基于机器学习的跨平台缓存划分方法研究

邱杰凡[1)] 贾逸哲[1)] 华宗汉[1),2)] 曹明生[3)] 范 菁[1)]

[1)](浙江工业大学计算机科学与技术学院 杭州 310023)

[2)](中国人民银行桐庐县支行 杭州 311500)

[3)](电子科技大学网络与数据安全四川省重点实验室 成都 610054)

摘 要 多核处理器的最后一级缓存(Last Level Cache,LLC)采用共享机制,当多个程序并行随机访问LLC时,可能引起访存冲突,进而导致系统整体性能的大幅下降.已有研究者试图通过引入缓存划分,合理安排不同程序对LLC的访问时机,解决访存冲突问题.然而,现有的缓存划分方法主要采用启发式算法,即通过不断"试错"寻找缓存划分最优方案,寻找过程具有不确定性.这种不确定性可能导致划分开销过大甚至出现无法收敛的问题.为此,我们提出了一种基于机器学习的跨平台缓存划分方法MLPart.该方法借助少量运行参数,利用决策树和序列到序列模型预测剩余各个划分方案的性能,直接找到最优的划分方案,从而保证划分开销相对稳定且有效避免了无法收敛的情况.此外,实验表明如果计算平台之间存在着微小配置差异,即使运行相同程序所产生的运行参数也存在较大差异,因此单一模型通常没有足够的泛化能力适应不同的平台.为解决该问题,我们分别利用迁移学习技术和微调技术优化决策树和序列到序列模型,使其能以较低的计算开销完成不同平台的模型快速迁移部署.我们以当前主流的Intel至强(Xeon)处理器平台展开实验,实验结果表明MLPart能够通过少量划分步骤找到性能最佳的缓存划分方案;并且,与采用启发式缓存划分方法KPart,采用基于贝叶斯优化的缓存划分方法CLITE以及采用机器学习的缓存划分方法C&A相比,MLPart对工作集的性能提升更高且更为稳定.

关键词 缓存划分;递归神经网络;决策树;序列到序列模型;迁移学习

中图法分类号 TP18 **DOI号** 10.11897/SP.J.1016.2023.02097

A Multi-platform Cache Partitioning Method Based on Machine Learning

QIU Jie-Fan[1)] JIA Yi-Zhe[1)] HUA Zong-Han[1),2)] CAO Ming-Sheng[3)] FAN Jing[1)]

[1)](*Department of Computer Science and Technology, Zhejiang University of Technology, Hangzhou* 310023)

[2)](*The People's Bank of China Tonglu Sub-branch, Hangzhou* 311500)

[3)](*University of Electronic Science and Technology of China, Chengdu* 610054)

Abstract The Last Level Cache (LLC) is shared in a multi-level-cache and multi-core CPU. If multi-programming randomly access LCC, it incurs accessing conflicts, and results in severe performance degradation. Currently, researchers focus on cache partitioning to reduce LLC conflicts. Existing cache partitioning methods adopt heuristic algorithm which needs uncertain steps to optimal performance. Thus it may bring up excessive partitioning overheads or even failure to converge in the partitioning process. To this end, we propose MLPart, a machine learning-based multi-platform adaptive cache partitioning method. With a few of running parameters, MLPart employs the decision tree and sequence-to-sequence models respectively to

收稿日期:2022-07-12;在线发布日期:2023-05-08.本课题得到国家重点研发计划"现代服务业共性关键技术研发及应用示范项目"(2018YFB1402800)和浙江省自然科学基金(LY20F020026)资助.**邱杰凡**,博士,副教授,中国计算机学会(CCF)会员,主要研究领域为操作系统、无线传感器网络、人工智能.E-mail:qiujiefan@zjut.edu.cn.**贾逸哲**,硕士研究生,主要研究领域为嵌入式操作系统、智能计算.**华宗汉**,硕士,主要研究领域为嵌入式操作系统、计算机体系结构.**曹明生**,博士,副研究员,主要研究领域为人工智能、网络空间安全.**范 菁**,博士,教授,主要研究领域为服务计算、人工智能.

predict the performance of the different partitioning policies, and finally find out the best-performance partitioning policy with certain partition steps. In addition, the experiments show that even the same program runs in different computing platforms and result in the totally different runtime parameters. It is hard to ensure that trained models have sufficient generalization ability to be deployed in different platforms. MLPart also adopt transfer learning and tuning techniques to optimize the decision tree and sequence-to-sequence models for reducing the training overheads on other platforms. The experimental results illustrate that MLPart only needs a few of partition steps to find the optimal cache partitioning policy on Intel Xeon processor. The overall performance improvements using MLPart are highest and most stable in the KPart, CLITE and C&A.

Keywords cache partition; recurrent neural network; decision tree; sequence to sequence; transfer learning

1 引 言

当前主流的多核处理器的最后一级高速缓存(Last Level Cache,LLC)多采用共享模式,即并行运行在多核处理器上的所有程序按需共同使用LLC,以保证在程序进程之间的公平性.然而,实际使用缓存的数量对不同应用程序性能和可调度性的影响并不相同[1-2].经典的缓存替换策略仅关注数据的被访问情况,并不考虑数据与具体应用程序之间的关系[3].这意味着已经存储在LLC上的程序数据会由于近期访问少被其他程序的数据覆盖.当要访问已被覆盖的程序数据时,必然出现缓存缺失现象,引发工作集中多个并行程序在LLC上的冲突.这种冲突会直接导致部分程序性能的严重下降,进而影响工作集的整体性能[4-8].

目前可解决LLC冲突的一种思路是缓存划分,即通过为重要程序划分特定LLC空间,可以确保工作集中的特定程序不受其他程序干扰,保护其性能.如果为工作集中的所有程序进行缓存划分,也可以在一定程度上保证工作集的整体公平性[9].在前期的工作中,研究人员分别基于软件或硬件的方法实现了缓存划分[10-20],但这两类方法都存在一些问题.首先,基于软件的缓存划分方法通常依赖于页着色(Page Coloring)技术[10,12-17],该技术无法与巨页(Huge Page)兼容[21],并且重新调整划分方案的开销较高.其次,基于硬件的缓存划分方法[11,18-20]虽然可以克服上述不足,不过需要在底层增加相应的硬件专用模块,通用性较差,并不适合在商用处理器中大规模应用.来自Intel公司与Cavium公司的研究人员将两种方法结合,在其高性能处理器上实现了以缓存路(cache way)为最小缓存划分单位的粗粒度缓存划分方法[22-23],在兼顾通用性的同时克服了基于软件的缓存划分方法的不足.然而,缓存路的粗粒度会导致划分方案相对有限,在实际应用过程中,往往划分给应用程序的LLC与其所需不匹配,而没有得到足够LLC的应用程序性能进一步下降.

为此,研究人员对基于缓存路的缓存划分方法进行了深入研究.当前基于缓存路的方法存在两方面的不足[21,24-25].首先,这类方法监控的运行参数过少,例如KPart[21]只监控Cache缺失和每时钟周期指令数(Instruction Per Clock,IPC)等参数,根据各个运行参数与阈值的比较,再对程序分类,并在运行过程中根据分类结果实施缓存划分,虽然易于实现,但是仅仅参考一个或少数几个运行参数就得出结论是片面的,容易做出错误的判断.其次,这类方法[21,25]一般采用启发式的思想,导致划分开销具有不确定性.这是由于启发式缓存划分方法[24]无法直接确定高性能缓存划分方案,需要不断依据性能监视器的反馈调整缓存划分方案,直至接近或达到预期性能,在特定情况下划分开销本身会超过缓存划分所带来的性能提升收益,得不偿失.

近年来,随着处理器性能的大幅提升,机器学习技术也被应用于计算资源的调度[26-28].以缓存划分为例,由于机器学习基于统计思想,需要先建立划分模型.与启发式缓存划分方法相比,在运行过程中,能够以预测的方式找到相对最优的划分方案,总体划分开销可控.且划分模型通常对数据维度并不敏感.因此,在构建模型时可以考虑更多程序的特征行为,有利于更加精确满足不同程序的动态缓存需求.

然而,直接将机器学习技术应用于缓存划分存在两方面的不足.首先,现有基于机器学习的资源

调度方法一般不会采用启发式算法中较为常见的先对程序分类再以分类结果为依据划分缓存（先分类后划分）的做法，而是直接根据每个程序的访存行为为程序划分缓存[21,29-31]. 但受到处理器中缓存路的数量限制，这种精细化的划分策略，反而使得划分策略僵化，引起一些不必要的缓存冲突，导致吞吐量提升有限. 反而，先分类后划分的做法可以使同类程序缓存共享相同的缓存路，形成更加灵活的资源划分和共享策略，避免依据程序单独划分所造成的分配资源不足或浪费. 不仅如此，先分类后划分能够更加充分利用不同类别程序的访存行为特征，提供更合理的划分策略. 其次，机器学习技术需要预先构建缓存划分模型，如果采用离线学习方式，一旦程序运行平台参数发生变化，且模型的泛化能力不足，则性能不可避免会受到影响，而重构模型开销较大，最终模型重构引起的额外计算开销与缓存划分所带来的吞吐量提升并不相称. 为此，已有研究者[32-33]提出利用在线学习技术，通过多次动态修正模型，避免因平台参数变化引起的划分模型重构，从而降低模型适应新平台所需的计算开销. 但是在线学习在一些应用场景中的整体收敛速度偏慢，有可能进一步增加总体划分开销.

针对上述问题，我们提出了一种基于机器学习的跨平台缓存划分方法MLPart. 在MLPart中，我们首先应用决策树（Decision Tree，DT），根据程序使用缓存时对其他程序的干扰程度以及程序本身对缓存的敏感程度，将程序自动分为强干扰程序，缓存敏感程序，以及缓存不敏感程序. 其次，利用递归神经网络的变种模型——序列到序列模型（Sequence to Sequence，Seq2Seq）预测缓存划分方案下工作集的总体性能，根据最高性能的缓存分配方案对缓存进行划分. 此外，由于程序运行平台之间配置不同（如CPU核数、内存频率和内存容量等存在差异），导致同一个应用程序在不同的平台上运行时的运行参数不同. 如果在目标平台上重新采集数据并训练数据集，开销过大. 为此在MLPart中，我们对迁移学习技术TrAdaboost[34]进行了改进，通过少量的数据采集和模型训练开销对权重进行调整，以适应新平台的数据分布，避免了因泛化能力不足导致的程序分类模型重新构建，并利用微调技术[35]完成缓存划分模型的跨平台优化

本文的贡献点如下：

（1）提出一种基于决策树的程序分类方法. 该方法将程序按照对缓存的使用特点分类，为缓存划分提供依据，通过提高缓存划分的灵活性降低对缓存资源的浪费.

（2）提出一种基于序列到序列模型的缓存划分方法. 该方法能够在保证缓存划分开销可控的前提下，找到最优或次优的缓存划分方案. 经实验评估，该方法在提升工作集整体性能上，相较于已有的三种缓存划分方法CLITE，KPart和C&A分别提升了11.31%、12.96%和1.67%.

（3）应用改进的TrAdaBoost迁移学习技术和微调技术，实现模型的跨平台移植. 以少量数据采集开销，达到了与源平台性能基本一致的划分效果，有效降低了跨平台部署模型的开销. 实验表明，模型跨平台后，其性能相较于上述三种缓存划分方法，仍能分别提升13.13%、13.46%、4.8%.

本文第2节介绍了缓存划分的相关研究工作，以及将机器学习应用于缓存划分面临的挑战与难点；第3节介绍了具体实验平台，并通过实验阐述了应用先分类后划分策略的动机；第4节详细介绍了MLPart的整体框架和具体实现；第5节给出了实验结果并进行了分析；最后在第6节总结我们的工作.

2　相关工作

本节在第2.1节中总结了缓存划分的意义以及先前缓存划分工作中存在的不足；在第2.2节中概括了机器学习在计算资源划分中的挑战.

2.1　缓存划分的发展

缓存中的页面置换策略（如最近最少使用策略）[3]程序的性能指标作为页面置换的依据. 这意味着在缓存中，任意程序的数据都有很大的概率会被其他程序的数据替换，引起工作集整体的吞吐量下滑. 特别是如果工作集中存在流式程序时，该类程序可能频繁地造成缓存缺失，需要替换其他程序的数据块，引起系统整体性能与公平性的大幅下降.

