体系结构大作业

摘要

这篇论文里介绍了实现了在 L2C 上的基于 GHB 的步长预取算法，以及在 LLC 上的 LFUcache 替换算法。介绍了 GHB 的一些实验原理，和在代码编写过程中的思路。LFU 是一个比较经典的算法，根据这个算法的逻辑，进行了 LFU 替换策略的编写。使用 ghb+lfu 组合，进行实验测试，基于数据集的不同，调整 GHB 的步幅，可以发现实验效果会有很大差别。之后做了扩展，写了其他缓存级别上的预取，别切进行了测试和实验分析。对于替换策略，基于 Cache-based side channel attacks 漏洞，实现了一种安全替换，并进行了结果分析。

1 问题描述

- 正确编译 ChampSim，并且完成最少一个预取器和一个 cache 替换策略的添加。这里推荐在 L2 cache 上实现 GHB 步长预取器，在 LLC 上实现 LFU cacheline 替换策略。你还需要完成一个详尽的实验报告，里面要包括你模拟器的具体实现和实验效果分析。

- 实现多种预取器和 cache 替换策略并且进行性能比较，采用多项性能指标来解释不同策略的优缺点。实现更多复杂策略的组合，并且详细分析引入的代价和性能提升效果。

2 基于 GHB 的步长预取算法

2.1 算法介绍

在微处理器的整个发展过程中，基础半导体技术和微体系结构的发展趋势都大大缩短了处理器的时钟周期。同时，主存储器技术的主要趋势是朝着更高密度的方向发展，存储器存取时间比处理器周期时间减少得少得多。当以处理器时钟周期测量时，这些趋势显著增加了主存储器的延迟。为了避免因长时间的内存访问延迟而造成的巨大性能损失，微处理器依赖于一个更高的高速缓冲存储器体系。不幸的是，由于有限的高速缓存容量(以及在较小程度上，有限的关联性)，高速缓存并不总是有效的。为了至少部分克服高速缓冲存储器的限制，可以将数据预取到高速缓存中。最简单的预取方法是顺序的，也就是说，它们访问紧跟在当前高速缓存行之后的高速缓存行。GHB 的结构如下图2.1所示:

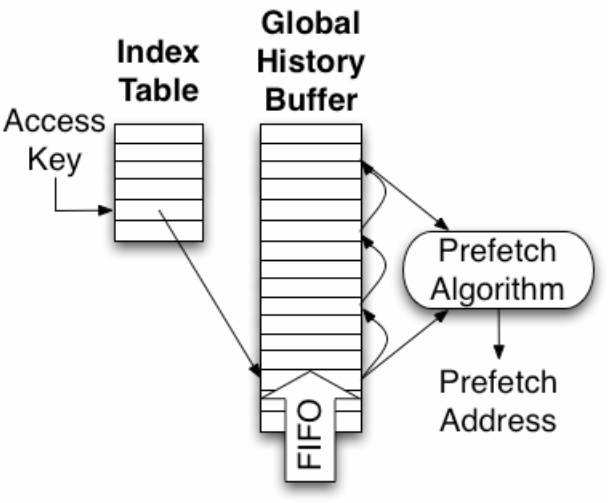

图 2.1: GHB 结构

一般来说，预取表存储预取历史效率低。首先，表数据可能会变得陈旧，从而降低预取的准确性 (程序在被逐出之前实际使用的预取的百分比)。其次，当多个访问键映射到同一个表条目时，表会发生冲突。减少表冲突最常见的解决方案是增加表元素的数量。但是，这种方法增加了表的内存需求，并增加表中保存的陈旧数据的百分比。第三，表中每个条目都有固定的 (通常很少) 历史记录。为每个条目添加更多的预取历史记录为有效的预取创造了新的机会，但是额外的预取历史记录也会增加表的内存需求及其陈旧数据的百分比，这两者一起会抵消这些优势。GHB 将表键匹配与预取相关历史信息的存储相分离。整体预取结构有两个层次。

- 一种索引表 (IT)，与传统的预取表一样，用一个键来访问。该关键字可以是加载指令的 PC、高速缓存未命中地址或某种组合。索引表中的条目包含指向全局历史缓冲区的指针。

- 全局历史缓冲区 (GHB) 是一个 n 项 FIFO 表 (实现为循环缓冲区)，保存 n 个最近的 L2 未

命中地址。每个 GHB 条目存储一个全局未命中地址和一个链接指针。链接指针用于将 GHB 条目链接到地址列表中。每个地址列表都是具有相同索引表关键字的按时间顺序排列的地址序列。

