软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.005815]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

S-Bridge：面向性能非对称异构多核处理器的负载均衡代理机制

赵姗[1,2], 郝春亮[1], 翟 健[1], 李明树[1]

[1](中国科学院软件研究所基础软件国家工程研究中心,北京 110190)

[2](中国科学院研究生院,北京 100190)

通讯作者: 郝春亮, E-mail: chunliang@iscas.ac.cn

摘要: 近年来,在移动计算环境中异构多核处理器已经逐渐成为主流.与传统同构的处理器设计相比,此类异构多核处理器以更低的功耗成本满足设备的计算需求.但是,异构环境下CPU核之间的微架构差异也为操作系统中的一些基本方法提出了新的挑战.本文面向性能非对称异构多核环境下调度的负载均衡问题,从系统层面提出了一种负载均衡机制 S-Bridge,可以减少处理器微架构差异以及任务执行需求差异对传统负载均衡带来的影响.S-Bridge的主要贡献是从系统层提供了通用的、适配异构性的负载均衡相关接口,使任意调度器都能方便地与异构多核处理器系统进行适配.本文基于CFS和HMP调度器在ARM平台上进行实验,同时在X86平台上进行S-Bridge通用性的验证,结果表明,S-Bridge可以支持不同真实平台和内核版本的快速实现,平均性能提升超过15%,部分情况下可达65%.

关键词: 异构多核;异构调度;负载均衡;统一指令集异构多核

中图法分类号: TP302

中文引用格式: 赵姗,郝春亮,翟建,李明树.S-Bridge：面向性能非对称异构多核处理器的负载均衡代理机制研究.软件学报.http://www.jos.org.cn/1000-9825/5815.htm

英文引用格式: Zhao S, Hao CL, Zhai J, Li MS. S-Bridge: The CPU load balancing agent for performance asymmetric multicore processors. Ruan Jian Xue Bao/Journal of Software, (in Chinese).http://www.jos.org.cn/1000-9825/5815.htm

S-Bridge：CPU Load Balancing Agent for Performance Asymmetric Multicore Processors

ZHAO Shan[1,2], HAO Chun-Liang[1], ZHAI Jian[1], LI Ming-Shu[1]

[1](Institute of Software Chinese Academy of Science, Beijing 100190, China)

[2](Graduate University of Chinese Academy of Science, Beijing 100190, China)

Abstract: In recent years, heterogeneous multi-core processors have gradually become the mainstream in the mobile computing environment. Compared with the traditional processor design, they can meet the computing needs of devices at a lower power cost. Microarchitecture differences between the CPU cores also pose new challenges for some basic methods in the operating systems. In this paper, in order to resolve the load balancing problem of heterogeneous scheduling, a new load balancing mechanism called S-Bridge is proposed, which reduces the influence of the processor microarchitecture and the task requirement diversity. The main contribution of S-Bridge is to provide an universal, heterogeneity-aware load balancing interface, so that any scheduler can easily adapt to the heterogeneous multi-core processor systems. The experiments based on CFS and HMP on the X86 and ARM platforms show that S-Bridge can be implemented on different platforms with different kernel versions. The average performance increases by more than 15%, and in some best cases 65% is achieved.

Key words: heterogeneous multi-core; heterogeneous scheduling; load balancing; single-ISA heterogeneous multi-core

基金项目: 国家自然科学基金青年基金(61305054)

收稿时间: 2018-08-04; 修改时间: 2018-11-15; 采用时间: 2019-01-10; jos 在线出版时间: 2020-04-21

1 介绍

性能非对称性异构多核处理器（AMPs,asymmetric multicore processors）在同一处理器中集成不同微架构设计的 CPU 核心（Core）,这些核心具有不同的特性,比如高性能、低功耗,可以协同工作提供良好的性能功耗比（能效）[1-3].目前典型的代表是 ARM 的 big.LITTLE[4]架构设计（例如：高通骁龙 855 和华为麒麟 980）,普遍应用在移动终端场景下,可以根据手机上不同应用的需求[5]选择不同的 CPU 核心（Core）处理,以便达到性能与功耗的平衡.

然而,异构多核处理器设计在提供高能效的同时,也为操作系统的任务调度带来诸多挑战[1],任务调度作为提高多核系统性能和资源利用率的重要手段[6],其中一个重要的问题是 CPU 调度的负载均衡[7][8].由于传统负载均衡主要是针对同构多核处理器设计,其基本思想是保证负载在各个核之间均匀分布,当分布不均时,需要将任务从高负载核心迁移到低负载核心[9][10].然而在异构环境中,不同核之间的微架构存在差异,相同任务在不同类型核上运行的相对负载不尽相同,无法再以任务的平均分布情况作为负载均衡的单一判别标准.同时,由于不同类型任务对于 CPU 核的利用情况差异显著[2-3],传统的负载均衡会导致不合理的决策.

基于此问题, 已有研究工作是面向异构环境进行调度模块本身的设计和优化[7][11-15],为异构环境提供了可用的调度和负载均衡方案; 然而由于缺乏平台级的改进, 无法对传统环境下已有的调度器形成支持, 因而降低了系统调度机制适配的可扩展特性. 在上述背景下,本文提出一种新的负载均衡机制 S-Bridge,在系统层实现对 CPU 微架构和任务异构性的感知;继而对所有调度器提供相应开发接口,协助调度器进行异构感知的负载均衡.

本文的主要贡献如下：（1）提出一种新的负载均衡机制 S-Bridge,在不修改调度算法的前提下,在系统层面实现接口和参数,进行处理器核心和工作负载的异构性的适配.S-Bridge 最显著的贡献是在异构环境下可以协助没有异构支持的调度器进行快速适配和性能优化;（2）提出一种新的负载度量模型,在传统负载度量方法的基础上进行扩展,基于对任务运行性能的经验分析,为负载均衡决策提供感知决策;（3）在不同的内核版本上针对不同的调度算法实现 S-Bridge,验证 S-Bridge 的有效性和通用性.

本文第 1 节介绍研究背景和主要贡献;第 2 节介绍异构调度的相关工作;第 3 节描述异构多核环境下的负载均衡问题;第 4 节详细介绍 S-Bridge 的设计与实现;第 5 节介绍相关实验及结果讨论;第 6 节是结论及展望.

