimal replacement policy，没有 bypass。请注意，在没有预取的情况下，从 L2 中驱逐出的块的 L2UC 不能为 0。即使存在预取的情况下，这样的块也可能只是由于过早或不正确的预取而存在。在第 3.4 节中，我们将讨论在这些块进行 LLC 分配时如何处理它们。图 4(a)中的数据显示，在 LLC 中分配的大约 58%的块只在 L2 中观察到一次使用。下一个 L2UC bin 贡献了约 30%的 LLC 分配。其余的分配至少有三个 L2UC。虽然这些数据没有提供任何关于 dead 和 live 分类的 insight，但它们确实证实了每个 L2 block 维护 2-bit 的 L2UC 对于所有实际应用都是足够的。认识到 L2UC 只是一个 filtered（或 sampled）access count，我们还查看了 L1 中的 L1 cumulative use count（CUC），它是从 block 被带入 L1 时开始计数，直到被从 L2 中驱逐（这对应于每个 L2-LLC trip 的 L1 use count）。我们发现每个缓存块需要 4 位来正确地维护 CUC。接下来，我们将探讨在 CUC 中增加的准确性（与 L2UC 相比）是否有助于提升 dead 和 live blocks 的分类。

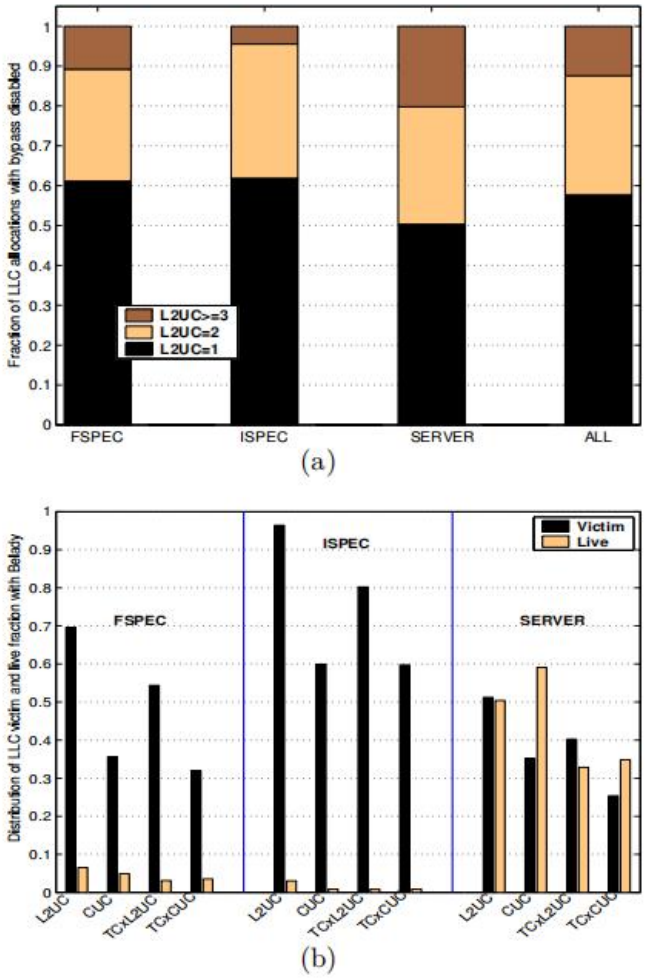


Figure 4: (a) Distribution of LLC allocations with L2 cache use count. (b) Median of victim and live block fractions in the most prominent victim bin for four bin classes. The victim fraction is the victim count of the most prominent victim bin out of all LLC victims across all bins, while the live fraction is computed over the LLC allocations done from the most prominent victim bin only. The data for (a) and (b) are collected in the presence of optimal replacement with no bypass (Belady) in the LLC.

设计好的 bypass 和 insertion age 分配算法需要知道在一个最佳设置下的 dead/live blocks 分布。探索这一点的一种方法是在 LLC 执行 Belady's optimal replacement 时观察 victims 的分布，因为 optimal victim selection 与 optimal deadth prediction 是同义的。为了获取好的 LLC victims 分布，我们在 LLC 中执行了 Belady's optimal replacement 且不带 bypass，同时维护 3 个 L2UC bins（L2UC=0 被排除），15 个 CUC bins（CUC=0 被排除），以及 TC bins 和 L2UC bins/CUC bins 的交叉乘积（即，6 个 TC×L2UC bins 和 30 个 TC×CUC bins）。这就创建了 4 个 bin 类，即

L2UC，CUC，TC×L2UC，和 TC×CUC。我们想知道这些 bin 类中哪一个可以作为识别 dead 和 live blocks 的好特性。解决这个问题的一种方法是，在每个 bin 类中，识别出 victims 数量最多的 bin。显然，这样的 bin 将会捕捉到最多的 optimal victims。然而，我们希望这个 bin 中包含的 live 块很少，从而减小伤害 live 块的概率。目标是识别一个 bin 类，它的 victim coverage 最大而 live coverage 最小。我们下面来实现这个。

当一个块被分配到 LLC 时，它在这四个 bin 类中的 membership bin 是根据它的 TC、L2UC 和 CUC 值来决定的。举例来说，带有 TC=0、L2UC=2 和 CUC=10 的块将落入 L2UC=2 bin、CUC=10 bin、TC×L2UC=（0,2）bin 和 TC×CUC=（0,10）bin。当一个 block 被 optimal replacement policy 从 LLC 中驱逐时，该 block 的四个 membership bin 的 victim counts 分别增加 1。对于每一个 trace，我们在每个 four bin classes C（C∈{L2UC、CUC、TC×L2UC、TC×CUC}）中识别出了覆盖最大 victims 比例的 bin（V^C_{max}）。对于不同的 traces，这四个已识别的 bins（每个 bin 类中有一个）可能是不同的。对于这四个已识别的 bin 中的每一个（每个 bin class C 中都有一个），我们还记录了 live 比例（L^C），计算为属于一个 bin 的块所经历的 LLC 命中的数量和从该 bin 中完成的 LLC 分配的数量之比。

图 4(b)显示了每个 bin class C 的三个 trace 类别的 V^C_{max} 和 L^C 的中位数。Victim 比例在 L2UC 最高，其次是 TCxL2UC，同时 live 比例在 TCxL2UC 中最小，尤其是 server traces。在 server traces 中，L2UC 和 TCxL2UC 之间的 live 比例差距远远大于 victim 比例的差距。假设最小化 live 比例与最大化 victim 的覆盖范围同样重要，我们决定使用 TCxL2UC bin 来推断 bypass candidates 和 insertion age。我们将这些 bins 称为 TC-UC bins，将 L2UC 称为 UC。令人鼓舞的是，high median victim 比例在 TC×L2UC 跨应用程序类别本质上表明每个 trace 至少存在一个 TC-UC bin，这样 dead block 对此有很强的关联性（membership 的可能性：FSPEC 为 0.55，ISPEC 0.80，SERVER 为 0.40）。此外，在 FSPEC（0.03）和 ISPEC（小于 0.01）中，将属于这样的 bin 的 live block 错误分类为 dead block 的可能性也很小，而对于 SERVER 来说，它约为三分之一。我们的算法试图动态地学习这个 bin 和任何其他 prominent dead bins。尽管图 4(b)中显示的统计数据总结了每个 trace 的聚合观察行为，但需要注意的是，在相同的应用程序 trace 中，prominent dead bins 可能