2.2 实验设计

为了实现步幅预取 (PC/CS)，GHB 结构以恒定的步幅检测加载指令。使用加载指令的 PC 作为索引表的索引，创建的地址列表是给定 PC 的地址序列。可以通过计算地址列表中连续条目之间的差异来计算负载的跨度。实验中设置恒定步幅为 5，而且在实验过程中发现，步幅从 3 变成 5，效果会好很多。

在代码编写的过程中，一个简单的状态机维护 GHB 地址列表并协调 GHB 访问以遍历地址列表。对于每个新的未命中，GHB 以 FIFO 方式更新。未命中地址被放入由头指针指向的 GHB 条目中，如图2.2, 其链接条目被赋予索引中的当前值。然后用指向新添加条目的头指针更新索引表链接条目。最后，头指针递增以指向下一个 GHB 条目。

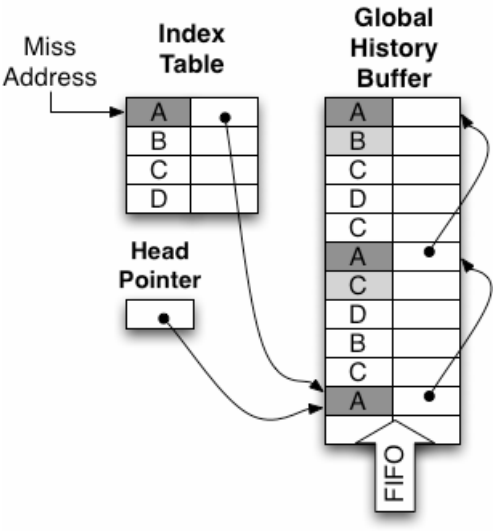

图 2.2: GHB 全局地址关联

算法的具体运行过程是根据当前缓存行与先前缓存行之间的间距（地址差）来预测将访问的下一个缓存行的地址。如果当前缓存行和先前缓存行之间的间距与先前缓存行和再前一个缓存行之间的间距相同，则使用命名为 PREFETCH_LOOKAHEAD 的常量，这里设置为 4，来确定要提前预取多少个缓存行，并在 L2 缓存中填充预取的缓存行。当预取的缓存行不在与当前需求访问地址相同的 4KB 页面中时，预取就会停止。

3 LLC 上实现 LFUcache 替换算法

3.1 算法介绍

LFU (Least Frequently Used) 是一种常用的高速缓存淘汰策略，用于在缓存满了之后删除最少使用的项目。

在 LFU 缓存中，每个缓存项都有一个访问次数计数器。当缓存被访问时，该计数器会自动增加。当缓存满了之后，LFU 缓存会删除访问次数最少的项目。

LFU 缓存的优点是它能够快速淘汰最少使用的项目，使得缓存能够保存更多有价值的数据。然而，LFU 缓存也有一些缺点。因为它只考虑了过去的访问次数，所以对于一些突然变得非常流行的数据来说，LFU 缓存可能不会立即更新它的内容，导致缓存的命中率降低。

LFU 缓存的实现可以使用哈希表和双向链表。哈希表用于快速查找缓存项，而双向链表则用于按照访问次数从小到大排序缓存项。这样，每次当缓存满了之后，我们就可以从双向链表的头部开始删除节点，直到缓存有足够的空间为止。

当我们谈到缓存回收算法时，我们需要主要关注对缓存数据的 3 种不同操作。

- 在缓存中设置 (或插入) 一个项目.
- 检索 (或查找) 缓存中的项目；同时增加其使用计数 (对于 LFU)
- 从高速缓存中逐出 (或删除) 最不常用的 (或作为逐出算法的策略) 项目

对于可以在 LFU 高速缓存上执行的每个字典操作 (插入、查找和删除), 所提出的 LFU 算法具有 O(1) 的运行时间复杂度。这是通过维护两个链表来实现的；一个用于访问频率，一个用于具有相同访问频率的所有元件。哈希表用于通过键访

问元素 (为清晰起见，下图中未显示)。双向链表用于将表示一组具有相同访问频率的节点链接在一起 (如下图中的矩形块所示)。我们将这个双向链表称为频率列表。这组具有相同访问频率的节点实际上是此类节点的双向链表 (在下图中显示为圆形节点)。我们将这种双向链表 (对于特定频率是本地的) 称为节点列表。节点列表中的每个节点都有一个指向其在频率列表中的父节点的指针 (为清晰起见，图中未示出)。因此，节点 x 和 y 将有一个指针指向节点 1，节点 z 和 a 将有一个指针指向节点 2，依此类推...

