计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
62(4): 989−1002, 2025

本期推荐

GroupUCP: 按需动态调节的细粒度缓存划分策略

张传奇[1,2,3]　王　卅[1,2]　孙凝晖[1,2,3]　包云岗[1,2,3]

[1](处理器芯片全国重点实验室(中国科学院计算技术研究所)　北京　100190)

[2](中国科学院大学计算机科学与技术学院　北京　101408)

[3](北京开源芯片研究院　北京　100080)

(zhangchuanqi@ict.ac.cn)

GroupUCP: An On-Demand Fine-Grained Cache Dynamic Partition Strategy

Zhang Chuanqi[1,2,3], Wang Sa[1,2], Sun Ninghui[1,2,3], and Bao Yungang[1,2,3]

[1](*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[2](*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 101408)

[3](*Beijing Institute of Open Source Chip*, *Beijing* 100080)

Abstract　With the advancement of modern computer technology, the memory wall problem is getting more and more severe. Under this background, the last-level cache in multi-level memory hierarchy becomes a key resource affecting system performance. In recent years, various researches have optimized the last-level cache by means of size expansion and dynamic resource management. Way-partitioning technique is the main method of cache resource management, which optimizes system performance by partitioning the cache into ways and allocating them to each application. However, it is coarse-grained and requires all sets of caches to follow the same way-partitioning strategy. In fact, applications may have different space demand on different sets, and the way-partitioning technique restricts the space utilization of the cache, resulting in a waste of cache resources. In this paper, we propose an on-demand fine-grained cache resource management technique, GroupUCP, whose design idea is to aggregate individual cache sets into groups based on the different space demand of each application on each set, using dynamic grouping and real-time evaluation. Each group can be allocated space on demand independently, thus improving cache utilization and overall system performance. Experiments demonstrate that GroupUCP achieves finer-grained on-demand resource allocation using less hardware resources than the traditional UCP approach and achieves higher system performance improvement in cache-sensitive application combinations which shows imbalance space demand of cache.

Key words　multi-core processor; shared cache; dynamic partitioning; dynamic group; metadata compression

摘　要　随着现代计算机技术的进步,内存墙问题越发严重.在此背景下,多级缓存中的末级缓存成为了影响性能的关键资源.近年来各项研究通过拓展尺寸,以及动态资源管理的手段优化末级缓存.路划分技术是缓存资源管理的主要方法,通过将缓存按路为单位划分后分配给各个应用使用,实现系统性能优化.然而路划分粒度较粗,要求缓存的所有组(set)都遵循同样的路划分方案.实际上,应用在不同组可能会有不同的空间需求,路划分技术限制了缓存的空间利用,造成资源浪费.GroupUCP是一种按需调节的细

收稿日期: 2024-01-29; **修回日期**: 2024-07-19

基金项目: 中国科学院战略性先导科技专项(XDA0320000, XDA0320300); 国家自然科学基金项目(62090022, 62172388); 国家电网有限公司科技项目(SGJSXT00TYJS2100414)

This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (XDA0320000, XDA0320300), the National Natural Science Foundation of China (62090022, 62172388), and the Science and Technology Project of State Grid Corporation of China (SGJSXT00TYJS2100414).

通信作者: 王卅(wangsa@ict.ac.cn)

粒度缓存资源管理技术,其设计思路是根据每个应用对各缓存组的不同需求,采用动态分组和实时评估的方式,将各个缓存组聚合成组,分组进行按需分配.这一设计允许各个组进行独立的路划分分配,从而提高缓存使用率和整体系统性能.实验证明,相较于传统的 UCP 方法,GroupUCP 利用更少的硬件资源实现了更细粒度资源按需分配,在对缓存资源敏感且需求不均衡的应用组合下获得了更高的系统性能提升.

关键词 多核处理器;共享缓存;动态划分;动态分组;元数据压缩

中图法分类号 TP302

DOI: 10.7544/issn1000-1239.202440061 **CSTR:** 32373.14.issn1000-1239.202440061

现代计算机随着技术的进步,CPU 处理速度不断提高,但是与此同时内存访问速度并没有得到相应的提高,在这种趋势下,CPU 性能与内存性能之间的差距越来越大,造成了著名的“内存墙”问题[1].为应对内存墙问题,现代处理器通常采用多级缓存结构减少访存次数,提升整体性能.在这样的结构之下,多级缓存中位于访问内存前的最后一级——末级缓存(last level cache, LLC)就成为一个影响性能的关键资源[2].

近年来针对 LLC 整体的优化和研究主要有 2 个思路:1)拓展 LLC 尺寸.近期芯片公司 AMD 推出了 3D V-Cache 技术,将 LLC 的尺寸大幅度拓展,使得对于 LLC 容量有更高需求的科学计算类任务,如电子设计自动化、有限元分析等,都得到了极大的性能提升[3].2)LLC 动态资源管理.现代处理器普遍是多核架构,在云计算等多租户共享的场景下,当多个不同租户的应用在多核处理器上同时运行,应用之间会在访存通路的多个共享资源,如 LLC、内存带宽等,产生资源竞争,导致应用性能下降[4-6].因此,Intel, ARM 等芯片厂商相继推出自身的 LLC 资源管理技术,可以让管理员按需动态划分 LLC 资源给不同应用,隔离应用之间的干扰,让资源得到合理分配,提升系统性能[7-8].当前主流 CPU 都采用了路划分技术作为 LLC 资源管理技术[9-11].

支持路划分技术的处理器通常会为用户提供软件/硬件接口,用户可以通过接口将组相连缓存(set-associative cache)以路(way)为粒度划分成不同大小,指定给各个应用使用.每当应用需要新插入一个缓存块时,仅能在其被分配的几路中进行替换选择并插入,由此便限制了应用在缓存内的占用空间,实现了应用间干扰隔离和性能保障.路划分技术的原理与主流组相连缓存的结构直接对应,易于理解和使用,为上层用户提供了 LLC 的划分隔离能力,但如何确定应用的资源需求,按需调节 LLC 容量,依旧是一个关键挑战.在软件层面上,已有工作[7-8,12-13]尝试通过性能建模、反馈调节等方式动态调整各个应用的 LLC 容量.但是软件策略常常需要较长的调节周期,反馈速度慢,容易无法及时响应应用的需求变化.另一方面,以 UCP[14] 为代表的工作就尝试直接在硬件层面进行预测和调节.UCP 设计了新的采样监控模块,监控应用对每个路的利用情况,来进行路划分的分配.

然而路划分技术一直存在粗粒度的问题.当前商用服务器的 LLC 路划分,1 路大小通常为 512 KB~2 MB,已有研究[15] 发现在云函数的场景下,有的应用,如 autocomplete, ocr-img, markdown 等,对 LLC 容量的需求在 2 MB 以下,不到大 LLC 的 1 路大小,按路调节势必造成资源浪费.不仅如此,路划分本质上要求缓存中的所有组(set)都遵循同样的路分配方案,然而实际上从 set 的粒度来看,应用在不同 set 上可能具有不同的空间需求[16-19].文献 [18] 中的实验表明,应用对于每个 set 的空间需求差异巨大,少则仅需 1, 2 路,多则需要数十路.这就意味着传统路划分技术对所有 set 进行相同调控的策略过于粗粒度,无法同时兼顾需求多的部分 set 与需求少的部分 set,限制了缓存的空间利用.同时,随着技术进步,LLC 的尺寸不断增长,如 AMD 的 3D V-Cache 下 LLC 的每路尺寸已经达到了 6 MB[2],依旧采用粗粒度路划分会使得 LLC 资源利用低效问题更加严重.

本文提出一种按需调节的细粒度缓存资源管理技术 GroupUCP,在保持路划分简洁设计的基础上,增加按 set 粒度对应用空间进行细粒度调节的硬件管理机制,有效提升缓存使用率和系统整体性能.GroupUCP 的设计思路十分直接:由于每个应用对于各个 set 的需求不同,参考 UCP 的应用空间需求评估方法,可以在 set 粒度为它们根据实际需求划分缓存资源,并根据运行时的需求变化进行动态调整.但考虑到 set 粒度调节带来的巨大开销,不同于 UCP,我们将 set 进行动态分组,实时评估应用在每个分组的空间需求,对每组 set 进行独立的路划分策略调控.

然而实现这个想法仍需要解决 3 个问题:

1)set 分组粒度.如果分组过细,会带来过度的开

销，并且可能因为局部信息不足，做出错误的判断；如果分组过粗，则相对原路划分方案提升较小. 本文提出了一种低开销的可变分组方法，并能够动态优选分组粒度.

2）全局最优决策. 确保分组方案在全系统层面性能最优也是一个关键问题. 分组方案是根据组内局部信息，如组内缓存命中（hit）来做决策，而各个组内达到局部最优未必会实现全系统层面，如各应用的每周期指令数（instruction per cycle, IPC）的性能最优（1.3 节详述）. 本文提出了一种动态回归验证的方法，通过验证分组决策是否给全局性能带来了更好的收益，在分组决策与全局决策之间进行优选.

3）分组监控开销. 进行全局最优决策需要对每个分组都进行资源监控，这会引入很多监控部件，带来大量开销. 本文设计一种缓存块内存储元数据（metadata in cache block）的方法，有效降低了分组监控的开销.