会随着时间的推移而变化。

3 Bypass and Insertion Policies

本节讨论了 exclusive LLC 的 bypass 和 insertion 算法的设计和实现。首先，讨论了我们所有的算法所使用的动态学习框架，然后给出了这些算法。

3.1 General Framework

Bypass 和 insertion 决策应该基于 TC-UC bin 中的 dead block 和 live block 的数量。请注意，在 exclusive LLC 中分配的块如果在被 L2 recall 之前就被驱逐，则被分类为 dead；否则该块将被归类为 live。理想情况下，我们希望了解每个 TC-UC bin 中的 dead and live populations。根据一个进入的 block 的 membership bin 和该 bin 的 dead 和 live populations，我们要决定是否 bypass 这个 block，或者如果不 bypass，应该分配什么 initial age。为了进行这一学习，我们每 1024 个 LLC sets 提供 16 个样本 sets，观察每个 TC-UC bin 的 dead and live populations。这些 sets 将被称为 observers。observers 分配所有块，并基于传入块的 1-bit TC 值实现静态 insertion age 分配方案。我们将在第 3.3 节中介绍这个 age 分配方案。

对于每个 LLC bank 的每个 TC-UC bin，observers 维护两个值，即对 observers 的 dead 和 live 分配之差（D−L）和对 observers 的 live 分配（L）。我们的算法需要为每个 LLC bank 配置 8 个 D−L 和 8 个 L 计数器，对应于 8 个 TC-UC bins。当一个块到达 LLC 并分配给其中一个 observer 时，块的 TC-UC bin b 是根据块的 TC、UC 值（由从 L2 中的驱逐消息携带）决定的。当分配块时，observer 将 bin b 的 D−L 计数器增加 1。在命中 observer set 中的 block B 时，observer 将 B 所属的 bin 的 D−L 计数器减少 2，并将 bin 的 L 计数器增加 1。observer 为每个 block 维护 3 个 bits，以记住被分配的块所属的 bin。然而，非 obsever set 不需要存储任何此类信息。非 observer set 在分配块时，首先根据块的 TC、UC 值确定块的 membership bin，然后查询该 bin 的 D−L 和 L 计数器。返回的 D−L 和 L 值被输入到 bypass 和 insertion 算法中。

当更新 LLC bank 中的 D−L 和 L 计数器时，observer 还在该 LLC bank 中维护 TC-UC bin 的最大（D−L）、最小（D−L）、最大（L）和最小（L），不包括 UC=0 bin。

除此之外，所有 TC-UC bin 上的总 D−L（不包括 UC=0 bin），由每个 LLC bank 维护。我们将把它称为$\sum_{UC\neq 0}(D-L)$。我们的一种 insertion 算法要求 observer 在所有具有正 UC 的 TC=0 bin 上维护 aggregate L。我们将把它称为$\sum_{TC=0,UC\neq 0}(D-L)$。最大值、最小值和聚合值的更新大多发生在 LLC 活动的关键路径之外。每个 bank 的 N 个 LLC 分配的所有 D-L 和 L 计数器（包括最大值、最小值和总值）都减半，以便维护 temporally-aware 的指数平均值。N 等于每个 LLC bank 的 observer set 的数量乘以 LLC 的 associativity。即使每个计数器有两个字节的存储开销，整个计数器开销也很小。我们的 simulation 使用 8 位 D-L 和 L 计数器用于单线程配置，9 位计数器用于多编程配置。最大值、最小值和总和寄存器的大小都是相应的。此外，每个 L2 block 存储三个额外的 bit，以维护块的 TC 和 UC 值。

3.2 Bypass Algorithms

好的 bypass 算法可以 bypass 具有高的 D-L populations 和足够低的 L populations 的 bin 的输入块。更具体地说，如果$(D-L)_b \geq 1/2(\max(D-L)+\min(D-L))$且$L_b \leq 1/2(\max(L)+\min(L))$，则一个属于 TC-UC bin b 的拥有计数器值为$(D-L)_b$和L_b的输入 block 作为 bypass candidate。然而，我们发现在某些情况下，D−L 的整体大小很高，以至于即使第二种情况（足够低的 L）失败，也可以在没有任何性能下降的情况下进行 bypass。因此，如果$(D-L)_b \geq 3/4\sum_{UC\neq 0}(D-L)$，我们就覆盖了上面比较的结果。更仔细选择的低于 3/4 的权重可以进一步改善 bypass 性能。下面总结了我们的 bypass 算法，其中 bypass 是一个布尔值变量。

$$\begin{aligned} bypass \quad = \quad & ((D-L)_b \geq \frac{1}{2}(\max(D-L)+\min(D-L)) \\ & \text{AND } L_b \leq \frac{1}{2}(\max(L)+\min(L))) \\ \text{OR} \quad & ((D-L)_b \geq \frac{3}{4}\sum_{UC\neq 0}(D-L)) \end{aligned} \tag{1}$$

如果传入块在目标 set 中发现一个 invalid 的 way，并且根据上述公式 bypass 为真，则将其以 insertion age 为零填充到 LLC 中。换句话说，在 LLC 中，总是要利用 invalid way，但有 insertion age 为 0 的情况下，更推荐 bypass。另一方面，如果 bypass 为真，并且在目标 set 中没有 invalid way，则将 bypass 传入的块。被

bypass 的块的处理方式与 LLC victim 的处理方式完全相同，它模仿了 LLC 驱逐协议。

为了减少性能损失的风险，我们对 set samples[27]进行 dueling，并且总是将 obsevers 的 bypass 算法与 no-bypass 算法进行对决。为此，除了 observer sets 之外，我们还提供了相同数量的 LLC sets（每 1024 个 LLC sets 16 个），它们总是执行我们的 bypass 算法。我们已经观察到，不使用任何决斗的 bypass 策略的静态版本会降低几个应用程序的性能。

3.3 Insertion Algorithms

我们提出了三种复杂性逐渐增加的 insertion age 分配算法。这些算法被应用于那些由公式(1)计算出的 bypass 为假的块。我们假设有一个 2-bit 的预算来维持每个 LLC block 的 age。我们的算法只适用于数据块，指令块总是以最高的 age 填充，即 3。我们的 LLC 替换策略首先在目标 set 中寻找一个 invalid way。如果没有，它会使最小 age 的块成为 victim，并在插入新块之前用这个最小值减小这个 set 的所有 age，以反映正确的相对 age 顺序。通过选择具有最小物理路径 id 的块，打破了最小 age 的块之间的联系。