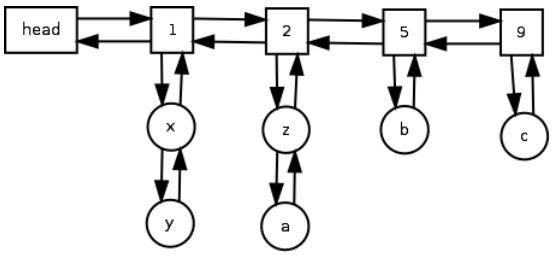

图 3.3: 包含 6 个元素的 LFU 字典

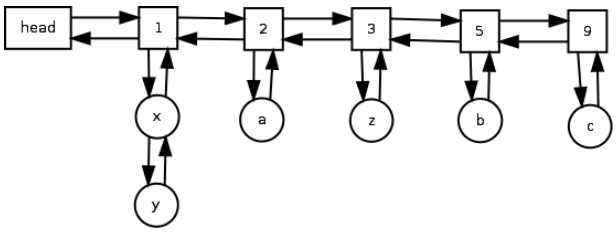

图 3.4: 再次访问带有键 “z” 的元素后

3.2 实验设计

在 LFU 缓存中，每个缓存块都有一个访问次数计数器。当缓存块被访问时，该计数器就会增加。当缓存满了之后，LFU 缓存会删除访问次数最少的缓存块。

在这段代码中，LFU 缓存使用了一个二维数组 times 来存储每个缓存块的访问次数。当缓存块被访问时，它的访问次数就会增加；当缓存满了之后，LFU 缓存会找到访问次数最少的缓存块，并将其删除。

在 llc_find_victim 函数中，LFU 缓存遍历了 times 数组，找到了访问次数最少的缓存块。在 llc_update_replacement_state 函数中，LFU 缓存更新了缓存块的访问次数。

最后，llc_replacement_final_stats 函数用于在模拟结束时输出统计信息。

llc_update_replacement_state 函数负责更新 LFU 缓存的状态。该函数首先检查缓存命中情况，如果没有命中，就将访问次数设为 1。如果命中，就将访问次数加 1。

然后，该函数还更新了一个计数器 time_counter，并在 time_counter 的值到达 50 时将另一个计数器 miss_counter 设为 0，其中还调用函数更新了 LRU 缓存的状态。

llc_find_victim 函数负责找到 LFU 缓存的淘汰块。该函数遍历了二维数组 times，找到了访问次数最少的缓存块。如果找到了合法的淘汰块，就返回该块的位置；否则返回 0。

4 GHB+LFU 实验结果

```
Core_0_L2C_total_access 5020869
Core_0_L2C_total_hit 3513711
Core_0_L2C_total_miss 1507158
Core_0_L2C_loads 1377846
Core_0_L2C_load_hit 1018353
Core_0_L2C_load_miss 359493
Core_0_L2C_RFOs 90454
Core_0_L2C_RFO_hit 10607
Core_0_L2C_RFO_miss 79847
Core_0_L2C_prefetches 3432827
Core_0_L2C_prefetch_hit 2365235
Core_0_L2C_prefetch_miss 1067592
Core_0_L2C_writebacks 119742
Core_0_L2C_writeback_hit 119516
Core_0_L2C_writeback_miss 226
Core_0_L2C_prefetch_requested 5133491
Core_0_L2C_prefetch_issued 5133361
Core_0_L2C_prefetch_useful 1213997
Core_0_L2C_prefetch_useless 396028
Core_0_L2C_prefetch_late 193
Core_0_L2C_average_miss_latency 163.63373
```

图 4.5: 482 数据集 L2C 结果

```
Core_0_LLC_total_access 1612282
Core_0_LLC_total_hit 531799
Core_0_LLC_total_miss 1080483
Core_0_LLC_loads 359301
Core_0_LLC_load_hit 123527
Core_0_LLC_load_miss 235774
Core_0_LLC_RFOs 79847
Core_0_LLC_RFO_hit 19822
Core_0_LLC_RFO_miss 60025
Core_0_LLC_prefetches 1067784
Core_0_LLC_prefetch_hit 365612
Core_0_LLC_prefetch_miss 702172
Core_0_LLC_writebacks 105350
Core_0_LLC_writeback_hit 22838
Core_0_LLC_writeback_miss 82512
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_useful 1466
Core_0_LLC_prefetch_useless 1029620
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 185.61427
```