一般认为程序性能的主要瓶颈在于缓存命中率[36]. 因此，以程序性能指标为依据的缓存划分能够使程序的一部分数据长时间保存在LLC中以提高缓存命中率，保证了工作集中程序的整体性能. 此外，通过对工作集中的程序合理地划分LLC，也可以保证工作集的整体公平性. 不少研究者借助软件或硬件的方法[10,12-20]实现了LLC的划分，但是这些方法都有各自的问题. 基于软件的方法普遍以页着色技术作为实现缓存划分的手段. 该技术不需要专门的硬件模块支持，但存在着诸多缺陷. 首先，该

技术无法与巨页(huge page)技术兼容.巨页技术增大了单一页表项覆盖的内存区域,从而降低页表缓存的缺失,最终提升系统访存性能.以页着色为基础的缓存划分所带来的性能提升可能远不及巨页技术所带来的性能提升[21].其次,页着色在动态调整过程中需要执行页面复制操作,时间开销较大.再次,页着色技术控制程序占用的LLC受到程序内存足迹的限制,即程序的内存足迹越大,其被划分的LLC也必须越大.而在一些情况下,会出现程序占用大量内存并只占用少量LLC的情况.最后,页着色的适配较为繁琐,适配过程需要确定用于映射缓存组(Cache Set)的地址位,进一步增加了适配时间.

另外,已有研究人员提出了诸多基于硬件的缓存划分方法[11,18-20].这些方法能够克服基于软件的缓存划分方法的不足,但是这些方法需要专门的硬件模块支持,通用性较差,很难直接应用于商用处理器中.来自Intel公司的研究人员结合上述两种方法,在处理器上实现了以缓存路(cache way)为最小分区的粗粒度缓存划分[37],即缓存分配技术(Cache Allocation Technology,CAT).Cavium公司的研究人员也实现了类似的技术[23].CAT避免了基于软件的缓存划分方法带来的巨额开销,且无需再引入专用硬件模块,兼顾了划分方法的通用性.但是这种以缓存路为最小分区的粗粒度划分方法限制了划分方案的数目,导致分配给工作集中应用程序的缓存资源与程序所需资源并不一定匹配.

为了克服CAT的不足,研究人员提出了各种基于CAT的缓存划分优化方法.然而,这些方法多数基于启发式算法实现,即需要根据性能监视器的反馈结果不断调整资源划分方案,直到整体性能满足要求为止[21,24-25].以CoPart[25]中的缓存划分为例,其构建了一个有限状态机,根据检测到的缓存缺失与命中情况以及IPC变化,使应用程序在增加缓存路,减少缓存路以及维持现状三个状态中不断转换.KPart[21]则借助变更缓存资源划分,获取每个程序加速比随缓存分配的变化曲线,并根据变化曲线对程序分类;然后,结合启发式算法和Lookahead算法[4],根据程序的分类分配缓存资源,以获得最优资源划分方案.

然而,启发式算法的总体划分开销具有一定不确定性.这是由于为了防止漏过合理的高性能划分方案,每一次调整的步长会被限制.而缓存划分的搜索空间较大,可能需要多次调整才能找到符合预期性能的缓存划分方案.其次,启发式算法只借助少数运行参数作为依据用于判断是否需要调整.参数一般基于若干特定程序的分析实验获得,如果参数种类过少也会导致判断依据片面化.

2.2 机器学习在计算资源划分中的应用

考虑到机器学习算法的复杂度,计算机体系结构领域较少直接应用机器学习方法.然而,随着处理器性能的提升以及GPU和AI智能芯片的大量应用,近年来,一些研究开始借助机器学习在实现了多维度的联合计算资源上划分.例如,CLITE采用贝叶斯优化方法,可以对CPU核,LLC以及内存带宽进行联合划分[32].C&A[38]采用支持向量机(SVM)根据性能变化特征将程序进行分类,并利用贝叶斯优化器调度LLC,以保证具有相同性能变化特征的应用共享LLC的同一部分.此外,Nishtala等人借助深度强化学习方法实现了对CPU核及其频率动态调整[33].Song等人利用机器学习近似Belady的MIN算法,该算法通过预测每个页面未来的调用时间来最小化缺页次数[39].然而,机器学习方法直接应用到缓存划分中仍然存在以下难点:(1)机器学习需要利用程序在平台运行数据训练专用的模型,不同的程序运行平台配置不同,诸如处理器的算力,存储设备的传输速率等配置因素均影响了一个程序的运行参数的变化.如果训练集与测试集的数据源自不同的平台,那么模型的有效性会受到影响.(2)模型设计难度较高,面对复杂的计算机体系结构的问题,无法依赖于单个机器学习模型解决,可能需要多个机器学习模型协同工作.(3)采样开销较大,由于程序的多样性以及可调节的配置较多,数据集的制作可能需要高昂的采样开销.

3 研究动机

本节重点阐述了我们采用先程序分类后缓存划分的动机.首先,我们在第3.1节介绍本文使用的实验平台、基准程序以及相应工作负载的性能指标;在第3.2节我们分析程序是否应该独占部分LLC的问题,并验证程序的先分类后划分的有效性;最后第3.3节讨论程序分类的依据问题.

3.1 缓存划分实验设置

实验平台的硬件配置.本文使用的实验平台包含源实验平台和目标实验平台,前者用于第3节的实验与分析,以及之后第5节的性能评估;后者主要用于第5节跨平台移植的LLC划分方法性能评估.两个实验平台的具体配置如表1所示.

表1　实验平台配置

特征	源实验平台	目标实验平台
CPU	Intel® Xeon® E5-2697 v4	Intel® Xeon® E5-2630 v4
核心数量	18(36个逻辑核)	10(20个逻辑核)
主频	2.3 GHz	2.2 GHz
一级Cache与二级Cache	64 KB×18与256 KB×18	64 KB×10与256 KB×10
LLC	45 MB(20路)	25 MB(20路)
内存	128 GB(32 GB×4)	128 GB(32 GB×4)
最大内存带宽	76.8 GB/s	68.3 GB/s
OS版本	CentOS 7.6.1810	CentOS 7.8.2003
Linux内核版本	4.18.0	3.10.0

我们的源实验平台和目标实验平台采用Intel至强(Xeon)系列处理器.该系列处理器能够提供动态分配缓存所需的工具链资源,即资源调配技术(Resource Director Technology,RDT)包括:缓存监控(Cache Monitoring Technology, CMT),内存带宽监控(Memory Bandwidth Monitoring, MBM),缓存分配(Cache Allocation Technology, CAT),内存带宽分配(Memory Bandwidth Allocation, MBA)等划分缓存所需的工具.

测试应用程序.我们选用SPECCPU 2006[40]和SPECCPU 2017[41]中的部分基准程序作为测试程序.二者均是处理器子系统测试工具,旨在确立平台性能评判标准.这些基准程序在缓存上的访存行为差异较大,因此我们选择在访存行为上具有代表性的部分基准程序作为测试以及分析用例.实验中选用的基准程序详见附录中(1).

性能指标.为了比较与分析程序工作集(或工作负载)的整体性能与公平性,我们采用了两个量化指标.对于工作集的整体性能,我们采用加权加速比(*Weighted Speedup*,*WS*),加权加速比可以衡量工作负载的整体吞吐量.对于公平性,我们采用工作负载中性能下降最大的程序表示(*Max Slowdown*,*MS*).具体的表达式如下:

$$WS=\sum\frac{IPC_{together}}{IPC_{alone}} \tag{1}$$

$$MS=MAX(\frac{IPC_{alone}}{IPC_{together}}) \tag{2}$$

其中,$IPC_{together}$是程序与其他程序并行运行时的IPC,IPC_{alone}是程序单独运行时的IPC.可见某程序与其他程序并行运行时,其IPC不受影响,则$IPC_{together}$等于IPC_{alone}.这时程序整体性能WS达到最大,等于所有并行运行程序数量.

3.2　LLC划分测试

商用处理器中LLC的缓存路数量有限,大多数情况下只有11路到20路.如图1(a)所示,如果存在5个并行程序,并且每个程序单独占用一部分LLC,

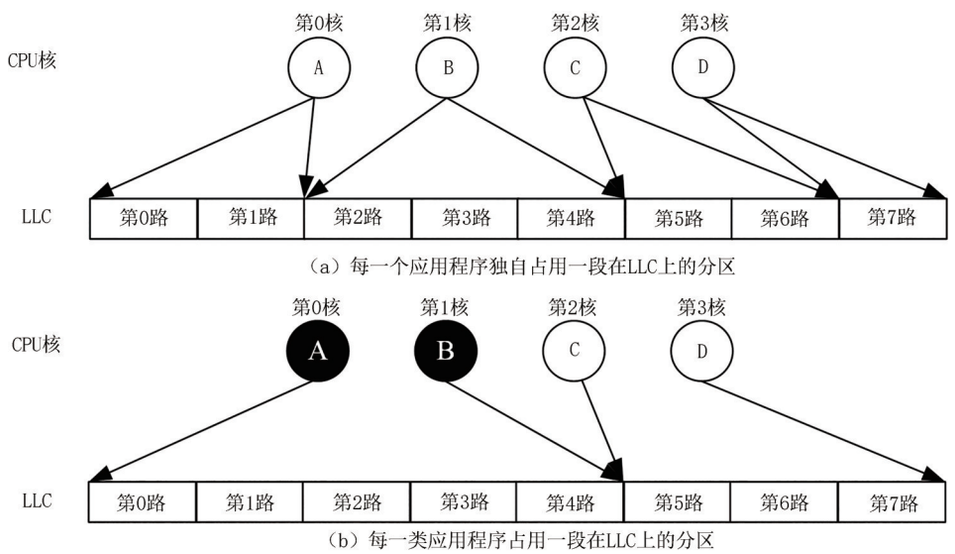


图1　缓存划分示意图

那么每个程序平均占用的LLC只有2到4路左右.而限制程序只占用少量的LLC会抵消Cache划分带来的优势,甚至会大大降低程序的性能.

实际上以缓存路划分缓存是粗粒度的,若每个程序单独占用一部分LLC,缓存将难以被合理划分.因此,如果将具有相似访存行为的程序分类后,在每个程序类别内部共享LLC,就可以在一定程度上消除这种以缓存路为最小划分单位的粗粒度划分方式带来的劣势,如图1(b)所示.KPart[21]根据缓存共享的兼容性对程序分类,每个类别内部以缓存路为单位共享一部分LLC.我们尝试根据应用程序性能变化特征的相似性,对其进行分类,并以程序类别为单位进行Cache划分,每个程序类别内部共享相同的LLC缓存路.

为了验证上述情况的真实性,我们从SPEC-CPU2007和SPECCPU2016中选取了19个基准程序,构建了4个工作集(见附录#B1-#B4),每个工作集包含了6个应用程序.在构建工作集时,我们会在每个工作集中放入1-2个前期测试得到的强干扰程序,剩余的程序则随机挑选.工作集内程序并发执行时,处理器在短时间内达到最大载荷.随后,我们采集了各个程序的运行参数,并根据LLC-IPC曲线,判断该应用程序最佳的LLC占用量,并以此作为这些运行参数(特征)的标签.最后,我们通过得到的数据训练一个机器学习模型,并将之与简单实现的启发式算法、不划分(不管理)方法一同运行在源实验平台,并对比工作集的整体性能.最终的实验结果如图2所示,我们可以发现,不管是启发式算法还是机器学习算法,工作集的整体性能甚至不如不管理方法的整体性能.因此,我们认为,如果每个

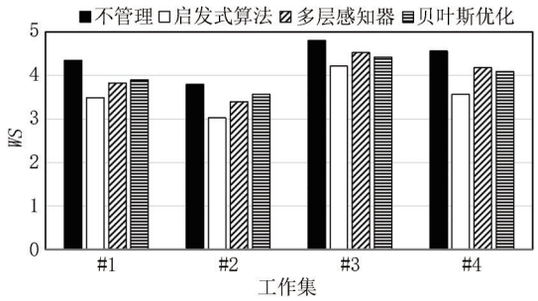


图2 程序独占部分LLC时的方法性能对比

工作集中的程序在运行时都独占部分LLC,将无法发挥缓存划分带来的优势.

既然程序单独占用部分LLC几乎无法带来性能提升,那么我们猜测,将程序分类并使同一类别的程序共享部分LLC或许可以提升工作集性能.据此猜想,我们依据是否是流式程序[42],对程序作了简单分类,确保同一类别的程序共享同一分区的LLC,并借助简单的启发式算法寻找划分方案.该类划分方法的简要示意图如图1(b)所示,位于第0核的A程序与位于第1核的B程序为一类,位于第2核的C程序与位于第3核的D程序为另一类.其中,A程序与B程序共享LLC中的第0路到第4路,C程序与D程序共享剩余的LLC.结果显示,该方法下的*WS*比不管理情况下的*WS*平均高出2.7%.

