硕士研究生学位论文

题目：**CAPS：基于部分共享的高速缓存优化框架**

姓　　名：黄子翟

学　　号：1401214258

院　　系：信息科学技术学院

专　　业：计算机软件与理论

研究方向：系统虚拟化技术

导　　师：罗英伟教授

2017年6月

版权声明

摘要

随着处理器多核技术的广泛应用，多个程序可以在不同的核上并发执行。然而，并发执行的程序会在底层共享缓存（LLC）层面产生竞争，从而出现严重的性能下降。如何有效地对共享缓存进行调控和优化是被学术界广泛研究的一个问题。现有的解决方案大多数依赖硬件层面对缓存进行划分，从而满足性能或服务质量（QoS）等方面的要求。但是，这些硬件方案都只能在模拟器中进行模拟，并没有在真实系统中实现。直到最近英特尔在服务器级处理器中引入了高速缓存分配技术（CAT）。高速缓存分配技术基于缓存的路（Cache Way），所以只能在粗粒度上进行缓存分配。直接使用高速缓存分配技术进行不重叠的划分，在并发运行的线程数较多时，并不能很好的满足各种各样的优化目标。为了克服这些限制，我们刻意地将部分划分进行重叠，通过精确的重叠控制来提升分配的粒度。

本文提出了一个支持部分共享的高速缓存分配优化框架（CAPS）。它可以在较细的粒度上实现对缓存占用的控制。CAPS基于英特尔高速缓存分配技术，并且可以在真实系统中运行，它主要包含预测和分配两个模块。预测模块负责在任意重叠的缓存分配情况下对每个并发执行程序的缓存失效率和周期指令数进行预测，分配模块负责在给定一个优化目标后生成一个优化分配策略。CAPS可以支持多种优化目标，并且在并发程序数较大时也有很好的兼容性。在本文中我们实现了五种优化策略：失效数最小化，吞吐量最大化，以及三种性能下降指标的最小化。我们对75组并发工作负载进行实验评估，每组负载包括4到15个SPEC CPU2006测试程序。平均来看，相比于自由竞争使用LLC，CAPS可以降低16.96%的失效数，提升11.11%的吞吐量，减少8.16%的平均性能损失，在兼顾公平和性能的指标上提升8.17%，将性能下降最严重的程序提升23.24%。

关键词：多核，高速缓存分配技术，部分共享，优化

CAPS: Cache Allocation with Partial Sharing

Huang Zihui (Computer Science)
Directed by Prof. Luo Yingwei

ABSTRACT

In a multicore system, simultaneously executed programs may suffer from performance degradation due to contention on the shared last level cache (LLC). Effective management of LLC has attracted significant research attention. Existing solutions often rely on hardware cache partitioning to ensure performance and quality of service. However, none of these hardware partitioning schemes had been implemented on a real system until Intel introduced Cache Allocation Technology (CAT) to its commodity processors recently. CAT itself implements way partitioning and thus can only allocate at a coarse granularity. It does not scale well for a large thread or program count to serve their various performance goals effectively. We overcome these limitations by deliberately and precisely sharing part of the allocations among programs and cores.

In this paper, we propose Cache Allocation with Partial Sharing (CAPS), a framework that manages shared cache occupancy at a fine granularity. It is implemented on top of CAT, and runs on the real system. CAPS consists of two parts: (1) a prediction model that estimates miss rates and IPCs of a multiprogrammed workload under any partially-overlapping CAT scheme, and (2) a simulated annealing algorithm that outputs a near-optimal solution given a specific performance goal. CAPS is able to support a wide range of performance targets and can scale to a large core count. We demonstrate its flexibility by implementing five policies targeting average MPKI, IPC throughput, average slowdown, fair slowdown and maximum slowdown, respectively. Our evaluation, with 75 workloads ranging from 4-program to 15-program co-run, shows that on average, CAPS can reduce average MPKI by 16.96%, increase throughput by 11.11%, cut average slowdown by 8.16%, improve fair slowdown by 8.17%, and lower maximum slowdown by 23.24%, when compared to full-sharing.

KEYWORDS: Multicore, CAT, Partial-sharing, Optimization

目录

第一章　序言

随着多处理器、多核技术的迅猛发展，多核处理器（CMP）中的核数也越来越多。核数增加一个显而易见的优点就是可以支持更多的程序并发执行，然而这也导致了存储访问的瓶颈变得尤为突出。由于中央处理器（CPU）的运算速度远远高于其访问内存的速度，为了缓解这个速度差异，通常在CPU与内存之间会设有多级高速缓存（Cache）。在多核处理器中，每个核会有1到2级私有缓存，同时所有核会共享一个底层缓存（LLC)。由于缓存大小有限，并发执行的程序会对这层共享缓存进行竞争，竞争的结果就是每个程序的性能都或多或少受到损失。

目前CPU大多采用基于LRU的缓存替换策略，它对所有的缓存访问，不管来源于哪个核，都“一视同仁”。这就带来了一个问题：一个污染性高的程序会占据大量的缓存空间，从而压缩了其他程序的缓存使用，导致它们的失效率升高，性能下降。长期以来，国内外对这个问题展开了大量的研究工作，但是并没有真正解决，尤其在真实系统环境下。前人的研究大多依赖于硬件/软件层面上对共享缓存进行划分，通过对缓存插入和替换策略的调优来改善系统效率。然而，它们都有一个共同的缺点：无法应用到真实系统上。

基于硬件的方法主要是在模拟器中进行实现 [7, 9, 10, 18–20, 27, 32]。然而模拟器存在多方面的不足，比如运行速度较慢、准确度不够等等。通常一个基于模拟器的实验会运行几十亿条指令，这仅仅相当于实际运行的几秒钟。一个程序的结构和行为的复杂性无法在这么短的执行时间里体现出来。所以我们认为对于一个缓存优化方案，必须要经过长时间的执行才能证明其有效性。

基于软件的方法主要是通过页面着色技术（Page Coloring）。它通过操作系统对页面映射进行控制，从而完成对缓存占用的控制。页面着色技术本可以应用在真实系统中，然而现代处理器纷纷改用哈希算法，而不是原先一一对应的方式，来进行内存地址到缓存块的映射。所以页面着色技术目前也不再适用。

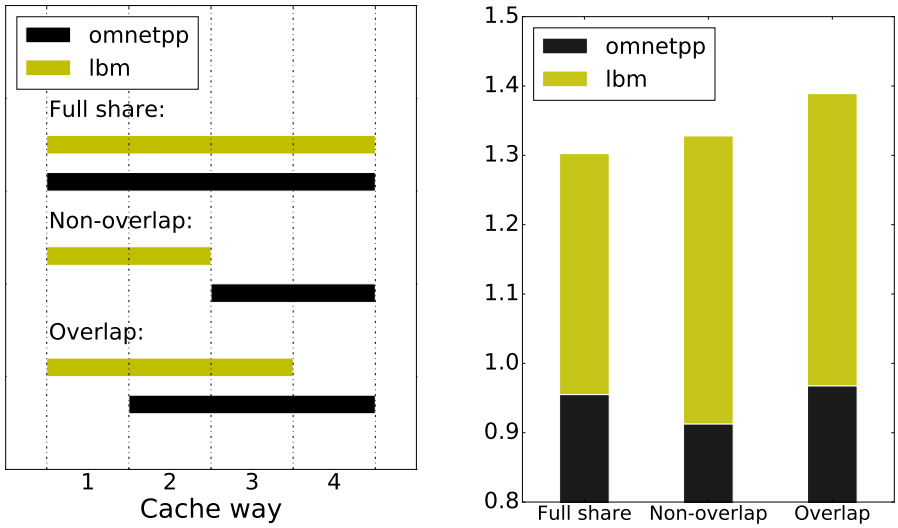
缓存调控的另一个关键点是分配的粒度。粗粒度分配技术一般是基于缓存路的划分（Way Partitioning） [7, 9, 10, 18–20, 27, 32]。一个路往往含有多个缓存块（Block），划分的时候不能把一个路切开分给两个不同的线程，并且由于体系结构的限制，路的个数不能无限制增加。所以基于路的缓存划分涉及的硬件改动较小，但是粒度较粗。另外一些研究提出了细粒度划分技术 [4, 16, 21]。虽然细粒度划分提供了更强的可扩展性和灵活性，但是需要更加复杂的设计和更多的硬件改动。考虑到设计的简洁性，大部分研究还是采用了粗粒度的路划分技术。正如前文所说，路划分下分配的最小单位

是路的大小。在核数/线程数增加的情况下，它的效率就会下降，因为最优的划分很可能会切在一个路的中间。极端情况下当核数/线程数等于总路数，每个线程有且只能有一个路的缓存分配，这就彻底失去了分配的灵活性。一些研究试图在保留路分配的基础上，细化分配粒度。Probabilistic Shared-Cache Management（PriSM）通过精确控制驱逐（Eviction）概率来把粒度变得更细 [16]。Cooperative Cache Partitioning（CCP）通过时域共享，在不同的时间片用不同的分配方案，来细化分配粒度。不过，这些研究都需要或多或少的改动硬件，所以只是在模拟器中进行实验，而暂时无法应用到实际系统中。