2 相关工作

异构多核处理器最主要的优势是不同类型的核心可以满足不同特定应用的需求[1].因此,在任务调度时,需要感知异构所带来的计算能力的差异性和任务特性,为任务选择更合适的核心.近年异构支持的调度优化出现很多研究工作,主要从满足性能[7][11-14][16-21]、能效[22-26]、公平性[15][27-28]等优化目标提出调度算法.由于本文主要是性能为主要优化目标,下面将重点介绍已有研究工作.

HASS[18]是静态调度算法,在调度之前借助编译器的反馈优化技术提前对程序进行分析,并在二进制文件中保存架构签名,主要包括程序在不同配置核上的访存信息,作为程序是否受益大核的依据.文献[7][11][19]根据应用的运行时状态（比如性能事件信息）进行调度策略的优化,文献[19]通过将线程分配到不同类型核上运行一段时间周期性的进行 IPC 的采样,根据在大核和小核上运行的 IPC 加速比进行线程的迁移,为了避免 IPC 变化所带来的频繁迁移,调度决策依据历史 IPC 和当前 IPC 的加权平均.文献[7]建立 CPI 栈模型,由核内阻塞、核外阻塞和真正执行占用的周期数组成,通过在不同核上采样证明当线程 CPI 栈以执行周期为主时,线程具有大核偏好(BIAS),反之则具有小核偏好,在系统不均衡的情况下,选择最合适的任务迁移到目标核上.文献[11]根据周期性的统计性能事件（包括：LLC 缺失数,指令数,指令之间的依赖距离分布等）建立栈模型,结合硬件的架构参数进行 MLP 和 ILP 的预测,并根据预测结果做出调度决策.文献[20]提出一种异构感知的负载均衡策略,保证核上运行的负载与核功耗成比例,并通过优先使用大核的策略提升系统执行性能,通过在线监控线程的常驻工作集预测线程迁移的代价,并根据迁移代价进行线程迁移的优化.文献[21]采用跟文献[20]相类似的大核优先的调度策略和类似的效果,两者的主要区别是文献[20]在提升性能的同时保证公平性.文献[12]基于间隔分析（interval

analysis）理论模型通过动态获取程序性能数据构建 CPI 栈来统计线程在不同核上的 IPC,并形成以不同 CPU 核配置和 IPC 组成的性能矩阵,作为线程分配和迁移的依据.文献[13-14]通过指令执行由于访存被阻塞的时间建立栈模型,通过大小核的加速比模型作为调度的依据.

文献[15][24]针对动态异构多核处理器,根据应用的运行时状态进行处理器参数的动态调整,文献[15]基于 CFS 提出新的 HFS（heterogeneity-aware fair scheduler）算法,增加集中式任务队列支持逻辑核的快速分配和调整,同时进行公平性决策时候增加核计算性能的因素,实现在利用异构多核性能优势的基础上保证公平性,但是文献没有考虑不同特征应用计算资源的需求,而且异构的适配需要在 CFS 算法的基础上进行修改.其余工作主要针对静态性能非对称异构多核处理器,以下主要针对性能优化目标的工作进行分析.文献[16]提出基于稳定匹配算法的调度技术,维护动态的线程任务和核的优先级表,作为调度的依据.文献[17]提出一种迭代启发式调度算法,在满足功耗线程的情况下提高吞吐量.

以上工作主要是针对调度算法本身,程序分析和调度决策本身具有较强的耦合度,缺乏通用性.而且 HASS 虽然对于有稳定执行状态的负载,尤其是异构度明显的负载有较好的效果,但最主要的限制是无法感知程序变化的执行阶段,而且无法考虑运行状态（比如共享内存状态）的影响.文献[19]和文献[7]都是 IPC 驱动的动态调度算法,需要通过 IPC 采样获取在不同核上的性能数据,信息获取依赖于不同微架构的硬件支持,而且随着核类型的增加可扩展性也比较差.文献[11]的性能预测模型从 PMC 单元无法直接获取,需要增加额外硬件支持.同时,文献[7]虽然通过很少的代码修改可以在多数 Linux 调度器上实现,但是仅对负载均衡时任务选择的策略进行改进和增强.文献[25]虽然是负载均衡算法的优化,但是需要通过修改 Linux 内核的负载均衡算法加入处理器利用率信息,使动态调频和负载均衡更好的协同工作.而不同于以上工作提出或者优化具体调度算法的方法,本文从另外一个角度提供一种代理机制为各种调度算法提供异构的支持.这种机制跟以上工作最显著的不同是不需要直接修改调度算法本身,而是基于运行时信息形成规则影响负载均衡决策,同时提供架构无关的接口与调度器进行交互,适配不同的调度器,可以做到平台无关性和通用性.

同时,HMP（Heterogeneous Multi Processing）是 Linaro 针对 big.LITTLE 架构开发的异构感知的调度算法,在实现上与 CFS 调度算法具有相同的入口,重新定义调度域为大小核两个域,在此基础上改进负载均衡策略,将负载重的任务迁移到大核域内执行,将负载轻的任务迁移到小核域内执行,采用任务处于可运行状态所占用 CPU 时间比率作为负载轻重的依据.由于 HMP 这套机制在 2014 年后的成功商业化,目前大部分 big.LITTLE 设计的移动终端设备上（比如三星 Exynos5430 和 5433 等）都采用了 HMP 机制.因此,本文实验部分将优化工作与此主流的异构调度器进行比对,由于 HMP 调度器在 Linux 内核原有负载计算方法上扩展,对于任务特性的感知在 HMP 中没有考虑,将通过本文的工作对 HMP 进行扩展.