实验结果表明，与 UCP 相比，GroupUCP 能够利用更少的硬件资源，实现更细粒度的 LLC 资源按需分配，在对缓存资源敏感且需求不均衡的应用组合下获得更高的系统性能提升.

1 研究背景及相关工作

1.1 数据中心 LLC 的资源管理

在当下的互联网基础设施中，数据中心扮演着至关重要的角色[20]. 为了节约成本提高资源利用率，数据中心的运营者通常会使用虚拟机、容器等技术将多个应用混和部署在同一个物理服务器上，但是让应用自由混合运行会造成严重的共享资源竞争，带来严重的应用间干扰，因此数据中心共享资源管理近年来一直是学术界和工业界研究的热点话题[21-22]. 而 LLC 作为数据中心中重要的硬件共享资源，更是需要精准高效的资源管理机制.

要实现 LLC 资源精准高效的管理，首先要实现 LLC 资源的细粒度划分隔离，其次应当根据应用 LLC 的需求，按需调节 LLC 资源分配. 已有大量工作分别从软件和硬件层面尝试从 LLC 资源划分和 LLC 划分策略展开研究.

1）LLC 资源划分

软件层面最知名的 LLC 资源划分工作是页着色机制[23-29]. 它们通过修改操作系统的内存管理机制，对应用空间进行特殊分配“着色”，从而让应用的地址空间在 LLC 内只能映射到特定 set，以此达到划分 LLC 的效果. 虽然此类工作仅需修改系统软件即可使用，但是它们并没有在现在的数据中心中得到广泛应用，有 3 个原因[7]：①需要对操作系统的虚拟内存管理进行大量的修改，并且和大页等技术相冲突，抵消掉 LLC 划分带来的收益；②在重新调整分配空间时会带来大量的开销；③要求 LLC 的分区分配与物理页面的颜色分配处于相同的比例，这意味着一些占用了绝大部分内存空间的低局部性应用将不得不占用绝大部分的 LLC 分区. 这些问题都使得页着色这种纯软件 LLC 管理方案难以在数据中心中被真正使用.

硬件层面的 LLC 资源划分工作可以分为 2 个大类：路划分技术[9-11] 与其他技术[30-33]. 路划分是当前最广泛应用的 LLC 资源管理技术，其一方面对用户友好，用户仅需通过掩码指定应用可以使用的缓存路范围即可生效；另一方面易于实现，路划分技术仅影响 LLC 替换与插入缓存块的工作流程，逻辑简单，在硬件上易于实现. 各主流芯片厂商均推出了基于路划分技术的 LLC 资源管理系统，并提供了易于使用的软件接口支持，如 Intel CAT[9]，AMD PQoS[10]，ARM MPAM[11] 等. 但如前面讨论，路划分的粒度较粗，难以实现更精准高效的 LLC 资源管理. 学术界也尝试修改过 LLC 的硬件结构，提供了更细粒度的 LLC 划分[30-33]，但是这些工作对目前的 LLC 结构修改过大，难以实现在现有处理器的 LLC 中，同时还要求管理者对新式的 LLC 底层结构非常了解才能做好划分，因此也很难在现实数据中心中部署.

2）LLC 划分策略

以上无论是软件还是硬件的 LLC 划分机制，都是提供了对 LLC 划分的能力，但如何探知应用的 LLC 容量需求，按需调节 LLC 资源分配实现高效资源管理仍是一个关键挑战. 已有工作在软件层面与硬件层面都设计相应的划分策略.

①软件层面. Brock 等人[34] 从理论上分析了缓存划分的最优化方案，并提出了一种基于动态规划的算法来求得最优化划分方案. dCat[8] 建立了一套简单的反馈调节流程，通过采集应用的 IPC 以及 LLC 的丢失（miss）数目等数据，判断应用目前对 LLC 资源的使用情况，以此来动态指挥 LLC 资源在不同应用之间的分配. KPart[7] 通过推测应用两两组合共同运行时的性能，对表现相似的应用进行聚类，让同一类的应用共享同一个 LLC 划分分区来有效提升 LLC 的使用率，提升系统整体吞吐率. DCAPS[12] 通过更高效的方法统计应用对 LLC 的使用情况，并以此为依据

让应用之间的分区可以全部或部分重叠,以此提高LLC的使用率,达到提升系统吞吐率等各种目标. DRLPart[13]使用了机器学习中强化学习的方法,通过收集各种实时性能数据,训练神经网络来生成优化的LLC分区划分策略.这些软件策略可以生成相对不错的划分策略,但是它们的调节周期都(与硬件直接调控相比)较长,同时需要通过离线分析或者在线长时间数据采集等方式收集先验知识,并且生成策略也需要消耗不少的计算资源,应用到实际场景可能会降低资源利用效率.

②硬件层面. UCP[14]是硬件生成路划分策略方面的代表性工作,它利用了LRU替换算法的特性,为每个核心设置了一套独立的使用率监测模块,推测核心对LLC每一路的使用率,从而推测出最优的分配策略,提升系统性能.与软件策略相比,UCP具有监测数据准确、调节快速等优点,但是,UCP的数据观测以及调控仍然受限于路划分技术的粗粒度问题,在某些应用上的性能提升有限.本文工作沿用了UCP提出的数据监测原理以及资源分配策略生成算法,在set分组的新设计下使用并获得了比UCP更好的性能提升.

1.2 UCP的动态划分原理

UCP进行动态划分使用的核心技术包括对路效用(utility)的定义与监测,以及用来计算资源分配的前瞻(lookahead)算法.

1)路效用. 只有准确评估每个应用对缓存资源的实际需求,才能精准按需划分缓存资源,实现最大化系统整体性能.因此,UCP提出“路效用”的概念,即每一路缓存资源能够为应用贡献的命中数量,称之为路的效用.基于监测到的不同应用对于不同路的效用信息,用户可以通观全局设计效用最大的缓存划分策略.具体来讲,UCP基于LRU算法提出路效用概念. LRU作为缓存替换算法中的常用算法,具有栈特性[14],即对于某一串访问序列中的某个访问请求来说,如果其在N路的缓存下访问结果是命中,那么其在大于N路的缓存下访问结果同样也会是命中.因此,UCP形式化地定义路效用为:假设一个缓存分区总共有N路,则设置N个计数器,分别为$H_1, H_2, \cdots, H_N$,分别对应LRU序列中从最晚使用的位置到最早使用的位置,假设某个应用线程t的访问请求在缓存发生命中时,在对应LRU位置的计数器中加1进行记录,最后第i个计数器的值就对应了线程t在第i路上的效用,记为$U_{t,i}$. 效用定义背后所蕴含的意义可以进行直观的解释:假定我们为某个应用线程t的缓存分区增大空间,从a路增加到b路,那么依据LRU的栈特性,在a路下命中的请求依然会命中,增大空间所能提高的命中数为

$$\Delta Hit_{t_{i=a}}^{b}=\sum_{a}^{b}U_{t,i}. \tag{1}$$

因此,最大化总效用就是最大化缓存提供的命中数,也就是提升缓存的吞吐能力.假设分配决策为线程t分配了W_t路,那么线程t的总效用可以记为

$$U_t=\sum_{i=1}^{W_t}U_{t,i}. \tag{2}$$

划分算法的优化目标就是依据收集到的效用信息,制定划分策略来最大化总效用:

$$U_{\text{total}}=\sum_{t=1}^{T}U_t. \tag{3}$$

同时为了避免各个线程互相干扰,采集各个线程使用完整缓存的数据,UCP引入了一个与原缓存相分离的效用监控(utility monitor, UMON)模块,为每个线程独立地进行效用统计.其包含上文所述的计数器,以及辅助标记目录(auxiliary tag directory, ATD). ATD仅记录访存请求的标记(tag),具有和缓存相同的相联度,使用LRU替换算法,以此来精确统计每个线程独立使用缓存时的效用数据.为了能降低ATD带来的大量开销,UCP通过使用动态组采样(dynamic set-sampling, DSS)[35]的方法,大幅度减少了ATD需要监控的set数目,从而降低了存储开销.

2)前瞻算法. 为了求得最大化式(3)的最优划分方案,可以使用穷举法遍历所有划分方案,然而文献[36]已经证明这个问题属于NP难问题,随着待分配的应用线程数目增多,该问题的求解时间将会巨幅增长.因此UCP提出了前瞻算法来快速近似求解最优分配方案.在式(1)的场景下,定义边际效用(marginal utility, MU)为进行缓存分配调整的单位收益值,如式(4)所示,假设调整应用分配的LLC空间从a路变为b路.

$$MU_{ta}^{b}=\sum_{i=a}^{b}U_{t,i}/(b-a). \tag{4}$$

前瞻算法以MU为启发式优化指标,让每一步分配能得到的最大的MU值,逐步完成空间分配. UCP会每间隔一小段时间后(如5 M或10 M时钟周期,称为决策周期),依据更新的效用数据执行前瞻算法及时进行划分策略的更新.