在本文中，我们提出了CAPS（Cache Allocation with Partial Sharing），一个基于部分共享的缓存分配优化框架。CAPS在真实系统上实现了细粒度的缓存分配，是一个纯软件的框架。CAPS依赖于英特尔高速缓存分配技术（CAT）。CAT是英特尔最近才在服务器处理器中全线引入的功能，首次在商业处理器上实现了对高速缓存的管理。CAT也是基于路的划分，所以本身是一个粗粒度的分配技术。CAT技术中可分配的路数非常有限，在最高端的处理器上也只有20个路可供分配，每个路包含数个MB的缓存空间。为此，我们提出了通过空间共享的方式来细化分配粒度。分配间的部分重叠是被CAT所允许的。通过精确地控制分配以及之间的重叠，我们可以实现一个较细粒度的缓存管理。

我们用一个小例子来说明部分共享如何优于完全共享和不共享的分配。两个SPEC CPU2006的测试程序470.lbm和471.omnetpp分别运行在两个核上，它们共享一个4路高速缓存，每路含有2816KB的缓存资源。我们在真实机器上实验了所有可能的CAT分配方案，然后选择最优的重叠方案和不重叠方案。最优这里指的是IPC（周期指令数）吞吐量的最大化，也就是这两个程序的IPC之和最大。图 1.1 (a) 展示了这三种方案的分配布局，从上到下分别是：传统的完全共享LLC（full share），最优的不重叠分配（non-overlapping），和最优的重叠分配（overlapping）。图 1.1 (b) 比较了这三种方案的IPC吞吐量。不难看出，部分重叠的分配方案比另外两种方案都要更优。完全重叠下两个程序的平均IPC分别为0.347465和0.954963，不重叠下分别为0.421380和0.912704，部分重叠下为0.415268和0.967625。部分重叠相比于完全重叠，两个程序的IPC都获得提升，而相比于不重叠分配，虽然470.lbm的IPC稍微损失了一点，但471.omnetpp得到了很大的提升，所以加总的IPC吞吐量也获得了提升。

我们更深入的研究表明，随着核数/并发线程数的增加，CAPS生成的部分重叠方案对完全重叠和不重叠划分的优势会愈发明显。CAPS包含两个部分：（1）一个预测模型可以较为准确地预测出在任意重叠CAT分配下（完全共享和不共享划分是两个特例）的并发程序的缓存失效率和IPC，（2）一个优化算法在给定一个优化目标后生成



(a) 分配布局　　(b) IPC吞吐量

图 1.1　完全重叠、不重叠和部分重叠方案的比较示例

一个优化分配方案。借助于英特尔CAT技术，我们在真机上实现了CAPS并在实际系统中进行了实验评估。在评估实验中，我们把每个并发的程序都运行完毕，同时一些性能指标通过硬件性能计数器（Performance Counter）和高速缓存监控技术（CMT）来收集和记录。CAPS具有高度的灵活性，可以支持很多优化目标和策略。我们在CAPS上实现了5种策略分别锚定5个优化指标：（1）每个程序平均的1000指令失效数（Average MPKI）；（2）IPC吞吐量（Throughput)；（3）相比于独占缓存，每个程序平均的性能损失（Average Slowdown）；（4）兼顾公平的性能损失（Fair Slowdown）；（5）最大的性能损失（Maximum Slowdown）。实验结果表明，相比于自由竞争，在Average MPKI这个指标上，CAPS平均可以降低16.96%的失效数，最好情况下可以降低23.1%；对于Throughput，CAPS平均提升11.11%，最好情况下提升高达31.3%；对于Average Slowdown，CAPS在平均程度上可以减小性能下降8.16%，最优情况下达到11.18%；对于Fair Slowdown，平均可以被减小8.17%，最好情况13.2%；对于Maximum Slowdown，平均可以提升23.24%，最好情况下高达33.42%。

本文提出的CAPS缓存分配优化框架的主要特点和优势有以下几点：

- 在细粒度上实现缓存分配。通过精确的控制分配重叠，CAPS突破了way-partitioning技术的分配粒度限制，实现了细粒度的控制，同时也没有带来额外开销。细粒度分配可以带来更好的优化效果。
- 支持多种优化策略。过往的研究对于不同的优化目标需要制定不同的策略，而在CAPS中，只需要简单地适配优化函数，就可以实现一个新的策略。这大大增强了优化的灵活性。

- 具有良好的可扩展性。随着线程数/核数的增加，可扩展性对于一个优化框架来说格外重要。过去的解决方案多是针对双核/双线程的情景，对于如今动辄十几个核的处理器就会力不从心。而CAPS对于核数少与多的情况都提供了良好的支持性。
- 可以在真实系统中实现。CAPS是一个纯软件的框架，通过CAT技术得以在真实系统上实现。相比于模拟器，我们在真机上可以进行更加充分的实验。同时，CAPS也更容易被应用到实际生产环境中。

据我们所知，CAPS是国内外第一个基于缓存分配技术在真实系统上实现的普适性缓存优化框架，它开创性地利用部分共享缓存空间来增强优化效果。本文的后续部分如下安排：

在第二章中，我们介绍了有关高速缓存的重要概念，CAT技术的背景知识，以及国内外相关工作的研究进展。

在第三章中，我们将重点介绍CAPS的性能预测模型。我们讲探讨该预测方法的基本原理，给出算法设计，并阐述实现中的关键点。

在第四章中，我们提出CAPS的优化算法。我们会阐述该算法如何在给定条件下生成一个优化算法，并给出算法伪代码。同时我们会对CAPS中实现的5个优化策略进行进一步分析。

在第五章中，我们通过大量的实验对CAPS进行全方位的评估。评估先从平均的角度给出综合评估分析，再从个别实验样本的角度进行细致分析。

最后，我们在第六章中，总结了全文的工作成果，并提出未来的研究方向。

第二章　研究背景和目标

2.1　高速缓存的相关概念

在本节中，我们将列举并解释在本文中出现的与高速缓存相关的一些重要概念。

多级缓存

因为内存访问的延迟较高，为了弥补CPU与内存之间的速度差异，现代计算机通常都采用了层次化的高速缓存架构，我们称之为多级缓存（Multi-level Cache）。按照缓存距离CPU 的远近，依次将其称为L1、L2、L3 等。特别的，我们将体系结构上最远离CPU、存取速度最慢的那一级称为底层缓存（Last-Level Cache，LLC），其余的统称为高层缓存（Higher Level Cache）。在多核处理器中，每个处理器核（Core）通常具有各自独立的高层缓存，我们将它们称为核的私有缓存（Private Cache）。底层缓存却通常被多个核（甚至所有核）所共享，我们称其为共享缓存（Shared Cache）。这种设计既有助于在一定程度上保证核与核之间的隔离，同时又能使各个核在底层缓存上进行高效的数据交换，并简化处理器的硬件复杂度。

在多级缓存中，如果某级缓存所存储的数据同时存在于其下一级缓存中，我们就将这下一级缓存称为包含型缓存（Inclusive Cache）。反之，如果在同一时刻相邻两级缓存中没有重复的数据单元，我们就将较低的那级缓存称为排除型缓存（Exclusive Cache）。尽管包含型缓存造成了一定的空间浪费，但却有助于提高缓存同步（Cache Coherence）时的性能。这是因为如果发现要同步的数据不在该缓存中，我们就没有必要把同步操作再向上层传递。而且，对于共享的包含型缓存，我们还可以在其缓存单元中记录该数据还存放在哪些私有缓存中，而避免向所有上层缓存发出请求。所以，目前的商用处理器多采用包含型缓存的设计。

如图2.1所示是一个典型的多级缓存设计，英特尔近年的微处理器架构，包括都Sandy Bridge、Haswell、Skylake，都采用了这一设计。每个处理器核拥有两层独立的私有缓存，第一层私有缓存包括L1 D-cache和L2 I-Cache，分别负责数据和指令缓存；第二层私有缓存L2 Cache。L1、L2缓存每个核都有且独占。底层缓存LLC，也可以被称为L3 Cache，为所有核所共享。L1、L2的缓存大小通常在KB级，L3/LLC的大小通常在MB级，高端处理器中可达数十MB。

