计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(1): 15−27, 2026

# 面向多线程应用的智能缓存分配方法

何铭健　王　桦
（武汉光电国家研究中心（华中科技大学）　武汉　430074）
（mingjianhe@hust.edu.cn）

# Intelligent Cache Partitioning Method for Multithreaded Applications

He Mingjian and Wang Hua
（*Wuhan National Laboratory for Optoelectronics* (*Huazhong University of Science and Technology*), *Wuhan* 430074）

**Abstract**　Multithread programming models are widely employed to facilitate application execution on multiple cores. However, interference caused by competition for the last level cache (LLC) among concurrently executed threads can lead to performance degradation. Intel cache allocation technology (CAT) provides a mechanism to assign cache ways to cores for achieving cache isolation. Unfortunately, prior studies on CAT-based partitioning are not applicable to multithreaded applications due to two reasons. Firstly, they are tailored for multi-application scenarios rather than single-application scenarios involving multiple related threads. Secondly, they are designed for improving IPC (instructions per cycle), which is not a suitable performance indicator in multithreaded scenarios. To bridge this gap, we present LPart, an intelligent cache partitioning technique for multithreaded applications that significantly improve throughput in multicores with way-partitioned caches. LPart leverages deep reinforcement learning to assign an appropriate number of LLC ways for different threads within an application, without needing any prior knowledge of the application characteristics. We evaluate LPart on a microbenchmark, Redis, a commercial production-ready distributed storage system, and multiple application scenarios. The experimental results indicate that, compared with the default configuration, LPart achieves speedups up to 26.9%, 8.1%, 9.8%, and 24.1% respectively without requiring any changes to the operating system.

**Key words**　cache partitioning; multicore architecture; multithreaded application; deep reinforcement learning; resource allocation

**摘　要**　多线程编程模型被广泛用于促进多核上的应用程序执行。然而，并发执行的线程对最后一级缓存（last level cache, LLC）的竞争造成的干扰可能会导致性能下降。英特尔缓存分配技术（cache allocation technology, CAT）提供了一种机制为不同线程实时分配缓存。然而，有2个原因导致现有分配方法并不适用于多线程应用。首先，它们是为多应用场景量身定制的，而不是为涉及多个相关线程的单个多线程应用场景设计的。其次，它们旨在提高每周期指令数（instruction per cycle, IPC），这不是一个在多线程场景中合适的性能指标。为了解决这个问题，提出了LPart，这是一种用于多线程应用程序的智能缓存分配技术，通过分配缓存显著提高了系统的吞吐量。LPart利用深度强化学习为应用程序中的不同线程分配适当数量的缓存空间。在微基准测试、Redis、商用分布式存储系统和多种应用场景上评估LPart的实验结果表明，与默认配置相比，LPart分别实现了26.9%，8.1%，9.8%，24.1%的性能提升。

**收稿日期**：2025-01-01；**修回日期**：2025-08-07
**基金项目**：国家自然科学基金项目 (62172180)
This work was supported by the National Natural Science Foundation of China (62172180).
**通信作者**：王桦（hwang@hust.edu.cn）

**关键词** 缓存分配; 多核架构; 多线程应用; 深度强化学习; 资源分配

**中图法分类号** TP302

**DOI:** 10.7544/issn1000-1239.202550001 **CSTR:** 32373.14.issn1000-1239.202550001

随着集成电路工艺趋近物理极限, 多核架构成为突破“功耗墙”的主流解决方案[1]。现代处理器通过集成数十个计算核心, 利用线程级并行提升指令吞吐量。然而, 多个线程在共享缓存时可能会引发争夺, 进而导致性能下降的问题[2-4]。

已有研究从硬件和软件 2 个维度提出缓存优化方案[5]。硬件方法通过修改硬件来实现缓存分配、精细化缓存插入和替换策略[6-8], 并通过仿真实验进行评估。软件技术则通过操作系统级的页面着色、内存重映射等手段优化缓存利用率[9-11]。近期, 英特尔公司推出了一种缓存分配技术(cache allocation technology, CAT), 该技术在运行时将最后级缓存(last level cache, LLC)分配给不同的线程。基于 CAT, 研究人员提出了几种资源管理方案以实现不同的目标, 如优化系统吞吐量、提高客户端满意度或确保公平性[12-14]。这些研究的实验结果表明, 这些方法可以显著提高多应用场景中的吞吐量或公平性。

本研究发现, 除多应用混部场景的缓存资源竞争外, 单一多线程应用内部同样存在显著的缓存干扰现象。现有基于 CAT 的划分策略难以直接迁移至该场景, 其核心局限性源于多线程与多应用环境的 2 个本质差异。第 1 个差异是数据共享机制的影响。传统 CAT 技术通过应用级缓存敏感性分析, 为高缓存依赖的应用分配更多 LLC 空间, 以提高系统每周期指令数(instruction per cycle, IPC)总和。该策略隐含应用间数据独立性假设, 当应用间无数据交互时可实现有效优化。然而, 多线程应用内部存在复杂的线程间数据共享机制, 一个线程的缓存行为可能通过共享内存空间直接影响其他线程的缓存命中率。第 2 个差异是线程间逻辑依赖的干扰。多线程应用中线程间普遍存在显式或隐式的同步机制(如屏障同步、生产者-消费者模型), 导致线程执行具有强时间关联性。这种逻辑依赖使得线程性能不仅取决于自身缓存配额, 还与其他线程的执行进度密切相关。传统基于 IPC 的分配策略无法捕捉这种动态依赖关系。此外, 多线程应用的性能评估体系与传统方法存在本质差异。衡量多线程应用性能的关键指标不是线程 IPC 总和, 而是系统吞吐量(如事务处理量、数据吞吐量)。IPC 对于多线程应用来说可能具有误导性, 程序的性能与 IPC 并没有直接关系。在多线程应用中, 每个线程负责不同的任务, 对应用性能产生不同的影响。IPC 的总和并不反映应用的性能, IPC 的增加不一定代表性能提升, IPC 的减少也不一定代表性能下降。这种非线性性能映射关系使得传统基于 IPC 的优化目标失去指导意义。

针对这一挑战, 本文提出了一个端到端的智能缓存分区系统—— LPart, 它利用深度强化学习的方法对缓存分区和系统性能的关系进行学习, 解决了多线程场景下的缓存争用的问题。本文的主要贡献包括 3 个方面:

1)干扰量化研究。通过微基准测试揭示多线程应用内部存在缓存干扰, 最佳分配方案的性能比共享缓存方案高, 其提升可达 28.2%。

2)动态聚类策略。提出了基于数据并行的线程聚类算法, 将训练开销降低到 1/6。

3)端到端强化学习框架。这是第 1 个使用深度强化学习在线程间进行缓存分配的端到端自动缓存分配系统。通过大量实验, 我们展示了 LPart 在运行时动态适应不同工作负载的能力。

## 1 背景介绍

### 1.1 相关工作

在硬件支持实时缓存分配技术(如 Intel CAT)成熟之前, 缓存分配研究主要通过硬件架构级优化[6-8]或操作系统级干预[9-11]实现。这些研究的实验结果表明, 缓存分配可以用来提高性能和公平性。

随着现代处理器集成硬件缓存分区功能, 研究重点转向如何利用硬件级资源隔离特性提升系统性能。由于商用处理器实现的分配最小粒度是路(way), 其过粗的粒度未能显著提升性能。KPart[2]设计了一种聚类算法来解决这个问题。KPart 利用在线分析获取每个应用的缓存资源和性能表现的曲线。通过估算共享缓存分区的性能损失, KPart 生成了多种聚类方案, 并自动选择最佳聚类方案并在这些聚类之间分配缓存以提高系统吞吐量。DCAPS[15]则通过 3 个组件实现缓存分配方案: 1)能够实时构建工作负载缺失率曲线的软件工具; 2)可以估计应用在