3 异构多核处理器下的负载均衡问题

3.1 传统负载均衡的设计

在主流的 Linux 操作系统中,调度器中的负载平衡针对同构多核系统（SMP）设计,目标是通过在各个核之间均匀分配负载,使各个核之间处于平衡的状态.传统负载均衡的实现采用 pull 和 push 两种方式进行负载均衡,以 pull 方式为例,当前 CPU 运行队列为空的时候,触发负载均衡,调度器将找到负载最重的 CPU,并移动该 CPU 运行队列中的任务到空闲核上.其中,所有 CPU 都被对称对待,默认具有相同的处理能力（sched_capacity_scale）.每个 CPU 工作负载是指运行队列（runqueue）中所有的线程负载的总和.每个线程负载的度量方法随着调度算法的发展而不断演化,最初定义为线程的负载权重（根据线程的优先级定义权重值）,后来发展为负载跟踪度量标准（Load Track Metric）[28],根据历史负载的衰减来跟踪负载,但是,基础的负载依然是线程的负载权重.该度量方法主要基于线程的优先级和平均 CPU 利用率统计线程的负载.传统的负载均衡主要基于CPU对称性和任务对称性的假设,在CFS调度算法中,假定相同优先级的任务具有相同的基础负载权重,而不会考虑任务属性.对于相同优先级而且一直占用CPU的任务,将具有相同的负载,但是优先级相同的任务由于对 CPU 资源需求的不同（比如计算密集型和内存密集型）,在负载均衡的过程中,如果被选择迁移,不能一视

同仁的分配在所有不同类型的目标核上.因此,在异构多核环境下,CPU 的处理能力和任务特性不同对 CPU 造成的负载不同且变化.

3.2 负载均衡异构适配问题

在异构多核处理器环境中,使用传统同构环境的负载均衡算法会导致执行效率问题.其原因是,异构多核处理器是一个相对复杂的架构设计,每种类型的核之间的微架构存在差异[8],包括流水线设计、缓存设计等差异.以 ARM big.LITTLE 设计（包括 ARM Cortex A15 和 A7）为例为例,这两种类型核在微架构方面存在显著差异.A7（小核）主要用于低功耗处理,采用 8-10 级顺序（In-Order）流水线设计;A15（大核）主要用于性能处理,采用 15-24 级乱序（Out-Of-Order）流水线设计.微架构设计的不同导致 A15 和 A7 的处理能力大不相同,在运行处理器运算能力基准程序 Dhrystone 的情况下,A15 可以达到 A7 的 1.9 倍[4].此外,不同的任务类型适合运行的核类型也不同,例如,计算密集型任务更适合在乱序窗口、取指宽度大的乱序核上运行,因此在 A15 上受益明显,更适合迁移到大核上运行;访存密集型任务由于访存延时无法充分利用流水线的并发设计,更适合在小核上运行.因此,基于 CPU 和任务对称性的假设下的传统负载均衡在异构多核处理器上会产生不准确的负载均衡决策.

在目前主流的 Linux 操作系统中,针对异构处理器环境下负载均衡问题,有如图 1 所示两种（Sys1、Sys2）常见的系统状态.一是图中 Sys1 所描述思路,即继续使用为同构环境设计的经典调度器;这些调度器可以在异构处理器环境下正常运行,但由于未针对异构环境进行适配,可能出现前文所述的低效情况.因此,操作系统中常用的调度算法,比如：先进先出（FCFS）、时间片轮转（RR）、最高优先级（HPF）、完全公平（CFS）等,在异构处理器环境都同样面临着异构适配的问题.二是图中 Sys2 所述系统状态,即使用异构处理器专用调度器,其优点是对异构环境的适配较好,其缺点是缺乏通用性.

在以上背景下,本研究认为为异构处理器环境的负载均衡提供系统级的支持是一种可行的、更为通用的方案是.其思路如图 1 中 Sys3 所示,使用专用定制的系统级接口以及参数,传统调度器也可以有效的与异构硬件环境进行适配,且可以保留原有操作系统调度机制的通用性.

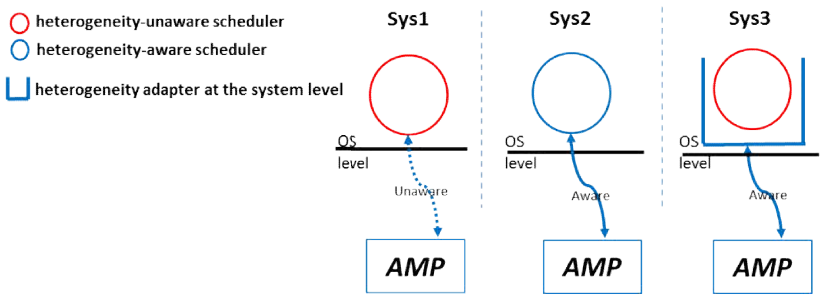

Fig.1 The load balancing problem in heterogeneous multi-core environment

图 1 异构多核环境下负载均衡的适配问题

4 S-Bridge

本文从系统层面提出了一种负载均衡的增强代理机制 S-Bridge,其主要思路是基于针对任务在不同架构类型核上相关执行信息的学习,实现独立于调度算法的异构感知层.S-Bridge 在系统层面提供接口和参数,通过负载扩展因子对线程的基础负载进行动态适配,协助调度器的负载均衡进行决策的优化,将任务分配到更加合适的核上运行.

4.1 系统设计

S-Bridge 的主要思想是在系统中设计实现一种增强代理层,在不修改现有负载均衡算法的前提下,设计一系列异构感知的接口和参数,供调度器使用,协助负载均衡进行异构感知和适配.为了保证 S-Bridge 架构无关性,设计独立的模块与硬件交互获取任务性能事件（PMC[29],Performance Monitoring Counter）信息.S-Bridge 总体设计如图 2 所示,主要核心功能由三部分组成：架构性能收集器、CPU 异构配置收集器和规则生成器,规则生成器基于一定的模型进行适配规则的产生,模型是独立可替换的,本文采用自己提出的可扩展负载模型.