相联度

缓存中一个最小的存储单元称为缓存块（Cache Line）。缓存的相联度（Associativity）决定了某个内存地址的数据能存放在哪些缓存块中。

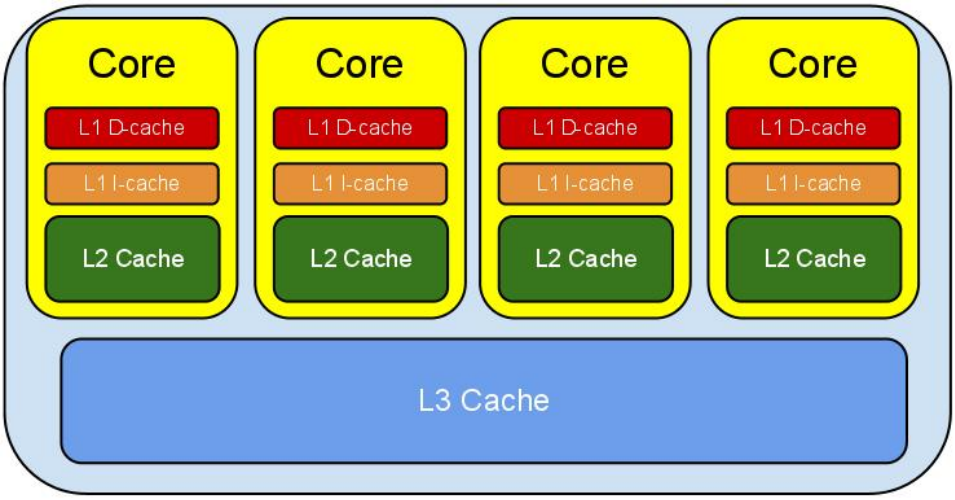


图 2.1　多级缓存的示例

相联度为1的缓存称为直接映射缓存（Direct Mapped Cache）（图2.2(a)）。对于直接映射缓存，硬件会将内存地址的一部分作为索引，将其对应到一个唯一的缓存块。因此，每块内存数据都只能存放在一个特定的缓存块中，而如果两个内存地址的索引部分相同，就将导致冲突（Conflict）。

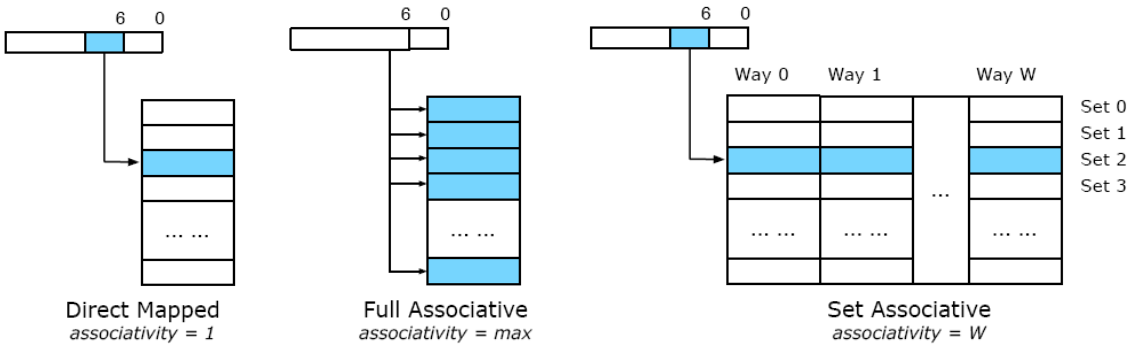
如果缓存的相联度和缓存块的总数相同，则称之为全相联缓存（Full Associative Cache（图2.2(b)）。在全相联缓存中，一个内存地址能缓存在任何一个缓存块中，硬件通过全相联比较器确定该内存地址被缓存的位置。全相联缓存能够使缓存空间得到更有效的利用，但却需要极大的硬件复杂度。

组相联缓存（Set Associative Cache）是以上两种类型的混合（图2.2(c)）。它按照相联度将缓存块平均分为若干缓存组（Set），同组内的各缓存块称为路（Way），路的数目就是缓存的相联度。分配缓存时，硬件首先根据内存地址的一部分确定该数据所对应的缓存组，数据可以缓存在该组内的任何一个缓存块中。硬件设计人员可以通过控制相联度，在冲突失效率及硬件复杂度之间做出权衡。现代处理器多采用这种设计。

管理策略

为了管理全相联缓存和组相联缓存中自由的缓存空间，硬件通常以队列的逻辑形式来组织这些缓存块。当缓存失效（Cache Miss）时，新数据将插入队列的什么位置取决于缓存的插入策略（Instertion Policy）。当要缓存的内容超过缓存容量时，硬件就会按照某种策略从队列中淘汰一个缓存块，这个策略称为替换策略（Replacement Policy）。当缓存命中（Cache Hit）时，如何调整命中的缓存块在队列中的位置取决于缓存的晋升策略（Promotion Policy）。

由于局部性原理，最近最少使用（Least Recently Used，LRU）是最为被广泛使用



(a) 直接映射　　(b) 全相联　　(c) 组相联

图 2.2　缓存的相联度

的缓存队列组织形式。在LRU 队列中，我们将队首的位置称为MRU（Most Recently Used），将队尾的位置称为LRU。那么其管理策略就可以概括为：新数据插入到MRU的位置，从LRU 的位置淘汰数据，命中的数据将晋升到MRU的位置。

由于程序的数据访问在时间和空间上会有局部性的特性，即如果一个信息项正在被访问，那么在近期它很可能还会被再次访问；以及在最近的将来将用到的信息很可能与现在正在使用的信息在空间地址上是临近的。所以LRU策略可以提供较好的缓存管理效率。严格的LRU在实现上比较困难，且会带来较大的额外开销，所以现代处理器多采用近似LRU的管理策略。

2.2　缓存划分的相关研究

在LRU策略下的共享缓存LLC中，不管访问来源与哪个核都被一视同仁，也就是说所有核在竞争使用LLC。然而，竞争的结果往往不是效率最高的结果。因为某些高污染程序，比如流媒体应用，往往会占据大量的缓存资源，从而导致同时运行的其他程序因为没有足够的缓存而性能下降。如何高效地管理和优化共享缓存是学术界非常关心的一个问题。优化共享缓存的一个关键点就是通过缓存的划分，来改善系统的整体性能。我们广泛调研了关于缓存划分技术的相关研究，将其归纳为硬件技术和软件技术两大类。

硬件技术

基于硬件的研究主要通过改良缓存硬件、优化缓存管理策略等方法来完成缓存的优化。

Suh等首先提出了动态缓存划分的思想来优化共享缓存的利用率 [25–27]。他们采用了一种软硬件结合的方法，首先由修改过的硬件动态构造出一个失效率曲线，然后操作系统利用该曲线求出使得总失效率最低的一种划分方案，最后再由CPU通过修改

缓存替换策略完成缓存分割。这种方法带来了很大的硬件开销。

Qureshi等对上述方法加以改进 [**qureshi2006utilityïijN̆qureshi2007adaptive**]，他们将增加单位缓存后某个程序减少的失效数称为效用（Utility），并实现了以最大化全局效用为目标的缓存划分方法。其主要优化包括：为每个核构造单独的失效率曲线图，以提高决策的准确度、用组采样（Set Sampling）来降低硬件开销，以及优化寻找最优划分的贪心算法。试验表明，该方法能够用较小的硬件开销实现平均情况下11%的性能提升。

Rafique等给出了一种通用的、让操作系统自由控制缓存配额的软硬件接口 [20]。他们设计了该接口的体系结构支持，并在模拟器上评测了若干策略。他们认为，这种策略与机制相分离的方法具有更强的适用性。

Srikantaiah等提出了一种自适应组独占（Adaptive Set Pinning）的方法 [24]。它能自动根据需要，将某些缓存组分配给某个处理器核独占使用一段时间，从而同时减少核与核间由于同步操作导致的缓存失效，以及处理器核自身由于容量限制或地址冲突导致的缓存失效。

Xie 等提出了一种称为PIPP的缓存“伪划分”技术 [32]。他们首先用Qureshi 的方法得到一个最优的划分方案，再通过调整缓存的插入策略和晋升策略，使缓存的分配动态平衡在想要的划分比例。他们的实验结果表明，这种方法避免了刚性划分所引起的缓存空间利用不当，能够综合Qureshi工作的优点。

此外，还有一些研究针对不同的优化目标。Iyer等提出了基于服务质量（Quality of Service, QoS）的优化模型。针对多核平台下任务多样性的新趋势，QoS 存储架构能够根据应用的优先级，可控地分配缓存空间和内存带宽等资源 [9]。Kim 等提出了以性能公平性（Fairness）为目标的共享缓存管理策略 [10]。Hsu等总结了基于性能、服务质量以及公平性这三方面的优化目标 [7]，并提出相应的优化策略。

上述这些研究，虽然在优化目标、策略及算法上有所不同，但实现缓存划分的方式都是基于路的划分（Way Partitioning）。路划分技术是将缓存的路（Way）分配给各个核，它的最小划分单位为一个缓存路。之所以路划分技术被广泛使用是因为其设计简单，不需要很复杂的硬件修改。但是它的不足之处在于分配的粒度较粗，一个路往往会有很多缓存块，这在核数较多时会显得力不从心，因为最优划分很可能会且在一个路的中间。对于一个处理器而言，缓存的路数在设计之处就已经确定，而且数目不会太多，当核数增加时，分配的灵活性将会大大下降。当核数等于路数时，每个核有且只能被分配一个路，这就彻底失去了灵活性，只能起到隔离的效果。