图 4.6: 482 数据集 LLC 结果

```
Core_0_L2C_total_access 12049240
Core_0_L2C_total_hit 9397552
Core_0_L2C_total_miss 2651688
Core_0_L2C_loads 2651631
Core_0_L2C_load_hit 2322623
Core_0_L2C_load_miss 329008
Core_0_L2C_RFOs 20
Core_0_L2C_RFO_hit 0
Core_0_L2C_RFO_miss 20
Core_0_L2C_prefetches 8296075
Core_0_L2C_prefetch_hit 5973415
Core_0_L2C_prefetch_miss 2322660
Core_0_L2C_writebacks 1101514
Core_0_L2C_writeback_hit 1101514
Core_0_L2C_writeback_miss 0
Core_0_L2C_prefetch_requested 16152350
Core_0_L2C_prefetch_issued 16152350
Core_0_L2C_prefetch_useful 3565486
Core_0_L2C_prefetch_useless 40
Core_0_L2C_prefetch_late 162204
Core_0_L2C_average_miss_latency 286.94546
```

图 4.7: 462 数据集 L2C 结果

```
Core_0_LLC_total_access 3752963
Core_0_LLC_total_hit 217920
Core_0_LLC_total_miss 3535043
Core_0_LLC_loads 166804
Core_0_LLC_load_hit 9650
Core_0_LLC_load_miss 157154
Core_0_LLC_RFOs 20
Core_0_LLC_RFO_hit 0
Core_0_LLC_RFO_miss 20
Core_0_LLC_prefetches 2484864
Core_0_LLC_prefetch_hit 143950
Core_0_LLC_prefetch_miss 2340914
Core_0_LLC_writebacks 1101275
Core_0_LLC_writeback_hit 64320
Core_0_LLC_writeback_miss 1036955
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_useful 0
Core_0_LLC_prefetch_useless 3495552
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 217.07566
```

图 4.8: 462 数据集 LLC 结果

这些统计信息涉及 L2 缓存和 LLC 的访问，命中，缺失，加载，预取和写回操作。从结果中可以看到在 482 数据集中 IPC 测试更好，但是在 462 数据集上的精准率更高，可以看到 prefetch_useless 数值更低，在实验过程中发现，ghb 设置的步长大小对结果影响也比较大。

5 其他的预取和替换策略

5.1 多组件预取器

5.1.1 算法介绍

这里涉及到 L1C，L2C 和 LLC。该预取器采用三个不同的组件以分层的方式组织，以解决访问模式的多样性。每一级预取器使用不同数量的控制流和数据流信息。第一级预取器利用指令指针 (IP) 和由当前 IP 获得的访问流中的当前增量 (即控制流和数据流) 的联合特征，以高置信度预测增量序列。为了保持高覆盖率，当基于 IP 增量的预取器不能提供高置信度预测时，调用第二级预取器，该预取器采用基于 IP(即，仅基于控制流) 的步距预测器。如果基于 IP 的步距预测器也不能有把握地发现任何步距模式，则在预取器的第三级触发具有自适应程度 (仅基于数据流) 的下一行预取器。然而，当与 L1 高速缓存预取器一起工作时，合并在 L2 高速缓存中的相同的三级预取器提供了边际效益。此外还使用一种技术来优化预取所消耗的 L1 缓存查找带宽，从而增强了三级缓存预取器。预取器采用三个不同的组件，它们以分级方式组织，利用组件中不同数量的控制流和数据流信息。这些组件是基于 IP-delta 联合特征的 delta 序列预预测器、基于 IP 的跨距预取器和自适应度 next_line 预取器。

5.1.2 实验设计

预取器结构如下图5.9

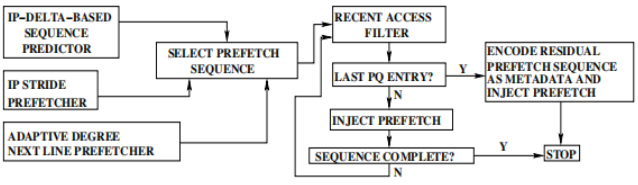

图 5.9: 462 数据集 LLC 结果

基于 IP 增量的序列预测器首先基于它们的源 IP 对所有按需访问进行分类。接下来，对于来自

特定源 IP 并且相对于来自同一 IP 的最后访问具有增量的给定访问，预测器预测接下来的 d 个增量的序列。两次访问之间的增量被定义为当前访问的偏移量 (即，页面内的高速缓存块数) 和上一次访问的偏移量之间的差。使用该预测器的动机来自于这样的观察，即在来自特定 IP 的访问中，在给定增量之后出现的增量序列经常在相同的增量之后重复.