1.3 UCP存在的问题与本文动机

UCP[14]为了降低硬件开销,最后选择了对所有

set 进行统一的全局决策，牺牲了细粒度的调节. 然而，文献 [16–19] 已经证明，很多应用对缓存的各个 set 的实际空间需求差异很大. 使用全局决策对缓存 set 进行统一资源监测和空间划分，可能造成某些 set 的分配空间不足，而某些 set 空间资源过剩，造成缓存空间资源使用效率低下.

然而，我们发现，即使忽略硬件开销，以每个 set 为粒度进行缓存空间划分，并不一定比全局统一决策收益更高. 我们在 GEM5[37] 全系统模拟器中，配置了一套 4 核测试系统（详见 3.1 节），并从 SPEC CPU2017[38]，GAP benchmark[39]，Tailbench[40] 中抽取应用组成 16 组混合测试以尝试验证这个问题. 按照文献 [14] 中 UCP 的设计，共提出了 3 个方案，UMON-local，UMON-global，UMON-DSS. 其中 UMON-global 与 UMON-DSS 为 1.2 节所描述的方案，分别如图 1(c)和图 1(d)所示. 而 UMON-local 如图 1(a)所示，为每个 set 建立 ATD 和效用计数器，并细粒度地独立为每个 set 生成划分策略.

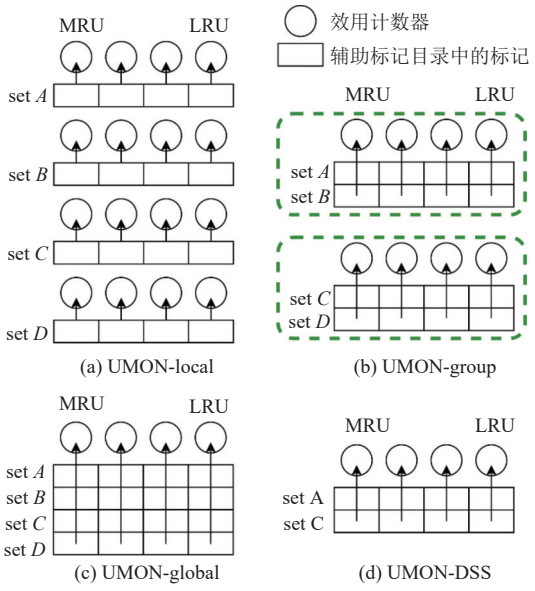


Fig. 1 Different types of UMON

图 1 不同类型的 UMON

从划分粒度来看，UMON-local 显然比其他 2 种方案具有更高的灵活度，只是由于空间开销太大而未能得以使用. 我们实现了 UMON-local 与 UMON-global，比较两者相对无 LLC 管理策略下给系统吞吐率带来的提升（具体测量方式见 3.1 节）.

实验结果如图 2 所示，UMON-local 方案在绝大多数情况下的性能表现都比 UMON-global 方案更差，更细粒度的划分策略并没有带来绝对的性能提升. 我们分析，这是因为虽然 UMON-local 方案针对每个 set 都制定了划分策略，能让每个 set 都按需灵活地调节划分空间，但是由于其做出决策的信息仅仅来源于 set 局部的短期信息，无法依据全局信息获知长期趋势，陷入了“短视”的局部最优.

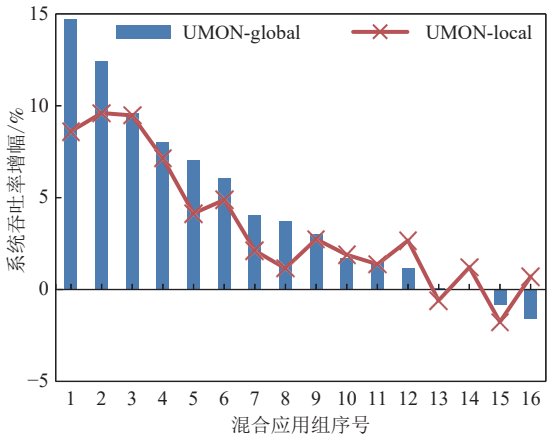


Fig. 2 Performance comparison of UMON-global and UMON-local

图 2 UMON-global 与 UMON-local 性能对比

而全局策略则受限于传统路划分的粗粒度问题，无法依照各个 set 的不同空间需求做出调节. 因此，本文提出了将 set 进行动态分组的管理方案 UMON-group，如图 1(b)所示，这样既优化了路划分的粗粒度，又可以收集到足够多的区域历史信息，从而能够根据区域中多个 set 的历史做出本区域使用情况的预计，能够在一定程度上避开 UMON-local 方案的“短视”问题，对缓存进行按需调节的细粒度资源管理.

2 GroupUCP 的设计

本文提出了 GroupUCP，目标是实现一种能够细粒度按需分配、低开销的 LLC 空间资源动态分配机制，来提高 LLC 的利用率和提升系统整体吞吐. GroupUCP 的整体思路是：由于不同应用使用 set 空间情况的不同，我们对路划分技术的粒度进行优化，允许在不同 set 上使用不同的路划分调节，达到细粒度按需分配的效果. 同时为了充分整合 set 级别的细粒度信息并降低对每个 set 都进行监控的开销，我们在每个决策周期中，将 set 进行动态分组，依据之前周期的效用数据，对每组的空间划分策略进行动态调节，并引入验证措施动态反馈调整分组粒度，保证以极低的性能影响降低大量硬件开销. 图 3 展示了 GroupUCP 的设计概括，其包含 3 个重要部分：

1）set 动态分组方法. 对 LLC 的 set 进行低开销分组，对各个分组独立进行全应用需求监测，指定空间划分，并能够动态进行不同粒度分组，按效用最大化

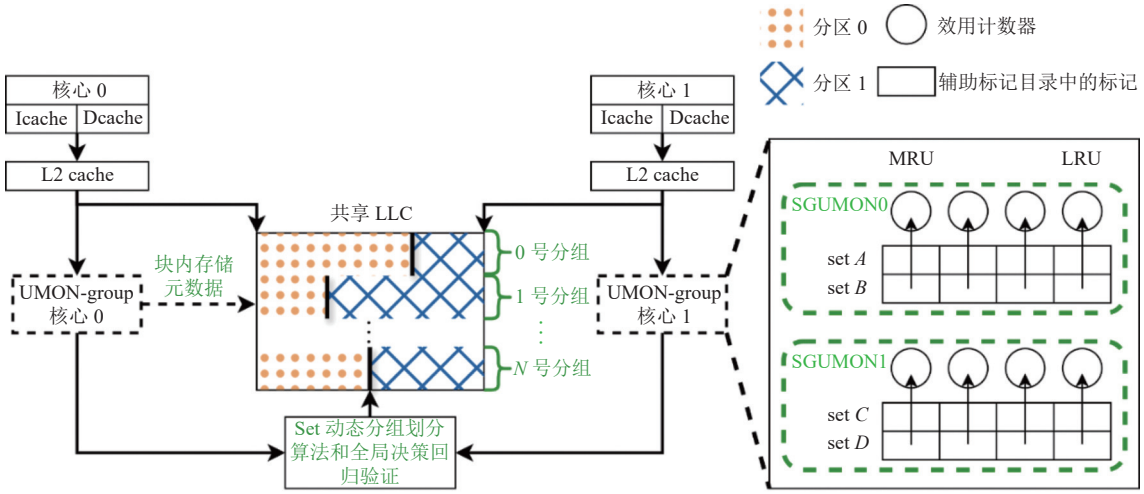


Fig. 3 Illustration of GroupUCP

图 3 GroupUCP 示意图

优选分组粒度, 从而实现细粒度按需分配的效果.

2)全局决策回归验证方法. 验证分组方案性能表现, 避免个别应用性能受到严重影响, 及时跳出局部最优, 确保缓存利用率与系统整体吞吐率得到提高.

3)缓存块内存储元数据. 借用小部分缓存的数据块来存储 UMON 的元数据, 实现极低硬件开销的同时完成分组决策.

2.1 set 动态分组方法

set 动态分组的设计需要考虑: 1)组织分组的方式; 2)分组决策的步骤; 3)确定分组粒度的算法.

正如 1.3 节讨论, UCP 使用全局策略对所有 set 进行同样的路划分, 忽视了局部上每个 set 的实际空间需求, 粒度粗; 而实验表明, 以单个 set 为粒度进行监控和划分, 会造成由于单个 set 的局部历史信息不足而做出错误决策. 为了避免陷入信息不足导致错误判断, 同时兼顾不同 set 局部具有的不同空间需求, 本文提出了对 set 进行动态分组决策的方法 GroupUCP, 以及动态进行分组粒度选择的算法.

GroupUCP 方法是将缓存当中 set 索引(set index)临近的若干个 set 分成一组, 将每一组视作一个完整的缓存, 对各个组进行原有 UCP 方法中的缓存路效用监控与路划分空间分配, 每组依据组内收集到的效用监控数据制定本组的划分策略, 不同组之间相互独立, 如图 3 所示.