与其他应用部分共享缓存分区时的缓存占用情况的模型；3）近似最优缓存分配方案的搜索算法。

近期研究进一步探索缓存分配与其他系统特性的协同优化。CPpf[16]提出预取感知的分区策略。dCat[4]则会周期性地对工作负载进行分类，并动态重新分配缓存空间以提升系统性能。CPA[17]通过识别应用中的阶段实现了更高的性能提升。Lucia等人[18]针对非包容式LLC的缓存污染问题，提出了Cache-Poll，考量了并发应用程序的污染行为和缓存需求后，动态地为不同的应用分配L3缓存空间。

此外，还有研究者针对系统公平性来研究。CoPart[19]动态分析应用的特征，并根据特征给不同应用分配LLC和内存带宽，以显著提高整体公平性。LFOC将应用分为3类，并模拟最优缓存聚类方案以提高公平性[20-21]。SATORI[22]通过高效平衡计算核心、LLC方法、内存带宽和功率限制资源，实现了吞吐量与公平性的最佳平衡。Navarro-Torres等人[23]发现了一种特定类型的应用，在其分配的LLC空间减少时性能影响最小，但内存带宽消耗显著增加，对系统整体性能造成了负面影响。基于这一发现，他们提出了对特定应用的LLC空间使用该应用和对内存流量进行限制的策略。Divide&Content[24]通过动态线程调度和缓存分区优化NUMA多核系统的资源公平性，实验显示优化后系统不公平性降低17%以上。

在云环境中，通常存在多个租户或应用同时运行并伴随着资源争用的情况。针对这种情况，研究人员提出了一系列资源调度方案，以确保服务质量目标[25-27]。ELIS[28]系统旨在解决微服务在数据中心内和跨数据中心的调度问题，确保服务质量，同时最小化网络带宽和计算资源使用。Orchid[29]通过上下文多臂老虎机和轻量级在线学习算法，对资源分区问题进行建模并学习到最优分区配置。多任务计算中，同样存在共享缓存争用影响任务执行性能的情况。Scalpel[30]提出基于共享缓存局部性的任务分组和2级调度策略。通过计算任务的内存足迹来量化共享缓存局部性，并以启发式方法分组任务，减少组内争用，同时保持负载平衡。FaaSCtrl[31]利用强化学习动态调整实时优先级和核心分配，使无服务器平台的尾延迟降低36.9%并使标准差减少44.6%。SurgeGuard[32]结合内核级快速响应和用户态策略处理微服务负载激增时的快速扩展，降低了8%资源使用量。Mosaic[33]提出微架构资源分区和调度器，使无服务器吞吐量提升225%并使功耗降低22%，同时保持与传统工作负载的兼容性。

### 1.2 多线程应用缓存争用机制分析

本研究发现，多线程应用内部的缓存资源竞争显著影响系统性能，其作用机制与多应用场景存在本质差异。为揭示共享缓存策略的次优性，本文从程序特性和线程交互2个维度展开分析，并通过微基准测试实验来验证理论假设。

1）关键路径线程主导性。多线程程序的性能由关键路径线程（critical path thread）决定，其定义为程序执行过程中延迟最长的线程[34]。加速关键路径线程可带来最大的系统性能增益。然而，共享缓存策略基于全局访问热度进行缓存，可能导致关键路径线程的数据因局部性较低被提前替换。此时，对关键路径线程的相关数据进行保护可以提高关键路径线程的性能从而提高整体性能。

2）线程交互的双重效应。线程间交互可分为建设性交互（constructive interaction）和破坏性交互（destructive interference）。当一个线程缓存的数据被另一个线程使用时，称为“建设性交互”。而当一个线程在缓存数据时将其他线程未来可能使用的数据逐出缓存，这被称为“破坏性交互”。在缓存部分线程的数据时，缓存效益可以潜在地提高其他线程性能。同时，限制部分侵略性强的线程缓存数据可以减少缓存污染，从而实现性能提升。线程交互的影响下，仅仅对关键线程分配更多缓存并不能实现最大的性能提升。我们需要考虑线程间的交互才能得到最理想的分配方案。

为量化上述特性对缓存分配的影响，本研究设计6组单生产者-单消费者微基准测试。微基准测试是一组具有轻微数据访问变化的单生产者-单消费者程序。生产者（producer）生成随机值，基于随机值从数组$A$读取数据，并将随机值和读取值的总和放入无锁队列。消费者（consumer）从无锁队列中读取数值，并根据该值索引数组$B$中相应的数据。数组中的数据均为int类型，基于随机值从数组$A$和数组$B$中完整读取数据被视为一个事务，无锁队列由生产者和消费者共享。这2个线程使用Linux sched setaffinity系统调用映射到不同的核心。我们在一台装有Intel® Xeon® CPU E5-2630 v4的服务器上进行实验。该处理器具有10个核心，主频为2.20 GHz，20路25 MB的三级缓存。表1描述了6个微基准测试之间的具体差异。

图1展示了6种微基准测试在完全共享（full share）与线程隔离下的静态最优方案（optimal under isolation）配置下的事务处理速率TPS（transactions per second）

**Table 1 Details of Microbenchmarks**

**表 1 微基准详细说明**

| 微基准实验编号 | 基准情况 |
|---|---|
| 1 | 生产者和消费者访问大小为 30 MB 的同一数组。 |
| 2 | 生产者和消费者分别访问大小为 25 MB 的不同数组。 |
| 3 | 生产者和消费者分别访问大小为 25 MB 和 50 MB 的数组。 |
| 4 | 生产者首先生成 1 个随机数，然后根据随机数访问数组 *A* 中的 1 个整型数据，然后访问数组 *B* 中的 1 个整型数据。消费者只会访问数组 *B* 中的数据 1 次。数组 *A* 和数组 *B* 的大小分别为 1 000 MB 和 30 MB。 |
| 5 | 生产者和消费者各自读取数据 2 次，数据读取有一定的重叠。生产者分别从数组 *A* 和 *C* 中读取 1 个整型数据，而消费者分别从数组 *B* 和 *C* 中读取 1 个整型数据。数组 *A*，*B*，*C* 的大小分别为 15 MB，15 MB，10 MB。 |
| 6 | 与微基准实验 5 基本一致，只是数组 *A*，*B*，*C* 的大小分别更改为 10 MB，25 MB，10 MB。 |

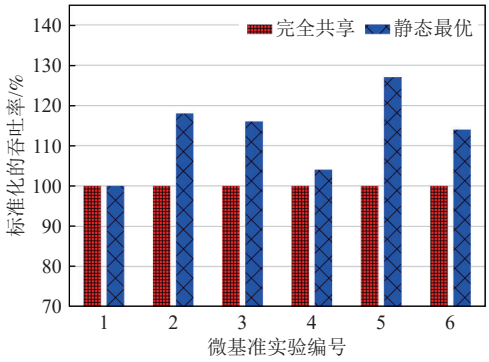



Fig. 1 Standardization performance of different workloads under two cache allocation schemes

图 1 不同工作负载在 2 种缓存分配方案下的标准化性能

对比。实验结果表明，除数据完全共享的微基准实验 1 外，其余场景下的隔离缓存策略均实现性能增长，最高可达 28.2%。下面从程序特性和线程交互 2 个角度分析这种收益的来源。

1)关键路径线程优先的重要性。在微基准实验 2 中，尽管生产者与消费者具有相似的访问模式(独立 25 MB 数组)，但生产者因包含随机数生成的计算开销而成为程序关键路径。通过将 LLC 的 17 路分配给生产者(消费者 3 路)，最终使整体 TPS 提升 18.7%。这验证了优先关键路径线程策略的有效性。

2)关键路径线程的动态性。当消费者数组大小扩展至 50 MB(微基准实验 3)时，最优分配方案调整为消费者 12 路及生产者 8 路。从优化缓存命中率的角度来讲，消费者数据的重用距离增长了，应该少分配一些缓存才能最大化缓存命中率。然而实际是消费者的命中率在微基准实验 3 中比在微基准测试 2 中更低，使其在原有最优的配置方案下(生产者 17 路，消费者 3 路)成为了关键路径线程，因此将更多的缓存分配给消费者可以获得性能提升。该现象表明，多线程程序的关键路径线程会随着负载变化而发生变化。