为了进一步提高LLC共享的效率,我们尝试对程序分类展开研究.我们选择以*Xalan*、*H264ref*以及*Soplex*作为背景程序,将SPECCPU[40-41]中的基准测试程序与背景程序并行运行30秒,并计算加权加速比(*WS*),实验结果如图3所示.通过大量实验,我们在分析运行时程序参数以及性能指标之间的关系基础上,尝试将工作集中的程序分为三类:强干扰程序、缓存敏感程序以及缓存不敏感程序.

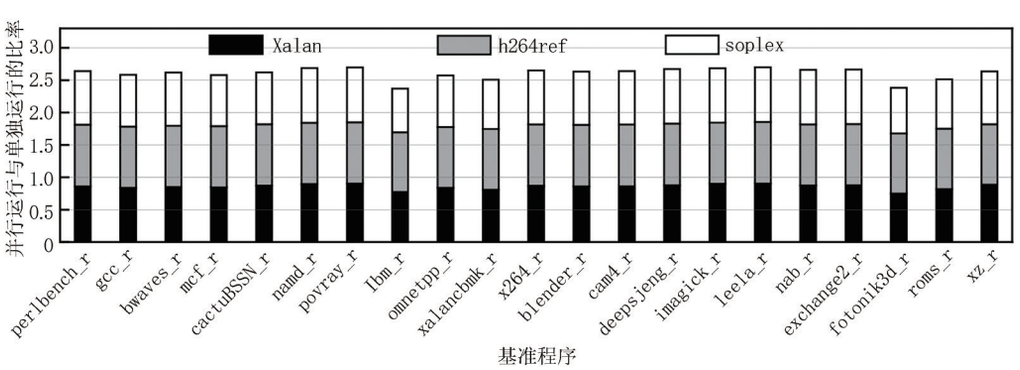


图3 与各程序并行运行的背景程序性能

强干扰程序被定义为会造成其他程序性能大幅下降的程序,如果对该类程序不加以缓存分配控制,而是按需分配缓存,会导致其他程序在短时间内大量页缺失,引起整体性能的下降.从图3中可以发

现，*lbm_r*，*fotonik3d_r*等程序明显对背景程序有较强的干扰．其中，对背景程序*Soplex*的干扰，导致其性能下降了25%．这类具有强干扰的程序在缓存划分中应当被隔离，避免造成其他程序性能的严重下降．因此强干扰程序的缓存被首先划分．

除了强干扰程序以外，根据缓存路数对程序的影响程度，非强干扰程序可进一步分类为缓存敏感程序和缓存不敏感程序．我们将基准程序单独运行在源实验平台上，并测试从占用1路到占用所有20路LLC时，IPC与Cache缺失之间的关联性表现．我们选取了部分具有代表性的程序作为示例，如图4所示．可以看到，*perlbench_r*和*Gee_r*随着可以占用的LLC不断增加，IPC上升明显且Cache缺失大幅下降．这说明，占用的LLC越大，该程序的吞吐量越高．因此可以得知，该程序对LLC较为敏感，是一个典型的缓存敏感程序．作为对比，无论占用的LLC怎么变化，*nab_r*和*Leela_r*的IPC以及Cache缺失均没有显著变化．这说明，该程序对LLC资源调整并不敏感，在满足性能要求的前提下，可以尝试为此类缓存不敏感程序尽量减少LLC分配．

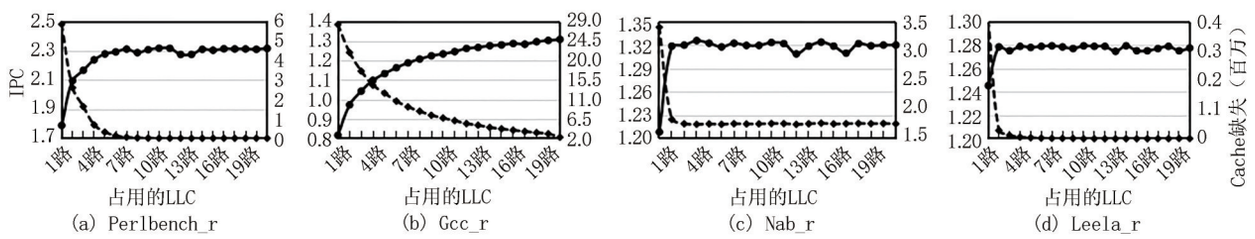


图4　程序对Cache的敏感程度

此外，相较于强干扰程序，缓存敏感程序并不会造成其他程序性能的严重下降，我们可以从图4中看到*perlbench_r*的IPC与占用的LLC成正比，而在图3中，这类程序并没有对背景程序造成干扰，所以无需对缓存敏感程序进行隔离．

3.3　运行参数相关性分析

我们尝试借助Pearson相关系数，分析IPC，缓存缺失，内存带宽等程序的运行参数与程序类别之间的相关性．Pearson相关系数（Pearson Correlation Coefficient）常常用于衡量定距变量间的线性关系，可表示为[43]

$$\rho_{X,Y}=\frac{cov(X,Y)}{\sigma_X\sigma_Y} \tag{3}$$

其中，*cov*表示协方差．σ表示针对连续变量的标准差．利用Pearson系数分析结果如图5所示．相关系数为正数反应参数之间为正相关，为负数反映参数之间为负相关，而等于0表示不相关．相关系数的绝对值越大，参数与参数之间的关联越紧密．

从图5中可知，按照三种类别分类程序时，程序的类别与程序的运行参数均有一定的相关性．但是CPU利用率和程序所在核的频率与程序类别之间的关系比较弱．这是因为基准测试程序都是单线程程序，我们限定每个程序仅占用一个核，程序的CPU利用率基本维持在100%左右以及程序所在核的频率变化也不明显．此外，IPC与程序类别的关联也不强，某一种LLC划分状态下的IPC并不能直接与程序类别产生联系．但如果知道其他LLC划分状态下的IPC，则能够判断程序的类别，如第3.2节中的缓存敏感与缓存不敏感程序判定．但这需要同步测试其他程序在不同分配方案下的IPC表现，将会造成较高的采样成本，使得无法做出快速的方案决策．剩余的运行参数均与程序类别具有一定的相关性．

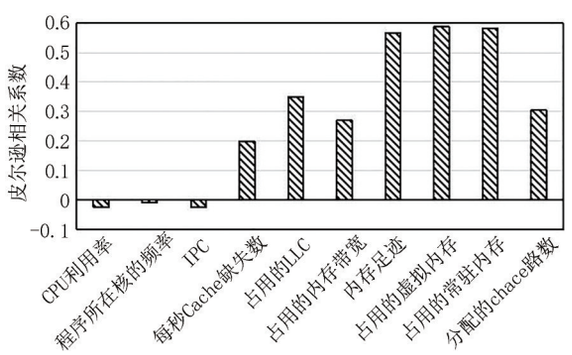


图5　程序类别与运行参数的Pearson相关系数

这说明，应该尽可能参考多的运行参数分析程序的类别．而引入更多运行参数，这也意味着程序的模型分类将变得越加复杂．机器学习凭借强大的学习能力，可以构建各个参数与程序类别等之间的关联．为此，我们尝试先通过机器学习方法对多个运行程序进行分类；再基于分类结果，预测不同LLC划分方案下的程序整体性能，最终找到最佳的LLC划分方案．

4 MLPart的实现

在本节中，我们设计并实现了一种基于机器学习的跨平台缓存划分方法MLPart，该方法主要由程序分类模型，缓存划分模型以及模型移植模块构成，我们分别在第4.1-4.3节中介绍其原理及实现.

4.1 基于决策树的程序分类模型

关于决策树. 当前被广泛应用于分类问题的机器学习算法主要有决策树(Decision Tree，DT)、支持向量机(Support Vector Machine，SVM)、朴素贝叶斯分类器(Naive Bayesian Classification)、K近邻分类方法(K-Nearest Neighbor，KNN)、神经网络(Neural Network，NN)等等. 其中，决策树是一种特殊的分类器[44]. 该分类器基于无次序无规则有类别标记的数据集，推导可以借助决策树表达的分类规则. 由于决策树基于决策规则，故能够对未知的数据分类. 决策树的优点在于计算复杂度不高，输出结果易于理解，对中间值的缺失不敏感，以及可以处理不相关的特征数据；其容易过拟合的缺点，可以通过剪枝的方法克服. 因此，我们将决策树应用于程序分类问题.

在第3.2节中，我们将程序分为强干扰程序，缓存敏感程序以及缓存不敏感程序. 为程序划分类别实际上成了一个多分类问题. 为了提高分类的精度，并且与后面的迁移学习技术相兼容，我们将多分类问题转化为两个二分类问题，即设计两个决策树DT-A与DT-B，借此解决这个多分类问题. 设计的基本思路是，借助DT-A，将程序划分为强干扰程序与非强干扰程序，之后借助DT-B，将非强干扰程序划分为缓存敏感程序与缓存不敏感程序.

构建决策树的基本思路是：(1)将所有数据看作一个节点；(2)遍历每个特征的每一种分割方式，找到最好的分割点；(3)分割成两个节点；(4) 对两个节点分别继续执行2-3步，直到每个节点的纯度达到要求为止. 在MLPart中，纯度利用基尼不纯度(Giniimpurity)衡量，其公式为

$$Giniimpurity = 1 - \sum_{i=1}^{n} P(i)^2 \tag{4}$$

其中，n是应用程序类别的数量，$P(i)$为该应用程序类别所占的比例.

我们采用预剪枝技术为了防止决策树的过拟合以及模型过于庞大的问题. 预剪枝技术的基本思路是限制决策树的过度分枝，即在决策树构建过程中明确控制决策树的大小，以此简化决策树的结构. 我们采用一种较为灵活的方式控制决策树的大小，即在训练过程中不断调整决策树的最大深度，通过权衡测试集的准确率与决策树大小，确定最大深度.

数据的采集与处理. 首先，我们基于第3.2节中的实验，对标准测试集中的所有程序分析、归类并定义如下：如果一个程序与*Soplex*、*Xalan*和*H264ref*并行运行时，三个程序的平均性能下降之和超过了20%，那么该程序是强干扰程序；如果一个程序单独运行时，其占用的LLC从1路到所有N路，其平均性能上升能够超过10%，那么该程序是缓存敏感程序；除此以外的程序，可以被归类为缓存不敏感程序.

我们从图3中可以看出，*lbm_r*，*fotonik3d_r*等程序明显对背景程序有较强的干扰，导致*Soplex*的性能下降了25%，因此需要对其进行缓存划分的隔离. 我们从图4可以看到，*perlbench_r*和*Gee_r*随着可以占用的LLC不断增加，IPC分别上升了29%和62.5%，而*nab_r*和*Leela_r*的变化幅度均未到达这一数值，将其归类为缓存敏感程序. *nab_r*和*Leela_r*的变化幅度均未超过10%，可以归类为缓存不敏感程序.

其次，我们采集了大量供决策树训练与测试的数据. 样本数据的采集是从已经完成分类的程序中，挑取n个程序(n=4，5，6)，将其并行运行在程序运行平台上. 使用高速缓存监控技术(Cache Monitor Technology，CMT)，内存带宽监控技术(Memory Bandwidth Monitor，MBM)以及Linux原生工具，采集每个程序的相关特征. 之后，根据各类的定义完成分类，并打上标签. 最后，我们依据每个特征的最大值与最小值，将特征值进行了归一化处理，保持其值在0与1之间.

4.2 基于递归神经网络的缓存划分模型

由于缓存不敏感程序不需要过多的LLC资源，我们分配给该类别程序1-2路LLC. 这样分配的理由有两个，一是因为缓存不敏感程序被划分的LLC超过或等于2路时性能没有明显变化，如图4(c)和4(d)所示. 二是因为缓存不敏感程序基本不受并行运行的程序影响，所以多个缓存不敏感程序共享2路LLC性能基本不受影响.

而对于强干扰程序与缓存敏感程序各自需要占用的缓存路的数量，实际上是一个较为复杂的决策问题，我们尝试利用递归神经网络解决该问题，即

设计一个能够预测各种划分情况下的整体性能的模型. 这里的整体性能被定义为加权加速比(Weighted Speedup，*WS*).