1)分组方式. 我们将 set 索引中高位相同的 set(也就是地址上相邻的 set)分为同一组, 这样可以从 set 索引中快速获取分组索引(group index), 如图 4 所示, 从而近乎无开销地进行分组组织, 这种组织方式可以方便地指定分组粒度为 1/2, 1/4, 1/8, 1/16 等不同

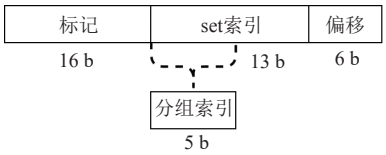


Fig. 4 Example of the group index under 32 groups

图 4 32 个分组下的分组索引示例

大小, 易于对分组粒度的大小进行快速调节. 在这种组织下, 粗粒度的分组逻辑上由多个细粒度分组组合而成, 因此粗粒度分组的统计数据可以直接通过整合细粒度分组的统计数据得到, 不必再进行额外的独立统计, 得以高效观察分组粒度动态变化的性能数据.

2)分组决策的步骤. 具体来说, 在将各 set 分组之后, 为新形成的每一个 set 组都配套建立 ATD 以及效用计数器, 形成 set 组的效用监控器(set group utility monitor, SGUMON), 利用其内部的 ATD 来监控并记录每一个核心线程对该 set 组的访问请求历史情况, 并在命中该组 ATD 的同时为对应 LRU 位置的效用计数器加 1 进行记录. 在每一个决策周期开始时, 对每个 set 组都独立地进行缓存空间的划分决策. 每组依据 SGUMON 的记录, 执行上文中提到的前瞻算法, 快速计算出在该组内能得到的最大效用以及对应的分配方案. 在划分决策计算完毕后, 将各组的分配方案按分组索引存储下来. 在实际使用中, 每当新的缓存请求进行空间分配时, 依照其地址对应的分组索引查找对应分组的分配方案, 进行缓存块的分配.

3)分组粒度决策算法. 为分组方案选择适当的分组粒度也是分组方案的关键问题, 本文提出了一套动态优选分组粒度的流程, 来进行分组粒度的决策,

具体流程见图 5. 在决策周期开始时，尝试进行各种粒度的划分，并记录各粒度分组给出的分配方案作为备选；在决策周期结束前，依照周期开始时各粒度分组给出的方案，计算各分组方案在全缓存上的总效用，优选其中最高者作为下一周期实际选取的划分粒度，在整个决策周期的实际执行中，统一按照上一周期结尾优选的粒度作为本周期的划分方案来执行.

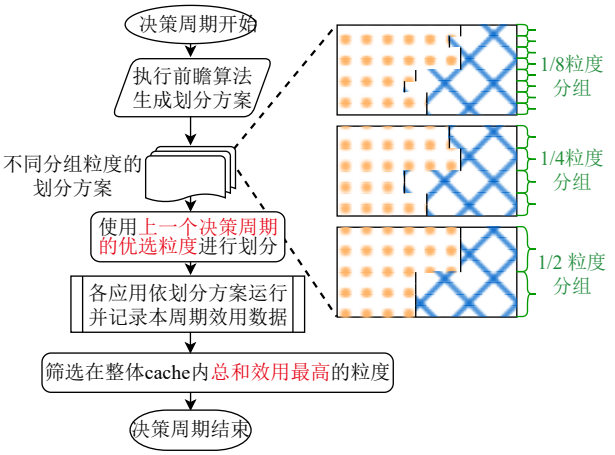


Fig. 5 Illustration of the selection process for optimal group granularity

图 5 优选分组粒度流程示意图

2.2 全局决策回归算法

本质上，set 动态分组方法是对各个分组进行局部优化来试图超越原 UCP 全局方案的性能. 而为了确保分组方案能够真正在全局系统层面提升性能，避免过度寻求缓存层面每组的局部最优，影响应用整体性能表现，本文引入了回归验证来检验分组方案的性能表现，通过与原 UCP 全局方案进行系统整体性能的回归对比验证，能够及时跳出局部最优决策，从而有选择地开启分组划分，确保系统全局吞吐性能提升.

在 2.1 节中我们提出了动态优选分组粒度的算法，极端情况下，全局方案也可以视作将所有 set 分成 1 组为粒度，可以纳入到上面的流程中进行总效用的比较来进行优选. 但是，这样仅能对缓存的期望效用进行比较来优选，而对于全局系统层面的性能提升的判定，则应当以 IPC 指标作为依据. 与通过 UMON 结构能够获取到的期望效用不同，IPC 是无法通过缓存的统计数据直接预测出来的，只有按不同的划分方案真实运行应用，才能比较出 IPC 的变化. 如果枚举各种粒度的划分方案并实际运行，则会使得系统在各种次优方案上运行很长时间，影响性能表现；并且各个方案测量的 IPC 时间间隔很远，可能由于应用自身行为发生变化而失去可比性，造成错误判断.

因此，为了能够快速准确地收敛到最优方案，本文选取了 2 个维度的指标作为不同阶段的判断依据，并提出了一个带有回归比较验证的优选分组粒度流程. 1)为了能够在尽量多的粒度方案中进行初步筛选，本文选择缓存的期望效用作为粒度的初步筛选指标，这样可以避免对系统性能造成影响，同时快速做出趋势性的判断；2)为了尽可能减少试探运行的次数，本文选择仅将分组方案与全局方案的优选粒度进行对比，这样可以尽可能地减少比较次数，并明确局部优化是否带来全局整体的性能提升.

为了整合这 2 个维度的指标进行方案指导，本文设计了一个表示分组置信度的饱和计数器 *Signal* 来记录指标以比较结果的历史数据. 如图 6 所示，在每个决策周期中运行各应用时，如 *Signal* 饱和就选择分组方案进行分配，反之则维持全局方案进行分配. 在每个决策周期结束，进行粒度优选时，如果分组方案胜出则将 *Signal* 的值加 1，反之则减 1. 特殊的情况是，每次从全局方案进入分组方案的第 1 个周期结束时，计算本周期分组方案的提交指令数 *Inst* 相对于上一周期全局决策的平均加速比(假设系统共有 N 个核心线程)：

$$AvgSpeedup = \frac{1}{N}\sum_{i=1}^{N}\frac{Inst_i}{Inst_{i,\text{last}}}. \tag{5}$$

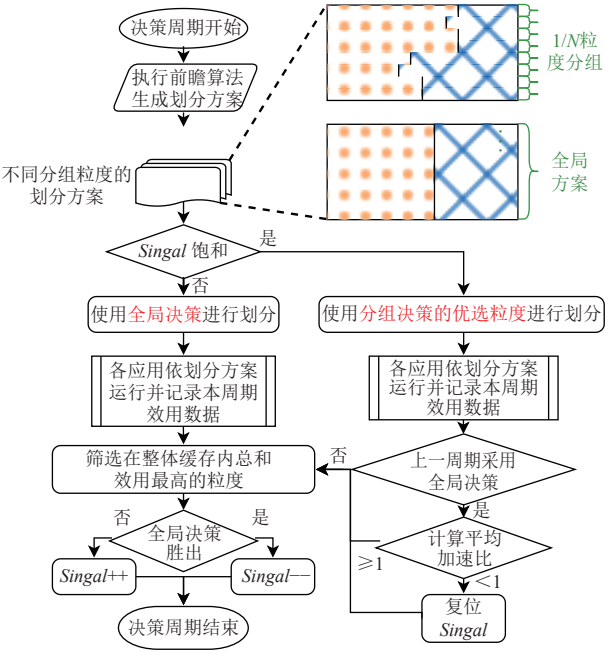


Fig. 6 Illustration of the selection process for optimal group granularity with regression validation

图 6 带有回归验证的优选分组粒度流程示意图

平均加速比如果小于1(意味着分组方案造成系统整体吞吐的下降),则将 *Signal* 复位. *Signal* 饱和计数器表示了系统对分组方案的信心,经过多次初步筛选,分组指标足够明确的时候才选择分组方案,这样可以避免频繁切换策略造成应用的性能抖动,并且能尽量减少针对不同方案测量 IPC 并比较的次数.

2.3 缓存块内存储元数据

细粒度动态 set 分组必然引入大量额外的监控开销,本文提出一种利用缓存块数据段存储监控信息的方法,有效降低了监控电路的开销. 如上文所述,分组决策方案依赖于各组的 SGUMON 所提供的数据,与原 UCP 全局共用一套 UMON 的方案相比,GroupUCP 的 UMON 电路数目成倍增长,带来了大量的硬件存储开销. 本文提出在缓存块内存储元数据的解决方案,将 SGUMON 的元数据存储于小部分缓存块中,有效降低了存储开销.

如同 1.2 节所述, UMON 需要存储的监控数据包含 2 个部分: ATD 以及效用计数器. 其中效用计数器在每个 SGUMON 内部共享一套,且每套计数器的数目仅为 $N_{\text{thread}} \times N_{\text{way}}$,因此同原 UCP 方案一样,将效用计数器设计为独立存储,其存储开销非常小,近乎可以忽略. 而针对开销巨大的 ATD,本文从最小化硬件开销和最小化性能开销 2 个角度做出了如下简化:

1)ATD 最小化硬件开销. 使用哈希(Hash)函数对 ATD 所存储的标记进行裁剪,并将 ATD 存储在部分缓存数据块内. 根据文献 [18, 41] 介绍,在进行 ATD 的构建时,使用裁剪、异或等简单的哈希操作,可以将标记有效缩小(如缩小至 11 b)的同时不影响监控精度,从而大大节省 ATD 的存储空间. 如图 7 所示,将 ATD 中所需要的标记以及 LRU 状态记作一个 entry,每个 entry 占用 2 B 的空间. 每个核心线程所需的 entry 数目应当与缓存的路数相当,然后将 entry 排布在一个缓存的数据块中,将其视作正常的数据存储在缓存中,这样存储 ATD 就不需要任何额外的硬件存储开销. 在本文的实验系统中, LLC 为 16 路组相连,数据块大小为 64 B, 只需要挪用 2 个缓存的数据块就足以存储 4 个核心的 ATD.