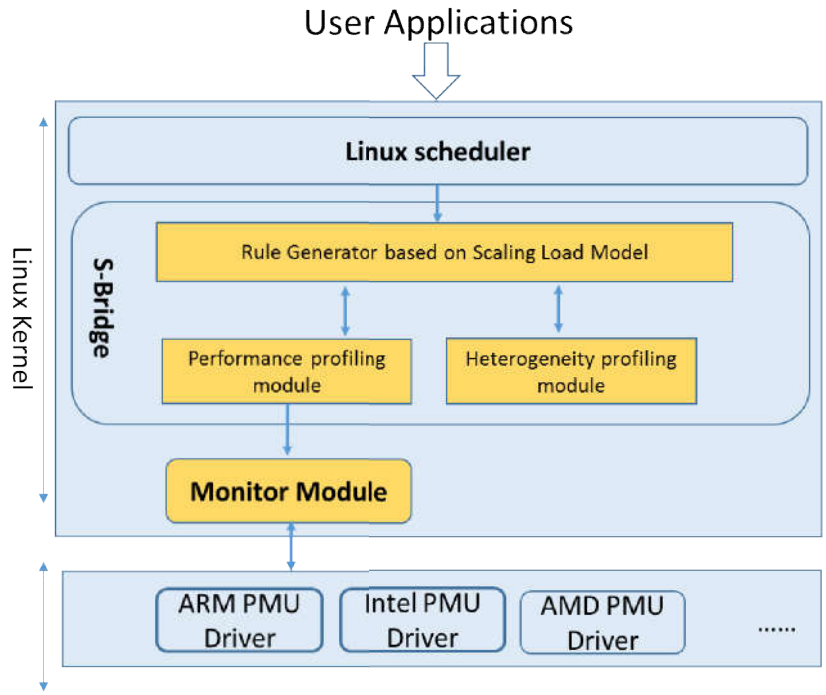

Fig.2 The overall design of S-Bridge

图 2 S-Bridge 总体设计

架构性能收集器的主要目的是在线程进行上下文切换（比如调度和迁移的）时,对线程的性能数据进行收集、分析和预测.它提供一系列回调函数用于实现如下功能：

（1）从硬件性能监控计数器（PMU）获取性能数据;

（2）将这些性能数据比如 LLC 访问缺失率、执行指令数和指令的周期数等转化为内部定义的数据结构进行保存;

（3）基于上面收集的信息进行线程性能数据的分析与预测.在线程调度和迁移的时间点,调用相应的回调函数.

CPU 异构配置收集器提供 API 来检测不同类型 CPU 核的参数,如 CPU 核 ID,缓存大小等,根据收集的信息设置 CPU 核处理能力的初始值.

规则生成器主要是接收架构性能和 CPU 异构配置收集器的数据,基于可扩展负载模型进行规则的产生.本文主要基于可扩展负载的度量模型计算线程的扩展因子,并更新扩展因子矩阵.扩展因子矩阵用来保存每个线程在不同核上的负载扩展因子,并通过接口传给调度器使用,在负载均衡的时候,通过影响线程负载的计算,反映每个线程适应异构多核处理器环境的真实负载.同时,出于性能方面的考虑,扩展因子矩阵主要是用来避免对相同任务性能的重复度量,对于运行的任务如果在矩阵中已经存在相应的项则不再进行度量.

4.2 可扩展负载度量模型

如上所述 S-Bridge 结构的实现基础是可扩展负载度量模型.如 3.1 部分讨论,在同构多核环境下,由于核设计对称,传统基于优先级和平均 CPU 利用率的负载度量方法不需考虑核处理能力和任务特性的区别.然而在异构多核环境下,由于核微架构设计不同,任务资源需求不同,在不同类型核上的性能表现有很大差异,任务在各类型核上形成的负载也有差异,以上两种因素直接影响负载均衡决策.因此,本研究提出一种新的负载度量模型,基于对任务运行性能的经验分析,在原有负载度量方法的基础上进行扩展,考虑不同类型核之间处理能力差异（CPU 因子）以及任务在不同核上的性能差异（任务因子）,从而实现异构感知.该模型包括两个部分：

（1）任务的性能模型,基于 CPI（Cycle Per Instruction）[30]栈模型对于任务的计算需求进行度量,即任务执行所有指令的时钟周期中,真正执行（而不是由于访存或者其它 CPU 资源不足所阻塞）所占用的时钟周期比例被用来表示程序在大核上运行的受益程度.

（2）负载模型,此模型基于程序性能模型和 CPU 核之间处理能力差异进行负载扩展因子的评估.

4.2.1 程序性能模型

该性能模型基于 CPI 栈分析程序在大核上运行的受益程度.CPI 栈是经典的被广泛采用的微架构性能评估模型,CPI 栈主要由基础执行的 *CPI* 和由各种阻塞事件（比如访存缺失）占用的 *CPI* 组成,因此,通过基础执行的时钟周期和由于外部阻塞所占用的时钟周期来计算 CPI 栈.如公式(1)所示,总的 *CPI* 由CPI_B和CPI_S组成,CPI_B表示真正用来执行指令的周期数,表示用来处理阻塞事件的周期数,CPI_S是无法有效利用 CPU 的时间.本文通过硬件性能事件程序执行的周期数（Cycles）以及指令数（Instructions）比值计算 *CPI*,CPI_S通过公式（2）计算,其中M_{ref}表示每条指令的评价平均访存次数,C_{miss}表示最后一级缓存（LLC）访问缺失率,$C_{penalty}$表示缓存访问缺失所带来的时间惩罚,$C_{penalty}$等于访问内存延时.

$$CPI_S = CPI_B + CPI_S \qquad (1)$$

$$CPI_S = M_{ref} * C_{miss} * C_{penalty} \qquad (2)$$

根据以上的定义,CPI_B通过 1-CPI_S计算,本文定义 *CPI_B* 表示 CPU 计算资源的需求,由CPI_B在 CPI 栈所占的比例计算而来.为了对性能模型的合理性进行评估,本部分针对 CPU SPEC2006 的 41 个测试程序进行了实验,图 3 表示所有测试程序的 CPI 栈信息与加速比的关系,可以发现程序的 *CPI_B* 和加速比的曲线具有非常相似的趋势,CPI_B高的测试程序具有高的加速比,表明通过采用CPI_B来表示程序对于大核的受益程度从而用来估计任务因子这种方法是可行且合理的.在图 3 中两条竖直虚线之前的区域与趋势存在偏离,此区域中的程序集中为不同输入集的 gcc 程序,由于输入集的不同造成了 CPI 栈信息的差异.为了避免这种噪声的影响,本文将所有程序的加速比范围划分为 0.1 的区间,在进行任务因子估计的时候,在同一区间的程序具有相同的任务因子值.