当基于 IP 增量的序列预测器由于 IP 增量表缺失或低置信度预测而不能提供预测时，利用来自 IP 表的步距预测。这种策略避免了覆盖范围的损失。这种预测没有任何额外的开销，因为无论如何都要查找 IP 表。在通过插入当前增量来更新匹配 IP 表条目的 FIFO 列表之后，如果 FIFO 列表中的最后两个增量是相同的，则在 IP-增量表由于未命中或低置信度而不能提供预测的情况下，该增量用于生成长度为 d 的预取序列。我们还采用从 IP 表预测的这个增量来完成否则由于 IP 增量表的预测序列中的低置信度增量而提前终止的预取序列.

5.1.3 实验结果

```
Core_0_L2C_total_access 5020869
Core_0_L2C_total_hit 3513711
Core_0_L2C_total_miss 1507158
Core_0_L2C_loads 1377846
Core_0_L2C_load_hit 1018353
Core_0_L2C_load_miss 359493
Core_0_L2C_RFOs 90454
Core_0_L2C_RFO_hit 10607
Core_0_L2C_RFO_miss 79847
Core_0_L2C_prefetches 3432827
Core_0_L2C_prefetch_hit 2365235
Core_0_L2C_prefetch_miss 1067592
Core_0_L2C_writebacks 119742
Core_0_L2C_writeback_hit 119516
Core_0_L2C_writeback_miss 226
Core_0_L2C_prefetch_requested 5133491
Core_0_L2C_prefetch_issued 5133361
Core_0_L2C_prefetch_useful 1213997
Core_0_L2C_prefetch_useless 396028
Core_0_L2C_prefetch_late 193
Core_0_L2C_average_miss_latency 163.63373
```

图 5.10: 482 数据集 L2C 结果

```
Core_0_LLC_total_access 1612282
Core_0_LLC_total_hit 531799
Core_0_LLC_total_miss 1080483
Core_0_LLC_loads 359301
Core_0_LLC_load_hit 123527
Core_0_LLC_load_miss 235774
Core_0_LLC_RFOs 79847
Core_0_LLC_RFO_hit 19822
Core_0_LLC_RFO_miss 60025
Core_0_LLC_prefetches 1067784
Core_0_LLC_prefetch_hit 365612
Core_0_LLC_prefetch_miss 702172
Core_0_LLC_writebacks 105350
Core_0_LLC_writeback_hit 22838
Core_0_LLC_writeback_miss 82512
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_useful 1466
Core_0_LLC_prefetch_useless 1029620
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 185.61427
```

图 5.11: 482 数据集 LLC 结果

```
Core_0_L2C_total_access 5495605
Core_0_L2C_total_hit 2843912
Core_0_L2C_total_miss 2651693
Core_0_L2C_loads 2651631
Core_0_L2C_load_hit 981048
Core_0_L2C_load_miss 1670583
Core_0_L2C_RFOs 20
Core_0_L2C_RFO_hit 0
Core_0_L2C_RFO_miss 20
Core_0_L2C_prefetches 1742440
Core_0_L2C_prefetch_hit 761350
Core_0_L2C_prefetch_miss 981090
Core_0_L2C_writebacks 1101514
Core_0_L2C_writeback_hit 1101514
Core_0_L2C_writeback_miss 0
Core_0_L2C_prefetch_requested 11747280
Core_0_L2C_prefetch_issued 11560823
Core_0_L2C_prefetch_useful 2286590
Core_0_L2C_prefetch_useless 102
Core_0_L2C_prefetch_late 1587863
Core_0_L2C_average_miss_latency 163.08008
```