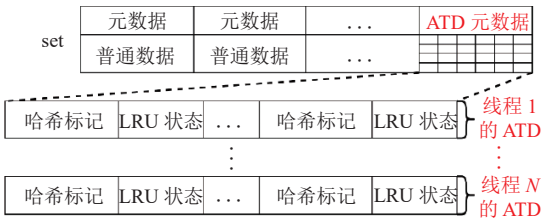


Fig. 7 Illustration for storing ATD metadata in cache lines

图 7 在缓存块存储 ATD 元数据的示意图

2)ATD 最小化性能开销. 为了能够进一步最小化挪用缓存块存储 ATD 对缓存性能的影响,本文工作使用了 DSS 方法[34] 减少需要监控的 set 数目,选取效用最低路的数据块存储 ATD 元数据,具体操作见算法 1. 在每个决策周期结束时,增添一个额外的步骤,筛选出效用最低的几路,在被采样的 set 上挪用这些空间用于保存 ATD 的元数据. 由于仅仅需要对极小部分被选中采样的 set 进行特殊的空间挪用分配方案,因此对系统整体性能的影响非常微小,同时极大地降低了硬件设计的存储开销.

算法 1. set 空间挪用筛选算法.

输入: 各个核心线程的路分配方案 ***alloc***, 各个核心线程的效用数据 ***utility***, 需要挪用路的总数 w;

输出: 在待监控 set 上的路分配方案 ***sampledAlloc***.

① ***sampledAlloc*** = ***alloc***; /*复制原分配方案为初始值*/

② while $w > 0$

③ for each t in threads

④ *lastUsedWay* ← *sampledAlloc*[*t*];

⑤ *lastUtility*[*t*] ← *utility*[*lastUsedWay*];

⑥ end for /*记录每一个线程中被分配的最后 1 路的效用值*/

⑦ *tChoice* ← *findMinIndex*(***lastUtility***); /*筛选所有分配的最后 1 路中效用最小的那一个*/

⑧ *sampledAlloc*[*tChoice*]——; /*将其分配空间缩小 1 路留作 ATD 元数据使用*/

⑨ w——;

⑩ end while

⑪ return ***sampledAlloc***.

3 实验评估

本节将评估相比于 UCP 方法, GroupUCP 能够带来的整体性能提升和开销情况. 具体包括: 1)通过静态分组和动态分组的性能表现及分析,证明细粒度按需分配确实可以在部分场景上带来性能收益; 2)回归验证方案的性能表现证明回归验证方案可以有效避免局部最优导致的性能下降; 3)将仅使用 DSS 采样的 GroupUCP 和使用 DSS 采样+缓存块存储元数据的 GroupUCP, 与使用 DSS 采样的 UCP 的性能进行对比; 4)LLC 尺寸倍增情况下 GroupUCP 的性能表现; 5)通过 GroupUCP 与 UCP 的硬件开销对比,证

明本文方案在提升性能的同时具有很低的开销.

3.1 实验配置及测量指标

1)实验配置. 本文选取的实验负载来自 SPEC CPU2017[38], GAP benchmark[39], Tailbench[40] 三个测试集. 对于 SPEC CPU2017 测试集中的应用, 我们使用 ref 作为输入集; 对于新兴的图计算 GAP 测试集中的应用, 我们使用节点尺寸为 2^{20} 的输入集; 对于在线服务类 Tailbench 测试集中的应用, 我们使用提供的默认参数来运行(注: masstree, silo, shore 包含指令集相关的代码, 无法移植到我们的测试平台上). 我们依照文献 [7, 42] 的分类, 从中挑选出 16 个缓存敏感型的应用, 这些应用的 IPC 会随着路划分分配的不同空间而变化, 且最大变化幅度会超过 10%. 由于实验需要应用混合成大量的负载组合, 为了缩短测试时间, 我们使用 SimPoint 技术[43] 为每个应用制作采样点, 并选取每个应用权重最大的 SimPoint 采样点代表该应用进行混合实验. 我们使用均衡随机(balanced random)抽样方法[44], 组成了 256 组 4 核测试负载. 我们指定预热长度为 250×10^6 的指令数, 当核 0 执行到此长度后, 进入正式的测试区域, 打开各调控策略, 当核 0 再次运行 250×10^6 指令之后结束测试.

我们使用 GEM5 模拟器[37] 搭建了 4 核测试平台, 参数配置如表 1 所示.

2)测量指标. 本文所研究的核心指标为系统吞吐率, 使用加权加速比(weighted speedup)作为量化指标[45-46], 对于 N 个应用, 其计算方法为:

$$WeightedSpeedup=\frac{1}{N}\sum_{i=1}^{N}\frac{Time_i}{Time_{i,\mathrm{NoPart}}},\tag{6}$$

其中 $Time$ 表示执行时间, NoPart 表示基础配置, 即不对 LLC 使用任何调控策略, 直接将应用混合运行, 让它们自由共享 LLC.

Table 1 Simulator Configuration

表 1 模拟器配置

部件	配置
核心	4 核 RISC-V 指令集 O3 处理器, 主频 2.66 GHz
1 级指令缓存	32 KB, 4 路组相连, 命中延迟 3 周期, LRU 替换算法
1 级数据缓存	32 KB, 4 路组相连, 命中延迟 3 周期, LRU 替换算法
2 级缓存	256 KB, 8 路组相连, 命中延迟 5 周期, LRU 替换算法
3 级缓存	8 MB, 16 路组相连, 命中延迟 35 周期, LRU 替换算法
内存	DDR3 DRAM, 1600MT/s, 8×8

3.2 分组技术的性能表现

我们试运行了 UCP 的全局方案, 选定每个决策周期的时间长度为 10×10^6 时钟周期, 本文进行的实验都选择了此决策周期长度. 本文首先将使用静态粒度分组方案的 GroupUCP-S 和动态粒度分组方案的 GroupUCP-D, 与 UCP-global 进行比较. 其中, GroupUCP-S 方案挑选 1/8 粒度为固定粒度; GroupUCP-D 限定粒度范围为 1/32 粒度到全缓存粒度. 结果如图 8 所示(横轴为混合应用负载按 UCP-global 性能提升降序排列), GroupUCP-S 与 UCP-global 相比, 256 组测试负载中的 190 组负载(占比 74.22%)在静态分组方案下获得了更好的性能提升(平均提升 0.33%, 最高提升 6.88%), GroupUCP-D 的 186 组负载(占比 72.66%)在动态分组方案下获得了更好的性能提升(平均提升 0.33%, 最高提升 7.32%). 相对于 NoPart, UCP-global 方案平均性能提升为 5.48%(最大 32.54%), 静态粒度分组的平均性能提升为 5.81%(最大 32.32%), 动态粒度分组的平均性能为 5.75%(最大 31.14%). 在全部的 256 组测

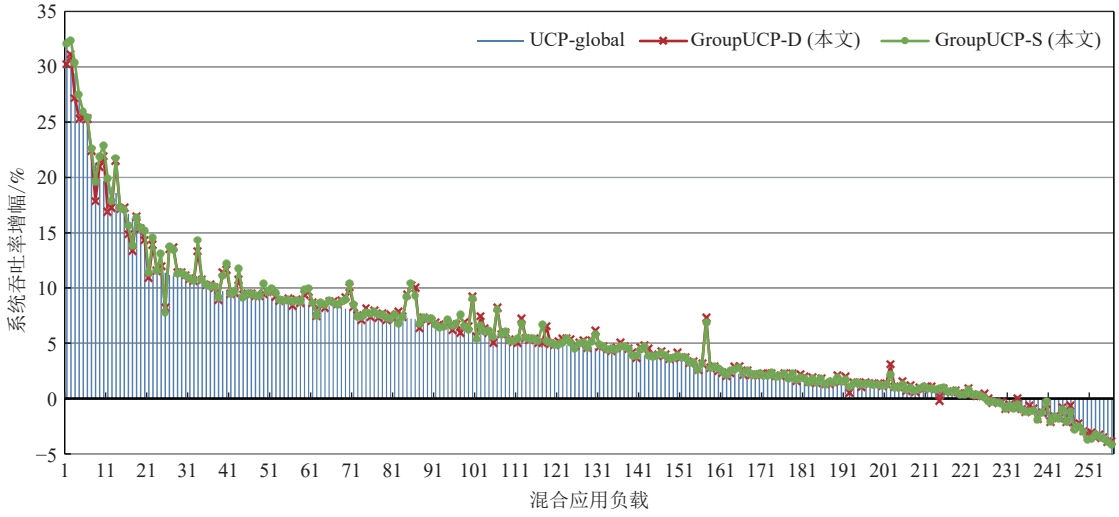


Fig. 8 Performance of GroupUCP-Static and GroupUCP-Dynamic

图 8 GroupUCP-Static 和 GroupUCP-Dynamic 的性能表现

试中, 有 31 组负载在 GroupUCP-S 方案下得到了超过 UCP-global 性能 1% 以上的增幅, 在 GroupUCP-D 方案下则有 30 组. 而在 GroupUCP-S 方案实验中, 有 5 组负载测试性能相对 UCP-global 下降了 1% 以上, 在 GroupUCP-D 方案中则有 10 组. 在本实验中, UCP 性能提升幅度与 GroupUCP 性能提升幅度相差最大的结果来自于测试组 moses-parest-omnetpp-cam4, 在原 UCP-global 方案仅能提升系统吞吐率 2.95% 的情况下, GroupUCP-S 和 GroupUCP-D 分别达到了 6.88% 与 7.32% 的系统吞吐率提升.