通过3.2节的实验，我们发现缓存敏感的程序在不断分配LLC的过程中，程序的性能是连续变化的，基本不会发生跳变，如图4(a)中展示的*perlbench_r*和图4(b)中展示的*Gee_r*. 这种程序性能连续变化的特性，让我们可以合理假设：状态n是强干扰程序共享n个缓存路(n在1到N之间)，并且缓存敏感程序共享剩余的N-n个缓存路，使用LLC的运行程序的整体性能可表示为$WS(n)$；状态$n+1$是强干扰程序共享$n+1$路LLC，并且缓存敏感程序共享剩余的N-n-1路LLC，使用LLC的运行程序的整体性能可表示为$WS(n+1)$. 我们可以认为，状态n与状态$n+1$这两种LLC划分方案之间的整体性能$WS(n)$和$WS(n+1)$也具有连续性，即两个状态之间有较强的关联. 我们尝试借助连续缓存划分方案之间整体性能*WS*的连续性，预测各种LLC划分情况下的*WS*.

受此启发，我们利用递归神经网络(Recurrent Neural Network，RNN)对工作集的整体性能预测. RNN是一类以序列数据为输入，在序列的演进方向上递归(Recursion)，所有节点(循环单元)按链式连接的递归神经网络. 我们可以输入前n种缓存划分方案，借助RNN预测后N-n种划分方案的*WS*. 因为后N-n-1种划分方案的*WS*也是一组前后相互关联的序列. 在MLPart中，我们采用RNN的变种模型—序列到序列模型(Sequence to Sequence，Seq2Seq)，预测各种划分方案的*WS*. Seq2Seq是一个编解码结构(Encoder-Decoder)模型，输入与输出均是一个序列[45]. 编码(Encode)部分将一个可变维度的输入序列变为固定维度的向量，解码(Decoder)部分将这个固定维度的向量解码成可变维度的输出序列.

Seq2Seq模型主要应用于自然语言处理领域，所以一般情况下输入序列与输出序列的维度并不加以限制. 不仅如此，输入序列与输出序列中每一元素的维度也不加以限制. 但在本问题中，我们的输入与输出都是不变化的. 因此，我们固定输入序列的维度为4，输出序列的维度为13. 本文中的Seq2Seq模型结构如图6所示.

在编码部分，v_i为模型的输入，每一个v_i代表强干扰程序共享i路LLC以及敏感程序占据剩余N-i路LLC时的特征向量；h_i是编码部分的隐藏层，每一个h_i均是长短期记忆(Long-Short Term Memory，LSTM)神经网络的记忆块(Memory Block)[46]；最后，编码部分输出一个向量c，作为解码部分的输入. 在解码部分，h_j'也是LSTM的记忆块；$WS(j)$为预测结果. 实际上由于缓存不敏感程序已经占用了2路LLC，我们分配给缓存敏感程序和强干扰程序的实际LLC路数为N-2. 因此$WS(m)$代表强干扰程序共享j路LLC以及缓存敏感程序占据剩余N-2-j路LLC时工作集的*WS*.

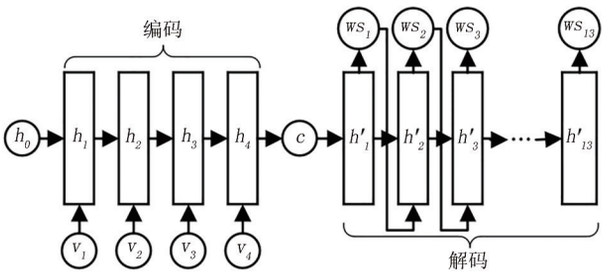


图6　Seq2Seq模型结构

数据采集与处理. 我们随机选取m个强干扰程序，n个缓存敏感程序以及p个缓存不敏感程序. 缓存不敏感程序始终固定分配共享2路LLC，强干扰程序依次分配1至17路LLC，缓存敏感程序共享剩余LLC. 在前4种缓存分配方案，即强干扰程序依次分配1至4路LLC中，我们采集了各个程序的运行参数，剩余的缓存划分方案中我们只采集各个程序的IPC.

我们根据程序的类别对前四种缓存划分方案中的运行参数进行处理. 对于同一类别程序的资源占用参数，例如各个程序1秒内的LLC占用情况以及内存占用情况等等，我们将其分别相加；对于同一类别程序的其他参数，例如各个程序1秒内缓存缺失数以及所占用的核的频率，我们将其分别求平均数. 因此，每一个缓存划分方案下的输入是一个向量，包括强干扰程序、缓存敏感程序及缓存不敏感程序的运行参数. 使用剩余LLC划分方案采集到的各个程序IPC计算工作集的*WS*. 最后，所有的数据做归一化处理.

执行流程. 在MLPart中，执行缓存划分为两个阶段，在第一阶段中，MLPart需要对程序分类；在第二阶段中，MLPart针对每个类别的程序确定最优的缓存划分策略.

在初始状态中，所有程序共享整个LLC. 首先，我们在该状态下采集各个程序的运行参数，例如各个程序的CPU的利用率，占用的内存，缓存缺失等参数. 在采样过程中，1秒采集一次运行参数，持续

3秒，并基于3次获得的数据取平均值．其次，我们将各个程序的运行参数输入DT-A中．根据DT-A的预测结果，我们甄别出工作集中的强干扰程序与非强干扰程序．最后，我们将其中非强干扰程序的运行参数再次输入DT-B中，从而区分缓存敏感程序与缓存不敏感程序．

由于Seq2Seq是RNN的变种，它可以将上文包含的信息保存在隐藏状态中，提高了算法对于上下文的理解能力，而连续分配缓存路与程序工作集性能必然是一条连续的曲线，存在前后关系．因此在缓存划分阶段，MLPart首先为缓存不敏感程序分配固定的2路LLC，为强干扰程序依次分配1至4路LLC，每次步长为1路，并让缓存敏感程序共享剩下的空余LLC．每次分配后，MLPart采集各个程序1秒的运行参数，并对这些运行参数做处理．此外，我们根据采集到的各个程序的IPC，计算工作集的*WS*．随后，本方法将处理后的参数输入Seq2Seq模型，预测强干扰程序占据5至17路以及缓存敏感程序占据剩余的空余LLC时的*WS*．

若采用更少的采样结果，如只采样分配1路至2路或1路至3路给强干扰程序，并预测其性能，将会降低MLPart对性能的提升，如图7所示．我们分别测试了使用1路至2路和1路至3路在A-D四个数据集上MLPart的综合性能提升，相较使用1路至4路的方案性能提升分别最大下降了11.38%和36.72%．若采用1路-5路甚至更多的路数作为Seq2Seq的输入，则由于需要预测的分配的缓存路至多只有13路，经过大量测试，我们发现其实与使用1路至4路在最终性能预测上结果相差不大，且还会增加采样开销，因此并不合适．因此，在20路缓存的处理器上，我们选择分配1路至4路给强干扰程序作为Seq2Seq的输入，搜寻其中使*WS*最大的划分方案，并据此展开LLC划分．具体MLpart的执行流程如图8所示．

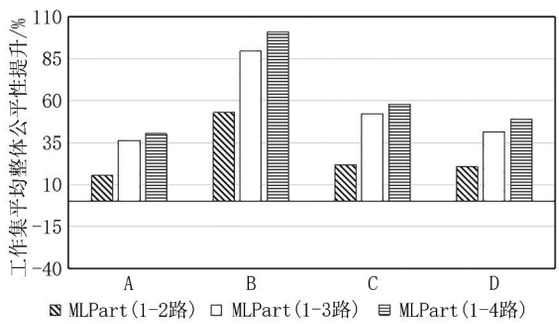


图7 不同采样路数对工作集簇的性能影响

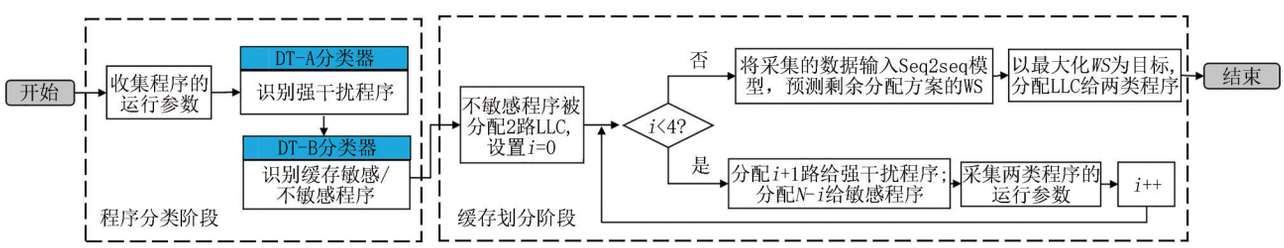


图8 MLPart执行流程图

4.3 模型的跨平台移植模块

在源实验平台上实现的程序分类模型和缓存划分模型移植到目标实验平台后，效果并不理想．这是因为如果计算平台的配置存在差异，即使是相同程序运行在不同配置的平台中，运行参数差异也较大，导致程序分类模型无法做出正确预测．

而重新生成模型，需要再一次在新的计算平台中采集大量相应的数据，其开销显然是令人难以接受的．为此，我们在MLPart中引入迁移学习，借助少量在新平台中采集的数据实现高精度的模型．迁移学习的目标是，借助源域(Source Domain)的知识学习目标域(Target Domain)的知识．在MLPart的模型迁移过程中，在源实验平台中采集的数据集为源域，在目标实验平台中采集的数据集为目标域．

我们希望已经采集大量数据的源域能对采集少量数据的目标域提供帮助．Dai等人提出的TrAdaBoost[34]，是迁移学习中的经典方法．该方法将AdaBoost的思想应用于迁移学习中，AdaBoost可以提高有利于目标分类任务的样本权重，降低不利于目标分类任务的样本权重，从而提高泛化能力．具体实现上，TrAdaBoost在训练过程中调整目标域和源域的样本权重，实现模型的迁移．如果目标域训练集的样本被分类错误，则说明决策树无法较好地对该样本进行分类，因此需加大该样本的权重，使经过训练的决策树能更好地解决目标域上的分类问题．当源域中的样本被分类错误时，则认为来自源域的样本难以对模型训练起到作用，因此会降低该样本所占的权重．

TrAdaBoost的初衷是增大在目标域中正向数据的权重，抑制负向数据的权重. 然而在实际应用中，源域与目标域数据分布可能存在着较大差异，因此，源域中一些数据的分布与目标域数据分布差异较大的数据也会被认为是正确数据，这些数据的存在会进一步降低模型迁移以后的分类准确率.

如图9所示，标浅数据和标深数据分别来自源域和目标域，不同的符号代表不同的类别. 图中被圈出来的数据属于源域且被正确的分类，但其分布情况与目标域差异较大，这类数据可以理解为是源域中的特殊样本. 实线为使用源域分类模型迁移后产生的分类模型边界，虚线为使用目标域数据训练模型产生的理想分类边界. 很明显，正是由于特殊样本的存在使得拟合曲线向左偏移，降低了分类准确率. 它们对目标域中的分类器起到的作用不大甚至会误导分类结果. 为了解决该问题，通常的做法是利用微调技术进一步调整迁移后的数据模型.

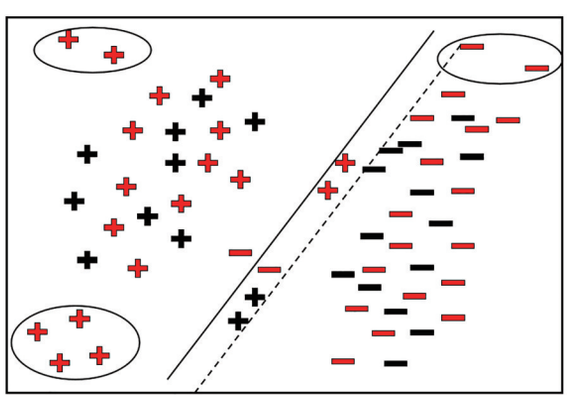


图9　差异性较大的源域与目标域数据分布

然而，由于程序分类所需要的数据分布差异性较为明显，我们尝试直接从源域数据入手，根据目标域的数据分布，调整源域的数据在迁移过程中对模型的影响权重，从而提高模型迁移后的分类精度. 为此，我们提出了一种基于权重调整的改进TrAdaBoost算法. 在改进算法中，我们首先确定目标域中数据分布的边界. 通过k-means算法找到目标域中不同类别数据的聚类中心，并遍历计算同类别数据与聚类中心的距离，选取其中的最大值D_{Max}作为分布边界. 采用欧氏距离(Euclidean Distance)计算分布中两个数据之间的距离. 在源域数据被分类后，若其与该类别聚类中心的散度$D(P,Q)$大于D_{Max}，则认为该数据与目标域分布差异较大，降低其权重. 其中P为聚类中心点，Q_i表示第i组数据. 可利用如下公式计算D_{Max}：

$$D_{Max}=\mathrm{Max}\left(\sqrt{\sum_{n=1}^{N}(x_n-T_n^i)^2}\right) \tag{5}$$

对于N维数据，x_n和T_n^i为聚类中心P和目标域数据集T中第i组数据的第n维数据，最终源域中的数据权重更新公式如下：

$$W_i^{t+1}=\begin{cases} W_i^t\beta, & D_{Max}<D(P,Q_i) \\ W_i^t\beta^{|h_t(x_i)-c(x_i)|}, & D_{Max}>D(P,Q_i) \end{cases} \tag{6}$$

$$\beta=1/(1+\sqrt{2\ln I/T}) \tag{7}$$

其中，W_i^t为第i组数据在第t次迭代时的权重值. $h_t(x_i)$和$c(x_i)$分别为第i组数据的分类结果和真实值. β为TrAdaBoost中的权重系数，I和T定义为目标域的总数据量以及总迭代次数.