图 5.12: 462 数据集 L2C 结果

```
Core_0_LLC_total_access 3752967
Core_0_LLC_total_hit 1101274
Core_0_LLC_total_miss 2651693
Core_0_LLC_loads 82721
Core_0_LLC_load_hit 0
Core_0_LLC_load_miss 82721
Core_0_LLC_RFOs 20
Core_0_LLC_RFO_hit 0
Core_0_LLC_RFO_miss 20
Core_0_LLC_prefetches 2568952
Core_0_LLC_prefetch_hit 0
Core_0_LLC_prefetch_miss 2568952
Core_0_LLC_writebacks 1101274
Core_0_LLC_writeback_hit 1101274
Core_0_LLC_writeback_miss 0
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_useful 0
Core_0_LLC_prefetch_useless 3842581
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 185.66642
```

图 5.13: 462 数据集 LLC 结果

这个策略 LLC 上的替换策略使用的是 LFU，可以看到因为多了其他级缓存的预取，在 482 数据集上的效果比只做了 L2C 上的 ghb 预取结果要

好。但对于这个多组件预取策略来说，在 L1 缓存中添加预取器比在 L2 缓存中添加预取器要好。

5.2 LLC 上的安全替换

5.2.1 算法介绍

Cache-based side channel attacks 是一种漏洞，它是在超线程（SMT）环境下攻击者利用包含和共享缓存的环境而产生的。在这种环境下，攻击者和受害者进程可能同时在两个不同的内核上的两个不同的线程上执行。例如，在 evict+reload 攻击中，攻击者（称为" 间谍"）会反复将" 探测地址" 清除并重新加载，并检查在两个操作之间受害者是否访问了该地址（如下图5.14所示）。如果受害者在间谍重新加载时访问了该地址，则间谍会经历缓存命中（可以作为减少的内存延迟来测量），否则会收到缓存未命中。通过分析这些命中和未命中，攻击者可以推断出受害者进程的内存访问模式。注意，探测地址（其访问模式可以揭示受害者程序信息的地址）是在离线阶段使用自动工具或手动方法识别的。受害者程序可以是任何加密算法，如 RSA 等。

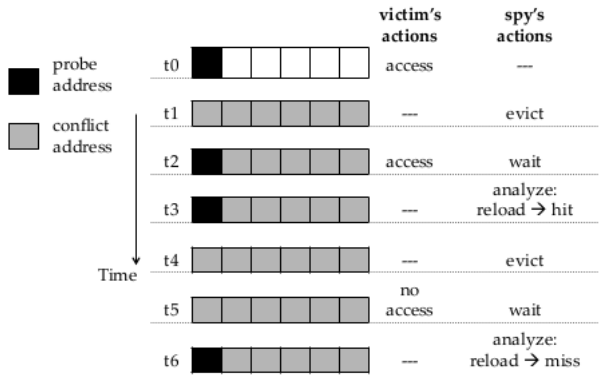

图 5.14: evict+reload

一旦意识到间谍进程由于共享缓存（通常是 LLC）的包容性质而从受害者进程的私有缓存中清除探测地址的能力，我们要做的就是尽可能减小选择被使用的包含受害者的概率。包含受害者是指因为与共享缓存（由间谍）发生冲突而需要从私有缓存中清除的行（由于包容性替换策略）。

5.2.2 实施细节

由于 ChampSim 本身并没有实现包容式替换策略，所以在 ChampSim 的缓存中的 CACHE :: handle_fill() 方法中实现了包容式替换策略。当在 LLC 级别进行 fill 请求时，底层 LLC 替换策略配备了查找受害者的方式来完成 handle_fill() 事务的最近的 MSHR（Miss Status Holding Registers）条目；由 MSHR.next_fill_index 给出。如果在 LLC 级别有效（即该块的有效位设置），当它被驱逐以容纳新条目时，如果上层缓存中存在相关块，我们必须向上层缓存发送后无效请求（取消设置有效位，本质上驱逐该块）。如果 fill 请求在 L2C 级别出现，则重复使用相同的策略。

添加了量化缓存包容性的指标-L1_backreq_counter，L2_backreq_counter，LLC_eviction_counter-这些指标在 ChampSim 上运行后与其余模拟统计信息一起打印。