一些研究试图改进路划分技术，提高分配粒度。Chang等提出了称为CCP共享缓存的技术 [3, 4]。这种方法从时间和空间两个维度上分配共享的缓存资源。它可以通过

控制处理器核使用某个缓存分区的时间片长短来保证公平性，并通过伸缩每个缓存分区的大小来控制服务质量。通过时间共享来细化划分粒度。Manikantan提出了PriSM通过精确地控制缓存失效概率来细化分配粒度 [16]。还有一些研究抛弃了路划分，采用更加复杂的硬件设计来实现细粒度缓存分配 [21]。

所有的硬件研究因为涉及到硬件修改，所以基本都是在模拟器中进行。

软件技术

软件方面主要依赖于“页面着色”（Page Coloring）这一技术。Lin等首先提出了“页面着色”技术 [11]，在Linux 操作系统上实现了无需硬件支持的缓存分区，它们提出了一种按照失效率和敏感度对应用程序使用缓存的特征进行分类方法，并利用该方法设计了若干组基准程序，评估了页面着色方法对他们的效果。之后多个学者对这一技术进行了更深入的研究和应用 [1, 13, 23, 28, 33]。

页面着色的基本原理是通过操作系统控制页面到缓存块的映射，从而限制缓存被分配的区域。这种技术可以在真实系统上通过软件实现，而且原则上来说可以将分配粒度细化到一个缓存块。然而，目前处理器纷纷采用哈希算法映射物理地址到缓存块，而不是过去的一一对应。在不知道哈希函数的情况下，就无法通过页面着色来控制缓存块的分配。所以现在页面着色技术已经不再适用。

2.3　英特尔高速缓存分配技术

英特尔在2016年发布的第四代至强处理器产品家族中全线引入了资源调配技术（RDT），高速缓存分配技术（CAT）是其中重要的组成部分。高速缓存分配技术首次在商用处理器上实现了对共享缓存（LLC）容量的管理。通过CAT技术，我们可以在软件层面为每个核分配可使用的缓存资源。下面我们简要介绍CAT的使用方法。

CAT通过一个被称为CLOS（Class of Service）的单元来控制缓存的分配。每个CLOS包含一个CBM（Capacity Bitmask）的容量掩码，代表该CLOS可以使用哪些缓存。CBM是由一串连续的1构成01串，1代表当前位这一路的缓存可以被适用，0代表不能被使用。将处理器核与某个CLOS绑定，该核就只能占用被CLOS中的CBM所指定的缓存。

图 2.3是一个CLOS与CBM的例子。CLOS[0]分配有第19到第16路的缓存，CLOS[1]分配有第15到第12路缓存，CLOS[2]分配有11到第6路缓存，这一部分与CLOS[3]发生了重叠。一个核与某个CLOS绑定后就会受到其限制了。

CAT借鉴了前人的研究成果，采用了路划分（Way Partitioning）技术这一设计。前人在模拟器中可以调整缓存路数，然而真实处理器中的路数是固定的。CAT可分配的路数非常有限，在最高端的CPU中仅有20路可供分配，每一路高达几兆字节的缓存空间，所以说CAT的分配粒度很粗。

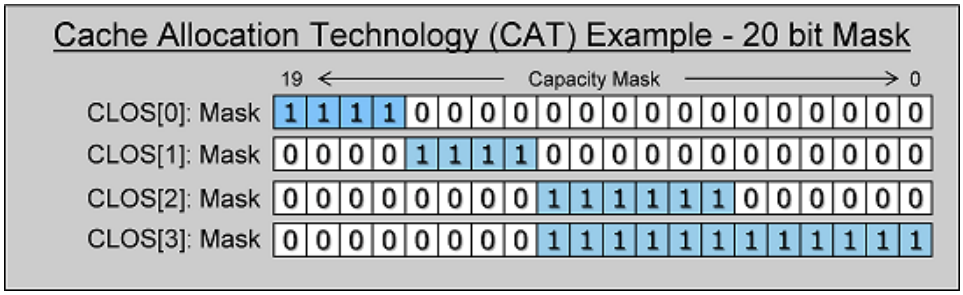


图 2.3　CLOS与CBM的示例

此外CAT技术与以往的技术有两点很大的不同：

- 分配必须是连续的。CBM必须包含连续的1，例如，“0111”是一个有效的分配，而“1011”则不行。
- 分配间允许重叠。例如，可以把“1110”分配给一个核，“0111”分配给另外一个核，中间两路缓存两个核所共享。

这两个特点意味着在CAT中分配的位置也需要被考虑。以往的研究通常只需要考虑分配“多少”缓存，因为分配不一定要连续而且不会重叠。而CAT中因为连续的要求和允许重叠，所以分配在“哪里”也同样重要。

目前关于CAT的研究都集中在QoS方面 [5, 6, 12]。它们主要通过提供给高优先级的程序足够的缓存资源来满足QoS要求，而让低优先级程序共享剩余的缓存。这种思路并不需要细粒度的缓存控制，所以CAT可以轻松胜任。

2.4　研究目标

随着处理器中的核数越来越多，共享缓存的竞争问题也愈发严重。目前，还没有一个可以在真实系统上可以运行的缓存优化框架，我们希望通过我们的努力实现这样一个框架的原型。我们称之为CAPS，它可以满足以下几个方面的要求：（1）**可以在真实系统中运行**；（2）**实现细粒度分配控制**；（3）**良好的可扩展性，在核数较多时依然有很好的适应性**；（4）**具备灵活性，支持多种优化策略**。

共享缓存分配优化的步骤一般包含以下三步：

1. 预测。首先需要在任意分配情况下，对每个并发程序的性能进行预测和评估。性能参数包括失效率（Miss Rate）和周期指令数（IPC）等等。预测的指标用来为后面的决策提供参考依据。
2. 决策。根据预测的指标，和优化目标，做出一个分配决策。针对不同的优化目标，可能会采取不同的策略和算法。
3. 实施。将决策的分配付诸实践，让分配真正产生效果。

若要在真实系统上实施分配，CAT技术是目前唯一的方法。所以CAPS必须倚仗于CAT作为最后实施分配的方法。而分配技术对前两个步骤，即预测和决策又有深远的影响，因为决策所得到的分配必须符合分配技术的要求，而预测又是为决策所服务。如上节所述，CAT技术要求分配必须连续以及有限的可分配资源都为实现细粒度、可扩展和灵活的分配优化带来了极大的困难。然而，CAT允许重叠这一特性却带来很大的操作空间。虽然重叠分配，尤其是部分重叠，为预测模型和分配决策带来了更大的挑战，但是我们通过研究探索，在CAPS中实现了一个全新的预测模型，以及一个基于模拟退火的决策算法，可以支持部分重叠下的CAT分配的性能预测和优化决策，让上述提出的四点目标得以满足。在下一章中，我们将首先介绍CAPS预测模型。

第三章　预测模型

3.1　模型概述

在本章中，我们将介绍CAPS的预测模型。对缓存失效率的预测是优化的基础，它的准确性会直接影响到整个优化框架的效果。由于优化决策依赖于预测模型提供的信息，不准确的预测结果可能会导致错误的分配决策。虽然前人对失效率预测进行了大量工作，但它们都不适用于CAT下的预测。因为允许部分共享，分配间可以部分重叠，这就大大增加了预测的难度。为了应对部分重叠的问题，我们为CAPS推导了一个全新的预测模型。

CAPS预测模型可以较为准确地预测出，多个并发执行的程序在任意CAT分配下，每个程序的缓存失效率（Miss Rate）和周期指令数（IPC）。一个CAT分配包括每个线程/核的分配（CLOS）的集合。CAPS预测模型的输入包括每个并发程序的失效率曲线（Miss Rate Curve，MRC）和访存指令占比（Accesses per Instruction, API），以及加载于它们身上的CAT分配。MRC和API可以描绘出一个程序的局部性和缓存访问频率等特征，这两个指标都可以通过离线采样分析得到，在第3.2节中会详细介绍。MRC刻画了失效率随缓存大小变化的情况，它是描述某个程序的缓存敏感度的一种有效手段。MRC是这样一条曲线，它的横轴是缓存占用，纵轴是失效率。API用来刻画程序的访存频率，它代表了程序对缓存的污染程度，通常访存频率越高，在竞争中越容易占据较多的缓存空间。