我们的第一个 insertion 算法的灵感来自于 TC_0 和 $TC_{\geq 1}$ 块中的 live 分布，如图 3(b)所示。该算法将所有 $TC_{\geqslant 1}$ 块的 insertion age 分配为 3，将所有 TC_0 块的 insertion age 分配为 1。这是我们的 observer set 所执行的策略，因为它不需要任何动态学习。我们将此策略称为 TC-AGE 策略。TC-AGE 策略类似于最初为 inclusive LLC 提出的 SRRIP 式静态算法[15]。在我们的 age 分配设置中，较低的 age 对应较高的替换优先级，SRRIP 算法在 inclusive LLC 中将一个 insertion age=1 分配给一个新分配的块，并当命中时，将其提升到可能的最高 age。在一个 exclusive LLC 中，已经看到 LLC hits 的是 $TC_{\geqslant 1}$ 块。

我们的第二个 insertion 算法继续分配最高 age，即 age=3 给 $TC_{\geq 1}$ 块，但它将更精细的分级 age 分配给 TC_0 块。要实现这一点，它需要 observer 了解到的 dead and live populations 的帮助。该算法认识到，属于低命中率的 bins 的 TC_0 块不应该得到一个正的 age。如果某个 bin b 满足 $D_b > xL_b$ 或等价的$(D-L)_b > (x-1)L_b$，则对于属于 bin b 的块，转化为上界为$\frac{1}{x+1}$的命中率（命中率是$\frac{L_b}{D_b+L_b}$）。如果一个传

入的块属于 TC=0 bin 且命中率很低，我们希望分配 insertion age=0。然而，我们发现在某些情况下，目标 bin 的命中率很低，但 bin 中仍然有相当多的 live blocks，即 L_b 高于一个阈值。在这些情况下，分配一个为 0 的 insertion age 的风险太大了。总的来说，我们将 insertion age=0 分配给 UC 为正的 bin b 的 TC_0 块，如果它满足

$$(D-L)_b > (x-1)L_b \text{ AND } L_b < \frac{3}{4}\sum_{TC=0,UC\neq 0}(L). \quad (2)$$

所有其余的 UC 为正的 TC_0 块的 insertion age 均为 1。我们将此策略称为 TC-UC-AGE 策略。我们对 x = 4,8 评估这个策略。

我们的第三个 insertion 算法类似于 TC-UC-AGE 策略，但它没有为所有不满足公式(2)的正 UC 的 TC_0 块分配 age=1，而是根据 live population 将它们从 age=1 到 age=3 进行分级。首先，该算法根据三个 TC=0/UC≠0 bins 的 L 值进行排序，并对 L 值最小且 age=1、L 值最大且 age=3 的 bin 进行标记。接下来，该算法确定传入块所属的 bin，并为该块分配相应的 insertion age。我们将把这一策略称为 TC-UC-RANK 策略。与 bypass 策略不同，我们的 insertion age 分配方案都不需要决斗，因为一个轻微错误的 insertion age 并不像一个错误的 bypass 决策那么有害。

3.4 Handling Prefetches

我们特别考虑了 UC=0 的 bins。正如我们所指出的，属于这些 bins 的块要么是过早但正确的预取结果，这些预取在 L2 中驻留期间未能看到 demand hit，要么是不正确的预取结果，在不久的将来不会看到需求命中。我们的 bypass 算法继续忽略这种情况，并且处理 UC=0 bins 的方式与处理其他 bins 完全相同。我们的 TC-AGE insertion 算法对 UC=0 块不做任何特殊的事情。另外两种 insertion 算法为属于 bin b 的一个 (TC=0,UC=0) 块分配 insertion age=0，如果它满足 $(D-L)_b > (x-1)L_b$（这里 b 是(TC=0,UC=0)）。所有其他(TC=0,UC=0)块的 insertion age 为 1。所有 (TC≥1,UC=0) 块的 insertion age=3。图 5 显示了我们的 bypass 和 TC-UC-AGE 逻辑图。

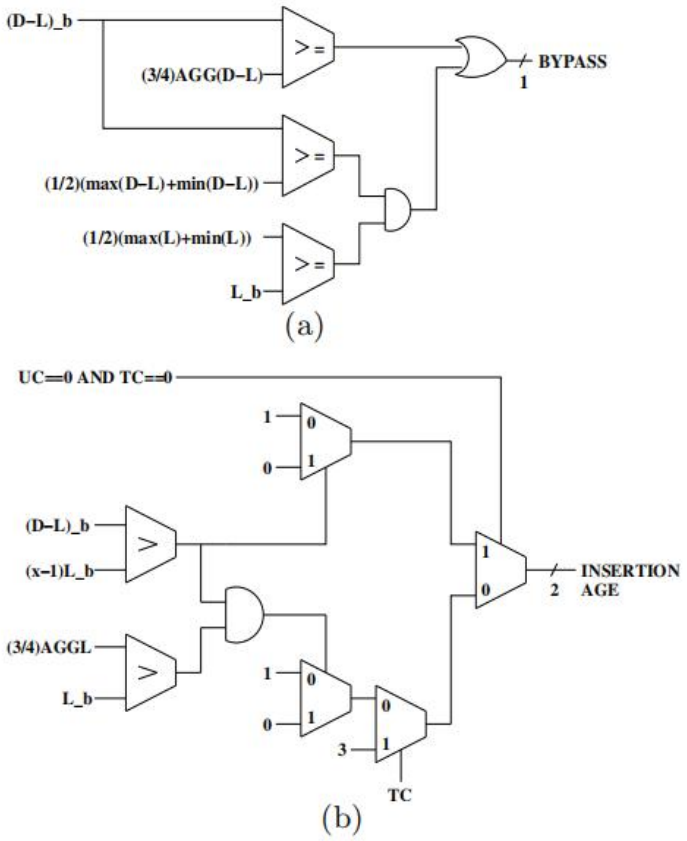


Figure 5: Logic diagrams for our (a) bypass and (b) TC-UC-AGE algorithms. $AGG(D-L)$ refers to $\sum_{UC\neq 0}(D-L)$ and $AGGL$ refers to $\sum_{TC=0,UC\neq 0}(L)$. Note that the existence of an invalid way in the target set can override the bypass decision and force an insertion with age zero.

3.5 Intrducing Thread-awareness

将我们的 bypass 和 insertion 算法升级到多线程环境需要为每个线程的每个 TC-UC bin 维护 D-L 和 L 计数器。每个线程也被分配了一组单独的 observers。被指定为一个特定线程的 observers 为这个线程执行 TC-AGE insertion 策略，如果启用了 bypass，则也为每一个其他线程执行 best emerging duel winner（与 TADIP-F[16] 类似）。我们每个线程中的每 1024 个 LLC sets 使用 4 个 observers。我们的计数器更新方案不需要在 LLC 中存储线程 id 来结合线程感知。我们在本文中假设每个 core 有一个线程。在 LLC 分配时，source L2 的 core id 是可用的，因为这些信息需要更新一致性目录，因此，可以增加适当的 D-L 计数器。在 LLC 命中时，请求者的 core id 是可用的，因此，适当的 D-L 计数器可以减少，而适当的 L 计数器