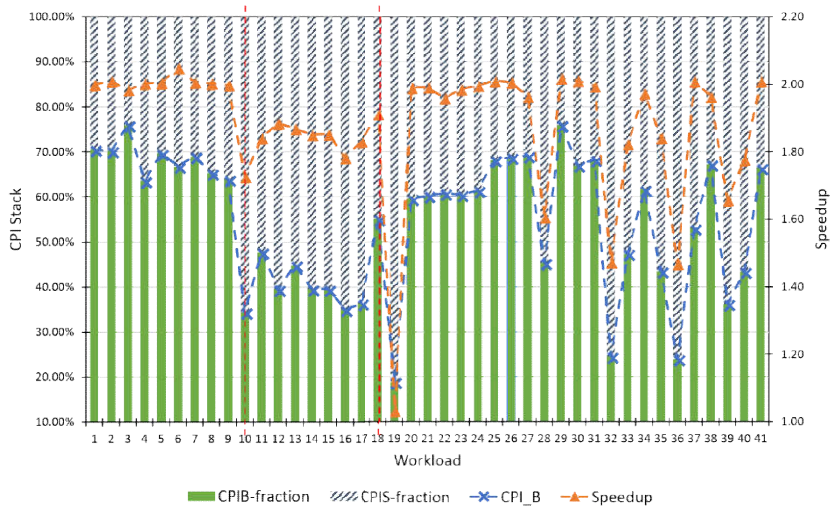

Fig.3 The correlation between the CPI Stack and the application speedup

图 3 CPI 栈模型与加速比的关系

4.2.2 负载模型

负载模型基于程序性能模型和 CPU 核之间处理能力差异进行负载扩展因子的计算.CPU 因子表示异构多核处理器中不同核之间处理能力的比值;任务因子表示不同类型任务之间 *CPI_B* 的比值.假设将APP_m在$Core_k$上的扩展负载因子作为参考,APP_i在$Core_j$上的扩展因子计算如公式（3）所示

$$LF_{ij} = C * Cap_{\frac{k}{j}} * CPI_{B\frac{i}{m}} \qquad (3)$$

$Cap_{\frac{k}{j}}$表示$Core_k$跟$Core_j$之间的处理能力的比值,也就是 CPU 因子,C 是通过测试得到的系数因子.$CPI_{B\frac{i}{m}}$表示App_i与App_m的 *CPI_B* 的比值,也就是任务因子.由于现有的异构多核处理器系统主要有两种类型的 CPU 核（比如 big.LITTLE）,本文选择在大核上运行的纯计算任务（*CPI_B* 为 1）作为参考,这种类型的任务的负载扩展因此假定为 1.

4.3 实现

本研究在 ARM big.LITTLE 平台（Cubieboard4 CC-A80）运行的 Linux 内核 3.4 版本上对 S-Bridge 进行了实现.同时,为证明 S-Bridge 的通用性和可移植性,在 X86 平台（Intel Core™ i7-2600K）运行的 Linux 内核 3.13 版本也进行了实现.

S-Bridge 提供一系列接口与调度器交互,在创建、调度或迁移线程的时候,通过 S-Bridge 架构性能收集器的接口进行线程性能数据的收集,基于可扩展负载均衡模型进行经验分析,并通过规则生成器生成的参数对线程的负载进行动态扩展,将线程分配到更加适合的核上运行.CPU 异构配置收集器提供接口获取或设置 CPU 核处理能力的值,由于平台大小核具有确定的微架构参数,本文对表示大小核之间处理能力差异的 CPU 因子提前测试并进行初始化.其中较为核心的架构性能收集器部分主要功能通过以下回调函数进行实现：（1）do_fork 和 do_exec：当线程被直接创建或者调用 exec()新建的时候,该函数为每个新建立的线程分配并初始化相应的数据结构,比如用来保存架构性能数据的结构,在扩展因子矩阵中分配相应的项,并将该线程的扩展因子的初始值设置为 1,这个值表示在做负载均衡的时候不会对线程的负载有任何影响,还是保持原有的基础负载.（2）do_migration：在负载均衡的时候,当线程被迁移时,线程的性能数据被实时的更新.该函数中读取性能数据的接口通过内核监控模块的加载进行初始化.（3）do_statistic：基于历史的性能数据进行分析和预测.为了简化,这

个函数在将来的工作中被定义和扩展.（4）factor_gen：根据扩展负载的度量模型生成线程的扩展因子,并更新扩展因子矩阵相应线程的项.扩展因子矩阵里主要包括产生缩放因子的基础上可扩展的负载度量模型和更新的缩放因子矩阵相应的条目.每个条目的信息包括 CPU 核类型、任务名称、不同阶段的负载因子等信息.

S-Bridge 为了做到独立和架构无关性,硬件性能数据的获取通过专门的内核监控模块来实现,用来访问和读取底层硬件性能事件计数器,在架构性能收集器的回调函数只是对全局读取函数的指针进行初始化,当内核模块加载的时候会对函数指针进行赋值.在实现过程中,由于 ARM 平台的 PMU 性能事件支持尚未完备,需要在 3.4 内核中增加对于特定硬件事件的支持.具体包括访存缺失数,分支预测错误数等.

5 实验

5.1 实验目标与设置

在本节中,由于 S-Bridge 最突出贡献是在不修改调度算法本身的前提下,协助异构多核处理器环境下没有异构支持的调度器进行快速适配和性能优化.同时,S-Bridge 是独立的架构无关的负载均衡代理层,具有方便的移植性,适用于不同的调度器.因此,实验拟分别在 ARM 和 X86 平台上对 S-Bridge 的有效性和通用性进行评估,实验对象是主流的调度器 CFS（没有异构感知）和 HMP（异构感知）算法.

（1）针对非异构感知的调度算法 S-Bridge 的有效性

本文在 big.LITTLE 设计的 ARM 平台上,以目前 Linux 中主流的 CFS 调度算法为实例进行 S-Bridge 支持前后的对比实验.实验基于 UltraOcta A80 处理器的 ARM 平台,包括 Cortex-A15（指定为大核,缩写为 B）和 Cortex-A7（指定为小核,缩写为 S）两种类型的核,如 3.2 描述,两种核具有不同的微架构设计,分别适用于高性能和低功耗的场景,大小核的频率分别为 1.608G 和 0.72G.该平台运行的内核版本为 Linux kernel 3.4.