因为CAT分配下，一个程序的缓存分配可能会与一个或多个其他程序的分配部分重叠，每个重叠部分就处于竞争使用状态，所以预测的关键在于弄清楚这些重叠部分的竞争结果，即程序在竞争下实际得到的缓存大小。我们通过一个迭代算法解决了这一问题，算法会在第3.3节中详细阐述。算法中的每次迭代相当于模拟一小段时间片中每个程序执行了一定的指令，在这个过程中，它们的缓存占用发生了改变。我们假设每个程序的访存模式都是稳定的，所以在均衡状态下，各个程序的缓存占用量也会达到稳定，这个稳定值就是我们需要的答案。根据真实占用和MRC很容易推导出每个程序的失效率，再根据失效率估计出IPC，就得到了模型的输出。值得一提的是，实际中每个程序的访存模式都会随着运行阶段的变化而变化，CAPS预测模型也可以适应于这种变化，但是需要在线的实时MRC采样。在本文中，我们只关注程序的平均性能，所以只使用离线采样的平均MRC和API作为输入。

在我们对4到15个程序的工作负载进行了多达750次实验，结果表明CAPS预测模

型具有较高的预测准确率，同时还能保持较低的额外开销，具体的实验评估见第五章。

3.2 离线采样分析

本节中，我们将介绍如何通过PIN这一工具离线采样得到程序的MRC和API。研究者们对于如何获取MRC进行了大量的研究，在CAPS中，我们借鉴了基于平均失效时间（Average Eviction Time，AET）的技术 [8]。任何MRC采样技术都需要程序的访存序列，它是构建MRC的基础。在本文中，我们使用PIN [14]这一工具对访存序列进行追踪。

Pin是一款针对x86 指令系统的二进制代码分析工具（Binary Instrumentation Tool）。它能够在不改变原有程序执行逻辑的前提下，在该程序的任何指令前后插入用户自定义的代码片段。Pin 包含引擎（PinEngine）和工具（PinTools）两个部分。PinEngine是一个不开源的可执行程序，是其核心部分，它负责完成二进制代码解析和改写。PinTools是由用户自己编写的一些函数库，定义了代码替换的具体规则、以及要插入的代码片段。当Pin执行时，PinTools 会以模块的形式被动态链接到PinEngine中，二者协同完成整个代码替换。

Pin与AET结合构建MRC的工作流程如下：

1. 被测试的基准程序作为输入被Pin 引擎读入翻译缓存，PinEngine 对它的二进制代码进行静态分析，标记出函数、基本块等；
2. 完成一批代码分析后，PinEngine 会自动调用PinTools 中注册的代码替换回调函数（Instrumentation Callback）。该函数根据用户自己的需要，扫描Pin分析出来的指令流，再调用PinEngine提供的代码替换接口，将自定义的指令回调函数（Execution Callback）插入到程序的指令流中。本例中，我们在所有访存指令之前加入了自己的代码filter_memop(ip, ea)；
3. PinEngine 将修改后的代码片段载入其执行缓存，并跳转执行它。
4. 执行到访存指令时，修改后的代码片段自动调用先前插入的指令回调函数filter_memop()，且PinEngine会计算出该指令的指令指针ip和被访问的内存地址ea，作为参数传递给回调函数。在少数情况下，如果该代码片段试图跳转到未翻译的程序代码，则Pin将获得控制权，并跳回步骤2。
5. filter_memop()的行为非常简单，它只是将ip和ea两个参数放入存放访存序列的缓冲区中（图中的Memory Trace），并返回步骤4继续执行代码片段。只有当缓冲区将要溢出时，才跳转到步骤6。
6. 当被测试的基准程序执行完毕或执行到指定时间点后，输出访存序列和访存指

令占比（API）。

7. 利用AET方法，输出最终的缓存失效曲线MRC。

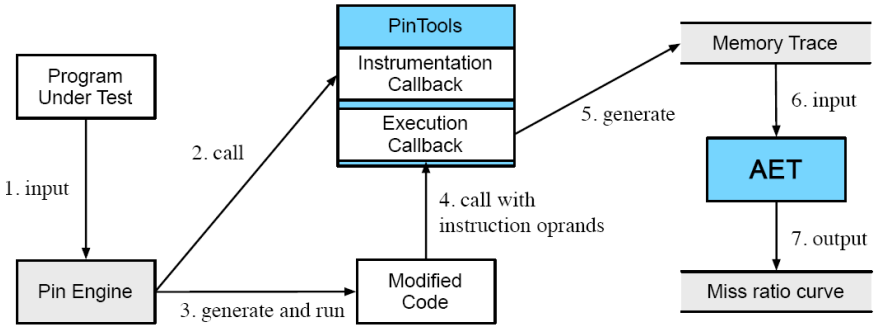


图 3.1　利用Pin和AET构建MRC的流程图

AET是一个先进的MRC采样技术，它可以在很低的时间空间开销下根据访存序列得到一个准确的MRC。虽然额外开销对于离线优化框架来说并没有那么重要，但我们仍然想控制时空开销，因为我们计划在未来将CAPS拓展到在线环境中。AET具有线性的时间复杂度，并且可以通过随机采样来减少运行时间，同时也能保持较高的MRC准确率。

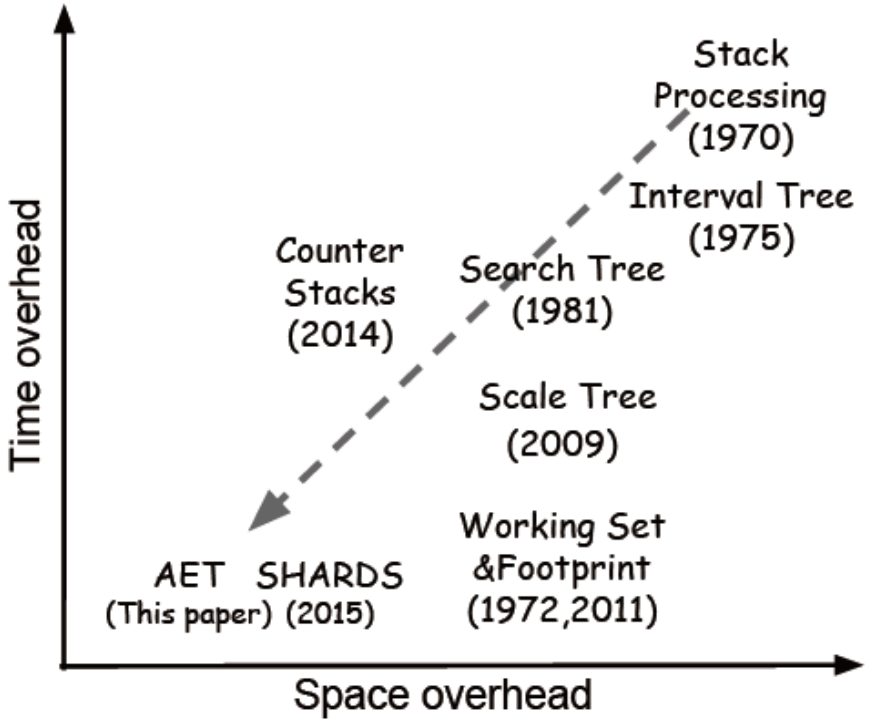


图 3.2　AET与其他技术的时间与空间开销对比

以往的MRC技术多是基于重用距离，它被定义为对同一数据的相邻两次访问间所

间隔的不同数据数。通过构造出重用距离直方图，然后累加得到MRC。但是完整地统计出重用距离直方图会带来巨大的时间和空间开销。从渐进意义上来说，对于N个读写访问到M个不同的地址，构造重用距离直方图的时间复杂度为$O(N\log M)$，空间复杂度为$O(M)$ 17。

基于AET的方法引入了平均失效时间（Average Eviction Time，AET）这一概念。失效时间（Eviction Time）被定义为最近一次访问到失效所经历的时间。LRU缓存可以被看作是一个栈，栈中数据按最近访问时间排序。最近访问多的在栈顶，最近访问少的在栈底。栈底被挤出去的就是被替换掉了。AET实际上就是缓存块从栈顶移动到栈底并出栈的平均时间。AET模型的输入是重用时间而不是重用距离，两者的差别在于，前者并不需要统计两次重用之间不同的访问次数，只需要统计总次数，所以可以通过随机采样的方式大大减少时间和空间消耗，因为只要采样的重用时间分布与真实的分布一样，AET一样可以得到准确的MRC。通过随机采样一小部分的访存，大量的时间和空间开销可以被节省下来。下面我们来推导AET模型。

我们设T_m是一个数据块在到达LRU栈的位置m的平均时间。显然，$T_0 = 0 \,\text{and}\, AET(c) = T_c$。注意这里是所有数据块的平均情况。