可以增加。此外，必须维护第 3.1 节中讨论的几个计数器的最大值、最小值和总和值。

4 Evaluation Methodology

我们的模拟是在一个 cycle-accurate 的 execution-driven 的 x86 模拟器上完成的。我们的 4GHz 4way 动态调节的乱序发射 core 模型紧密跟随 Intel Core i7 processor 的 core 微架构[12]。在整个研究过程中，我们假设每个 core 有一个物理线程上下文。每个 core 都有自己的 L1 和 L2 缓存。L1 指令缓存为 16KB/4 路关联，L1 数据缓存为 32KB/8 路关联。统一的 L2 缓存是 512kB/8 路关联的。L2 缓存相对于 L1 缓存是 non-inclusive 的，因为 L2 的 block 驱逐时总是查询 L1 缓存以获取最新的状态和数据，但是 L1 缓存可以选择保留块而不是无效化。对于单线程研究，我们将一个 2MB/16 路 exclusive LLC 划分为两个 bank，每个 bank 分别为 1MB/16 路。在多编程研究中，我们用私有 L1 和 L2 缓存对四个核建模，核心在一个环上连接。环的每个 core hop 都有一个共享的 2MB/16 路 exclusive LLC bank，从而导致一个总大小为 8MB/16 路的共享 LLC。在缓存层次结构的所有三个级别上的块大小都为 64 字节。我们为 L2 建模了 6 个周期的 hit latency（tag+data），为每个 LLC bank[10]建模了 8 个周期的 hit latency（tag+data）。ring hop 时间是一个周期。我们建立了一个一致性目录的模型，它可以容纳聚合的 L2 缓存 tag 数量的 8 倍，并且是 16 路关联的（与 LLC 相同）。一致性目录 bank 与 LLC bank 位于同一位置。对于所有的 simulations，我们建模了一个具有 core frequency 的双通道集成内存控制器，每个通道连接到一个 8 路存储的 DDR3-1866 DIMM。DRAM 部分（933 MHz）具有 64 位的突发长度和 10- 10-10 个接入周期参数。我们对每个 core 的激进的 multi-stream instruction 和数据预取器进行建模，它们将 blocks 带入 core 的 L2 缓存中。

我们的单线程 trace 从三个工作负载类别中提取，正如已经讨论过： FSPEC、ISPEC 和 SERVER。我们首先确定了 213 个具有代表性的动态代码区域，每个区域的长度接近 3000 万条动态指令，前面有数亿条 load/store 指令的 trace 来预热缓存。虽然大约 3000 万条动态指令的整个 trace 在详细的 cycle-accurate 计时模式下运行，但最后 600 万条指令用于测量 IPC 和其他性能指标。本文中评估的所有

策略都从预热 trace 开始执行，以确保详细的 cycle-accurate 测量阶段捕获一个稳态快照。在这 213 个区域中，我们选择了 97 个可能对非核优化敏感的区域（使用 baseline NRF，每千条指令至少有 5 次缺失）。在这 97 个 traces 中，我们有 44 个 FSPEC traces，跨越了 12 个应用，即 bwaves、cactusADM、dealII、GemsFDTD、lbm、leslie3d、milc、soplex、sphinx3、tonto、wrf、zeusmp。我们有 23 个 ISPEC traces，跨越 7 个应用程序：bzip2，gcc，gobmk，libquantum，mcf，omnetpp，xalancbmk。最后，我们从 SAP、SAS、SPECjbb、SPECweb2005、TPC-C、TPC-E 等应用程序中选择了 30 条 SERVER traces。

我们提供了 35 个 4 路多编程工作负载的结果，这是通过混合来自所有三个工作负载类别的四个具有代表性的单线程 traces 而准备好的。在一个混合过程中，每个线程在开始详细的性能模拟之前，首先执行它的预热区域。如果线程提前完成其性能模拟阶段，它将继续执行，以便我们能够正确地建模共享 LLC 的争用。当每个线程都完成了其性能模拟阶段时，该混合程序就会终止。

5 Simulation Results

5.1 Single-threaded Workloads

我们首先给出了在禁用硬件数据预取器时的仿真结果。图 6 总结了在三种单线程工作负载类别和总体(ALL)中归一化为 1 位 NRF 的几种策略的平均 IPC。

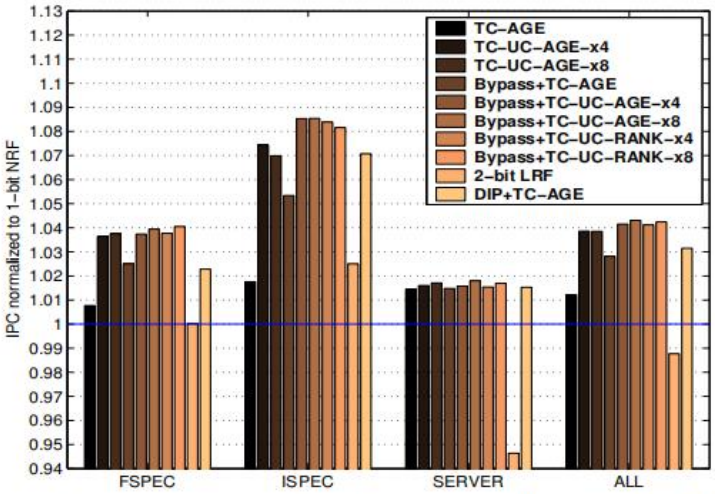


Figure 6: Summary of performance of several policies normalized to 1-bit NRF.

在每个类别中，最左边的三个 bar 显示了静态 TC-AGE insertion 和 x = 4,8 的基于

动态学习的TC-UC-AGE insertion的性能。为了避免不必要地增加策略bars的数量，我们将只在存在bypass的情况下显示TC-UC-RANK的性能。接下来的五个bars显示了LLC通过三种不同的insertion算法执行的性能。在这五种情况中，评估的策略（例如，Bypass+TC-UC-AGE-x8）总是与执行TC-AGE的observers对决，如果observers成为赢家，followers禁用bypass，但继续执行isnertion组件（例如，TC-UC-AGE-x8）。

我们还实验了一个2-bit近似的最近最少填充（LRF）的替换策略，该策略根据一个set中blocks的填充顺序（只能区分最后三个填充）对块进行排序。最后，每个工作负载类别中最右边的bar显示了在TC-AGE insertion存在情况下的动态插入策略（DIP）[27]的性能。该策略插入$TC_{\geq 1}$块时的age=3，并在插入TC_0块时对决insertion age=0和insertion age=1。该策略展示了一种实现DRRIP式动态策略 [15]的方法，它最初是为inclusive LLCs提出的。