由上式可知，若源域数据分布超过目标域内的边界，则权重值减小，反之，权重调整策略与TrAdaBoost相同，分类错误则权重减小. 经过实验测得，1000组数据计算边界的时间为39.003 ms，占用开销较小，具有较高的可行性. 具体算法流程如下：

算法1　基于权重调整的改进TrAdaBoost算法

输出：输出最终的分类器.

输入：源域和目标域数据集D_S和D_T，合并的训练数据集D=$D_S\cup D_T$，基分类算法Learner，总迭代次设置为T.

初始化：初始化权重$w^1=(w_1^1,\cdots,w_{n+m}^1)$，设置$\beta=1/(1+\sqrt{2\ln I/T})$，计算目标域中的数据边界$D_{Max}$.

Loop t=1:T

1. 设置$P^t=\dfrac{w^t}{\sum_{i=1}^{n+m}w_i^t}$;

2. 调用Learner，根据合并后的训练数据D以及D上的权重分布P^t得到分类器h_t;

3. 计算h_t在D_T上的错误率：$\varepsilon_{tar}=\sum_{i=n+1}^{n+m}\dfrac{w_i^t|h_t(x_i)-y_i|}{\sum_{i=n+1}^{n+m}w_i^t}$

IF $\varepsilon_{tar}>0.5$ THEN

END LOOP;

4. 设置新的权重

$$W_i^{t+1}=\begin{cases} W_i^t\beta, & D_{Max}<D(P,Q_i),i=1,\cdots,m \\ W_i^t\beta^{|h_t(x_i)-c(x_i)|}, & D_{Max}>D(P,Q_i),i=1,\cdots,m \\ W_i^t\beta^{-|h_t(x_i)-c(x_i)|}, & i=m+1,\cdots,m+n \end{cases}$$

而对于负责预测整体性能的Seq2Seq模型，我们采用微调技术[35]实现模型的跨平台优化. 该技术的优势在于：(1)不需要针对当前的新任务重新训练网络，大大节省了计算开销；(2)预训练的模型往往是基于大量数据训练的，这使模型更具鲁棒性，并提升了模型的泛化性能；(3)微调技术实现简单，可以

让我们更关注于模型本身.

微调技术的本质是冻结预训练模型中靠近输入的隐藏层,训练靠近输出的隐藏层.基于此,我们冻结Seq2Seq模型的编码部分,并借助少量采集的目标域数据,对Seq2Seq模型的编码部分重新训练,调整其权重.

5 实验评估

在本节中,我们对MLPart评估性能.在第5.1节中,我们展示了MLPart中程序分类模型的准确率和误差等指标,以及利用迁移学习优化后的分类模型性能.在第5.2节中,我们比较了不同缓存划分方法的性能与划分开销.最后,我们在第5.3节展示了跨平台移植后的方法性能.

5.1 程序分类模型性能

我们将在源实验平台采集的数据以7:3的比例随机分成训练集与测试集.如表2所示,我们总结了各个模型在测试集中的平均性能.两个决策树模型的准确率都较高,相对较低的DT-A准确率也接近97%,DT-B的准确率更是接近99.7%.此外,Seq2Seq模型的均方误差为0.386.

表2 MLPart的划分模型性能

指标	DT-A	DT-B	Seq2Seq
准确率	96.8568%	99.6850%	/
精确率	97.0415%	99.6851%	/
召回率	95.9204%	99.6849%	/
均方差	/	/	0.4946
平均绝对值误差	/	/	0.52434
平均时间开销	0.0010秒	0.0003秒	0.89秒
训练时间	0.993秒	0.972秒	55.2秒

在训练过程中,针对DT-A模型,我们总计选取99954组数据作为训练集,其中强干扰程序35837组数据,非强干扰程序64117组.对于DT-B模型,我们总计选取99932组训练集,其中缓存敏感程序44276组数据,缓存不敏感程序55656组数据.经过验证,DT-A与DT-B的训练时间均不超过1秒,Seq2Seq模型训练400个apoch的时间为55.2秒,平均每个epoch的训练时间不超过140毫秒.而决策树模型的时间开销基本是毫秒级的;调用Seq2Seq模型完成一次预测的时间开销在1秒以内.因此对于需要长时间运行的某个工作集来说,其包含的程序的执行时间一般远远超过1秒,两个模型的执行开销要完全能够被MLPart所带来的性能提升收益覆盖.

为了验证TrAdaboost的效果,我们在目标实验平台采集与源实验平台数量一致的数据,以1:19的比例随机分成训练集与测试集.我们利用TrAdaBoost技术对源实验平台的数据集中的数据进行权重调整并训练决策树,随后基于新的决策树在目标实验平台的测试集中做了准确率评估,结果如图10所示.由于源实验平台数据与目标实验平台数据存在一定的差异,导致仅使用源实验平台数据训练的模型无法有效地对目标实验平台中的数据进行分类.而目标实验平台划分的训练集数据量较小,因此仅使用目标实验平台训练集训练的DT-A模型准确率较低,在75%左右.在TrAdaBoost的帮助下,DT-A的准确率成功从75%左右提升到了97%;在DT-B中,TrAdaBoost也成功提升了3%左右的准确率.

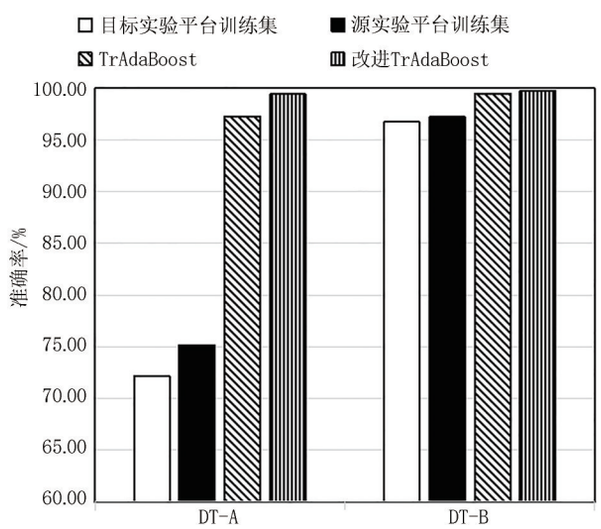


图10 迁移学习后的决策树性能

改进后的TrAdaBoost算法通过计算目标域内的数据分布边界,并比较源域数据与目标域边界之间的差异,剔除了源域内的冗余数据,使得模型能够更好地在目标域内拟合,进一步提升了模型在目标域内的训练效果.改进后的TrAdaBoost则在原有TrAdaBoost的基础上将DT-A的准确率提升了2.49%,达到99.49%,将DT-B的准确率提升至99.72%.准确率的提升使得模型能够更好的将目标域内的程序进行合理的分类,从而提升整体性能及公平性.

5.2 缓存划分方法性能评估

我们选取了以下方法与MLPart进行对比:

Baseline:该方法不对程序占用的LLC进行控

制，依赖于操作系统的资源调度器.

简单Heuristic：该方法依据并发执行的程序性能，对缓存资源进行实时划分. 该方法采用程序的加速比作为该程序是否缺乏缓存资源的度量标准. 每次划分过程中，该方法将加速比最高程序所占用的1路LLC划分给加速比最低的程序. Heuristic每1秒执行一次资源划分，直至当前程序组合的*WS*到达组合中程序数目的95%为止.

KPart[21]：该方法首先采集每个程序的缓存缺失随缓存划分变化的曲线，根据Whirlpool[47]的距离函数对程序进行聚类. 然后，将启发式算法与Lookahead算法[4]结合，为每类程序提供缓存划分方案，选取预计性能最好的划分方案作为最终的缓存划分方案.

CLITE[32]：该方法采样随机初始划分策略，并采集相应的程序性能参数. 借助贝叶斯优化技术，生成可能进一步提高程序性能的新划分方案. 经过多次采样，取能使性能达到最高的划分策略作为最终缓存划分方案.

C&A[38]：该方法使用SVM根据每个应用程序的性能变化特征将应用程序进行分类，并利用贝叶斯优化技术，把缓存分配和系统吞吐量之间的关系视为一个黑盒函数，将黑盒函数的最大化输出作为最优分配方案.

我们借助以下在源实验平台中的实验展示MLPart的性能：首先，我们在SPECCPU 2006[40]和SPECCPU 2017[41]中分别选取了5-8个程序，构成多个工作集. 各个工作集的构成如附录3所示. 我们使用A、B、C、D分别表示包含5个、6个、7个、8个程序的工作集簇. 测试过程中，我们针对A-D簇中的每个工作集进行单独测试，且工作集在测试过程中，需要同时启动所包含的程序，形成并发执行(concurrent execution)，但如何执行并行策略则由处理器决定，测试过程中不加以干预. 我们分别采用上述不同方法对单一工作集的缓存进行管理，并记录每种情况下的反映工作集整体性能的*WS*以及整体公平性的*MS*.

图11(a)和图11(b)分别展示了采用不同缓存管理方法时，与Baseline进行比较，对工作集整体性能的提升以及工作集整体公平性的提升. 与Baseline相比，将MLPart应用于A-D四个工作集簇，对于工作集的整体性能能够平均提高4.83%、14.13%、5.94%和8.16%. 将简单Heuristic方法应用之后，除了B工作集簇性能较Baseline提升2.51%外，在A、C和D三个工作集簇中，反而会使工作集的性能分别平均下降7.43%、5.70%和4.08%. KPart对四个工作集簇的整体性能平均提升为−1.10%、2.99%、1.94%和4.37%. 而CLITE对四个工作集簇的整体性能平均提升分别达到了−4.10%、4.68%、−8.03%、−0.23%.

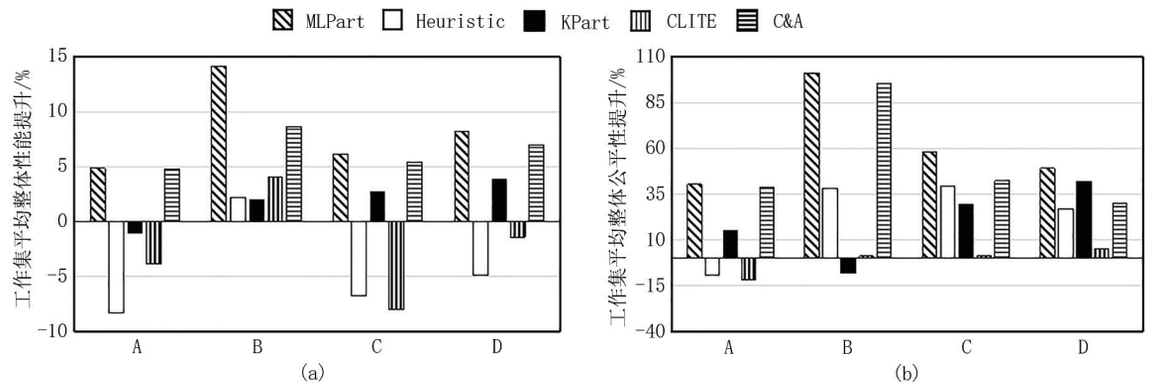


图11　针对不同程序数量的工作集簇测试：(a)平均整体性能提升；(b)平均整体公平性提升

在公平性方面，相较于Baseline，MLPart在包含5-8个程序的四个工作集簇中应用后，其整体公平性提升分别为：37.64%、98.17%、65.73%和47.73%；简单Heuristic的平均提升分别为−5.56%、39.46%、44.23%和27.36%；KPart平均提升分别为19.92%、4.95%、47.69%和48.85%；CLITE平均提升为−3.08%、4.44%、15.78%和12.54%. C&A平均提升分别为4.82%、8.63%、5.43%和6.94%. 实验表明，MLPart能够有效提升不同程序数目的工作集的整体性能和整体公平性，也说明了其具有良好的扩展性和通用性.