（2）S-Bridge 方法的通用性

除了（1）中的 ARM 平台上,本文同时在运行不同 Linux 内核版本的 X86 平台上进行对比实验.实验基于 Intel 4 核心处理器（Core™ i7-2600K）,双线程的 X86 处理平台,跟 ARM 不同的是,由于 4 个核的微架构设计相同,本实验主要通过设置不同的时钟频率来体现核的异构性,频率主要包括 3.2G（指定为大核,缩写为 B）和 1.6G（指定为小核,缩写为 S）.该平台运行的内核版本为 Linux kernel 3.13.

（3）S-Bridge 跟主流异构调度算法的对比

本文选择 big.LITTLE 平台上主流的 HMP 负载均衡算法作为实例进行 S-Bridge 支持前后的对比试验,分析在已经异构适配的调度算法上的效果和影响,同时与 HMP 对比进行 S-Bridge 潜在限制的分析.实验采用与（1）相同的平台.

本文主要选择 1B-1S,1B-3S 和 3B-1S 三种平台进行实验.如表 1 所示,在实验中采用的工作负载主要由不同特性的单线程（比如相对访存多和相对计算多）程序随机混合组成,在实验平台上同时并发的执行.这些程序分别选自 SPEC CPU2006[31]和 MiBench[32]测试套件,SPEC CPU2006 主要针对 X86 平台,而 Mibench 主要针对 ARM 平台.

Table 1 ARM 和 X86 平台工作负载的混合测试程序.

表 1 ARM 和 X86 平台的多程序随机混合负载

Platform	CPUs	Benchmarks
Intel Core™ i7-2600K	1B-1S	mcf, milc, Xalancbmk, soplex, astar, povray, perlbench, bzip2, h264ref, hmmer
	1B-3S	mcf, milc, Xalancbmk, soplex, astar, gcc, povray, gobmk, bzip2, named, h264ref, hmmer
(Cubieboard 4 CC-A80	1B-1S/3B-1S	cjpeg, say, qsort, dijkstra, tiff2bw, lout, bitcnts, crc, sha, patricia, toast, ispell, susan, rawcaudio, basicmath, rihndael, bf, fft,

S-Bridge 的效果通过工作负载中的所有测试程序在原始调度算法和带有 S-Bridge 的调度算法两种情况下的执行时间加速比进行衡量.测试时为了减少系统线程的影响,一方面尽可能的关闭运行的系统线程,另一方面,通过重复执行数百次工作负载,计算负载中每个程序的平均执行时间加速比.测试程序的性能数据主要通过 PMC 记录硬件性能事件,包括程序运行的指令数、周期数和 LLC 缺失数等.如 4.3.1 讨论,由于时钟频率差异也是影响 CPU 处理能力差异的因素,实验平台中每个核的频率通过 CPUFreq[33]技术被设置为固定的频率.

5.2 实验结果与分析

5.2.1 ARM 平台上 S-Bridge 对于 CFS 调度算法的效果

如图 4 和图 5 所示,当 S-Bridge 使能时,所有程序的平均性能提升超过约 68.4%,对于个别执行时间特别短的程序(比如 search_large)达到 100%.在程序运行过程中,更加适合在大核上运行的程序由于获得较多在大核上运行的机会而有相对明显的性能提升,比如 patricia_l 总体性能提升约 71.4%;而对于更加适合在小核上运行的程序 rijndael_s 性能提升约 50.5%,而且它本身执行时间也比较短.对于适合在大核上运行的执行时间短的程序比执行时间长的程序效果明显,由于伴随部分程序执行结束,系统的整体负载下降,会影响到 S-Bridge 效果,S-Bridge 在系统负载重的情况下效果会更加明显.1B-1S 和 3B-1S 有相似的效果趋势,平均性能提升均超过 70.3%.基于相同的 CFS 调度算法,相比于在 X86 平台(如图 6 和 7),S-Bridge 在 ARM 平台上效果更加明显,因为 ARM 平台上不同类型的 CPU 核微架构差异更大,而 X86 平台上仅是频率的差异.因此,实验结果总结如下:

- 基于 CFS,S-Bridge 集成后在很大程度上减少了异构环境下调度的随机性,所有程序的平均执行性能均有明显的提升.S-Bridge 对于没有考虑异构的调度算法效果明显;
- S-Bridge 在系统负载重的情况下效果会更加明显;
- S-Bridge 对于微架构差异大的异构处理器效果会更加明显.

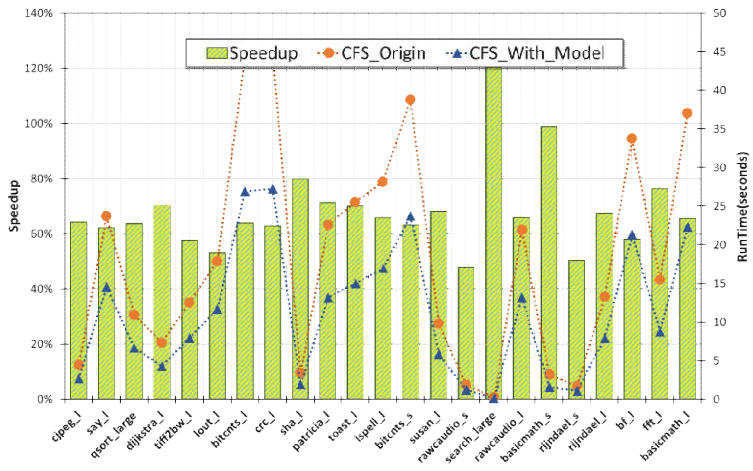

Fig.4 The speedup of 23 Mibench benchmarks running concurrently with S-Bridge and without it in Linux-3.4 scheduler on 1B-1S with 1.608G-0.72G. The average 68.4% performance improvement can be achieved.