TC-AGE策略在97个traces上的性能平均提高了1%以上（见ALL组）。这一结果促使我们对observers使用TC-AGE策略来代替NRF。TC-UC-AGE策略将整体性能提高了近4%，与NRF相比，ISPEC的平均性能提高了7%以上。总的来说，对于TC-UC-AGE，x = 4和x = 8之间没有性能差异。我们使用TC-AGE运行的bypass算法将整体性能提高了2.8%，其中ISPEC提高了令人印象深刻的5.3%。然而，这些数据表明，与bypass对决的TC-AGE 插入算法相比，单独的TC-UC-AGE插入算法可以获得更好的整体性能。尽管如此，bypass+TC-AGE策略仍然提供了一个有吸引力的设计点。LLC bypass加上x = 8的TC-UC-AGE提供了最好的性能。最好的组合，即Bypass+TC-UCAGE-x8将97条traces的总体IPC提高了4.3%，FSPEC、ISPEC和SERVER的个别改进分别为3.9%、8.5%和1.8%。相应地，它节省了7.6%、11.4%和8.4%的baseline LLC misses。LLC在SERVER类别中节省的IPC好处微不足道，因为这些工作负载在L1指令miss中浪费了很多周期（即使启用了指令预取器）。总的来说，当Bypass+TC-UC-AGE-x8策略作为L2缓存和LLC数据阵列存储的一部分计算时，需要不到0.5%的额外存储。表1总结了这一开销。

Table 1: Summary of overhead

State	Storage (bits)	Bits
TC and UC	3 per L2 cache block	24K
LLC age	2 per LLC block	64K
Bin identity	3 per obs. LLC block (16 obs. sets per 1024 LLC sets)	1.5K
16-entry obs. CAM (per 1024 LLC sets)	10 per CAM entry (partial set index)	320
TOTAL		89.8K

Bypass+TC-UC-RANK 的性能结果表明，添加基于 live populations 的 insertion age 排序机制并没有超过 x = 8 的 Bypass+ TC-UC-AGE 所提供的效果。事实上，在 ISPEC 类别中，ranking 机制会轻微影响性能，因为它不能区分 insertion age=3 的 TC_0 和 $TC_{\geq 1}$ 块。2-bit LRF 策略将 ISPEC 的性能提高了 2.5%，但 SERVER 工作负载降低了 5.4%。这一策略的主要缺点是，一个 block 的 age 在该 set 进行了 4 次填充后就会下降到 0，然后该 block 就有资格被驱逐。1 位 NRF 策略需要在将 block 的 age 重置为零之前有更高的预期填充数（参见第 1 节）。最后，DIP+TC-AGE 策略使整体 IPC 提高了 3.2%，而 ISPEC 提高了约 7%。接下来，我们将更详细地分析我们的最佳策略（Bypass+TC-UC-AGE-x8）的性能。

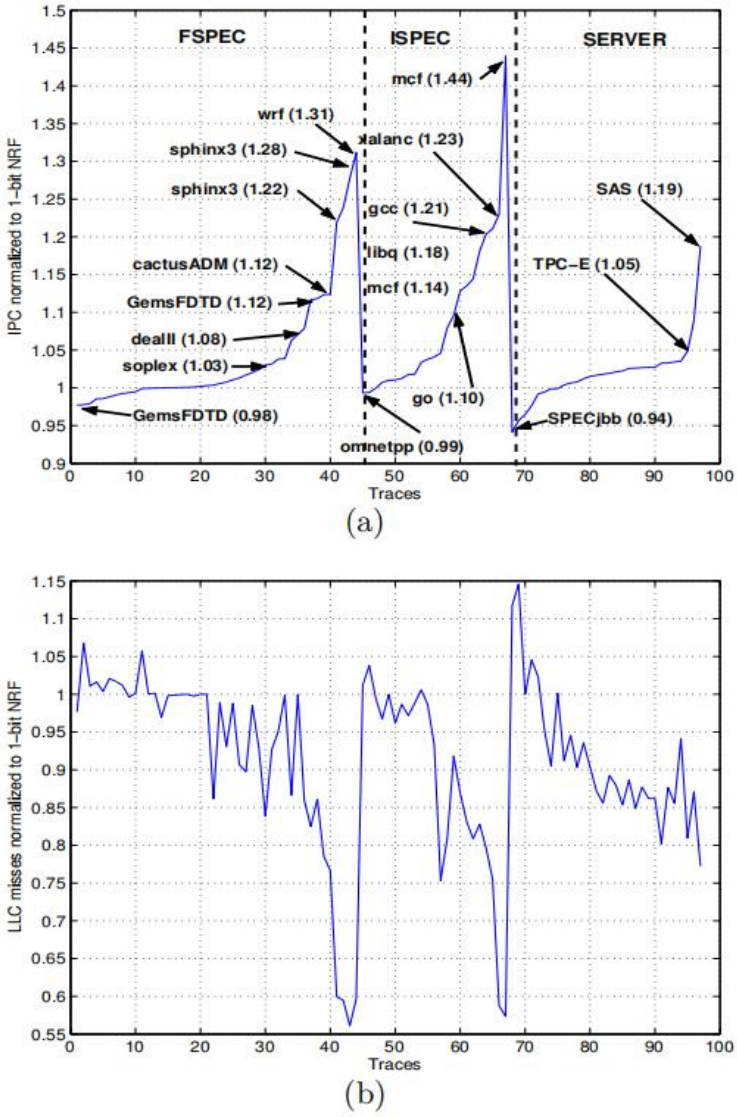


Figure 7: Distribution of (a) IPC improvements and (b) LLC misses of our best policy normalized to 1-bit NRF. We have shortened libquantum to libq.

图 7(a)和图 7(b)分别显示了与 baseline 1-bit NRF 相比，使用我们的最佳 LLC 策略(Bypass+TC-UC-AGE-x8)运行的 IPC 提升和归一化私有 traces 的 LLC misses 的详细信息。这三个类别中的每个 trace 都根据两条曲线中的 IPC 提升进行分类。其中一些 traces 也被标记在曲线上，括号内显示了它们的 IPC 提升。需要注意的是，同一应用程序的不同区域（例如 GemsFDTD）对我们的策略的反应非常不同，因此强调了需要模拟同一应用程序的多个区域。总的来说，FSPEC traces 显示性能最多提高了 31%，而性能损失最多为 2%。ISPEC traces 的 IPC 提升高达 44%，而最多损失 1%的性能。SERVER traces 显示 IPC 提升高达 19%，但也遭受高达 6%的性能损失（性能不佳的 SPECjbb trace 对 TC-UC-AGE 不友好）。LLC misses 的趋