为了详细对比不同方法对具体工作集的性能提升，我们针对包含6个程序的B簇(#B1-#B7)展开测试，测试结果如图12(a)所示. 可以看到，以Baseline

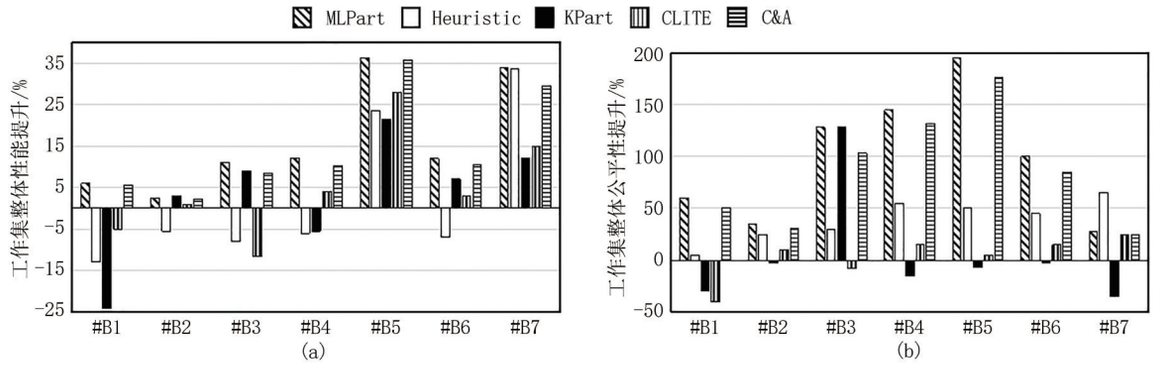


图12 针对包含6个程序组合的B工作集簇测试:(a)整体性能提升;(b)整体公平性提升

为基准,对比简单Heuristic,KPart,CLITE和C&A四种缓存管理方法,MLPart在对B簇中所有的工作集的整体性能均有提升.

相较于Baseline,简单Heuristic方法对性能的提升平均仅上升2.51%,其中在#B1-#B4以及#B7中甚至不如Baseline算法,也进一步验证了启发式算法具有一定的局限性.通过实验,我们发现简单Heuristic方法在划分缓存的过程中,需要不断尝试资源划分方案,很多时候无法迅速找到全局最优划分方案,且无法根据程序对于缓存资源的性能敏感程度进行划分决策,造成了性能下降.

KPart通过对缓存缺失曲线的采样,能够精准判断不同程序共享缓存所造成的冲突严重程度,并据此选择最优聚类程序组合.然而,对缓存缺失曲线详尽的采样却造成了大量的程序分配和资源调度开销.即KPart需要在线采集每个程序在不同缓存划分情况下的缓存缺失大小,难以在较短时间内产生高性能的调度决策,采样过程同样造成了程序组合的性能波动.并且KPart采用聚类方式完成程序分类,类别的数目则是基于程序性能确定.相似的程序性能可能使得类别的数量出现估计偏差,对聚类效果将产生较为明显的负面影响.KPart在工作集#B2-#B3以及#B5-#B7均有效提升了工作集的整体性能.但对于工作集#B1和工作集#B4,KPart的类别数目的估计并非最优,导致程序并未被分配到正确的类中,进一步影响了缓存的划分.由此可见,KPart虽然在一些情况下能起到较好的作用,但是如果对程序划分出现了问题,也会引起工作集的性能大幅下降.

C&A同样采用了先分类后划分的做法.其对工作集整体性能提升上优于Baseline,Heuristic,KPart和CLITE.然而在缓存划分方面,C&A使用贝叶斯优化技术划分缓存,即将缓存划分和系统吞吐量之间的关系视为一个黑盒函数,而在收敛的过程中由于数据分布的不同,对部分程序的分配方案并没有达到最优,因此对整体性能的提升弱于MLPart.

相应地,MLPart能够提供更可靠的缓存划分方案,并且Seq2Seq模型具有更好的鲁棒性,能够实现更公平的缓存划分.MLPart隔离了强干扰程序,避免了此类程序对其他程序造成的干扰,缩减了缓存不敏感程序所占用的缓存资源,提供了更多可用缓存资源给性能敏感程序.例如,在工作集#B5中,由于提供了更合理的缓存资源划分,本方法的性能相较于未划分情况显著提升了36.34%.

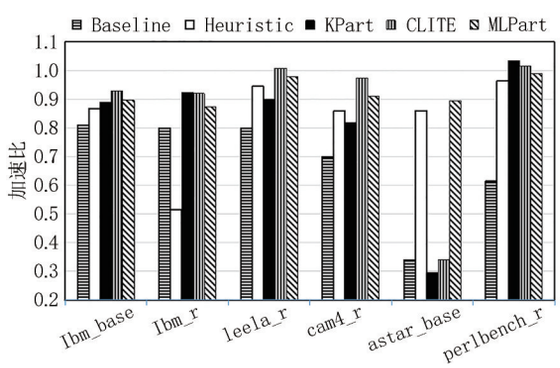


图13 工作集#B5的各程序性能加速比

图13进一步展示了工作集#B5中每个程序的加速比.可以看到,MLPart主要受益于对程序的精准分类和基于程序类别的缓存划分,将程序*astar*作为缓存敏感程序,并通过缓存划分使其能够得到更多的LLC,性能得到大幅提升.而其他程序并未受到强干扰程序的影响,且被划分的LLC也满足需求,因此未造成工作集整体性能的显著下降.

在缓存划分的开销方面,我们基于包含6个程序的B簇展开评估.由于MLPart方法仅需执行4次缓存划分调整(采集程序运行参数)即可通过Seq2Seq模型预测最优缓存划分方案,划分开销较

小. Heuristic方法需要达到目标WS才会停止缓存划分，因此划分开销存在大幅波动，需要平均执行90.34次缓存资源调整才能获得缓存划分方案. C&A的平均调度步骤为6.3次；KPart需不断调整资源划分，采集每个程序的IPC曲线和缓存缺失曲线，对于包含K个程序的工作集，共需执行$7\times K$次缓存划分调整；CLITE的平均缓存划分调整次数也达到了12.25次. 由于不同方法在缓存划分调整过程中的处理时间存在差异，仍需要以最终完成缓存划分所消耗的时间衡量最终的划分开销.

缓存划分的时间开销如图14所示. 我们应用五种方法，测试了所有数据集的缓存划分完成时间开销，并加以记录，形成了图14所示的水滴状柱体图. 其中五个柱体中心的横线代表了平均时间开销. 可以看到，简单Heuristic花费的时间最长且拟合时间波动较大，平均需要90.3秒完成缓存划分，这说明其单次采样和计算的开销尽管较低，但找到合适的缓存划分方案的次数多，也会导致缓存开销的大幅增加. KPart与CLITE的平均拟合时间分别为27.10秒与24.41秒. MLPart的平均拟合时间是17.99秒，相较于前两者方法分别节省了26.2%与33.5%的时间开销. 而C&A方法使用贝叶斯模型实现缓存划分，在划分时间上快于Seq2Seq模型. 但Seq2Seq模型能够更好地拟合数据，有利于预测出更加合理的缓存划分方案.

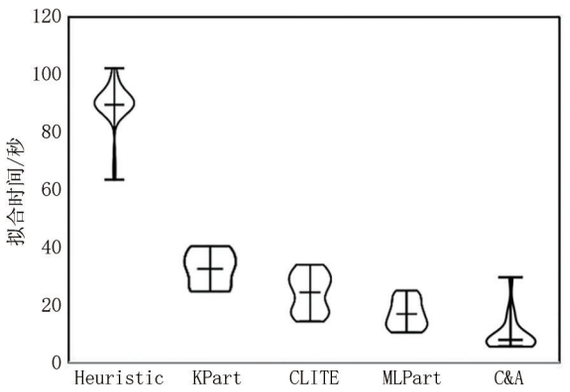


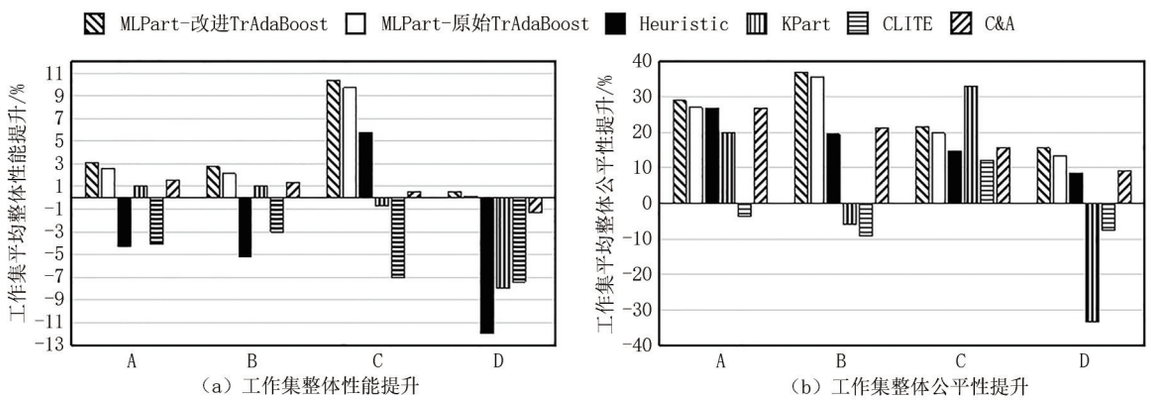
图14　各方法拟合时间

5.3　跨平台移植性能评估

利用TrAdaBoost和微调技术，我们可以实现对目标平台的模型迁移. 我们将不同缓存管理方法应用于目标平台(配置如表1所描述). 我们首先比较了原始的TrAdaBoost与改进后的TrAdaBoost对提升工作集性能和公平性的效果. 由于改进的TrAdaBoost大幅降低了源域中分布差异较大的冗余数据的权重，其提高了迁移后模型的分类准确率.

通过实验我们也可以看到，应用了改进TrAdaBoost的MLPart方法进一步提升了目标平台上工作集整体的性能和公平性. 相较于原始TrAdaBoost的MLPart，使用改进TrAdaBoost的MLPart在四个工作集簇上分别提升了0.56%、0.62%、0.65%、0.37%、0.55%的平均整体性能，而平均整体公平性分别提升了2.01%、1.12%、1.82%、2.1%. 而在B工作集簇中，使用改进TrAdaBoost的MLPart方法相较于原始TrAdaBoost的MLPart方法对#B1-#B7的整体性能平均提升了0.98%，整体公平性平均提升了2.49%. 实际上，与没有迁移学习算法的CLITE方法和C&A方法相比，使用改进TrAdaBoost的MLPart方法更加具有优势. 而更高的分类准确率也让MLPart方法相较于其他方法来说具有更好的划分效果.