实验结果表明, 分组方法整体带来了更好的性能提升, 并可以在部分测试集上实现比 UCP 方法更高的性能收益. 与静态分组相比, 动态分组能够在某些应用组合上得到更好的性能提升, 但由于分组变多引入更多开销, 单纯进行动态分组并不会带来明显的性能提升.

3.3 使用回归验证策略的性能表现

鉴于前述实验发现, 大范围动态分组带来的收益有限, 我们将动态粒度方案限定为 1/8 粒度或全缓存粒度, 并取消中间大小的粒度. 我们将引入回归验证策略之后的方案称之为 GroupUCP-S-RV, 将分组置信度 *Signal* 计数器的位宽设置为 2 b, 即从初始状态至少经过 3 个决策周期就可以收敛到分组方案. 结果如图 9 所示(横轴为混合应用负载按 UCP-global 性能提升降序排列), 与 3.2 节表现较好的 GroupUCP-S 方案相比, 带有回归验证的 GroupUCP-S-RV 有效减少了性能波动的情况. 回归验证方案的平均性能提升为 5.75%(最大为 32.28%), 两者性能提升相差最大的结果发生在测试组 sphinx-moses-parest-cc_sv, 原 UCP-global 方案提升系统吞吐率为 10.63%, 带有回归验证的分组方案达到 14.29% 的系统吞吐率提升. 在全部的 256 组测试中, 有 24 组负载在 GroupUCP-S-RV 方案下得到了相对 UCP-global 性能 1% 以上的增幅, 仅有 1 组测试性能相对 UCP-global 下降 1% 以上.

实验结果表明, 引入回归验证策略可以有效减少分组方法由于过度追求局部最优造成的系统整体性能下降, 同时还能保持分组决策带来的性能提升.

3.4 使用采样技术的性能表现

本文选择 UCP 文献中提出的采样规格参数来进行 UCP-DSS 的实验, 即共准备 32 个被采样的 set, 并同样使用来自于文献 [35] 的采样函数进行选择. 对于 GroupUCP 方案, 本文为其准备 256 个被采样的 set, 使用同样的采样函数, 对于 1/8 分组粒度来说, 每个分组恰好有 32 个被采样的 set, 依旧使用静态分组与回归验证策略组合. 为了验证缓存块内存储元数据对整体性能的影响, 我们同时进行了 SGUMON 独立设计, 不占用存储块的方案, 称为 GroupUCP-S-RV-DSS, 该方案与 SGUMON 占用存储块存储 ATD 元数据的方案一样, 称为 GroupUCP-S-RV-DSS-MIB 的实验. 结果如图 10 所示(横轴为混合应用负载按 UCP-DSS 性能提升降序排列), 在使用采样的场景下, 分组方案依旧能获得更好的性能提升, 其中 UCP-DSS 的平均性能提升为 5.48%(最大为 32.49%), GroupUCP-S-RV-DSS 方案的平均性能提升为 5.72%(最大为 32.37%), GroupUCP-S-RV-DSS-MIB 方案得到的平均性能提升为 5.73%(最大为 32.73%). 在全部的 256 组测试中, 有 27 组在分组方案下得到了相对 UCP-DSS 性能 1% 以上的增幅, 有 5 组测试性能相对 UCP-DSS 下降了

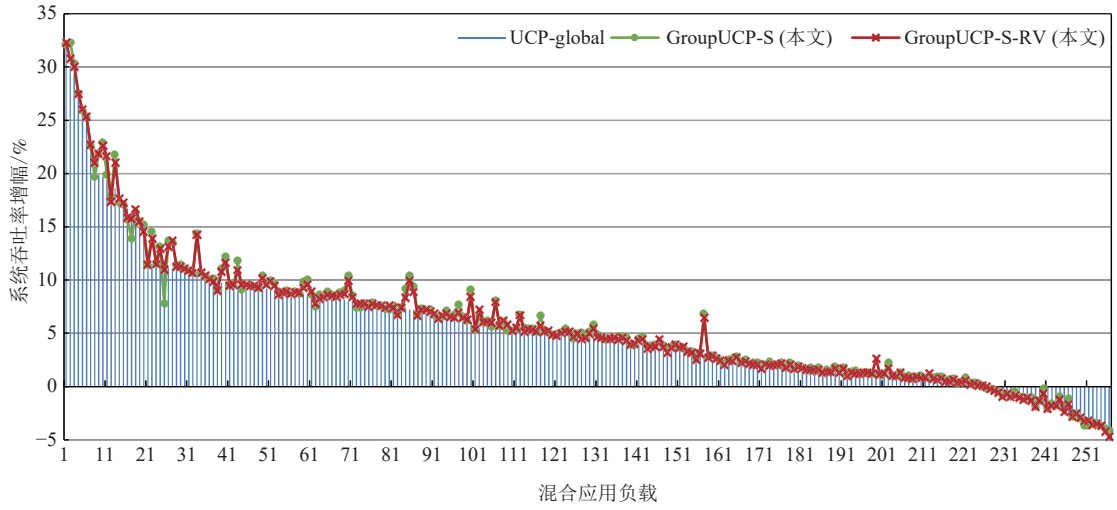


Fig. 9 Performance of GroupUCP-Static and GroupUCP-Static regression validation strategy

图 9 GroupUCP-Static 和引入回归验证策略的 GroupUCP-Static 的性能表现

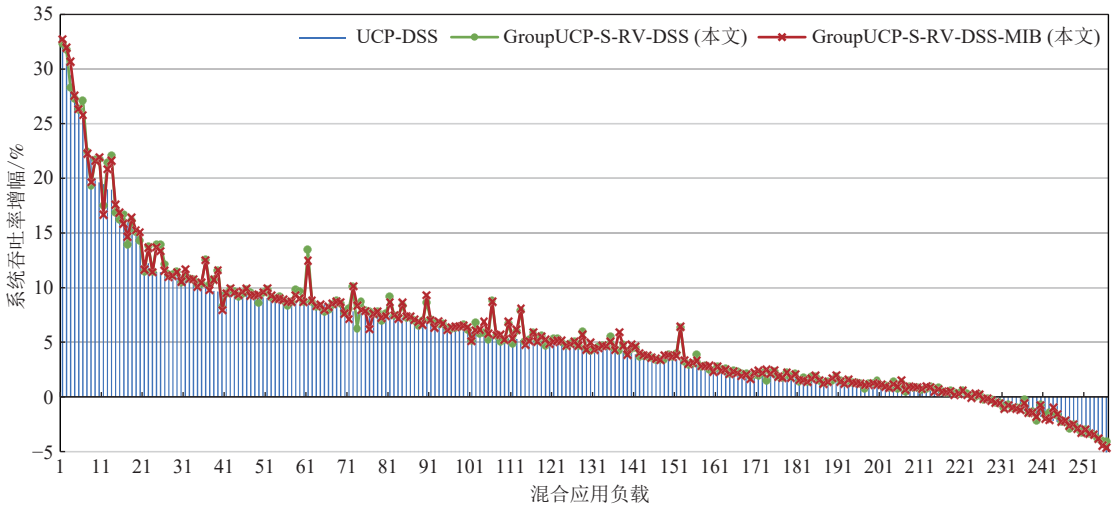


Fig. 10 Performance of GroupUCP with dynamic set sampling and storage metadata in cache block schemes

图 10 GroupUCP 进行动态组采样与缓存块存储元数据方案的性能表现

1% 以上. 性能提升的最大差值结果来自于测试组 xapian-xalancbmk-xz-lbm, 在原 UCP-DSS 方案提升系统吞吐率 8.58% 的情况下, GroupUCP-S-RV-DSS 达到了 13.46% 的系统吞吐率提升, 而 GroupUCP-S-RV-DSS-MIB 达到了 12.44% 的系统吞吐率提升.

实验结果表明, 在使用采样技术之后, GroupUCP 方法同样可以拿到比 UCP 方法更好的性能收益, 并在某些应用组合下可以拿到较大的性能收益. 而且, 在使用采样技术的前提下, 复用缓存块存储 ATD 元数据仅需占用很少部分 set 的缓存空间, 对系统整体性能影响非常小.