势与 IPC 提升的趋势很吻合。

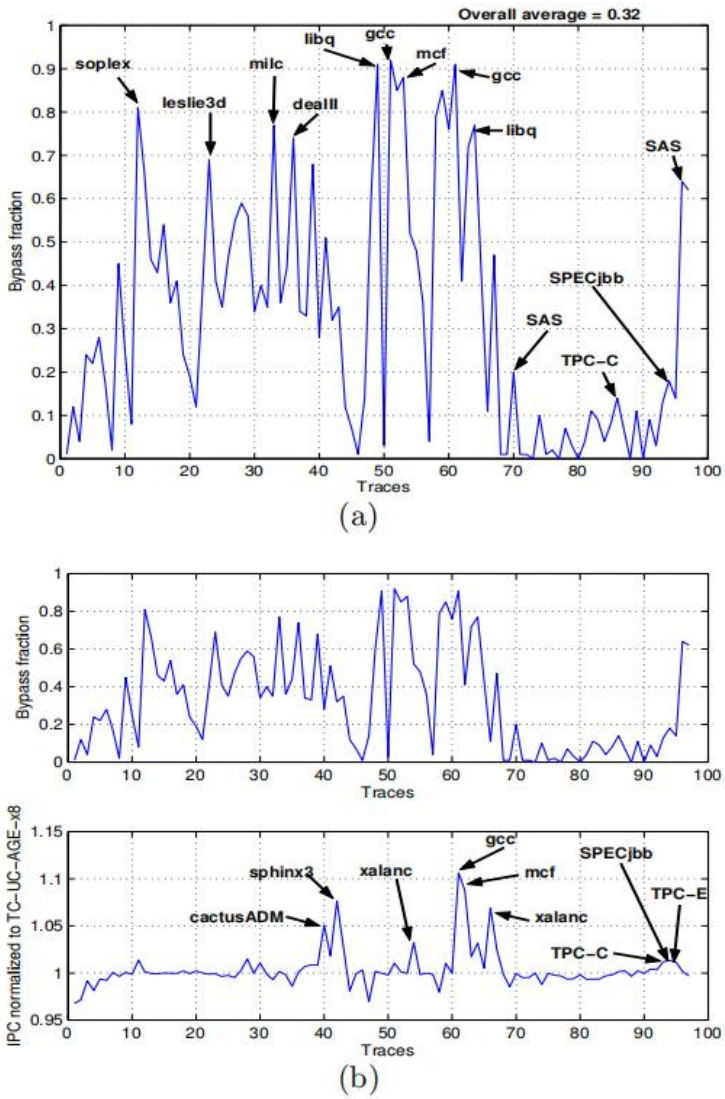


Figure 8: (a) Distribution of bypass fraction in our best policy. (b) Details of additional performance gains achieved by bypassing on top of TC-UC-AGE-x8.

接下来，我们量化了 LLC bypass 组件在我们的最佳策略（Bypass+TC-UC-AGE-x8）中的贡献。图 8(a)显示了对于每个 trace，在 LLC 分配时被 Bypass+TC-UC-UC-AGE-x8 策略 bypass 的 L2 缓存驱逐的比例。我们还识别了一些显示中等到高 bypass 比例的应用程序 traces。这些 traces 的分类与图 7(a)中显示的顺序完全相同。总的来说，在 97 条 traces 中，平均有 32%的 L2 缓存驱逐没有在 LLC 中分配。对于 FSPEC、ISPEC 和 SERVER 类别，bypass 百分比分别为 37%、52%和 11%。

为了进一步量化 LLC 在我们的最佳策略中 bypass 的性能影响，图 8(b)的下半部分显示了 Bypass+TC-UC-AGE-x8 相对于 TC-UC-AGE-x8 的 IPC，而上半部分再现

了 bypass 比例分布，以便于比较。一些从 LLC bypass 中获得明显收益的应用程序 traces 被标记在下半部分的图表上。很明显，就容量效益而言，SERVER 并没有从 LLC bypass 中获得太多的性能好处。然而，一些 FSPEC 和 ISPEC 的 traces 显示，由于 LLC bypass，IPC 有了显著的改善。高 bypass 比例并不一定会转化为性能改进，因为保留的块可能并不总是有足够小的重用距离，能够适应 LLC 的覆盖范围。尽管如此，如果我们的 bypass 方案是在 L2 缓存接口上实现的，那么我们令人印象深刻的 bypass 部分可以节省互连带宽，并导致进一步的性能改进。

图 9 显示了我们的最佳策略（bypass+TC-UC-AGE-x8）相对于 DIP+TC-AGE 策略的 IPC，并在曲线上标记了几个有趣的 trace points，以准确地显示我们的 gain 和 lose 的位置。这些 traces 的排序方式与图 7(a)中的方法完全相同。正如我们已经注意到的，我们看到同一应用程序的不同区域的行为不同（例如，GemsFDTD、libquantum、SAS）。总的来说，与 DIP+TC-AGE 相比，我们看到了几个 traces 显著增加，同时损失并不大。

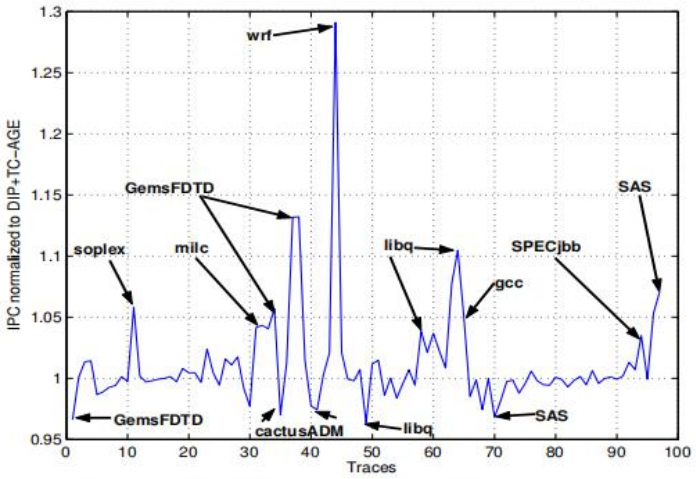


Figure 9: IPC of our best policy normalized to DIP+TC-AGE.

图 10(a) 和 10(b) 分别显示了针对 SPEC 2006 和 SERVER 工作负载的 bypass+TC-UC-AGE-x8 和 DIP+TC-AGE 之间的应用程序级比较。这些图表中所示的每个应用程序的归一化 IPC 图是通过取属于该应用程序的所有 traces 的归一化 IPC 的几何平均值来计算的。总的来说，对于 19 个 SPEC 2006 应用程序，我们的最佳策略比 1-bit NRF 的 IPC 提高了 5.4%，而对于 8 个 SERVER 应用程序，相应的改进是 1.9%。DIP+TC-AGE 所实现的改进分别为 4.1%和 1.1%。