设n是所有访存总数，$rt(t)$是重用时间为t的访存数量。$f(t)$是重用时间为t的访存占比。则：

$$f(t) = \frac{rt(t)}{n} \tag{3.1}$$

对于一个访存来说，$P(t)$是它的重用时间大于t的概率：

$$P(t) = \sum_{t+1}^{\infty} f(t) \tag{3.2}$$

现在数据块向LRU栈底部的每次移动可以被视为一个概率事件。从另一个角度，也可以理解成在单位时间数据块向下移动了$P(t)$。这可以理解为移动的速度，在位置m，平均达到时间为T_m，则平均速度$v(T_m)$为：

$$v(T_m) = P(T_m) \tag{3.3}$$

将速度做积分，就可以得到移动的距离。事实上，x从0积分到$AET(c)$，就是缓存的大小，所以有如下公式：

$$\int_0^{AET(c)} P(x)dx = c \tag{3.4}$$

该公式可以这样被证明：

$$
\begin{aligned}
&\sum_{m=0}^{c-1}\int_{T_m}^{T_{m+1}}(v(T_m)-\sum_{t=T_m}^{x-1}f(t))dx \\
&=\sum_{m=0}^{c-1}\int_{T_m}^{T_{m+1}}(P(T_m)-\sum_{t=T_m}^{x-1}f(t))dx \\
&=\sum_{m=0}^{c-1}\int_{T_m}^{T_{m+1}}(P(T_m)-(P(T_m)-P(x)))dx \\
&=\sum_{m=0}^{c-1}\int_{T_m}^{T_{m+1}}P(x)dx \\
&=\int_{T_0}^{T_1}P(x)dx+\int_{T_1}^{T_2}P(x)dx \\
&\quad\ldots+\int_{T_{c-1}}^{T_c}P(x)dx \\
&=\int_{0}^{AET(c)}P(x)dx
\end{aligned}
$$

根据AET，很容易构造出MRC。因为在缓存大小c时的失效率，就是重用时间大于$mr(c)$的概率。MRC构造公式如下：

$$mr(c)=P(AET(c)) \tag{3.5}$$

综上，我们就得到了一个高效的MRC构造方法。对于部分共享的情况，只有MRC是不够的，下一节我们将介绍如何通过一个迭代算法来求解重叠情况下的预测问题。

3.3　迭代预测算法

在分配没有重叠的情况下，得到MRC就完成了预测。因为分配的大小就是某个程序独自占用的的实际大小，根据MRC就可以直接得到失效率。此外，还有一些研究针对完全重叠的情况进行预测 [2, 8, 26, 30, 31]。然而，它们都不能适用于CAT下的部分重叠分配。简单地把部分重叠分配下的每个完全重叠片段看成是一个自由竞争的小缓存块是不正确的。因为CAT是通过缓存失效来驱动的，在CAT下，如果是一个缓冲命中，那么它可以命中在LLC的任何地方，即使是在这个核的分配之外，此时CAT不发挥作用。CAT只在缓存失效时才发生作用，此时该失效只能替换掉发起这个请求的核的分配之内的缓存。所以，对于某个核来说，它发出的访问请求并不是均匀分布在它

的分配中，而是它引起的失效均匀分布到它的分配中。因此，我们并不能脱离整体，而把每个完全竞争的缓存块当成一个独立的完全共享的缓存，而应该在整体上关注真实的缓存占用。

为此，我们推导了一个全新的模型，通过迭代来预测部分重叠情况下的缓存占用和失效率。首先，我们根据每个分配的起点和终点，将整个缓存空间划分成多个缓存段。每个缓存段是一个完全重叠子区域，它们组合在一起构成整个LLC。显然，总的缓存段数小于等于总的路数，因为最多情况下每一路都是不同的缓存段。分段过后，每个程序的分配区域可以被看成几个连续的缓存段。在每个缓存段中，分配中包含这段的程序互相竞争。我们预测算法的基本思路是，通过计算每一个缓存段中各个程序的实际占用，将它们汇总就可以得到总的真实占用。为此，我们需要搞清楚每个缓存段的竞争结果，才能知道每个程序的实际占用。

某程序占用的缓存大小和它造成的失效数息息相关，因为正是失效导致的替换抢夺了缓存空间。失效数与缓存占用同时存在着正向和负向两种关系。失效数越多，意味着更多的缓存占用，但另一方面，更多的缓存意味着更少的失效。这种关系类似控制论中的负反馈概念，最后会达到一个稳定的状态。这个稳定状态就是我们要求的状态。假设程序的访存模式是很定的，那么稳定状态时的缓存占用和失效率就是真实的占用和失效率。CAPS预测模型通过一个迭代算法求出这个稳定状态，算法伪代码如算法1所示。

我们默认每个程序的MRC和API已经得到。该算法的目标是通过迭代过程得到稳定状态下的实际缓存占用。得到实际缓存占用后，很容易通过MRC得到失效率，然后可以估计出IPC。作为迭代的初始状态，我们首先需要给出一个初始占用。这个占用可是随机的，并不影响最后的结果，但是会影响迭代收敛的速度。在CAPS中，我们在每个完全共享的缓存段中使用平均分配作为初始占用。在每次迭代中，我们先根据当前的占用结果计算得到失效率，然后我们再根据当前的失效率推导出下一阶段的占用。直到变化的程度小于一定的阈值，我们认为迭代收敛，此时的占用即是我们要求的稳定状态下的真实占用。

我们引入了一个迭代步长参数*Step*来控制收敛过程。*Step*模拟在冷启动中每次迭代的步长，它也可以被看成公式3.6中的周期数。更大的步长通常意味着更快的收敛速度，但是同时也可能造成某次迭代越过了均衡点，导致迭代在均衡点两侧跳动从而无法收敛。另一方面，较小的步长可能会影响收敛速度，降低预测算法的效率。在CAPS中，我们选择了一个较大的初始*Step*，然后逐渐地降低它，每轮迭代降低5%，直到设定的最低点。这样的话，我们可以在保证收敛到均衡点的同时提升了收敛速度。

下面我们将列出并阐述了预测模型中重要的公式。每轮迭代前半部分通过当前轮

Algorithm 1 Prediction Algorithm

Input: $MRC[i][]$ and $API[i]$ of each program i; a CAT scheme
Output: $MissRate[i]$, $IPC[i]$ for each program i

Partition cache space to shared intervals based on allocations' starting and finishing points
Initialize $occupancy[i][j]$ for program i in interval j = (size of interval j) / (number of programs sharing the interval)
while aggregate change of occupancies > threshold **do**
  /* occupancy to miss rate */
  foreach program i **do**
    $occ = 0$
    foreach interval j **do**
      occ += $occupancy[i][j]$
    end for
    $MissRate[i] = MRC[i][occ]$
    $IPC[i] = 1/(CPI_{base} + MissRate[i] * API[i] * MissPenalty)$
    $Miss[i] = MissRate[i] * API[i] * IPC[i] * step$
  end for
  /* miss rate to occupancy */
  foreach interval j **do**
    $TotalIntervalMiss = 0$
    foreach program i in interval j **do**
      $IntervalMiss[i][j] = Miss[i] * IntervalSize[j]/AllocationSize[i]$
      $TotalIntervalMiss$ += $IntervalMiss[i][j]$
    end for
    foreach program i in interval j **do**
      $occupancy[i][j] = occupancy[i][j] + IntervalMiss[i][j] * (IntervalSize[j] - occupancy[i][j])/IntervalSize[j] - (TotalIntervalMiss - IntervalMiss[i][j]) * occupancy[i][j]/IntervalSize[j]$
    end for
  end for
  if $step > minStep$ **then**
    $step = step * StepReductionRatio$
  end if
end while
return $MissRate[]$, $IPC[]$

的缓存占用来推导失效率、失效数和IPC。这主要涉及到以下两个公式：

$$Misses = MissRate \times API \times IPC \times Step \tag{3.6}$$

$$IPC = \frac{1}{CPI_{base} + API \times MissRate \times MissPenalty} \tag{3.7}$$

上一节的离线采样我们已经得到了每个程序的MRC，根据当前的缓存占用直接查阅MRC就可以得到失效率$MissRate$。有了$MissRate$可以根据公式3.6计算得到失效数$Misses$。而IPC比较难以估计，因为很多因素都可以影响到IPC。这里，我们通过公式3.7来做一个近似估计。CPI_{base}和$MissPenalty$通过真实机器上的实验来估计。CPI_{base}通过一个失效率很低的小benchmark来估计，而$MissPenalty$通过LLC的失效延迟来估计。

注意，公式3.6得到的是该程序总的失效数。因为英特尔处理器使用特殊的哈希函数来处理内存地址到LLC的映射，所以这些失效可以被认为在分配区域内随机分布，换句话说分布是均匀的。某个小缓存段中产生的失效占总体失效数的比例与它的大小占比是相同的。根据总的失效数和小缓存段大小占该程序分配区域的比例，就可以就可以得到每个缓存段的失效数。