3)线程交互的双重效应。微基准实验 4，5，6 的独特访问模式揭示了协同缓存效应的优化潜力。以微基准实验 4 为例(生产者随机访问 1 000 MB 数组 *A* 和 30 MB 数组 *B*，消费者仅访问数组 *B*)，按照前述分析，根据关键路径线程的需求，应该给予生产者更多的缓存空间。然而，该实验的最优隔离缓存分配方案是将 1 路分配给生产者，而将 19 路分配给消费者。这是因为消费者线程缓存的数据可以通过构造性交互提升生产者的性能，并且提高消费者的缓存命中率。

通过这个实验，我们可以发现在多线程程序中依然存在缓存争用的情况，通过给不同的线程分配合适的缓存空间可以提升系统的性能。然而，现有缓存分配方案并不能适应多线程程序内部的数据共享与变化，我们会在 1.3 节中详细说明。

### 1.3 现有分配方法不适用于多线程程序的原因

现有的解决方案通过启发式方法解决多个应用程序之间的缓存干扰。这些工作对应用程序进行分析，识别应用特性，然后通过启发式方法进行缓存空间的聚类和分配。尽管这些方法已被证明能够在多应用场景中提高性能，但在多线程应用中无法生效。

图 2 展示了负载 3 在不同方案下的指令处理和事务处理的加速效果。其中，完全共享是所有线程共享所有缓存空间的默认方案。KPart+基于 KPart，其指标由 IPC 修改为 TPS。此外，穷举法通过测试所有可能的配置方案，获取最佳的静态分配，且不同线程间没有共享缓存。实验结果揭示现有解决方案的两大核心局限性。

首先，现有解决方案不能学习到线程间复杂的依赖关系。如图 2 所示，现有方法远未达到最佳静态的效果。尽管 KPart+通过修改指标缓解了这一问题，但距离最佳静态方案仍有很大差距。这是因为现有方案是针对多应用场景设计的。它们对每个应用进行单独的缓存敏感性分析，并为缓存敏感的应

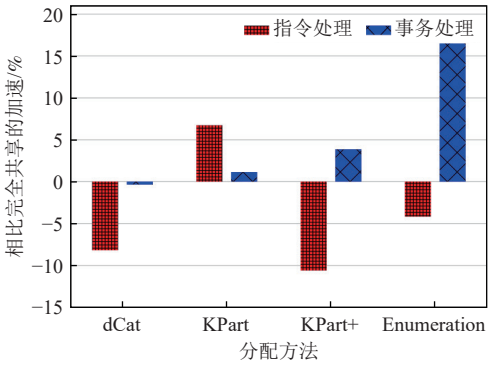



Fig. 2 Speedup effect of instruction processing and transaction processing under different allocation schemes

图 2 不同分配方案下指令处理和事务处理的加速效果

用提供更多的缓存空间，从而提升整体性能。这种做法是合理的，因为不同应用之间没有逻辑依赖和复杂的关系。与此不同的是，数据在同一程序的不同线程之间是共享的，并且线程之间可能存在逻辑依赖。一个线程的性能不仅受其 LLC 占用的影响，还受其他线程 LLC 占用的影响。此外，很难仅凭程序线程的加速比来计算程序的加速比。因此，现有的方法，通过学习程序性能与其缓存占用之间的关系，在多线程应用中表现不佳。为了学习不同线程之间复杂的关系，现有方法需要对所有分配方案进行分析，但是由于组合的数量庞大，这在实际中并不可行。

其次，IPC 对于多线程应用程序具有误导性。如图 2 所示，应用的指令处理加速与事务处理加速的关系复杂，例如在测试中，KPart 的 IPC 最高，但其性能比最佳静态方案低 15%。以 IPC 作为性能指标对单线程程序有着不错的效果。然而，在多线程应用中，每个线程负责不同的任务，对应用性能产生不同的影响。IPC 的总和并不能反映应用的性能。IPC 的增加并不一定代表性能提升，IPC 的减少也不一定代表性能下降[35]。

### 1.4 选择深度强化学习的原因

我们选择深度强化学习来解决多线程应用的缓存分配问题主要有 5 个原因：

1)深度强化学习具有强大的记忆能力。与现有的启发式方法不同，深度强化学习能够基于历史信息和当前状态快速制定分配方案，而无需在每个阶段重新学习。

2)深度强化学习具有强大的适应性，能够适应工作负载和优化目标的变化。一方面，启发式方法根据先前阶段的访问模式做出决策，更适用于过去的工作负载。当工作负载发生变化时，这些方法可能无法适应。深度强化学习则考虑不同的状态及其之间的转移，使其能够适应工作负载的变化。另一方面，优化目标可能会随着时间发生变化。理想情况下，基于深度强化学习的缓存分配器应能根据奖励函数的变化适应不同的优化目标。

3)深度强化学习具有更好的泛化能力。缓存分配与应用性能之间的关系复杂，现有的任何方法或公式都无法准确描述这种关系，深度神经网络具有捕捉这种关系的能力，从而对未见过的场景做出合理的分配方案。

4)深度强化学习具有更强的鲁棒性。性能波动在实际中很常见，巨大的方差会影响分配方法的性能。深度学习通过大量样本进行学习可以有效解决这个问题。

5)尽管深度强化学习训练过程对计算资源需求较高，但是我们提出了线程聚类的方法，使其训练成本大大降低至可接受范围。

## 2 将深度强化学习应用于缓存分配的挑战

尽管强化学习的基本框架是相同的，但在实际应用中仍然会遇到 3 个主要挑战。本节将讨论这些挑战以及解决方案。

### 2.1 训练中的低效率问题

在缓存分配中，深度强化学习面临的最大问题是搜索空间过大。当前大规模多线程软件通常拥有数百到数千个线程，为每个线程单独分配缓存空间会导致搜索空间庞大，使模型变得庞大且学习速度缓慢。例如，在为 20 个线程分配缓存空间时，每个线程可以选择将其缓存空间的起始或结束路数增加或减少 1，或者不改变其缓存分配，总共有 5 种动作。当所有线程组合在一起时，就有 $20^5$ 种可选择的动作。然而，本文发现某些应用中经常存在一些运行时间短暂且几乎没有数据访问的线程，改变它们的缓存大小对应用性能的影响很小。针对这一问题，本研究提出轻量级线程过滤机制与基于域分解的线程聚类策略，将搜索空间进行高效压缩。

1)轻量级线程过滤机制。在多线程应用中，不同的任务会导致线程表现出不同的特征。其中一些线程的 CPU 使用时间较短，数据访问较少。由于 CPU 使用时间短，这些线程对整个应用的性能影响较小。由于数据访问量较低，线程的性能对缓存空间的变化也不敏感。因此，对于 CPU 使用时间和数据访问量较低的线程，可以进行过滤以压缩搜索空间。

2)基于域分解的线程聚类策略。将多个线程聚

合到 1 组中以分配缓存空间可以有效降低算法的复杂性。与现有的分配方法不同，本文利用了多线程程序的一个特点以高效地完成聚类。在多线程程序中，通常会有相同任务的线程。多线程程序通过域分解将数据分成多个部分。相同任务的线程并行处理数据的一部分。在大多数情况下，由于每个子任务具有相同的工作和近似相等的数据大小，它们的内存访问特性没有明显差异。在这种情况下存在 2 种分配方案，分别是在线程之间平均分配缓存空间和共享缓存空间。