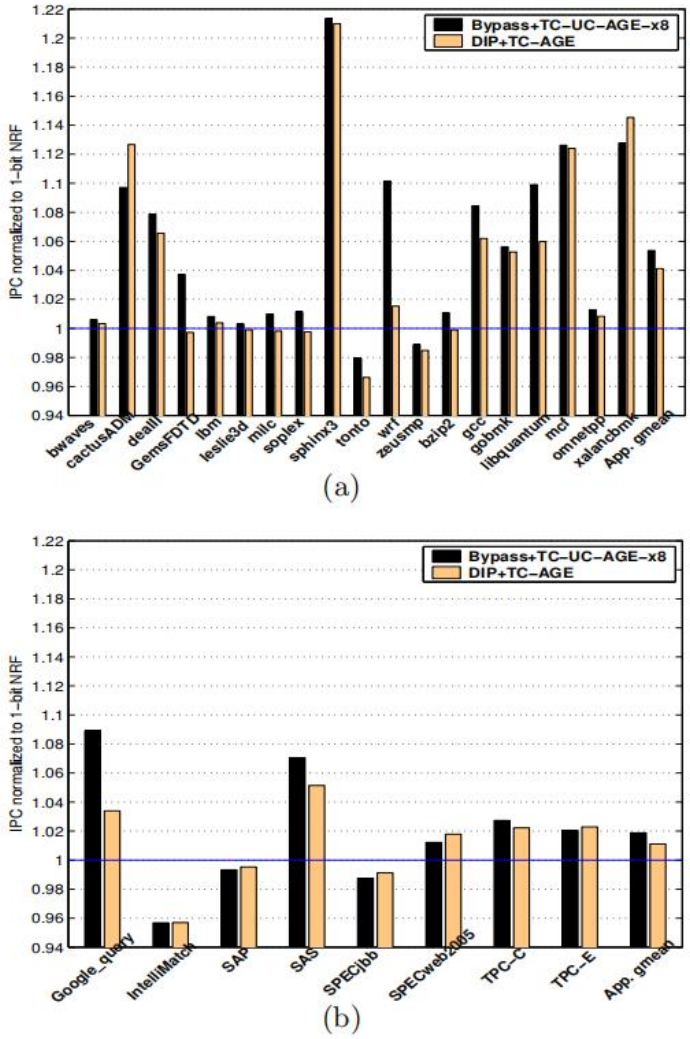


Figure 10: Details of IPC improvement achieved by our best policy and DIP+TC-AGE for (a) selected SPEC 2006 applications and (b) server applications.

最后，我们转向启用了激进的 multi-stream 硬件预取器的性能结果。图 11(a) 显示了启用预取器的 Bypass+TC-UCAGE-x8 与 1-bit NRF baseline 相比实现的 IPC 改进。

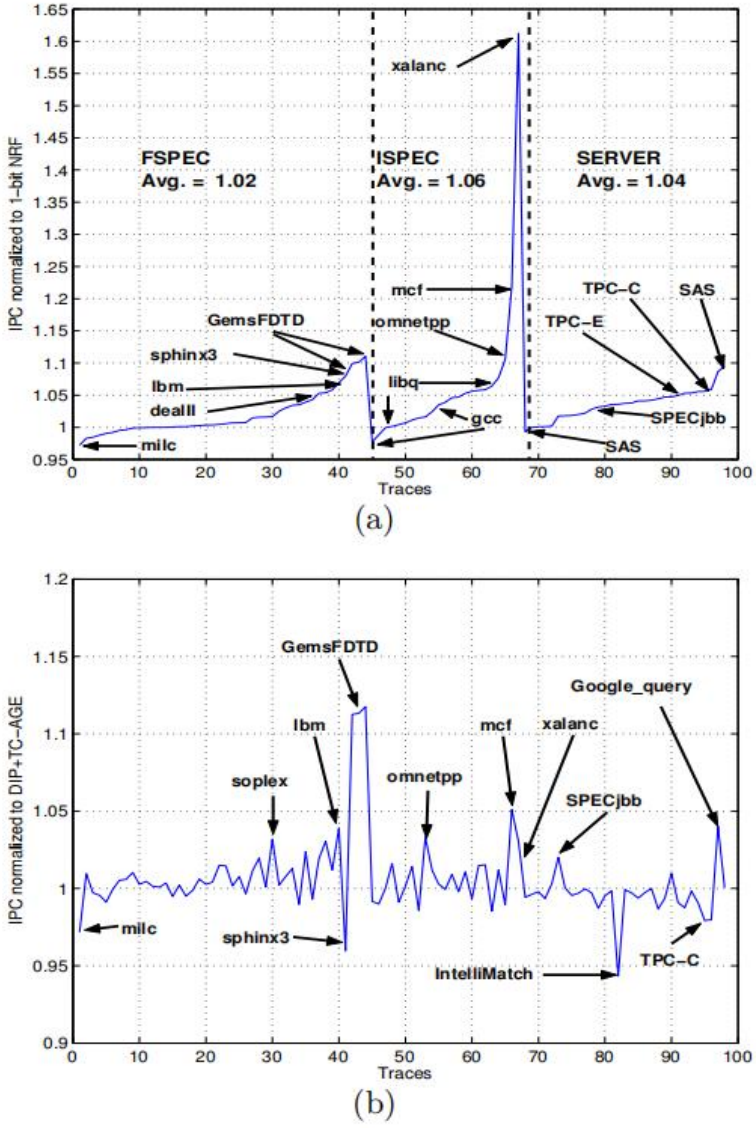


Figure 11: (a) Distribution of IPC improvements of our best policy normalized to 1-bit NRF. (b) IPC of our best policy normalized to DIP+TC-AGE. Both results are with prefetchers enabled.

在每个工作负载类别中，traces 按 IPC 提升进行分类。总的来说，对于 FSPEC，IPC 改进平均为 2%；对于 ISPEC，则为 6%；对于 SERVER traces，则为 4%。虽然与非预取的场景相比，FSPEC 和 ISPEC 的平均 IPC 改进有所下降（如预期的那样），但 SERVER traces 的提升有所上升。我们发现，我们对 UC=0 bins 的特殊处理（见第 3.4 节）显著地有助于 server traces，因为通常很难准确地为 server 工作负载预取数据。总的来说，在启用了预取器后，我们的最佳策略（bypass+TCUC-AGE-x8）在 97 条 traces 上实现的 IPC 改进为 3.4%。DIP+TC-AGE 的相应改善为 2.8%。在启用预取器的情况下，bypass +TC-UC-AGE-x8 实现的 bypass 比例平均为所有 L2 缓

存驱逐的 28%。在完整的 213 条 traces 集上，通过 Bypass+TC-UC-AGE-x8 实现的平均 IPC 改进为 2.4%，最大降低为 2.8%。

图 11(b)进一步总结了在存在预取的情况下，Bypass+TCUC-AGE-x8 归一化为 DIP+TC-AGE 的 IPC。这些 traces 的排序方式与图 11(a)中显示的方式相同。有明显收益或损失的 traces 被标记出来。图 12(a)和 12(b)显示了将我们的最佳策略和 DIP+TC-AGE 归一化到 1-bit NRF baseline 的应用程序级 IPC 改进。对于 SPEC 2006 的应用程序，我们的策略将 IPC 平均提高了 3.7%。SERVER 应用程序的相应改进为 3.6%。