每轮迭代的后半部分，我们通过当前的失效数和IPC，来更新缓存占用情况。对于每个缓存段，我们各个击破，因为每个缓存段的竞争情况都是不同的。那么如何根据失效数和IPC来推导新的缓存占用呢？West等在一篇论文中介绍了一种双线程下缓存占用实时预测的方法[29]，该研究是通过硬件实时抓取到失效率来预测两个线程的缓存占用情况。虽然使用场景与我们的场景并不相同，但也有很多共通之处。我们受其启发，建立了一个类似的定理用来计算多个程序的缓存占用。为了简化模型，我们假设所有程序都是单线程的，且在不同的核上执行。

定理：*考虑一个容量为C的LLC，被N个并发程序锁共享。每个程序目前分别占用了$C_1, C_2, ..., C_N$的缓存大小，并且在这一阶段分别产生了$M_1, M_2, ..., M_N$个失效。设M为失效数的总和，则对于程序i来说，它更新后的缓存占用为：$C_i' = C_i + \frac{C-C_i}{C} \cdot M_i - \frac{C_i}{C} \cdot (M - M_i)$.*

证明：首先，我们假设整个LLC空间已经被这N个程序充满。事实上，除了冷启动外，绝大多数时间LLC都是被充满的。这时，如下公式成立：

$$C = \sum_{i=0}^{N} C_i \tag{3.8}$$

其次，我们假设每个缓存块都有均等的概率被替换。虽然在LRU策略下，这个假设通常是不正确的。失效的访存会替换掉最近最少被使用（least-recently-used）的那个缓存块，这就意味着经常被访问的缓存块被替换的概率较小。然而，为了模型简洁性，我们仍然使用这一假设。事实上，这一假设不会给准确率带来很大影响[29]。

当程序i发生了一个失效时，它替换掉的缓存块属于其他程序的概率为：$\frac{C-C_i}{C}$，这就相当于把缓存块从其他程序那里抢夺过来。程序i在单位时间内总共产生了M_i个失效，所以因为失效而抢夺过来的缓存块数量为：$\frac{C-C_i}{C} \cdot M_i$。在另一方面，其他程序的失效也可能从它这里抢夺一部分缓存块。一个缓存块属于程序i的概率为$\frac{C_i}{C}$，其他程序产生的失效数为$(M - M_i)$，所以其他程序从它这里抢夺的缓存块数量为：$\frac{C_i}{C} \cdot (M - M_i)$。在这个阶段过后，该程序占用的缓存块数量变动即为，抢夺来的缓存块减去被抢夺走的缓存块：

$$\Delta C = \frac{C - C_i}{C} \cdot M_i - \frac{C_i}{C} \cdot (M - M_i) \tag{3.9}$$

更新后的缓存占用为：

$$C_i' = C_i + \frac{C - C_i}{C} \cdot M_i - \frac{C_i}{C} \cdot (M - M_i) \tag{3.10}$$

更新后的缓存占用仍然符合公式3.8，所有C_i'之和仍然为C。在CAPS预测模型中，我们对于某个缓存段，使用公式3.10计算每轮迭代更新后的缓存占用。然而将所有缓存段的占用加总，就得到了该程序在当前迭代轮的总缓存占用。此时就完成了一轮迭代。当缓存占用的变化率小于一定的阈值后，我们认为迭代已经收敛，我们要求的稳定状态已经达到。此时，模型输出每个程序预测的失效率和IPC。

第四章　分配优化

4.1　算法综述

本章节中，我们将介绍CAPS的优化算法。该优化算法基于上一章节的预测模型，可以针对一个优化目标，在较短时间内生成一个优化CAT分配。同时，该算法还支持不同的优化目标，我们在CAPS中实现了五个优化策略，在后文中会着重阐述。

缓存的优化问题可以被概括为一句话：给定一个优化目标，找到一个最优分配。但是在CAT技术下的优化问题将面临更大的挑战。相比于过去不考虑部分重叠和位置的分配问题，CAT下的分配拥有极其巨大的搜索空间。之前的优化算法只需要决定每个线程需要被分配多少缓存空间，而CAT下需要决定每个分配是从哪到哪，而且还允许部分重叠。搜索整个解空间显然是不现实的，在时间和空间开销上都是不被允许的。可以证明，在这种情况下找到一个最优分配是一个NP-hard问题。

因此，我们的算法并不寻求全局最优分配，而是只需要求解一个较优解。事实上，由于预测的准确率并不十分精确，所以一个全局上的绝对最优解并没有太大意义，反而在短时间内找到一个较优解有更大的意义。为此，我们从经典的模拟退火算法中吸取智慧，构建了一个基于“模拟退火”的优化算法。我们的实验表明该算法在任何优化目标下都能起到良好的效果，具体实验评估结果见第五章。

4.2　优化目标

优化的目标是一个优化策略锚定的指标，是驱动一个优化算法的重要动力。一个优化指标是对系统的总体优化目标的一个量化，不同的指标侧重点也不同，总的来说可以被概括为三个方面：性能（Performance）、公平（Fairness）和服务质量（QoS）。当然，一个指标也可以兼顾两个方面，但同时兼顾三个方面是不现实的 [7]。优化策略的目标就是将锚定的指标最小化或最大化，同时该指标也用来评估策略的有效性。

前人的研究中提出了许多指标来抽象多个并发程序的整体效能。这些指标大多依赖于IPC和失效率这两个参数，这也是CAPS预测模型会输出这两个参数的原因之一。我们希望我们的优化策略具有灵活性，可以很容易适应多个指标，而不用对不同的指标设计截然不同的策略。事实上，因为预测模型预测出了失效率和IPC，只要是基于这两个参数的指标，我们的优化策略都可以直接适配。在本文中，我们选择实现了五个指标作为样例。这五个指标涵盖了各种场合下的优化需求，包括上述所说的性能、公平和服务质量这三个方面。

我们在CAPS中实现的五个指标为：

- **平均失效数（Average MPKI）**：平均失效数Average MPKI代表平均每1000条指令的失效数（Misses Per 1000 Instructions, MPKI)。MPKI是系统评估中的常用指标之一，平均失效数代表所有并发程序的平均MPKI，它可以体现出该并发系统的缓存利用效率。较小的平均失效数意味着较高的缓存利用效率，所以针对该指标的优化策略目的就是让平均MPKI尽可能的小。另一方面，LLC缓存失效就意味着该访存指令需要访问内存，所以最小化MPKI也意味着降低内存总线的竞争。在下述公式中，我们定义$MissRate_i$为程序i在和别的程序并发执行时的失效率，$APKI_i$是程序i每1000条指令的访存指令数。Average MPKI是一个越小越好的指标。

$$AverageMPKI = \sum(MissRate_i \times APKI_i)/\#program \tag{4.1}$$

- **吞吐量（Throughput）**：吞吐量Throughput被定义为所有程序的IPC之和，这也是一个被广泛使用的指标。针对该指标的策略力求让系统整体的IPC吞吐量最大化。它把所有并发程序看成一个整体，使得整个系统的执行效率最高。但是同时，该指标可能会对一些本身IPC就比较低的程序不太公平，因为降低它们的IPC并不会对整体系统的IPC之和产生非常大的影响。在下述公式中，我们定义IPC_i为程序i在并发负载中的IPC。Throughput是一个越大越好的指标。

$$Throughput = \sum IPC_i \tag{4.2}$$

- **平均效率下降（Average slowdown）**：平均效率下降Average slowdown代表着在平均情况下，程序的在共享LLC与独占LLC的执行时间之比。因为相比于一个程序独占LLC，共享的情况下或多或少都会受到一定的性能损失，所以每个程序的Slowdown一定是大于1的，平均Slowdown自然也大于1。我们定义对于程序i来说，它的Slowdown为$SingleIPC_i/IPC_i$，这里$SingleIPC_i$指的是当它单独运行使用全部LLC时每周期执行的指令数（IPC），IPC_i是在多程序并发负载中的IPC。平均Slowdown的概念与前人研究中多次提到的另一个指标，加权效率提升（Weighted speedup），有很大相似之处 [19, 22]。Weighted speedup把并发执行的程序看成一个整体，使用speedup，这个slowdown的倒数，来概括这个整体因为并发带来的效率提升。但是我们认为，每个并发的程序还可以看作独立的个体，每个程序执行各自不同的任务，这样的话针对单个程序的slowdown更可以反映出程序的执行效率，因为共享LLC势必会导致性能下降，所以一定会引起每个程序或多或少