5.2.3 实验结果

```
Core_0_L2C_total_access 5020869
Core_0_L2C_total_hit 3513711
Core_0_L2C_total_miss 1507158
Core_0_L2C_loads 1377846
Core_0_L2C_load_hit 1018353
Core_0_L2C_load_miss 359493
Core_0_L2C_RFOs 90454
Core_0_L2C_RFO_hit 10607
Core_0_L2C_RFO_miss 79847
Core_0_L2C_prefetches 3432827
Core_0_L2C_prefetch_hit 2365235
Core_0_L2C_prefetch_miss 1067592
Core_0_L2C_writebacks 119742
Core_0_L2C_writeback_hit 119516
Core_0_L2C_writeback_miss 226
Core_0_L2C_prefetch_requested 5133491
Core_0_L2C_prefetch_issued 5133361
Core_0_L2C_prefetch_useful 1213997
Core_0_L2C_prefetch_useless 396028
Core_0_L2C_prefetch_late 193
Core_0_L2C_average_miss_latency 163.63373
```

图 5.15: 482 数据集 L2C 结果

```
Core_0_LLC_total_access 1612282
Core_0_LLC_total_hit 531799
Core_0_LLC_total_miss 1080483
Core_0_LLC_loads 359301
Core_0_LLC_load_hit 123527
Core_0_LLC_load_miss 235774
Core_0_LLC_RFOs 79847
Core_0_LLC_RFO_hit 19822
Core_0_LLC_RFO_miss 60025
Core_0_LLC_prefetches 1067784
Core_0_LLC_prefetch_hit 365612
Core_0_LLC_prefetch_miss 702172
Core_0_LLC_writebacks 105350
Core_0_LLC_writeback_hit 22838
Core_0_LLC_writeback_miss 82512
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_useful 1466
Core_0_LLC_prefetch_useless 1029620
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 185.61427
```

图 5.16: 482 数据集 LLC 结果

```
Core_0_L2C_total_access 11712748
Core_0_L2C_total_hit 9061060
Core_0_L2C_total_miss 2651688
Core_0_L2C_loads 2651631
Core_0_L2C_load_hit 2281355
Core_0_L2C_load_miss 370276
Core_0_L2C_RFOs 20
Core_0_L2C_RFO_hit 0
Core_0_L2C_RFO_miss 20
Core_0_L2C_prefetches 7959583
Core_0_L2C_prefetch_hit 5678191
Core_0_L2C_prefetch_miss 2281392
Core_0_L2C_writebacks 1101514
Core_0_L2C_writeback_hit 1101514
Core_0_L2C_writeback_miss 0
Core_0_L2C_prefetch_requested 16152350
Core_0_L2C_prefetch_issued 16152350
Core_0_L2C_prefetch_useful 3524218
Core_0_L2C_prefetch_useless 40
Core_0_L2C_prefetch_late 203472
Core_0_L2C_average_miss_latency 299.70585
```

图 5.17: 462 数据集 L2C 结果

```
Core_0_LLC_total_access 3752963
Core_0_LLC_total_hit 1101275
Core_0_LLC_total_miss 2651688
Core_0_LLC_loads 166804
Core_0_LLC_load_hit 0
Core_0_LLC_load_miss 166804
Core_0_LLC_RFOs 20
Core_0_LLC_RFO_hit 0
Core_0_LLC_RFO_miss 20
Core_0_LLC_prefetches 2484864
Core_0_LLC_prefetch_hit 0
Core_0_LLC_prefetch_miss 2484864
Core_0_LLC_writebacks 1101275
Core_0_LLC_writeback_hit 1101275
Core_0_LLC_writeback_miss 0
Core_0_LLC_prefetch_requested 0
Core_0_LLC_prefetch_issued 0
Core_0_LLC_prefetch_useful 0
Core_0_LLC_prefetch_useless 3697042
Core_0_LLC_prefetch_late 0
Core_0_LLC_average_miss_latency 311.96966
```

图 5.18: 462 数据集 LLC 结果

这个算法在两个数据集上差别不大，猜测是主要关注点在漏洞上，效率提升并不明显，这个相较于 ghb+lfu 的组合，差别也不大。

参考文献

[1] M. K. Aguilera, D. D. E. Long, J. C. Sancho, and D. G. Steel. An O(1) algorithm for implementing the LFU cache eviction scheme. In International Conference on Measurements and Modeling of Computer Systems, pages 85–96, 2003.

[2] M. Mohtashim, A. Raza, H. Fatema, and M. O. Ahmed. ”Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Cache-Based Side Channel Attacks.” In Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 671-676, 2017.

[3] S. K. Gurumani and N. Vijaykrishnan. ”Sangam: A Multi-component Core Cache Prefetcher.” In Proceedings of the International Conference on Computer Architecture, pp. 33-44, 2016.