为了验证哪种方法更优，本文通过实验分析了相同任务多线程应用的最优缓存分配方法。实验在 Intel Xeon CPU E5-2667 v4 机器上评估了 PARSEC 中的多个不同应用程序。为了缩小搜索空间，三级缓存的大小被限制在较低的空间，并使用少量线程来运行程序。图 3 展示了共享方案与隔离方案在相比，在不同缓存路数或线程数下的加速情况。由于线程之间没有明显差异，实验只比较了在相同线程数和缓存空间的情况下，完全共享和完全隔离 2 种方案。当隔离时，缓存空间在线程之间均匀共享。如图 3 所示，在所有测试中，共享方案的运行时间都少于隔离方案的运行时间。这一结果表明相同任务的多线程应用程序适合进行聚类处理。因此在为同一任务的多个线程分配缓存时，它们聚类在一起，并共享一定数量的缓存空间有效降低训练成本。

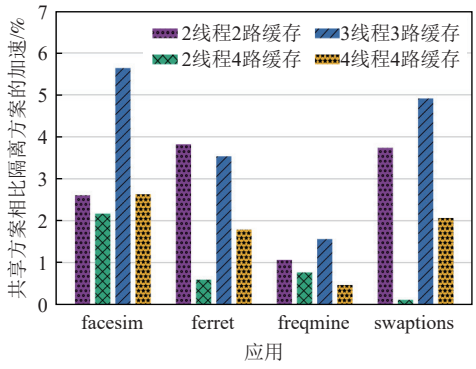



Fig. 3 Performance of sharing and isolation schemes under different number of cache paths and threads count

图 3 共享和隔离方案在不同缓存路数及线程数下的性能

## 2.2 调整迟缓问题

针对离散缓存分配动作空间导致的训练低效问题，本研究提出基于连续动作空间的策略优化框架。通过将离散的缓存路数调整转换为连续的资源配额分配，使模型能够在更少的迭代步骤中收敛到最优解。例如，在离散空间中调整缓存分配方案时，只能将线程的缓存空间的起始路数或结束路数增加或减少 1 个单位。而在连续空间中，我们可以将所有缓存空间(例如 20 个路)映射到 0~1 的范围内，并以任意单位调整(例如，在 20 个路中，增加 0.25 等价于增加 5 个路)，使得模型能够在更少的步骤中收敛到最优解。

## 2.3 性能波动抑制与采样策略优化

实际部署环境中，程序性能的非平稳波动对强化学习模型训练构成重大挑战。图 4 是内存数据库 Redis 性能波动测试的结果，从图 4 中可以看到性能存在显著波动：在 1200 s 稳定负载下，每秒处理请求数(requests per second, RPS)最大值(89 231)与最小值(41 678)的比值达 2.14。这种高方差，可能会导致模型收敛到次优的缓存分配策略，甚至完全无法收敛。

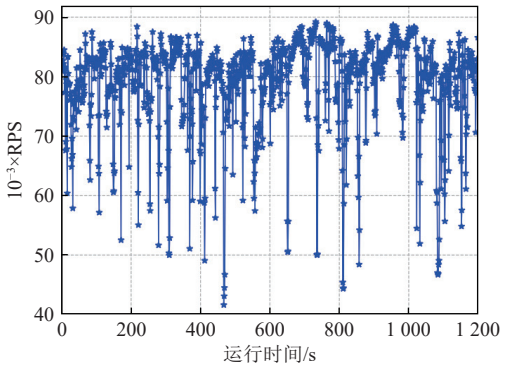



Fig. 4 Number of response requests per second of Redis under stable load

图 4 Redis 在稳定负载下的每秒响应请求数

为抑制性能波动的影响，本研究延长采样时间方法以确保模型学习的效率和准确性。图 5 展示了采样时间与 RPS 方差之间的关系。通过增加采样时间，方差可以有效减小。由于目前尚未有明确研究表明方差对模型学习的影响，本研究采用了一种简单的方法来选择合适的时间窗口，即选择图中的拐点(实验中是 15 s)。除了系统本身的波动，缓存热身

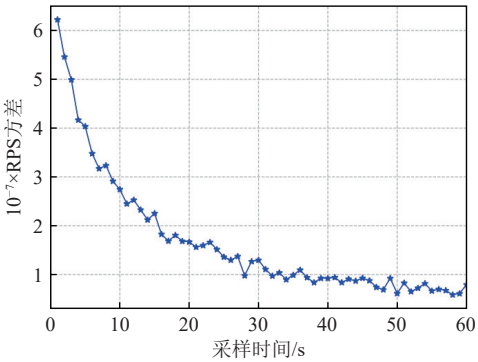



Fig. 5 Variance of RPS for Redis at different sampling time

图 5 Redis 在不同采样时间下的 RPS 方差

(warm up)加载进一步加剧性能波动。当缓存接收到新的分配方案时，缓存需要一定的热身时间来加载数据。不同场景下所需的热身时间不同，设置足够长的热身时间可以避免这个问题。

## 3 LPart 算法

针对多线程应用内部复杂的线程依赖与数据共享问题，本文提出基于深度强化学习的智能缓存分配方法——LPart。该方法以线程的性能监视计数器数据和缓存分配方案作为输入，以学习到的最佳的分配决策作为输出。图 6 是 LPart 的整体架构图。LPart 主要由 5 个部分组成，分别是应用分析器、应用监视器、内存池、深度强化学习模块和缓存控制器。其中，应用分析器负责对应用线程的解析和分组，应用监视器负责定期对应用运行情况进行采集，内存池负责保存强化学习的训练数据，深度强化学习模块负责学习缓存分配方案，缓存控制器负责实施缓存分配方案。

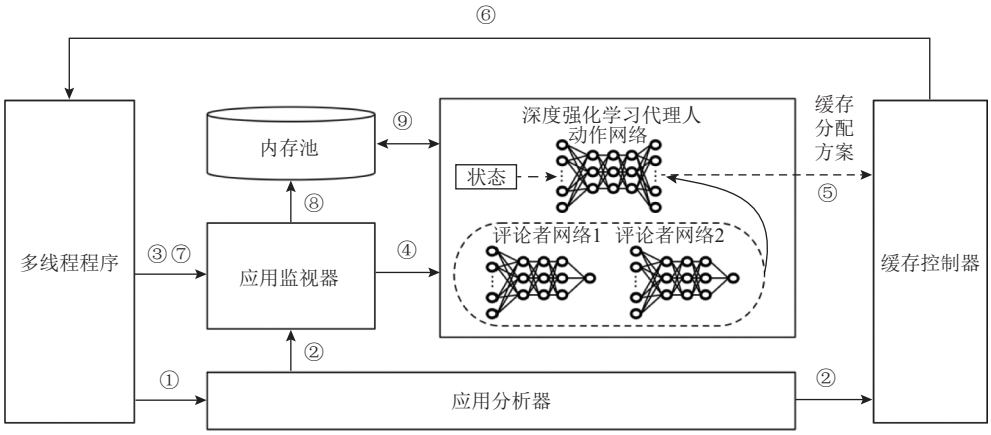



Fig. 6 Overall architecture of LPart

图 6 LPart 的整体架构

### 3.1 应用分析器

应用分析器负责选择应用的关键线程并将它们分组，以便后续进行缓存分配。应用分析器首先会收集每个线程的处理器使用情况、LLC 读取数和线程名称(图 6 中的步骤①)。随后，应用分析器会过滤掉处理器使用率低或三级缓存读取数低的线程，并将具有相同线程名称的线程聚为一类。线程名称是指在编程过程中由用户设置的名称。执行相同任务的线程由相同的函数创建，因此具有相同的线程名称。需要注意的是，LPart 没有根据线程采样得到的指标数值来进行聚类和分类，因为这种方法可能会由于负载不平衡、工作饱和不足等因素导致错误分类。最后，选择的关键线程和分组情况分别通知应用监视器和缓存控制器(步骤②)。