3.5 扩展至双倍 LLC 尺寸下的性能表现

为了验证 LLC 尺寸拓展后 GroupUCP 的性能表现, 我们将 LLC 尺寸翻倍, 同时维持其他参数(如相连度等)规格不变, 进行扩展性实验. 同样使用上个实验中的采样参数, 比较 UCP-DSS 方案与 GroupUCP-S-RV-DSS-MIB 方案, 结果如图 11 所示(横轴为混合应用负载按 UCP-DSS 性能提升降序排列), 其中 UCP 的平均性能提升为 0.35%(最大为 9.26%), 而 GroupUCP 的平均性能提升为 0.76%(最大为 8.80%). 在全部的 256 组测试中, 有 21 组得到了相对 UCP-DSS 性能 1% 以上的增幅, 有 1 组测试性能相对 UCP-DSS 下降了 1%. 另外, 在 23 组测试中, UCP-DSS 对系统吞吐率造成了负面影响, 而 GroupUCP 能带来正向提升.

实验说明, 在大尺寸 LLC 之下, 原有 UCP 方法使用全局调控粒度太粗的问题将会加重, 很有可能对系统吞吐率带来负面影响, 而 GroupUCP 则能通过更细粒度的调节, 为系统吞吐带来性能提升.

3.6 硬件开销

如同 1.2 节所说, UCP 类策略的硬件开销主要来

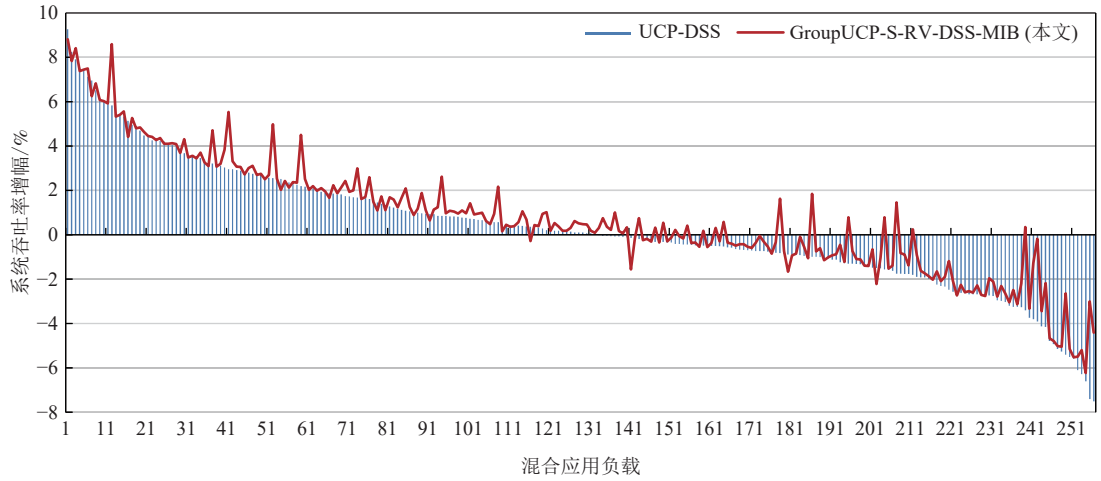


Fig. 11 Performance of GroupUCP under double-size LLC

图 11 双倍尺寸 LLC 下 GroupUCP 的性能表现

自于 UMON, 而 GroupUCP 通过缓存块内存储元数据的手段, 有效降低了 UMON 的开销. 使用缓存块存储其他数据的方式已有先进预取器实现[47-48], 不会带来高额的硬件开销. 我们将原 UCP 的 UMON-DSS 结构的开销与 GroupUCP 的 UMON-group 结构的开销进行对比, 对比结果列在表 2 中.

Table 2 Hardware Overhead Comparison Between UMON-DSS and UMON-group

表 2 UMON-DSS 与 UMON-group 的硬件开销对比

部件开销或数量	UCP	GroupUCP
每个 ATD entry 的大小	16 b (标签) + 4 b (LRU) + 1 b (有效位) = 21 b	11 b (标签) + 4 b (LRU) + 1 b (有效位) = 16 b
每个采样 set 需要存储 ATD 的额外开销	21 b×16 = 42 B	0 (块内存储)
采样 set 数量	32	256
ATD 额外开销	32×42 B = 1 344 B	0
每套效用计数器开销	16×2 B = 32 B	16×2 B = 32 B
效用计数器总开销	32 B	8×32 B = 256 B
开销总计	1 376 B	256 B

通过对比可以看出, GroupUCP 可以使用远小于 UCP 的硬件开销完成更细粒度的资源监控和更高精度的资源按需分配. GroupUCP 使用的核心前瞻算法与 UCP 相同, 仅需增添一些额外的状态机来进行状态的记录与转移判断, 这些并不会引入复杂的电路, 并且决策电路并非位于关键路径上, 可以进行多周期流水化设计, 在本文的实验环境中, UCP 完成决策预期需要的时间不超过 350 个时钟周期, 而 GroupUCP 完成决策预期需要的周期数不超过 3 000 个时钟周期, 而决策电路在每一个决策周期(如本文中实验环境为 10 M 时钟周期)仅需运行 1 次, 带来的额外开销可以忽视. 我们使用 CACTI[49] 对缓存的整体功耗以及 GroupUCP 的功耗进行估测. 实验表明, 全缓存的漏电功耗为 2 768.24 mW, 而 GroupUCP 部件的漏电功耗仅为 1.56 mW(小于整体功耗的 0.06%), 而且, 在使用了 DSS 技术之后, 监控电路仅在访问到被采样的 set 上才会运行, 对总体功耗的影响很小.

4 结 论

本文提出了 GroupUCP, 实现了一种能够细粒度按需分配, 并同时具有低开销的 LLC 空间资源动态分配机制来提升系统整体吞吐性能. 本文针对应用使用 set 空间情况的不同, 对路划分技术进行优化, 将 set 进行动态分组, 然后在不同分组上使用不同的路划分调节, 达到细粒度按需分配的效果. 并通过少量借用部分 LLC 数据块来降低大量硬件开销. 与 UCP 相比, GroupUCP 能够给系统带来更高的性能提升, 并且在大 LLC 设计下表现出更好的扩展性. 未来的工作可以关注如何更加高效地组织元数据存储, 以及更加复杂的动态分组方式.

作者贡献声明: 张传奇提出了设计思路和实验方案; 王卅撰写论文; 孙凝晖提出指导意见; 包云岗修改并审核论文.

参 考 文 献

[1] Wulf Wm A, McKee S A. Hitting the memory wall: Implications of the obvious[J]. ACM SIGARCH Computer Architecture News, 1995, 23(1): 20−24

[2] Wuu J, Agarwal R, Ciraula M, et al. 3D V-Cache: The implementation of a hybrid-bonded 64 MB stacked cache for a 7nm x86−64 CPU[C]//Proc of 2022 IEEE Int Solid-State Circuits Conf (ISSCC). Piscataway, NJ: IEEE, 2022: 428−429

[3] Advanced Micro Devices, Inc. 3rd Gen AMD EPYC processors with AMD 3D V-Cache technology deliver outstanding leadership performance in technical computing workloads[EB/OL]. (2022-03-21)[2023-11-03].https://www.amd.com/en/press-releases/2022-03-21-3rd-gen-amd-epyc-processors-amd-3d-v-cache-technology-deliver-outstanding

[4] Yang Ailin. Resolving noisy neighbors[EB/OL]. (2022-10-19)[2023-09-20]. https://www.intel.com/content/www/us/en/developer/articles/technical/noisy-neighbors-problem-in-kubernetes.html

[5] Kim S, Chandra D, Solihin Y. Fair cache sharing and partitioning in a chip multiprocessor architecture[C]//Proc of the 13th Int Conf on Parallel Architecture and Compilation Techniques. Piscataway, NJ: IEEE, 2004: 111−122

[6] Chen Shimin, Gibbons P B, Kozuch M, et al. Scheduling threads for constructive cache sharing on CMPs[C]//Proc of the 19th ACM Symp Parallel Algorithms and Architectures. New York: ACM, 2007: 105−115

[7] 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: 104−117

[8] Xu Cong, Rajamani K, Ferreira A, et al. dCat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service[C/OL]//Proc of the 13th EuroSys Conf. New York: ACM, 2018[2023-11-30]. https://doi.org/10.1145/3190508.3190555

[9] Nguyen K. Introduction to cache allocation technology in the Intel® Xeon®[EB/OL].[2023-11-20]. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-

technology.html

[10] Advanced Micro Devices, Inc. AMD64 technology platform quality of service extensions[EB/OL]. 2018[2024-01-01]. http://kib.kiev.ua/x86docs/AMD/MISC/56375_1.00_PUB.pdf

[11] Arm Limited. Memory system resource partitioning and monitoring (MPAM), for A-profile architecture[EB/OL]. 2022[2023-11-20]. https://developer.arm.com/documentation/ddi0598/latest

[12] Xiang Yaocheng, Wang Xiaolin, Huang Zihui, et al. DCAPS: Dynamic cache allocation with partial sharing[C/OL]//Proc of the 13th EuroSys Conf. New York: ACM, 2018[2023-11-30]. https://doi.org/10.1145/3190508.3190511