更具体地，我们采用改进后的TrAdaBoost方法，进一步从工作集的整体性能提升和公平性提升两个方面与另外四种方法展开了对比. 首先，我们对比对不同工作集簇的平均性能提升，结果如图15所示. 简单Heuristic方法在新平台上对工作集的提升

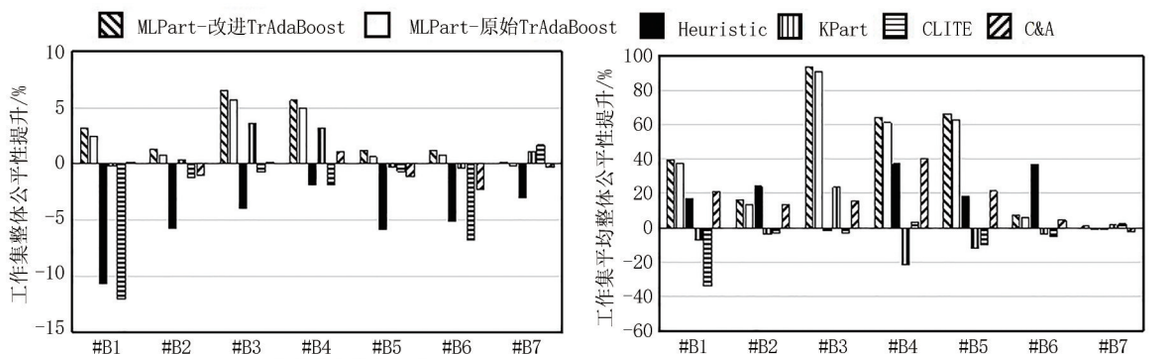


(a) 工作集整体性能提升　　(b) 工作集整体公平性提升

图15　在目标实验平台上针对不同程序数量的工作集簇测试

效果仍然不明显，除了在C工作集簇中相较于Baseline提升了5.57%，在其他工作集簇中基本只起到了负面作用.KPart在程序较少时帮助工作集提升了一定的性能，但是如果程序数量增多，该方法就无法起到性能提升的作用.而CLITE采用了机器学习的模型，但是由于没有迁移机制，其在新的平台上会产生较多错误的划分，应用到工作集之后，反而会严重影响工作集的整体性能，已经无法对工作集性能提升起到正向作用.C&A方法同样没有应用迁移学习，缓存划分效果依赖于对程序的分类结果，而在目标平台上错误的分类会严重影响划分后的整体性能.与这些方法相比，MLPart具有跨平台能力，因此能够持续在新平台为工作集带来整体的性能提升.

进一步，我们对比了各个方法在包含6个程序的B工作集簇(#B1-#B7)中的性能，结果如图16(a)所示.KPart相比于Baseline，平均提升了1.05%的性能.Heuristic的性能在目标实验平台中仍不理想.Heuristic控制下的工作集性能比Baseline的性能更差，平均性能下降达到5.18%.在工作集#B1中，Heuristic的性能下降甚至达到10.67%.而应用CLITE会导致平均3.10%的性能下降；C&A会导致平均0.87%的性能下降.MLPart对工作集整体性能的提升虽然在新的目标平台相较于在源实验平台中有所下降，但仍然优于其他四种方法.



(a) 工作集整体性能提升　　(b) 工作集整体公平性提升

图16　在目标实验平台上针对包含6个程序的B工作集簇测试

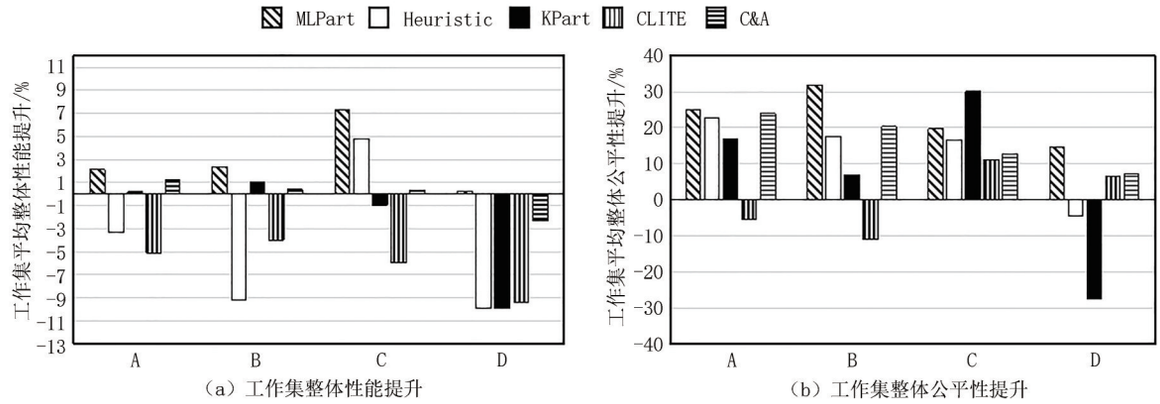
整体公平性评估如图16(b)所示，MLPart相较于Baseline，仍然在提升工作集的整体公平性.而对比Heuristic，KPart，CLITE和C&A四种方法，MLPart对#B1-#B7的平均整体公平性分别提升了22.27%、44.41%、48.13%和24.8%.KPart采用无监督聚类方式，为程序提供分类，在新平台上仍可以对部分程序提供足够的缓存资源；而MLPart迁移后的决策树分类模型，在新的平台上存在着少量的分类错误，也会影响工作集的整体公平性.

为进一步验证模型的跨平台能力，我们使用离散事件驱动全系统模拟器Gem5[48]，模拟了AMD两款具有代表性的处理器AMD EPYC 7451和AMD Ryzen 5 3600作为目标实验平台的核心处理器.Gem5是一个开源计算机架构模拟器，包括系统级架构以及处理器微架构.AMD EPYC 7451拥有24核48线程，最高加速时钟频率为3.2 GHz，具有64MB的三级缓存；AMD Ryzen 5 3600具有6核12线程，最高加速时钟频率为4.2 GHz，具有32 MB的三级缓存(具体配置详见表3).

实验结果如图17和图18所示，Heuristic方法由于不能迅速找到最优的缓存划分方案，因此仍无法稳定地提高工作集的平均性能，存在较大的波动.KPart尽管采用聚类方式对程序进行分类，但由于目标实验平台与源实验平台的巨大差异，也难以提高对程序的分类准确率，性能提升不如MLPart明显.C&A同样采用机器学习的方法对程序进行分类，但由于目标平台的数据分布与源平台存在差异，使用SVM无法对程序进行准确的分类，不具备很好的跨平台能力.而MLPart采用机器学习的方法，在对程序的分类及缓

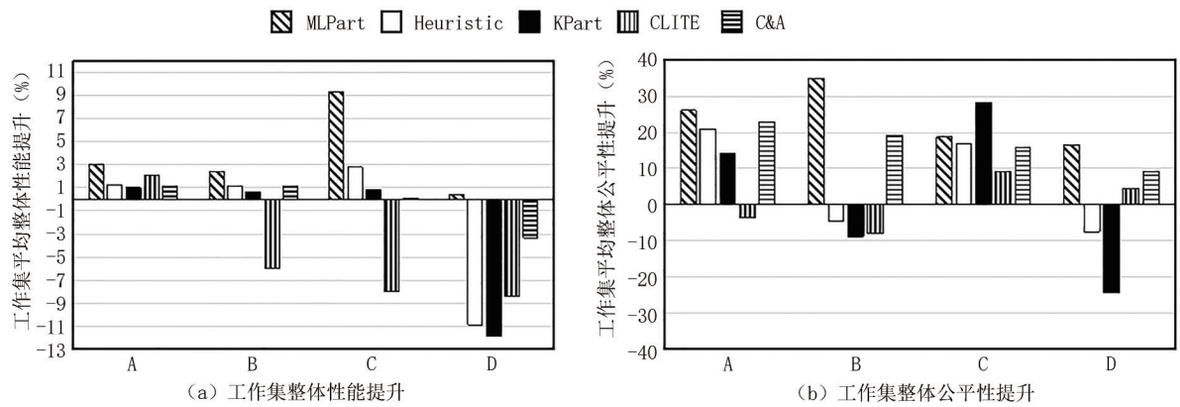
表3　目标实验平台配置

特征	目标实验平台	目标实验平台
CPU	AMD EPYC™7451	AMD Ryzen 5 3600
核心数量	24	6
主频	2.3 GHz	3.6 GHz
一级Cache与二级Cache	348 KB×24与3 MB×24	348 KB×6与3 MB×6
LLC	64 MB	32 MB
内存	128 GB(32 GB×4)	128 GB(32 GB×4)
最大内存带宽	170.6 GB/s	51.2 GB/s
OS版本	Ubuntu 20.04.6 LTS	Ubuntu 20.04.6 LTS
Linux内核版本	5.15.0-69-generic	5.15.0-69-generic



（a）工作集整体性能提升　　（b）工作集整体公平性提升

图17　在AMD EPYC 7451上针对不同程序数量的工作集簇测试



（a）工作集整体性能提升　　（b）工作集整体公平性提升

图18　在AMD Ryzen 5 3600上针对不同程序数量的工作集簇测试

存划分方案的预测上具有更好的鲁棒性，在应对不同数据分布的时候尽管相较于源平台会存在更多的分类错误，但仍能有效地提高工作集的平均整体性能. 相较于Heuristic、KPart、CLITE、C&A方法，MLPart在AMD EPYC 7451上工作集平均整体性能分别提升了7.42%、5.46%、9.15%和3.09%，在AMD Ryzen 5 3600上工作集平均整体性能分别提升了5.22%、6.17%、8.89%和4.02%. 实验结果证明在差异较大的实验平台上，MLPart仍能够实现有效的模型迁移，其通用性得到了进一步验证.

6　结束语

随着程序越来越多样化，访存行为也越来越复杂，程序运行平台中的存储资源层次加深，管理难度增大. 这些趋势都使得构建新型、易用、低开销的缓存管理方法越发重要. 为此，我们提出了一种基于机器学习的跨平台缓存划分方法MLPart，其具有如下优势：(1)缓存划分准确，能够有效提升工作集的整体性能和公平性. (2)划分开销较低，通过少数据量采集完成缓存划分最优方案预测，有效减少了缓存划分的次数. (3)利用迁移学习，基于少量采样完成模型更新，进一步增强了通用性. 另一方面，MLPart仍然需要处理器上执行实现，在一定程度上仍然会消耗处理器的算力资源. 然而，随着智能AI辅助芯片的发展和普及，更加复杂以及精度更高的机器学习算法可以直接由AI辅助芯片执行完成，降低处理器负担的同时，大幅提高处理器的执行效率. 因此在未来的工作中，我们将继续研究多种资源调度智能算法，并将其应用于AI辅助芯片.

参　考　文　献

[1] Agarwal A, Kaur J and Das S. Exploiting secrets by leveraging dynamic cache partitioning of last level cache//Proceeding of 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021: 1691-1696

[2] Araujo B A, Gracioli G, Kloda T, Hoornaert D and Caccamo M. Implementation and evaluation of adaptive cache insertion policies for real-time systems//Proceedings of 2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC), 2021: 1-8

[3] Xu J, Hu Q, Lee W C, et al. Performance evaluation of an optimal cache replacement policy for wireless data dissemination. IEEE Transactions on Knowledge and Data Engineering, 2004,

16(1): 125-139

[4] Qureshi M K, Patt Y N. Utility-based Cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches//Proceedings of 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06. Juan-les-PinsAntibes, France, 2006: 423-432

[5] Iyer R, Zhao L, Guo F, et al. QoS policies and architecture for cache/memory in CMP platforms. ACM SIGMETRICS Performance Evaluation Review, 2007, 35(1): 25-36

[6] Iyer R. CQoS: a framework for enabling QoS in shared caches of CMP platforms//Proceedings of the 18th Annual International Conference on Supercomputing. Malo, France,2004: 257-266

[7] Suh G E, Rudolph L, Devadas S. Dynamic partitioning of shared Cache memory. Journal of Supercomputing, 2004, 28 (1): 7-26

[8] Kim S, Chandra D, Solihin Y. Fair Cache sharing and partitioning in a chip multiprocessor architecture//Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. Orlando, USA, 2004 :111-122

[9] Lu X, Wang R and Sun X H. Premier: a concurrency-aware pseudo-partitioning framework for shared last-level cache// Proceedings of 2021 IEEE 39th International Conference on Computer Design (ICCD), 2021:391-394

[10] Liu L, Li Y, Cui Z, Bao Y, Chen M, Wu C. Going vertical in memory management: handling multiplicity by multi-policy// Proceedings of International Symposium on Computer Architecture. Minneapolis, USA, 2014:169-180

[11] Varadarajan K, Nandy S K, Sharda V, et al. Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions//Proceedings of 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture. Orlando, USA, 2006: 433-442

[12] Liu L, Yang S, Peng L, Li X. Hierarchical hybrid memory management in OS for tiered memory systems. IEEE Transactions on Parallel and Distributed Systems, 2019, 30 (10):2223-2236

[13] Liu L, Li Y, Ding C, Yang H, Wu C. Rethinking memory management in modern OS kernel for multicore systems: horizontal, vertical, or random? IEEE Transactions on Computers, 2016, 65(6):1921-1935

[14] Lin J, Lu Q, Ding X, et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems//Proceedings of 2008 IEEE 14th International Symposium on High Performance Computer Architecture, Salt Lake City, USA, 2008: 367-378

[15] Sherwood T, Calder B, Emer J. Reducing cache misses using hardware and software page placement//Proceedings of the 13th International Conference on Supercomputing. Rhodes, Greece , 1999: 155-164