### 3.2 应用监视器

应用监视器负责定期收集应用的状态(步骤③⑦)。监视器需要收集 2 种类型的数据。第 1 种是应用的整体端到端性能，例如每秒的输入、输出操作数(IOPS)、存储系统的吞吐量和延迟等。第 2 种是由基于硬件性能计数器的分析工具收集的每个线程的硬件事件。尽管在多线程应用中 IPC 与程序性能之间没有明显的关联，但收集性能监控计数器的数据可以帮助模型了解应用的当前状态，使模型能够具有更好的泛化能力，并在应用的不同阶段取得更好的结果。当应用监视器收集当前的应用状态时，会将其发送给深度强化学习代理以进行分区决策(步骤④)。

### 3.3 内存池

内存池负责存储从应用监视器收集到的训练样本(步骤⑧)，这些样本包含 4 类信息：第$t$次调整时的应用程序的状态$s_t$、由奖励函数计算得出的奖励值$r_t$、要实施的缓存分配策略$a_t$，以及在执行新的分配策略后应用程序的状态$s_{t+1}$。因此每个样本都是一个四元组，可以表示为$(s_t, r_t, a_t, s_{t+1})$。在采集到足够的训练样本之后，存储在内存池中的样本将会被分批次地输入到深度强化学习网络中进行训练(步骤⑨)，以消除样本之间的相关性。

### 3.4 深度强化学习模块

深度强化学习模块是整个系统的核心。一方面，它根据应用程序的当前状态提供推荐的决策以追求

更高的性能(步骤⑤)。另一方面，它从内存池中获取历史转换用于学习和更新策略，以提供更好的决策(步骤⑨)。

为了发现最佳的缓存分配方案，学习模块不仅利用现有知识来推荐缓存分配方案，还有一定概率探索未知状态，这被称为探索与开发。下面会分别介绍学习模块中的状态、动作、奖励函数以及网络。

状态的特征应该能很好地反映出应用程序的当前状态。LPart 选择了 4 个性能监控计数器来刻画，分别是周期指令数、分支错误数、缓存缺失数和缓存访问数。通过这 4 个指标组成状态标记应用所处的状态。需要注意的是，使用性能监控器的目的不是为了获取程序的性能，而是为了获取应用的状态和阶段，以帮助 LPart 做出决策。

在学习系统中的动作对应为缓存分配方案。每个分配方案应该包含每个线程组可以使用的缓存空间。由于每个缓存空间必须是连续的，我们使用元组$(L, W)$来表示一个缓存空间段，其中 $L$ 表示起始缓存路编号，$W$ 表示所占用的缓存路数量。将所有需要分配缓存空间的元组组合起来就是一个完整的缓存分配方案，即系统的动作。

奖励函数反映了由动作引起的环境状态变化所带来的奖励。通过定义奖励函数，可以将优化目标传达给学习模块，使其能够学习最大化累积奖励的最优或近最优策略。在计算奖励时，需要根据优化目标选择指标。例如，在我们的微基准测试实验中使用了事务处理速率及在 Redis 测试实验中使用了吞吐率。下面以第 $n$ 次调整的吞吐量 $T_n$ 为例，则对于动作 $a_n$ 的奖励 $r_n$ 为调整前后吞吐量之差，即

$$r_n = T_{n+1} - T_n \text{。}$$

在 LPart 中，我们使用了 3 个网络：1 个动作网络和 1 对评论者网络来进行学习。动作网络以状态作为输入，输出推荐的缓存分配方案。评论者网络用于评估特定状态下的动作性能，它以状态和缓存分配方案作为输入，输出方案的分数。LPart 包含 2 个评论者网络，这是为了减少 Q 函数估计误差而设计的。在训练过程中，评论者网络提供梯度以更新动作网络，使动作网络能够生成更好的动作策略。网络结构和特定参数值通过实验分析确定。深度学习算法详细步骤可见算法 1。

**算法 1.** LPart 中的深度强化学习算法。

输入：$(s_t, r_t, a_t, s_{t+1}), t = 0,1,\cdots,N$；

输出：训练好的动作网络 $\pi_\varphi$。

① 令 $n = 0$；

② 随机初始化评论者网络 $Q_{\theta_1}$，$Q_{\theta_2}$ 和动作网络 $\pi_\varphi$ 的权重；

③ 初始化目标网络参数 $\theta_1' \leftarrow \theta_1$，$\theta_2' \leftarrow \theta_2, \varphi' \leftarrow \varphi$；

④ 初始化记忆池 $\beta$；

⑤ 获取初始状态 $s_0$ 和初始系统吞吐量 $T_0$；

⑥ Repeat

⑦ 选择动作并添加探索噪声 $a_n \sim \pi_\varphi(s_n) + \varepsilon$，$\varepsilon \sim N(0,\sigma)$；

⑧ 执行分配动作 $a_n$，观察新状态 $s_{n+1}$ 和系统吞吐量 $T_{n+1}$；

⑨ 计算动作 $a_n$ 的奖励 $r_n = T_{n+1} - T_n$；

⑩ 将样本 $(s_t, r_t, a_t, s_{t+1})$ 存储到记忆池 $\beta$；

⑪ 从记忆池 $\beta$ 中随机抽取 $N$ 个样本，计算样本中状态 $s_{t+1}$ 的对应的动作 $\tilde{a} \leftarrow \pi_{\varphi'}(s_{t+1}) + \varepsilon$；

⑫ 计算由目标策略选择的动作的目标值：$y \leftarrow r_t + \gamma \min\limits_{i=1,2} Q_{\theta_i'}(s_{t+1}, \tilde{a})$；

⑬ 更新评论者网络 $\theta_i \leftarrow \arg\min\limits_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s_t, a))^2$；

⑭ 每隔 $C$ 步，更新目标网络；

⑮ $n = n+1$；

⑯ until $n > MAX_STEPS$。

### 3.5 整体工作流程

LPart 的工作机制主要包括离线训练和在线分配 2 个部分。LPart 首先在离线模式下收集数据并训练模型。然后，训练好的模型在线上应用，实时调整缓存资源分配。

在离线训练阶段，LPart 首先会唤醒应用分析器选出需要分配缓存的线程并分组，这个过程只会在初始化中执行 1 次。然后 LPart 会在应用运行中采集训练数据。在每个调整周期中(在本文实验中周期为 15 s)，LPart 会通过应用监视器采集到应用在这个周期的状态 $S$ 和奖励 $r$，并且与上个周期的状态和动作组成一个四元组 $(s_t, r_t, a_t, s_{t+1})$ 保存到内存池中，当内存池积攒到足够的样本时会开始对网络进行训练。

在 LPart 对软件进行在线分配时，只需要保留训练好的动作网络，在每个调整周期内，LPart 会通过应用监视器采集到当前阶段的状态 $S$，并把他输入到动作网络中，动作网络会输出一个缓存分配方案，并且通过缓存控制器实现该调整。

## 4 实验结果与分析

本节通过微基准测试、多线程应用和多程序场

景测试 3 个场景分别对 LPart 的有效性全面评估。

### 4.1 实验设计

1)实验配置。我们在一台支持路分区的服务器上进行评估，该机器搭载了 E5-2630 v4 @2.20 GHz 处理器。该处理器拥有 8 个物理核心和 16 个硬件线程，每个物理核心含有 32 KB 的私有 L1 I/D-Cache 和 256 KB 的 L2 缓存。所有核心共享一个 25 MB 的 20 路组相联的 L3 缓存。我们的机器配置有 64 GB 的 DDR4-2133 GHz 内存，操作系统为 Ubuntu-20.04，内核版本为 5.4.0。为了稳定性能测量，CPU 时钟频率被设置为最高水平。

2)对比方案。由于 LPart 是以线程为粒度进行缓存分配的方法，本文选择了 4 种多应用缓存分配方法(KPart、dCat、CPpf、完全共享)进行对比。其中，CPpf 利用应用预取敏感性进行缓存分配。然而，在我们的测试应用中，不同的线程在大多数情况下具有类似的预取敏感性，使得 CPpf 退化为完全共享。因此，我们没有在微基准测试和 Redis 测试中展示 CPpf 的结果。