[13] Chen Ruobing, Wu Jinping, Shi Haosen, et al. DRLPart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers[C]//Proc of the 30th Int Symp on High-Performance Parallel and Distributed Computing. New York: ACM, 2021: 175−188

[14] Qureshi M K, Patt Y N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches[C]//Proc of the 39th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2006: 423−432

[15] Shahrad M, Balkind J, Wentzlaff D. Architectural implications of function-as-a-service computing[C]//Proc of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2019: 1063−1075

[16] Rolán D, Fraguela B B, Doallo R. Adaptive line placement with the set balancing cache[C]//Proc of the 42nd Annual IEEE/ACM Int Symp on Microarchitecture (MICRO). New York: ACM, 2009: 529−540

[17] Zhan Dongyuan, Jiang Hong, Seth S C. Exploiting set-level non-uniformity of capacity demand to enhance CMP cooperative caching[C/OL]//Proc of 2010 IEEE Int Symp on Parallel & Distributed Processing (IPDPS). Piscataway, NJ: IEEE, 2010[2023-11-29]. https://ieeexplore.ieee.org/document/5470441

[18] Zhan Dongyuan, Jiang Hong, Seth S C. STEM: Spatiotemporal management of capacity for intra-core last level caches[C]//Proc of the 43rd Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2010: 163−174

[19] Rolán D, Fraguela B B, Doallo R. Adaptive set-granular cooperative caching[C]//Proc of the 18th IEEE Int Symp on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2012: 213−224

[20] Barroso L A, Hölzle U, Ranganathan P. The Datacenter as A Computer: Designing Warehouse-Scale Machines[M]. Cham: Springer, 2019

[21] Ma Jiuyue, Yu Zihao, Bao Yungang, et al. A programmable data plane design in computer architecture[J]. Journal of Computer Research and Development, 2017, 54(1): 123−133 (in Chinese)
(马久跃, 余子濠, 包云岗, 等. 体系结构内可编程数据平面方法[J]. 计算机研究与发展, 2017, 54(1): 123−133)

[22] Delimitrou C, Kozyrakis C. iBench: Quantifying interference for datacenter applications[C]//Proc of 2013 IEEE Int Symp on Workload Characterization (IISWC). Piscataway, NJ: IEEE, 2013: 23−33

[23] Sherwood T, Calder B, Emer J. Reducing cache misses using hardware and software page placement[C]//Proc of the 13th Int Conf on Supercomputing. New York: ACM, 1999: 155−164

[24] Tam D, Azimi R, Soares L, et al. Managing shared L2 caches on multicore systems in software[C]//Proc of Workshop on the Interaction between Operating Systems and Computer Architecture. New York: ACM, 2007: 26−33

[25] Jin Xinxin, Chen Haogang, Wang Xiaolin, et al. A simple cache partitioning approach in a virtualized environment[C]//Proc of 2009 IEEE Int Symp on Parallel and Distributed Processing with Applications. Piscataway, NJ: IEEE, 2009: 519−524

[26] Lin Jiang, 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 IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2008: 367−378

[27] Zhang Xiao, Dwarkadas S, Shen Kai. Towards practical page coloring-based multicore cache management[C]//Proc of the 4th ACM European Conf on Computer Systems. New York: ACM, 2009: 89−102

[28] Zhang Ludan, Liu Yi, Wang Rui, et al. Lightweight dynamic partitioning for last level cache of multicore processor on real system[C]//Proc of the 13th Int Conf on Parallel and Distributed Computing, Applications and Technologies. Piscataway, NJ: IEEE, 2012: 33−38

[29] Ye Ying, West R, Cheng Zhuoqun, et al. COLORIS: A dynamic cache partitioning system using page coloring[C]//Proc of the 23rd Int Conf on Parallel Architectures and Compilation. New York: ACM, 2014: 381−392

[30] Qureshi M K, Thompson D, Patt Y N. The V-Way cache: Demand-based associativity via global replacement[C]//Proc of the 32nd Int Symp on Computer Architecture (ISCA'05). Los Alamitos, CA: IEEE Computer Society, 2005: 544−555

[31] Varadarajan K, Nandy S K, Sharda V, et al. Molecular caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions[C]//Proc of the 39th Annual IEEE/ACM Int Symp on Microarchitecture (MICRO'06). New York: ACM, 2006: 433−442

[32] Beckmann N, Sanchez D. Jigsaw: Scalable software-defined caches[C]//Proc of the 22nd Int Conf on Parallel Architectures and Compilation Techniques. Piscataway, NJ: IEEE, 2013: 213−224

[33] Sanchez D, Kozyrakis C. Vantage: Scalable and efficient fine-grain cache partitioning[C]//Proc of the 38th Annual Int Symp on Computer Architecture (ISCA). New York: ACM, 2011: 57−68

[34] Brock J, Ye Chencheng, Ding Chen, et al. Optimal cache partition-sharing[C]//Proc of the 44th Int Conf on Parallel Processing. Los Alamitos, CA: IEEE Computer Society, 2015: 749−758

[35] Qureshi M K, Lynch D N, Mutlu O, et al. A case for MLP-aware cache replacement[C]//Proc of the 33rd Int Symp on Computer Architecture (ISCA'06). Los Alamitos, CA: IEEE Computer Society, 2006: 167−178

[36] Rajkumar R, Lee C, Lehoczky J, et al. A resource allocation model for QoS management[C]//Proc of the 18th Real-Time Systems Symp. Piscataway, NJ: IEEE, 1997: 298−307

[37] Binkert N, Beckmann B, Black G, et al. The GEM5 simulator[J]. ACM SIGARCH Computer Architecture News, 2011, 39(2): 1−7

[38] Bucek J, Lange K D, Von Kistowski J. SPEC CPU2017: Next-generation compute benchmark[C]//Proc of the 9th ACM/SPEC Int Conf on Performance Engineering. New York: ACM, 2018: 41−42

[39] Beamer S, Asanović K, Patterson D. The GAP benchmark suite[J]. arXiv preprint, arXiv: 1508.03619, 2017

[40] Kasture H, Sanchez D. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications[C]//Proc of 2016 IEEE Int Symp on Workload Characterization (IISWC). Piscataway, NJ: IEEE, 2016: 3−12

[41] Jain A, Lin C. Back to the future: Leveraging Belady's algorithm for improved cache replacement[C]//Proc of the 43rd ACM/IEEE Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2016: 78−89

[42] Jaleel A, Hasenplaugh W, Qureshi M, et al. Adaptive insertion policies for managing shared caches[C]//Proc of the 17th Int Conf on Parallel Architectures and Compilation Techniques (PACT). Piscataway, NJ: IEEE, 2008: 208−219

[43] Sherwood T, Perelman E, Hamerly G, et al. Automatically characterizing large scale program behavior[C]//Proc of the 10th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2002: 45−57

[44] Velásquez R A, Michaud P, Seznec A. Selecting benchmark combinations for the evaluation of multicore throughput[C]//Proc of 2013 IEEE Int Symp on Performance Analysis of Systems and Software (ISPASS). Piscataway, NJ: IEEE, 2013: 173−182

[45] Eyerman S, Eeckhout L. System-level performance metrics for multiprogram workloads[J]. IEEE Micro, 2008, 28(3): 42−53

[46] Snavely A, Tullsen D M. Symbiotic jobscheduling for a simultaneous multithreaded processor[C]//Proc of the 9th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2000: 234−244

[47] Wu Hao, Nathella K, Pusdesris J, et al. Temporal prefetching without the off-chip metadata[C]//Proc of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2019: 996−1008

[48] Wu Hao, Nathella K, Pabst M, et al. Practical temporal prefetching with compressed on-chip metadata[J]. IEEE Transactions on Computers, 2022, 71(11): 2858−2871

[49] Muralimanohar N, Balasubramonian R, Jouppi N P. CACTI 6.0: A tool to model large caches[EB/OL]. (2014-05-30)[2024-03-10]. https://www.researchgate.net/publication/242516869_Cacti_60_A_tool_to_model_large_caches



Zhang Chuanqi, born in 1996. PhD candidate. Student member of CCF. His main research interest includes data center computer architecture.

张传奇, 1996 年生. 博士研究生. CCF 学生会员. 主要研究方向为数据中心体系结构.



Wang Sa, born in 1986. PhD, associate professor. Member of CCF. His main research interests include cloud computing, operating systems, and system modeling and performance analysis.

王　卅, 1986 年生. 博士, 副研究员. CCF 会员. 主要研究方向为云计算、操作系统、系统建模与性能分析.



Sun Ninghui, born in 1968. PhD. Academician of Chinese Academy of Engineering. Fellow of CCF. His main research interests include computer architecture and high performance computing.

孙凝晖, 1968 年生. 博士. 中国工程院院士. CCF 会士. 主要研究方向为计算机系统结构、高性能计算.



Bao Yungang, born in 1980. PhD, professor. Member of CCF. His main research interests include datacenter architecture, agile design methodology of processor chips, and ecosystem of open-source processor chips.

包云岗, 1980 年生. 博士, 研究员. CCF 会员. 主要研究方向为数据中心体系结构、处理器芯片敏捷设计方法论、开源芯片生态.