图 4 CFS 在分别使能和关闭 S-Bridge 的情况下,23 个 Mibench 测试程序的平均性能提升约 68.4%,实验平台为 ARM 1B-1S,大小核的频率分别为 1.608G 和 0.72G,内核版本为 3.4

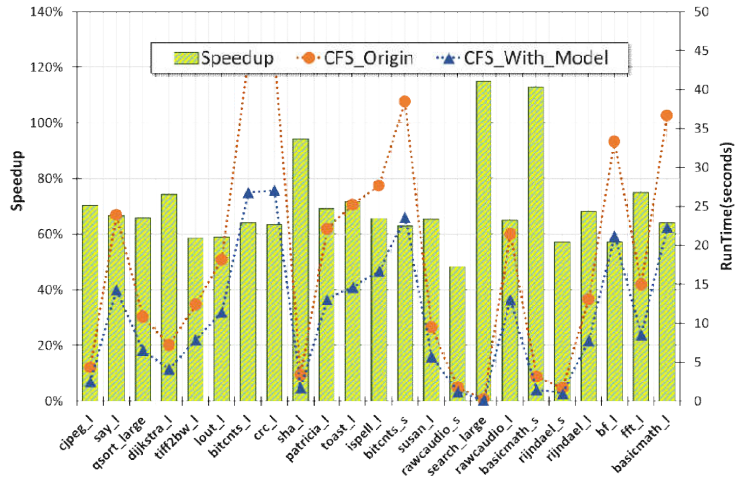

Fig.5 The speedup of 23 Mibench benchmarks running concurrently with S-Bridge and without it in Linux-3.4 scheduler on 3B-1S with 1.608G-0.72G. The average 70.3% performance improvement can be achieved

图 5 CFS 在分别使能和关闭 S-Bridge 的情况下,23 个 Mibench 测试程序的平均性能提升约 70.3%,实验平台为 ARM 3B-1S,大小核的频率分别为 1.608G 和 0.72G,内核版本为 3.4.

5.2.2 S-Bridge 的通用性（X86 平台）评估

图 6 和图 7 表示表 1 中的 X86 工作负载在 Intel 处理器上的平均执行时间及执行时间的加速比,当 S-Bridge 使能的时候,所有程序的平均性能超过 15%.最好的情况下性能提升超过 35%,比如以计算为主更加受益于在大核上运行的程序 hmmer 和 bzip2,但是以访存为主无法明显从大核受益的程序比如 mcf 的性能提升约有 11%,不是特别的明显.1B-3S 的情况下,有些测试程序没有被调度到大核上运行导致了性能的下降,比如 gcc,性能下降约 28%,milc 性能下降约 3%.由于 X86 的测试程序运行时间都比较长,本文对程序特性阶段的变化没有进行细粒度学习和预测,对于 gcc 这种执行计算和访存交替变换的程序,本应在大核执行的阶段没有及时被调度,性能的提升会受到影响.

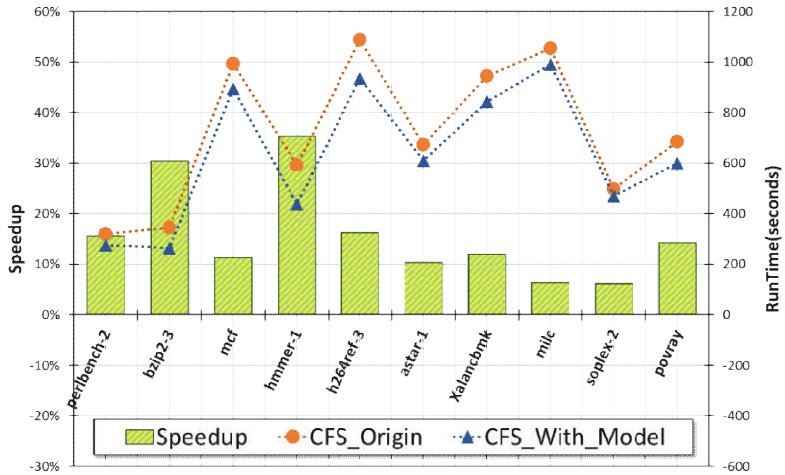

Fig.6 The speedup of 10 CPU SPEC2006 benchmarks running concurrently with S-Bridge and without it in Linux-3.13 scheduler on 1B-1S with 3.2G-1.6G. The average 15.8% performance improvement can be achieved.

图 6 CFS 在分别使能和关闭 S-Bridge 的情况下,10 个 CPU SPEC2006 测试程序的平均性能提升约 15.8%,

实验平台为 X86 1B-1S,大小核的频率分别为 3.2G 和 1.6G,内核版本为 3.13.

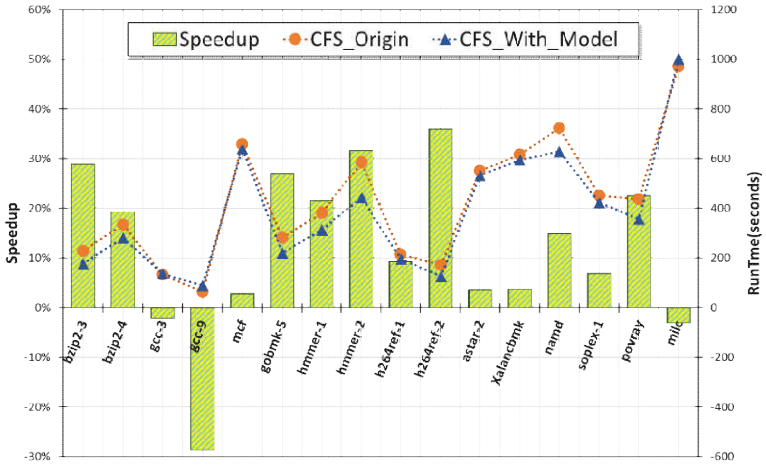

Fig.7 The speedup of 10 CPU SPEC2006 benchmarks running concurrently with S-Bridge and without it in Linux-3.13 scheduler on 1B-3S with 3.2G-1.6G. The average 17.5% performance improvement can be achieved

图 7 CFS 在分别使能和关闭 S-Bridge 的情况下,10 个 CPU SPEC2006 测试程序的平均性能提升约 17.5%, 实验平台为 X86 1B-3S,大小核的频率分别为 3.2G 和 1.6G,内核版本为 3.13.

5.2.3 S-Bridge 跟主流异构调度器的对比

本文以 big.LITTLE 处理器架构下的主流调度器 HMP 为基准进行 S-Bridge 效果的实验与对比,主要包括:

(1) S-Bridge 对 HMP 的影响

图 8 表示测试程序的平均执行时间和执行时间的加速比,当 S-Bridge 使能时,所有程序的平均性能提升约 2.3%.最好的情况下,basicmath_s 性能提升约 6%.其中有三个测试程序(lout_l、toast_l、rawcaudio_l)的性能略有下降,分别约为 0.37%,2.75%,2.51%.