3)评估指标。对于单应用程序，我们以应用程序的实际性能作为评估指标：微基准测试使用事务处理速率，Redis 使用请求处理速率，云存储系统使用吞吐率。对于多应用测试，由于被测试应用缺乏实时性能，我们与已有研究一致，选择使用 IPC 计算加速比，并报告加权加速比。

### 4.2 微基准测试

我们在 1.2 节提到的微基准测试上评估了 LPart。微基准测试的详细信息可以在表 1 中找到。图 7 显示了不同微基准测试下的事务处理速率。搜索算法通过穷举获得了不存在共享缓存空间的情况下的最优缓存分配方法。在所有的分配方法中，LPart 实现了最高的性能提升，最高可达 26.9%。在微基准测试 1 中，没有任何方法实现了显著的提升，因为 2 个线程以相同的访问模式访问相同的数据范围。在微基准测试 2, 3, 6 中，LPart 的性能高于搜索算法，这可以归因于 LPart 支持不同线程间对缓存空间的局部共享，在更细的分配粒度上获得了额外的性能提升。

### 4.3 多线程应用测试

我们分别在开源数据库 Redis 和商用分布式存储系统上验证了 LPart 的效果。

#### 4.3.1 Redis 实验测试

Redis 是一个开源的内存键值存储系统，被用作数据库或缓存。尽管具备高性能，但 Redis 的单线程架构一直备受争议。Redis 6 引入了 I/O 线程，同时保留了核心的单线程数据访问接口。我们在实验中使用了 Redis 6.0.10 版本，并将 I/O 线程的数量设置为 6，这意味着当 1 个主线程负责数据访问时，有 5 个线程负责读写 I/O 套接字。实验中禁用了 Redis 的持久化功能。

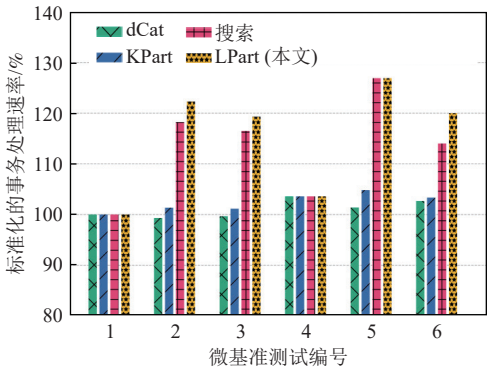



Fig. 7 Transaction processing rates of different algorithms under microbenchmarks

图 7 不同算法在微基准下的事务处理速率

在 Redis 的测试中使用了 2 种基准工具：Redis-benchmark 和 memtier-benchmark。使用 Redis-benchmark 来模拟同时由 50 个客户端执行的 SET 或 GET 命令。数据集由 1 000 万个大小为 1 KB 的对象组成，或者由 10 万个大小为 32 KB 的对象组成。我们还使用了来自 Redis Labs 的 memtier-benchmark 来生成 2 个混合工作负载以进行基准测试。第 1 个工作负载(负载 *E*)由 50% 的 SET 和 50% 的 GET 组成，而第 2 个工作负载(负载 *F*)则由 10% 的 SET 和 90% 的 GET 组成。表 2 列出了这些工作负载及其详细信息。为了避免不可预测的网络干扰，客户端和服务器在同一节点的不同插槽的 CPU 上运行，以避免干扰。

图 8 展示了不同分配方案在 Redis 下的吞吐量性能提升情况，图 9 展示了不同分配方案在 Redis 下的尾延时情况。LPart 在吞吐量上比完全共享提高了最多 8.1%，平均提高了 4.8%。通过不同的负载测试可以看到 LPart 能够在 Redis 不同负载下都取得最大的性能提升，尤其值得注意的是，LPart 将尾延迟降低了最多 17.8%，平均降低了 9.5%。这表明 LPart 在提高吞吐量的同时并没有牺牲服务质量。在所有场景中 dCat 的性能都是下降的，这是因为 dCat 专为多租户云设计，其强大的缓存隔离功能使其在多线程应用中表现不佳。而 KPart 的提升并不明显，这主要有 2 个原因。首先 KPart 在测试每个线程的缓存敏感度时没有考虑多线程应用中不同线程间的相互影响，也没有考虑关键路径线程，使其并未能显著提高性能。此外，KPart 的聚类算法取决于 2 个线程的缓存

**Table 2 Details of Redis Test Workloads**

**表 2 Redis 测试负载的详细说明**

| 负载 | 生成工具 | 数据大小/KB | 数据量 | 写比例/% | 读比例/% | 流水数 | 线程数 | 每线程客户数 |
|---|---|---|---|---|---|---|---|---|
| *A* | Redis-benchmark | 1 | 10 000 000 | 0 | 100 | 16 | 6 | 50 |
| *B* | Redis-benchmark | 1 | 10 000 000 | 100 | 0 | 16 | 6 | 50 |
| *C* | Redis-benchmark | 32 | 1 000 000 | 0 | 100 | 16 | 6 | 50 |
| *D* | Redis-benchmark | 32 | 1 000 000 | 100 | 0 | 16 | 6 | 50 |
| *E* | memtier-benchmark | 32 | 1 000 000 | 50 | 50 | 8 | 6 | 50 |
| *F* | memtier-benchmark | 32 | 1 000 000 | 90 | 10 | 8 | 6 | 50 |

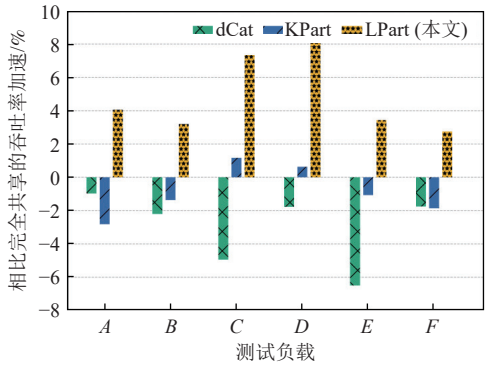



Fig. 8 Performance improvement of different methods under Redis

图 8 不同方法在 Redis 下的性能提升

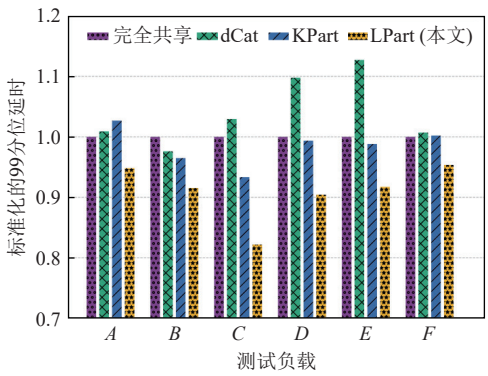



Fig. 9 Tail latency of different allocation schemes under Redis

图 9 不同分配方案在 Redis 下的尾延时

缺失频率, 其聚类效果受到测量误差影响。而 LPart 的聚类则是将多个相同任务的 I/O 线程进行合并, 更加准确。因此 LPart 的高性能可以归功于其高效的聚类以及考量了多线程程序特性。

在训练过程中, 模型在每次决策前会进行 1 000 次更新, 大约需要 20 s。模型收敛所需的步数不超过 166。具体而言, 我们在 Redis 上完成训练所花费的时间为 3 320 s。在线分配过程中, LPart 每 15 秒调整 1 次缓存, 并且在整个过程中使用的核心占用率不足 3%。

图 10 对比了在负载变化时完全共享缓存和 LPart 的性能。在测试中, 我们交替执行负载 *C* 和负载 *D* 各 2 次, 以模拟不同的工作负载。结果显示, LPart 的 RPS 比完全共享缓存高出 4.8%。从我们的观察来看, 每当负载发生变化时, LPart 在下一个周期内会输出学到的最佳分配方案。这表明 LPart 在工作负载变化时具有良好的适应性。