[16] Ye Y, West R, Cheng Z, et al. Coloris: a dynamic cache partitioning system using page coloring//Proceedings of 2014 23rd International Conference on Parallel Architecture and Compilation Techniques. Edmonton, Canada, 2014: 381-392

[17] Zhang L, Liu Y, Wang R, et al. Lightweight dynamic partitioning for last level cache of multicore processor on real system//Proceedings of 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies. Beijing, China, 2012: 33-38

[18] Albonesi D H. Selective cache ways: On-demand cache resource allocation//Proceedings of the 32nd Annual ACM/ IEEE International Symposium on Microarchitecture. Haifa, Israel, 1999: 248-259

[19] Chiou D, Jain P, Rudolph L, et al. Application-specific memory management for embedded systems using software-controlled caches//Proceedings of the 37th Annual Design Automation Conference. Los Angeles, USA, 2000: 416-419

[20] Xie Y, Loh G H. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. ACM SIGARCH Computer Architecture News, 2009, 37(3): 174-183

[21] El-Sayed N, Mukkara A, Tsai P A, et al. KPart: A hybrid cache partitioning-sharing technique for commodity multicores// Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). Vienna, Austria, 2018: 104-117

[22] Herdrich A, Verplanke E, Autee P, et al. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family//Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). Barcelona, Spain, 2016: 657-668

[23] Wang X, Chen S, Setter J, et al. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support//Proceedings of 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). Austin, USA, 2017: 121-132

[24] Chen S, Delimitrou C, Martínez J F. Parties: Qos-aware resource partitioning for multiple interactive services//Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems. Providence, USA, 2019: 107-120

[25] Park J, Park S, Baek W. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers//Proceedings of the 2019 Fourteenth European Conference on Computer System. Dresden, Germany, 2019: 1-16

[26] PenneyDrew, LiBin, SydirJaroslaw J., ChenLizhong, TaiCharlie, LeeStefan, WalshEoin, LongThomas. PROMPT: Learning dynamic resource allocation policies for network applications. Future Generation Computer Systems, 2023, 145: 164-175

[27] Lv, Wenkai, Yang Pengfei, Zheng Tianyang, Yi Bijie, Ding Yunqing, Wang Quan, Deng Minwen. Energy consumption and QoS-Aware Co-Offloading for vehicular edge computing. IEEE Internet of Things Journal, 2023, 10: 5214-5225

[28] Zhao H., Cui W., Chen Q. and Guo M. ISPA: exploiting intra-SM parallelism in GPUs via fine-grained resource management. IEEE Transactions on Computers. 2023, 72(5): 1473-1487

[29] Kundan S, Anagnostopoulos I. Priority-aware scheduling under shared-resource contention on chip multicore processors//

Proceedings of 2021 IEEE International Symposium on Circuits and Systems (ISCAS). Daegu, Korea. 2021: 1-5

[30] DuanK.. Y. Li. G. Wang and X. Liu. Improving server re-consolidation for datacenters via resource exchange and load adjustment// Proceedings of 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). Taormina, Italy, 2022: 635-644

[31] ShivamKundan, Marinakis Theodoros Anagnostopoulos Iraklis, DimitriKagaris. A pressure-aware policy for contention minimization on multicore systems. A Pressure-Aware Policy for Contention Minimization on Multicore Systems, 2022, 19(3): 1-26

[32] Patel T, Tiwari D. CLITE: Efficient and QoS-Aware Co-location of multiple latency-critical jobs for warehouse scale computers//Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA). San Diego, USA, 2020: 193-206

[33] Nishtala R, Petrucci V, Carpenter P, et al. Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services// Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA). San Diego, USA, 2020: 167-179

[34] Dai W, Yang Q, Xue G R, et al. Boosting for transfer learning//Proceedings of the 24th International Conference on Machine Learning. Corvalis, USA, 2007: 193-200

[35] Yosinski J, Clune J, Bengio Y, et al. How transferable are features in deep neural networks? //Proceedings of Advances in Neural Information Processing Systems. Montreal, Canada, 2014: 3320-3328

[36] Liu E Z, Hashemi M, Swersky K, et al. An imitation learning approach for cache replacement, arxiv, available in: https://arxiv.org/abs/2006.16239, 2020:1-14

[37] Intel-cmt-cat, https://github.com/intel/intel-cmt-cat 2020,8,18.

[38] Qiu, J, Hua, Z, Liu, L. et al. Machine-learning-based cache partition method in cloud environment. Peer-to-Peer Networking and Applications, 2022, 15(1): 149-162

[39] Song Z, Berger D S, Li K, et al. Learning relaxed belady for content distribution network caching//Proceedings of Networked Systems Design and Implementation. USENIX Association (NSDI 20), ClaraSanta, USA, 2020: 528-544

[40] SPEC CPU® 2006, https://www.spec.org/cpu2006/ 2020,8,18.

[41] SPEC CPU® 2017, https://www.spec.org/cpu2017/ 2020,8,18.

[42] Jaleel A, Hasenplaugh W, Qureshi M, et al. Adaptive insertion policies for managing shared caches//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. Toronto, Canada, 2008: 208-219

[43] Armstrong R A. Should Pearson's correlation coefficient be avoided? Ophthalmic and Physiological Optics, 2019, 39(5): 316-327

[44] Safavian S R, Landgrebe D. A survey of decision tree classifier methodology. IEEE Transactions on Systems, Man, and Cybernetics, 1991, 21(3): 660-674

[45] Li Z, Cai J, He S, et al. Seq2Seq dependency parsing// Proceedings of the 27th International Conference on Computational Linguistics. Santa Fe, USA, 2018: 3203-3214

[46] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735-1780

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

[48] Lowe-Power J, Ahmad A M, Akram A, et al. The gem5 Simulator: Version 20.0+. 2020. arxiv, available in: https://arxiv.org/abs/2007.03152

附录.

(1)我们使用了SPEC CPU 2006及2017中的如下程序进行实验测评.

程序集	基准测试程序
SPEC CPU 2006	400. perlbench, 401. bzip2, 403. gcc, 429. mcf, 435. gromacs, 436. cactusADM, 437. leslie3d, 444. namd, 445. gobmk, 447. dealII, 450. soplex, 453. povray, 454. calculix, 456. hmmer, 458. sjeng, 459. GemsFDTD, 462. libquantum, 464. h264ref, 465. tonto, 470. lbm, 471. omnetpp, 473. astar, 483. xalancbmk
SPEC CPU 2017	500. perlbench_r, 502. gcc_r, 503. bwaves_r, 505. mcf_r, 507. cactuBSSN_r, 508. namd_r, 511. povray_r, 519. lbm_r, 520. omnetpp_r, 523. xalancbmk_r, 525. x264_r, 526. blender_r, 527. cam4_r, 531. deepsjeng_r, 538. imagick_r, 541. leela_r, 544. nab_r, 548. exchange2_r, 549. fotonik3d_r, 554. roms_r, 557. xz_r

(2)图2所使用的工作集

工作集	程序
#1	400. perlbench, 459. GemsFDTD, 471. omnetpp, 483. xalancbmk, 507. cactuBSSN_r, 538. imagick_r
#2	444. namd, 450. soplex, 470. lbm, 523. xalancbmk_r, 544. nab_r, 549. fotonik3d_r
#3	473. astar, 502. gcc_r, 505. mcf_r, 541. leela_r, 548. exchange2_r, 549. fotonik3d_r
#4	401. bzip2, 429. mcf, 470. lbm, 500. perlbench_r, 526. blender_r, 557. xz_r

(3)图11-图15所使用的工作集

工作集簇（程序数量）	工作集编号	程序名
A(5)	#A1	447. dealII, 458. sjeng, 462. libquantum, 473. astar, 483. xalancbmk
	#A2	447. dealII, 462. libquantum, 473. astar, 526. blender_r, 557. xz_r
	#A3	435. gromacs, 462. libquantum, 464. h264ref, 500. perlbench_r, 557. xz_r
	#A4	447. dealII, 462. libquantum, 483. xalancbmk, 508. namd_r, 557. xz_r
	#A5	435. gromacs, 462. libquantum, 500. perlbench_r, 508. namd_r, 557. xz_r
B(6)	#B1	459. GemsFDTD, 473. astar, 511. povray_r, 544. nab_r, 554. roms_r, 557. xz_r
	#B2	435. gromacs, 436. cactusADM, 471. omnetpp, 549. fotonik3d_r, 554. roms_r, 557. xz_r
	#B3	444. namd, 459. GemsFDTD, 462. libquantum, 483. xalancbmk, 500. perlbench_r, 544. nab_r
	#B4	401. bzip2, 462. libquantum, 473. astar, 502. gcc_r, 541. leela_r, 549. fotonik3d_r
	#B5	470. lbm, 473. astar, 500. perlbench_r, 519. lbm_r, 527. cam4_r, 541. leela_r
	#B6	401. bzip2, 473. astar, 483. xalancbmk, 519. lbm_r, 531. deepsjeng_r, 549. fotonik3d_r
	#B7	500. perlbench_r, 502. gcc_r, 505. mcf_r, 511. povray_r, 526. blender_r, 549. fotonik3d_r
C(7)	#C1	462. libquantum, 473. astar, 483. xalancbmk, 519. lbm_r, 523. xalancbmk_r, 526. blender_r, 544. nab_r
	#C2	450. soplex, 462. libquantum, 473. astar, 505. mcf_r, 523. xalancbmk_r, 531. deepsjeng_r, 544. nab_r
	#C3	445. gobmk, 462. libquantum, 471. omnetpp, 500. perlbench_r, 523. xalancbmk_r, 538. imagick_r, 549. fotonik3d_r
	#C4	437. leslie3d, 444. namd, 462. libquantum, 473. astar, 483. xalancbmk, 500. perlbench_r, 538. imagick_r
	#C5	450. soplex, 462. libquantum, 464. h264ref, 500. perlbench_r, 508. namd_r, 519. lbm_r, 523. xalancbmk_r
D(8)	#D1	444. namd, 453. povray, 462. libquantum, 464. h264ref, 470. lbm, 500. perlbench_r, 502. gcc_r, 557. xz_r
	#D2	400. perlbench, 450. soplex, 454. calculix, 462. libquantum, 465. tonto, 500. perlbench_r, 549. fotonik3d_r, 557. xz_r
	#D3	403. gcc, 444. namd, 462. libquantum, 483. xalancbmk, 523. xalancbmk_r, 526. blender_r, 531. deepsjeng_r, 557. xz_r
	#D4	437. leslie3d, 447. dealII, 450. soplex, 462. libquantum, 471. omnetpp, 473. astar, 508. namd_r, 511. povray_r
	#D5	450. soplex, 453. povray, 456. hmmer, 458. sjeng, 462. libquantum, 471. omnetpp, 473. astar, 549. fotonik3d_r

QIU Jie-Fan, Ph. D., associate professor. His research interests include operation system, wireless sensor network and artificial intelligence.

JIA Yi-Zhe, M. S. candidate. His research interests include embedded operation system and intelligent computer.

HUA Zong-Han, M. S. His research interests include embedded operation system and computer architecture.

CAO Ming-Sheng, Ph. D., research assistant. His research interests include operation system, wireless sensor network and artificial intelligence.

FAN Jing, Ph. D., professor. Her research interests include service computing and artificial intelligence.

Background

In this paper, we conduct a search on the cache partitioning problem and proposes MLPart, a novel method for partitioning Last Level Cache (LLC) between co-located programs, aiming at improving the system's overall performance. Existing methods to the cache partitioning problem rely either on hardware methods, which requires extra hardware for online profiling, or software methods that suffer from heavy modification to the OS's virtual memory system and high partition overheads.

We propose a machine learning-based LLC partition method MLPart which adopts the idea to first classify programs and then partition LLC according the classes. By this, MLPart improve the flexibility and utilization of shared LLC. MLPart employs Decision Tree (DT) to classify the program according to the LLC accessing behaviors. Based on the program classes, it also employs Recurrent Neural Network (RNN) to predict overall performances of working set under different LLC partition scheme and via searching the max performances, the optimal partition can be found. Benefited from the capability of RNN, only a few of sampling data as input is enough, and thus MLPart effectively reduce the partition overhead, compared with previous methods. In addition, to avoid the effectiveness degradation of DT and RNN model, MLPart also leverage the transfer learning to reduce the cost of migrating models to other platforms.

This research work is supported by National Key Research and Development Project under Grant No. 2018YFB1402800 and Zhejiang Provincial Natural Science Foundation of China under Grant No. LY20F020026.