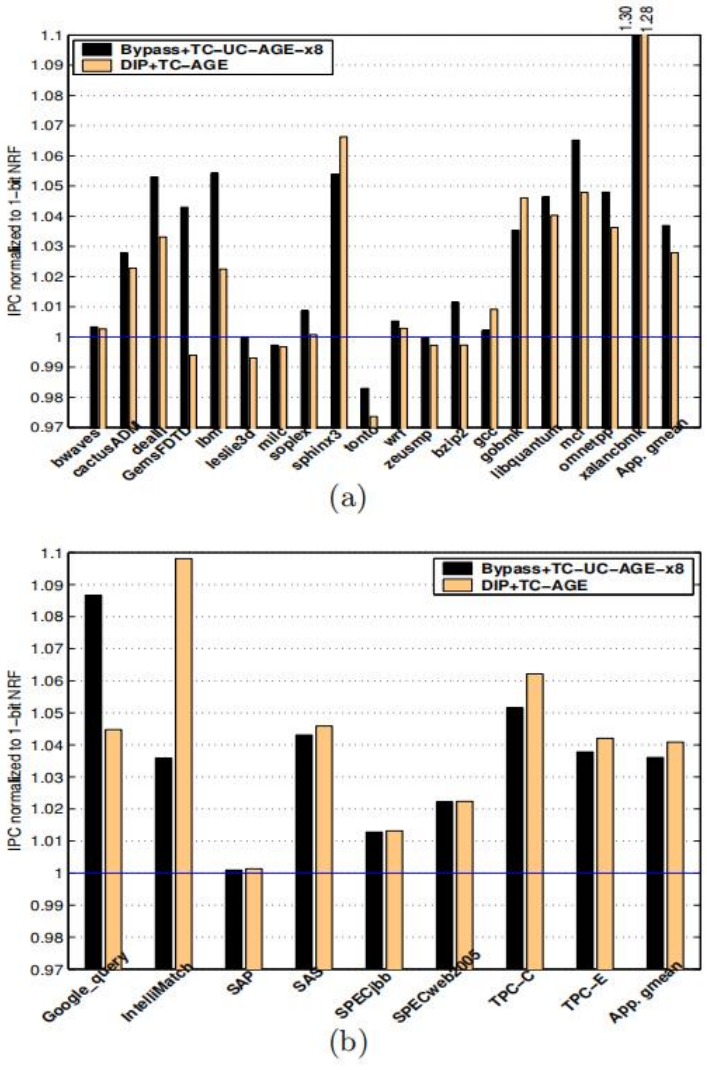


Figure 12: Details of IPC improvement achieved by our best policy and DIP+TC-AGE for (a) SPEC 2006 and (b) server applications in the presence of prefetchers.

5.2 Multi-programmed Workloads

图 13 总结了 4 路多编程工作负载的结果。

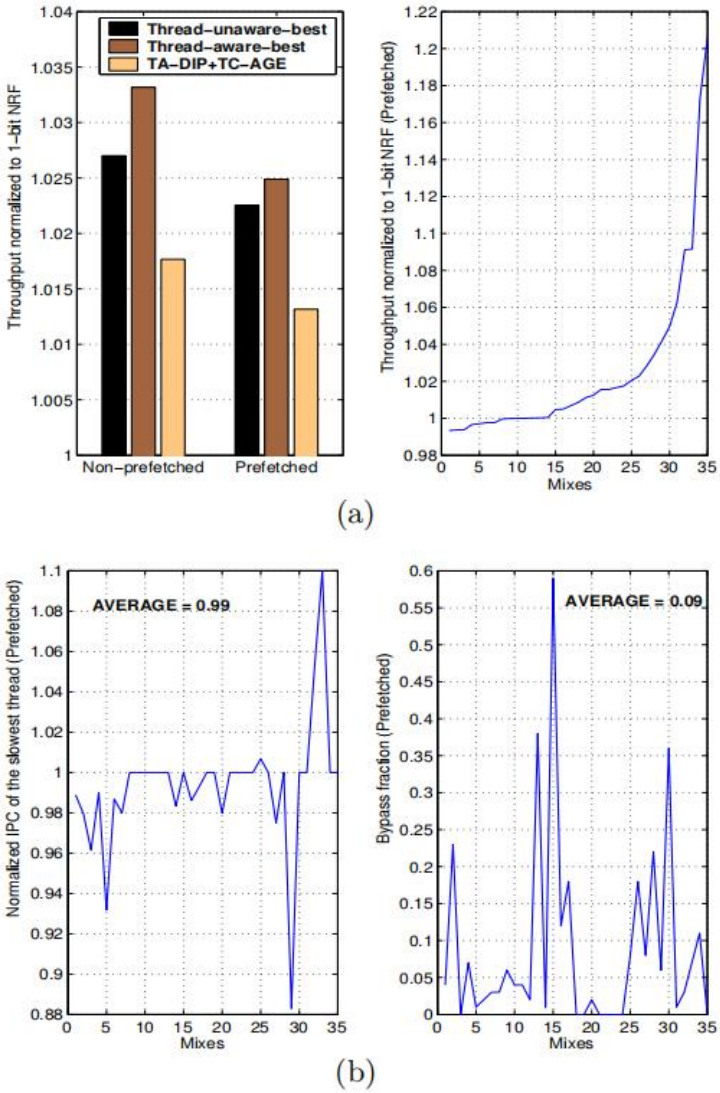


Figure 13: (a) Throughput improvements, (b) fairness and bypass fraction for the 4-way multi-prog. workloads.

图 13(a)的左半边在平均 IPC 或吞吐量提升方面（$\frac{\sum_i IPC_i^{Policy}}{\sum_i IPC_i^{Base}}$）评估了三种策略的性能，即线程无关的 bypass+TC-UC-AGE-x8、线程感知的 bypass+TC-UC-AGE-x8 和线程感知的 DIP+TC-AGE。基于线程感知的决斗机制借鉴了 TADIP-F proposal[16]。我们展示了非预取场景和预取场景的性能比较。图 13(a)的右半边量化了在启用预取器的情况下，对线程进行感知的 bypass+TC-UC-AGE-x8 的每次混合吞吐量改进。总之，在没有预取的情况下，线程感知会带来更大的性能提高。在存在预取的情况下，线程感知的 Bypass+TC-UC-AGE-x8 策略将吞吐量提高 2.5%，而线程感知的 DIP+TC-AGE 策略将吞吐量提高 1.3%。

任何单个线程的最大减速都应在一个可接受的范围内。图 13(b)的左半边量化了一个保守的公平性度量 $\min_i \frac{IPC_i^{Policy}}{IPC_i^{Base}}$，即，对于启用硬件预取器的线程感知的 bypass+TC-UC-AGE-x8 策略的每个混合中最慢线程的归一化 IPC。混合的排序方式与图 13(a)的右半边相同。除了一些混合之外，最慢的线程所经历的减速与基线相比在 2%以内，平均而言，是 1%。最后，图 13(b)的右半边详细说明了在启用硬件预取器后的线程感知 bypass+TC-UC-AGE-x8 所实现的 bypass 比例。虽然有几种混合物享有相当大的 bypass 比例，但平均比例是 9%。

6 Summary

这项工作做了一个重要的观察，即 LRU 及其近似在 exclusive LLC 中失去了意义，并提出了一些在三级 cache 中针对这种设计的选择性 bypass 和 insertion age 分配的设计选择。我们的 LLC bypass 和 age 分配决策是基于一个块的两个特性，当它在 LLC 中考虑分配时。第一个属性是 L2 和 LLC 之间的块从进入层次结构到被驱逐出 LLC 的 trip 次数（trip count）。第二个属性是一个块驻留在 L2 期间所经历的 L2 缓存命中次数（use count）。我们最好的方案是基于 trip count 和 use count 的 bypass 和 age insertion 方案的组合，配合激进的 multi-stream 预取器，在一个 2MB/16way 的 exclusive LLC 中，与 baseline 的 NRF 替换策略相比，将 97 个单线程 traces 的平均 IPC 提高了 3.4%。35 个 4way 多编程 mix 的吞吐量提升为 2.5%。