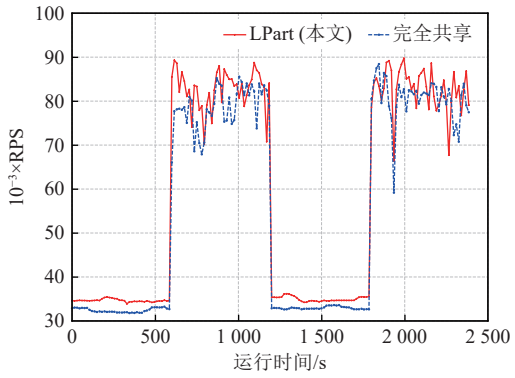



Fig. 10 Performance of full share cache and LPart under load changes

图 10 负载变化下完全共享缓存和 LPart 的性能表现

图 11 展示了 LPart 在有聚类和无聚类情况下的训练过程。图中的每个数据点表示 1 个调整阶段的应用性能。带有聚类的 LPart 的性能优于不带聚类的 LPart。带有聚类的 LPart 在大约 150 次调整后基本收敛并保持较高的 RPS。相比之下, 不带聚类的 LPart 并未达到最优性能。在我们的测试中, 不带聚类的 LPart 在接下来的 900 次调整中仍然没有收敛。这是因为不带聚类的 LPart 必须为更多的对象分配缓存空间, 导致其状态空间和探索空间急剧增长。因此, 尽管不带聚类的 LPart 花费了更多的时间进行学习, 但性能仍然不如带有聚类的 LPart。

### 4.3.2 商用分布式文件系统测试

除了 Redis 外, 我们还在商用的分布式存储系统上评估了 LPart。LPart 在系统吞吐量上相比完全共享提高了 9.8%。这表明在商用应用程序这种高度优化、硬件定制的场景下同样存在缓存干扰, LPart 可以

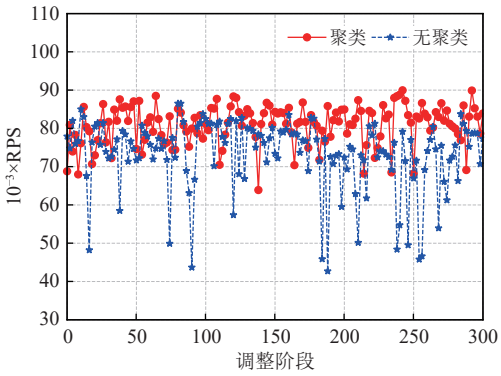



Fig. 11 Training with and without clustering

图 11 有无聚类的训练情况

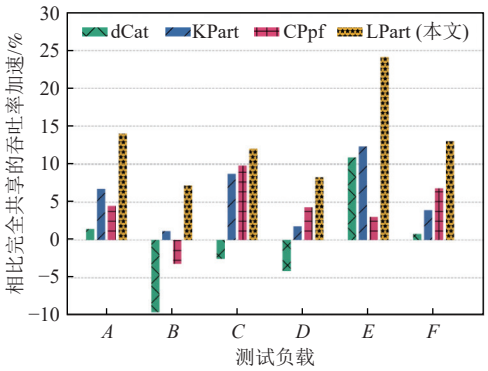



Fig. 12 Performance of different allocation methods in multi-application scenarios

图 12 不同分配方法在多应用场景下的性能

通过学习更智能的缓存分配方案获得更好的性能。

与此同时，其他分配方法并不能提高系统性能。这些方法将一个线程的缓存空间设置为其他线程的最小值，导致系统处于异常状态，因此无法提高系统性能。而 LPart 采用了一种探索性方法来识别应用程序中的关键线程。通过优化这些关键线程的缓存分配，LPart 提高了它们的性能，从而改善了应用程序的整体性能。此外，通过分析 LPart 的分配方案，编程人员可以发现应用中的关键线程以及需要缓存的数据，从而指导程序的后续代码优化。

### 4.4 多应用场景测试

虽然 LPart 是为多线程应用设计的，但它在多应用场景中仍然具有良好的学习能力。我们从 SPEC CPU2006 和 PBBS 基准套件中随机选择了 8 个应用程序，形成一系列混合工作负载。每个应用程序都是单线程的，并且独占一个核心。为了与先前的研究保持一致，缓存空间设置为 12 路，总大小为 15 MB。混合工作负载中的所有应用程序都同时启动并持续运行，直到每个应用程序完成至少 100 亿条指令。性能报告基于每个应用程序执行的前 100 亿条指令。

如图 12 所示，LPart 的性能改进最高，最大改进率为 24.1%。由于其严格的隔离特性，dCat 的性能改进不明显。CPpf 通过识别线程的预取敏感性来快速进行分配决策，几乎没有额外开销，在大多数情况下都是有效的。KPart 的整体性能仅次于 LPart，在具有显著流式应用程序的工作负载中往往能够获得更好的改进。例如，在负载 *C* 中，通过限制部分负载的缓存空间，使得部分对缓存需求大的应用获得了更好的性能表现。

LPart 在多应用程序工作负载中获得了最高的改进，这可以归因于 2 个主要因素。首先，LPart 的训练是根据大量的训练数据进行的，相比已有方法只根据 1 次的采样决策能够更加准确。通过观测应用的期望表现，能够有效规避应用由于阶段变化以及性能波动带来的影响。其次，LPart 支持局部共享，粒度更细，可以达到更理想的加速。从对应用场景的测试可以看出，LPart 能够适应各种负载，并且表现良好。作为代价，LPart 的训练需要大量的学习样本和学习时间，因此它只适用于一些固定的应用场景。

在 12 路缓存空间的基础上，额外对 9 路和 15 路的缓存空间进行测试。图 13 展示了不同方案的平均加速比。从图中可以看出随着缓存空间增大，所有方法的平均加速比都发生了下降，这是因为随着缓存空间增大，缓存争用的情况得到一定的缓解，因此分配方法所能获得的收益也随着下降。

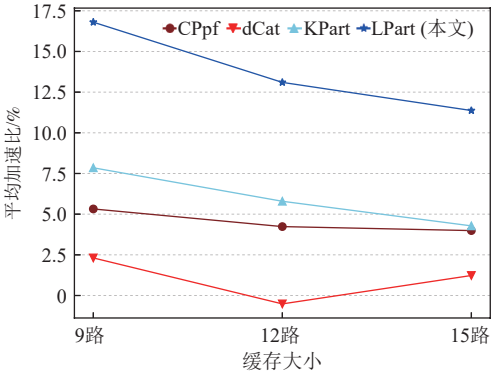



Fig. 13 Cache space sensitivity analysis in multi-application scenarios

图 13 多应用场景下的缓存空间敏感性分析

## 5 总 结

本文提出了多线程应用中存在的缓存争用问题并针对该问题提出了一种端到端的自动缓存分区系统 LPart，通过深度强化学习，LPart 可以识别出应用

所处的状态，并给出当前状态下的缓存分配方案。最后，大量的实验结果也证实了 LPart 的鲁棒性和有效性。当今，部分新处理器采用了截然不同的缓存设计层次结构，它们有更大的二级缓存和非包含式(non-inclusive)三级缓存。然而，这并不影响本文的研究意义。首先，尽管二级缓存有一定的扩大，但是绝大部分的缺失仍然要在共享的三级缓存中命中，而多线程应用内部的缓存争用问题一直存在，通过缓存分配方案可以有效管理三级缓存，提升系统的整体性能。在未来的研究工作中，我们会进一步考量更多的资源，如带宽，核心频率等，使得分配方法更全面、高效。此外，在多应用场景中，我们发现 LPart 的强适应能力使其能够应对各种复杂负载，但需要一定的学习样本和训练开销，因此更适用于一些固定应用的场景，这点与微服务场景高度契合。在未来，我们会把 LPart 应用于微服务，进一步扩大适用场景。

**作者贡献声明**：何铭健负责方案整体设计并撰写论文；王桦提出指导意见并修改论文。

## 参 考 文 献


[1] Asanovic K, Bodik R, Catanzaro B C, et al. The landscape of parallel computing research: A view from Berkeley [R]. Berkeley, CA: University of California, Berkeley, 2006