的Slowdown，我们把所有程序的Slowdown做算术平均，就可以得到该并发负载因为竞争LLC导致的整体性能下降。Average slowdown是一个越低越好的指标。

$$AverageSlowdown = \sum \frac{SingleIPC_i}{IPC_i} / \#program \tag{4.3}$$

- **公平效率下降（Fair slowdown）：**公平效率下降指标Fair slowdown兼顾了整体性能和公平性。这个指标借鉴了多个前人的研究经验 [4, 15]。如果只考虑公平性而无视性能是没有意义的，因为大家效率都很差的话，即使再公平也意义不大。我希望通过指标能在提升性能的基础上兼顾公平。与上一个指标Average slowdown不同之处在于，本指标被定义为各个程序Slowdown的调和平均。调和平均鼓励大家的相差尽量小，使得各个程序的slowdown得到均匀的改善。Fair Slowdown是一个越小越好的指标。

$$FairSlowdown = \#program / \sum \frac{IPC_i}{SingleIPC_i} \tag{4.4}$$

- **最大效率下降（Maximum slowdown）：**最大效率下降Maximum slowdown指代所有并发程序中的slowdown最大的那一个，它兼顾了性能与服务质量（QoS）。事实上，QoS是比较难以被量化的，因为判断哪些程序优先级较高、哪些程序优先级较低本身就比较主观。在本文中，我们不讨论程序间优先级不同这一主观因素，我们把并发负载看成一个木桶，把QoS定义成木桶中最短的那块短板，也就是Slowdown最高的那个程序。这种表达QoS的方法也被之前的研究者所使用 [16]. Maximum slowdown是一个越小越好的指标。

$$MaxSlowdown = \max(\frac{SingleIPC_i}{IPC_i}) \tag{4.5}$$

4.3　优化算法

Algorithm 2 Optimization Algorithm

Input: concurrent programs and a metric function
Output: a near-optimal CAT scheme

Profile $MRC[i][]$ and $API[i]$ for each program i
Initialize temperature T and a random allocation $scheme$
$IPC[]$,$MissRate[]$ = Predict($scheme$)
$metric$ = CalculateMetric($IPC[]$,$MissRate[]$)
while $T < T_{min}$ **do**
  $scheme'$ = RandomNeighbor($scheme$)
  $IPC[]$,$MissRate[]$ = Predict($scheme'$)
  $metric'$ = CalculateMetric($IPC[]$,$MissRate[]$)
  if $metric'$ is better than current $bestMetric$ **then**
    $bestMetric = metric'$
    $bestScheme = scheme'$
  end if
  if $metric$ is lower-is-better **then**
    $diff = metric' - metric$
  else
    $diff = metric - metric'$
  end if
  if $diff < 0$ **or** $\exp(-diff/(k * T)) \leq Random(0, 1)$ **then**
    $metric = metric'$
    $scheme = scheme'$
  end if
  $T = T * TemperatureReductionRatio$
end while
return $bestScheme$

第五章　实验评估

第六章　总结

参考文献

[1] Reza Azimi, David K Tam, Livio Soares *et al.* "*Enhancing operating system support for multicore processors by using hardware performance monitoring*". *ACM SIGOPS Operating Systems Review*, **2009**, *43*(2): 56–65.

[2] Dhruba Chandra, Fei Guo, Seongbeom Kim *et al.* "*Predicting inter-thread cache contention on a chip multi-processor architecture*". In: *11th International Symposium on High-Performance Computer Architecture*, **2005**: 340–351.

[3] Jichuan Chang. "*Cooperative caching for chip multiprocessors*". In: *In Proceedings of the 33nd Annual International Symposium on Computer Architecture*, **2006**.

[4] Jichuan Chang and Gurindar S Sohi. "*Cooperative cache partitioning for chip multiprocessors*". In: *ACM International Conference on Supercomputing 25th Anniversary Volume*, **2014**: 402–412.

[5] Liran Funaro, Orna Agmon Ben-Yehuda and Assaf Schuster. "*Ginseng: Market-Driven LLC Allocation*". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, **2016**: 295–308.

[6] Andrew Herdrich, Edwin Verplanke, Priya Autee *et al.* "*Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family*". In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, **2016**: 657–668.

[7] Lisa R Hsu, Steven K Reinhardt, Ravishankar Iyer *et al.* "*Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource*". In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, **2006**: 13–22.

[8] Xiameng Hu, Xiaolin Wang, Lan Zhou *et al.* "*Kinetic Modeling of Data Eviction in Cache*". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, **2016**: 351–364.

[9] Ravi Iyer. "*CQoS: a framework for enabling QoS in shared caches of CMP platforms*". In: *Proceedings of the 18th annual international conference on Supercomputing*, **2004**: 257–266.

[10] Seongbeom Kim, Dhruba Chandra and Yan Solihin. "*Fair cache sharing and partitioning in a chip multiprocessor architecture*". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, **2004**: 111–122.

[11] Jiang Lin, Qingda Lu, Xiaoning Ding *et al.* "*Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems*". In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, **2008**: 367–378.

[12] David Lo, Liqun Cheng, Rama Govindaraju *et al.* "*Heracles: improving resource efficiency at scale*". In: *ACM SIGARCH Computer Architecture News*, **2015**: 450–462.

[13] Qingda Lu, Jiang Lin, Xiaoning Ding *et al.* "*Soft-olp: Improving hardware cache performance through software-controlled object-level partitioning*". In: *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, **2009**: 246–257.

[14] Chi-Keung Luk, Robert Cohn, Robert Muth *et al.* "*Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, IL, USA: ACM, **2005**: 190–200. http://doi.acm.org/10.1145/1065010.1065034.

[15] Kun Luo, Jayanth Gummaraju and Manoj Franklin. "*Balancing thoughput and fairness in SMT processors*". In: *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on*, **2001**: 164–171.

[16] Raman Manikantan, Kaushik Rajan and Ramaswamy Govindarajan. "*Probabilistic shared cache management (PriSM)*". In: *ACM SIGARCH computer architecture news*, **2012**: 428–439.

[17] Frank Olken. *Efficient methods for calculating the success function of fixed-space replacement policies* [techreport], **1981**.

[18] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt *et al.* "*Adaptive insertion policies for high performance caching*". In: *ACM SIGARCH Computer Architecture News*, **2007**: 381–391.

[19] Moinuddin K Qureshi and Yale N Patt. "*Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches*". In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, **2006**: 423–432.

[20] Nauman Rafique, Won-Taek Lim and Mithuna Thottethodi. "*Architectural support for operating system-driven CMP cache management*". In: *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, **2006**: 2–12.

[21] Daniel Sanchez and Christos Kozyrakis. "*Vantage: scalable and efficient fine-grain cache partitioning*". In: *ACM SIGARCH Computer Architecture News*, **2011**: 57–68.

[22] Allan Snavely and Dean M Tullsen. "*Symbiotic jobscheduling for a simultaneous mutlithreading processor*". *ACM SIGPLAN Notices*, **2000**, *35*(11): 234–244.

[23] Livio Soares, David Tam and Michael Stumm. "*Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer*". In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, **2008**: 258–269.

[24] Shekhar Srikantaiah, Mahmut Kandemir and Qian Wang. "*SHARP control: controlled shared cache management in chip multiprocessors*". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, **2009**: 517–528.

[25] G Edward Suh, Srinivas Devadas and Larry Rudolph. "*A new memory monitoring scheme for memory-aware scheduling and partitioning*". In: *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, **2002**: 117–128.

[26] G Edward Suh, Srinivas Devadas and Larry Rudolph. "*Analytical cache models with applications to cache partitioning*". In: *ACM International Conference on Supercomputing 25th Anniversary Volume*, **2014**: 323–334.

[27] G Edward Suh, Larry Rudolph and Srinivas Devadas. "*Dynamic partitioning of shared cache memory*". *The Journal of Supercomputing*, **2004**, *28*(1): 7–26.

[28] David Tam, Reza Azimi, Livio Soares *et al.* "*Managing shared L2 caches on multicore systems in software*". In: *Workshop on the Interaction between Operating Systems and Computer Architecture*, **2007**: 26–33.

[29] Richard West, Puneet Zaroo, Carl A Waldspurger *et al.* "*Online cache modeling for commodity multicore processors*". *ACM SIGOPS Operating Systems Review*, **2010**, *44*(4): 19–29.

[30] Xiaoya Xiang, Bin Bao, Tongxin Bai *et al.* "*All-window profiling and composable models of cache sharing*". In: *ACM SIGPLAN Notices*, **2011**: 91–102.

[31] Xiaoya Xiang, Bin Bao, Chen Ding *et al.* "*Linear-time modeling of program working set in shared cache*". In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, **2011**: 350–360.

[32] Yuejian Xie and Gabriel H Loh. "*PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches*". In: *ACM SIGARCH Computer Architecture News*, **2009**: 174–183.

[33] Xiao Zhang, Sandhya Dwarkadas and Kai Shen. "*Towards practical page coloring-based multicore cache management*". In: *Proceedings of the 4th ACM European conference on Computer systems*, **2009**: 89–102.

附录 A　附件

pkuthss 文档模版最常见问题：

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，[**test-zh**]、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

致谢

pkuthss 文档模版最常见问题：

\cite、\parencite 和 \supercite 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：**test-en**，[**test-zh**]、[test-en, test-zh]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 openany 选项。

如果编译时不出参考文献，请参考 texdoc pkuthss“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在□一年/□两年/□三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日