[2] El-Sayed N, Mukkara A, Tsai P A, et al. KPart: A hybrid cache partitioning-sharing technique for commodity multicores [C]//Proc of the 24th IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2018. DOI: 10.1109/HPCA.2018.00019

[3] Park J, Park S, Han M, et al. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers [C]//Proc of the 27th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2018. DOI: 10.1145/3243176.3243211

[4] Xu Cong, Rajamani K, Ferreira A, et al. dCat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service [C]//Proc of the 13th Eurosys Conf. New York: ACM, 2018. DOI: 10.1145/3190508.3190555

[5] Mukkara A, Beckmann N, Sanchez D. Whirlpool: Improving dynamic cache management with static data classification [J]. SIGARCH Computer Architecture News, 2016, 44(2): 113–127

[6] Balasubramonian R, Albones D, Buyuktosunoglu A, et al. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures[C]//Proc of the 33rd IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2000. DOI: 10.1109/MICRO.2000.898075

[7] Chang Jichang, Sohi G S. Cooperative cache partitioning for chip multiprocessors[C]//Proc of the 21st Annual Int Conf on Supercomputing. New York: ACM, 2007: 17−21

[8] Wang Xiaodong, Chen Shuang, Setter J, et al. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support[C]//Proc of the 23rd IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2017. DOI: 10.1109/HPCA.2017.65

[9] Jiang Lin, Lu Qingda, Ding Xiaoning, et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems[C]//Proc of the 14th Int Conf on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2008. DOI: 10.1109/HPCA.2008.4658653

[10] Scolari A, Bartolini D B, Santambrogio M D. A software cache partitioning system for Hash-based caches[J]. ACM Transactions on Architecture & Code Optimization, 2016, 13(4): 1−24

[11] Zhang Ludan, Liu Yi, Wang Rui, et al. Lightweight dynamic partitioning for last-level cache of multicore processor on real system[J]. Journal of Supercomputing, 2014, 69(2): 547−560

[12] Funaro L, Benyehuda O A, Schuster A. Ginseng: Market-driven LLC allocation[C/OL]//Proc of the 2016 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2016[2024-12-07]. https://www.usenix.org/conference/atc16/technical-sessions/presentation/funaro

[13] Sun Gongjin, Shen Junjie, Veidenbaum A V. Combining prefetch control and cache partitioning to improve multicore performance [C]//Proc of the 33rd IEEE Int Parallel and Distributed Proc Symp. Piscataway, NJ: IEEE, 2019. DOI: 10.1109/IPDPS.2019.00103

[14] Xiang Yaocheng, Ye Chencheng, Wang Xiaolin, et al. EMBA: Efficient memory bandwidth allocation to improve performance on Intel commodity processor[C]//Proc of the 48th Int Conf on Parallel Processing. New York: ACM, 2019. DOI: 10.1145/3337821.3337863

[15] Xiang Yaocheng, Wang Xiaolin, Huang Zihui, et al. DCAPS: Dynamic cache allocation with partial sharing[C]//Proc of the 13th EuroSys Conf. New York: ACM, 2018. DOI: 10.1145/3190508.3190511

[16] Xiao Jun, Pimentel A, Liu Xu. CPpf: A prefetch aware LLC partitioning approach[C]//Proc of the 48th Int Conf on Parallel Processing (ICPP '19). New York: ACM, 2019. DOI: 10.1145/3337821.3337895

[17] Lucía P, Sahuquillo J, Selfa V, et al. Phase-aware cache partitioning to target both turnaround time and system performance[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31: 2556−2568

[18] Lucia P, Julio S, Salvador P, et al. Cache-Poll: Containing pollution in non-inclusive caches through cache partitioning[C]//Proc of the 51st Int Conf on Parallel Processing (ICPP'22). New York: ACM, 2022. DOI: 10.1145/3545008.3545083

[19] Park J, Park S, Baek W. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers[C]//Proc of the 14th EuroSys Conf. New York: ACM, 2019. DOI: 10.1145/3302424.3303963

[20] Garcia-Garcia A, Saez J C, Castro F, et al. LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores[C]//Proc of the 48th Int Conf on Parallel Processing (ICPP'

19). New York: ACM, 2019. DOI: 10.1145/3337821.3337925

[21] Saez J C, Castro F, Fanizzi G, et al. LFOC+: A fair OS-level cache-clustering policy for commodity multicore systems[J]. IEEE Transactions on Computers, 2022, 71(8): 1952−1967

[22] Roy R B, Patel T, Tiwari D. SATORI: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains[C]//Proc of the 48th Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2021. DOI: 10.1109/ISCA52012.2021.00031

[23] Navarro-Torres A, Alastruey-Benedé J, Ibáñez P, et al. BALANCER: Bandwidth allocation and cache partitioning for multicore processors[J]. The Journal of Supercomputing, 2023, 79(9): 10252−10276

[24] Bilbao C, Saez J C, Prieto-Matias M. Divide&Content: A fair OS-level resource manager for contention balancing on NUMA multicores[J]. IEEE Transactions on Parallel and Distributed Systems, 2023, 34(11): 2928−2945

[25] Chen Shuang, Delimitrou C, Martínez J F. PARTIES: QoS-aware resource partitioning for multiple interactive services[C]// Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2019. DOI: 10.1145/3297858.3304005

[26] Nishtala R, Petrucci V, Carpenter P, et al. Twig: Multi-agent task management for colocated latency-critical cloud services [C]//Proc of the 26th IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2020. DOI: 10.1109/HPCA47549.2020.00023

[27] Patel T, Tiwari D. CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers[C]// Proc of the 26th IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2020. DOI: 10.1109/HPCA47549.2020.00025

[28] Shi Jiucheng, Fu Kaihuang, Wang Jiawen, et al. Adaptive QoS-aware microservice deployment with excessive loads via intra- and inter-datacenter scheduling[J]. IEEE Transactions on Parallel and Distributed Systems, 2024, 35(9): 1565−1582

[29] Chen Ruobing, Peng Wangqi, Li Yusen, et al. Orchid: An online learning based resource partitioning framework for job colocation with multiple objectives[J]. IEEE Transactions on Computers, 2023, 72(12): 3443−3457

[30] Liu Song, Ma Jie, Zhang Zengyuan, et al. Scalpel: High performance contention-aware task co-scheduling for shared cache hierarchy[J]. IEEE Transactions on Computers, 2025, 74(2): 678−690

[31] Panda A, Sarangi S R. FaaSCtrl: A comprehensive-latency controller for serverless platforms[J]. IEEE Transactions on Cloud Computing, 2024, 12(4): 1328−1343

[32] Ghosh A, Yadwadkar N J, Erez M. Fast and efficient scaling for microservices with SurgeGuard[C]//Proc of the 2024 Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2024. DOI: 10.1109/SC41406.2024.00103

[33] Stojkovic J, Choukse E, Saurez E, et al. Mosaic: Harnessing the micro-architectural resources of servers in serverless environments[C]// Proc of the 57th IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2024. DOI: 10.1109/MICRO61859.2024.00103

[34] Muralidhara S P, Kandemir M, Raghavan P. Intra-application cache partitioning[C]//Proc of the 24th IEEE In Symp on Parallel & Distributed Processing. Piscataway, NJ: IEEE, 2010. DOI: 10.1109/IPDPS.2010.5470416

[35] Alameldeen A R, Wood D A. IPC considered harmful for multiprocessor workloads[J]. IEEE Micro, 2006, 26(4): 8−17



**He Mingjian**, born in 1992. PhD candidate. His main research interest includes cache and intelligent resource scheduling.

**何铭健**, 1992 年生。博士研究生。主要研究方向为缓存、智能资源调度。



**Wang Hua**, born in 1975. PhD, professor, PhD supervisor. Her main research interests include cloud storage, intelligent storage, and intelligent resource scheduling.

**王　桦**, 1975 年生。博士，教授，博士生导师。主要研究方向为云存储、智能存储、智能资源调度。