当 S-Bridge 使能的时候,针对 HMP 调度算法虽然有效果,但整体不是特别的明显,在实验中,HMP 算法通过 S-Bridge 模型进行不同任务类型的适配,但是由于 S-Bridge 中对于任务的阶段类型没有进行细粒度学习和预测,所以会影响到任务因子的适应性.在结果上,对于明显受益大核(比如 basicmath_s 和 qsort_large),效果会相对明显,但是对于比如 toast_l 和 rawcaudio_l 这样的程序,整个程序没有特别明显的以计算密集或者访存密集为主,而是阶段交替性出现不同的程序特征,由于阶段类型预测的自适应性没有支持,所以会影响到任务因子的评估而影响到调度决策,在某个执行阶段,本该调度到大核执行,反被分配到小核,从而导致程序性能不升反降.在后面的工作中会继续对任务阶段特性进行学习和预测.

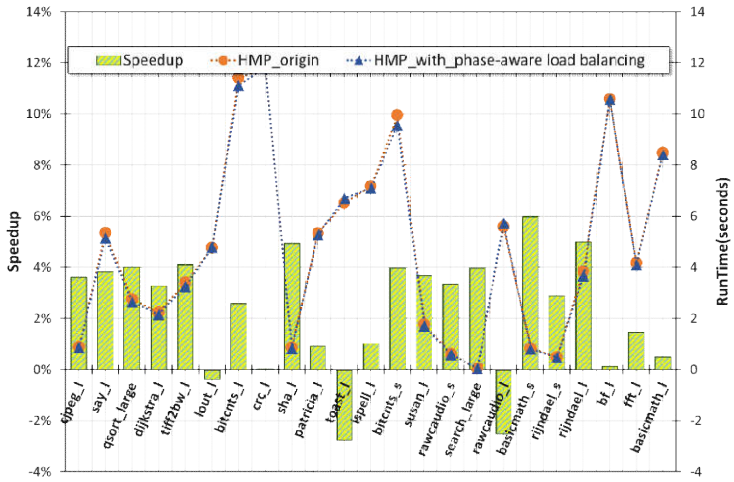

Fig.8 The speedup of 23 Mibench benchmarks running concurrently with S-Bridge and without it in Linux-3.4 HMP scheduler on 1B-1S with 1.608G-0.72G. The average 2.3% performance improvement can be achieved.

图 8 HMP 在分别使能和关闭 S-Bridge 的情况下,23 个 Mibench 测试程序的平均性能提升约 2.3%,实验平台为 ARM 1B-1S,大小核的频率分别为 1.608G 和 0.72G,内核版本为 3.4.

（2）在同构工作负载下 S-Bridge 与 HMP 的对比

图 9 表示在完全同构的负载场景下,没有异构支持的原始 CFS、使能 S-Brige 的 CFS 以及 HMP 工作情况下,工作负载执行完成的时间对比,负载的程序从 Mibench 中选择以访存为主的同类程序(比如 crc_l, rijndael_s, rawcaudio_s 等).在固定频率情况下,相比于原始 CFS 约有 5%的性能提升,而在 DVFS 情况下,由于大核和小核的频率在运行过程中伴随负载而变化,会导致 CPU 因子设置的不合理从而影响调度效果(具体会在 5.3.1 讨论),相比于原始 CFS 反倒略下降（加之考虑到本身的系统开销）.在此种情况下,没有 HMP 的效果明显,HMP 主要有针对性的将占用 CPU 时间长（超过预先设定的阈值）的任务迁移到大核上执行.在 S-Bridge 中,对于任务的阶段类型没有进行细粒度学习和预测,所以会影响到任务因子的适应性和灵敏度,而且工作负载如果完全是同类型访存为主的任务,S-Bridge 会考虑尽量将任务留在小核执行,从而无法充分利用大核资源.因此,S-Bridge 在后面需要考虑 CPU 因子根据频率变化自适应学习,以及任务阶段类型的细粒度学习和预测.

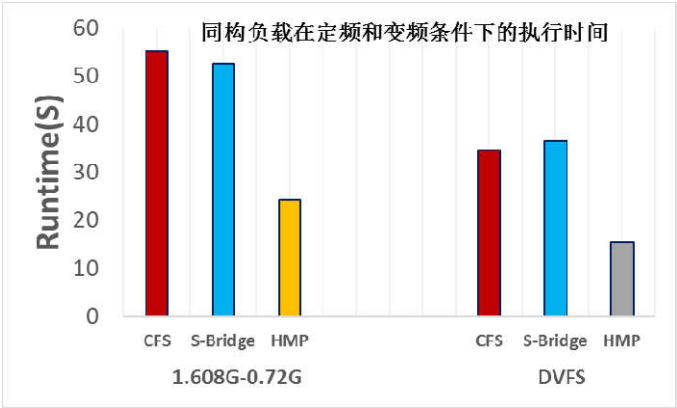

Fig.9 The runtime comparision of the homogeneous workload among the original CFS, the CFS with S-Bridge enabled and HMP on 1B-1S under the conditions of fixed frequencies with 1.608G-0.72G and DVFS. The benchmarks in the workload selected from the Mibench running concurrently are dominated by memory-intensive

applications such as crc_l, rijndael_s, rawcaudio_s and so on. The effect of S-Bridge is limited in this circumstance compared to HMP.

图 9　在完全同构负载场景下,没有异构支持的原始 CFS、使能 S-Brige 的 CFS 以及 HMP 工作情况下执行时间对比. 同构负载的程序主要来自 Mibench 中以访存为主的测试程序,比如 rijndael_s, rawcaudio_s 等. 相比专门的异构调度器 HMP,S-Bridge 没有明显效果.实验平台为 ARM 1B-1S,大小核的频率分别为固定频率（1.608G-0.72G）和自动变频,内核版本为 3.4.

5.2 部分实验结果总体表明：（1）S-Bridge 对于没有考虑异构支持的调度算法效果明显,很大程度上减少异构环境下调度的随机性;而且 S-Bridge 对于微架构差异大的异构处理器效果会更加明显;（2）S-Bridge 可以方便的在不同平台和内核版本上进行移植和实现,对不同版本的调度器起到异构适配的效果;（3）S-Bridge 能够与目前主流的异构调度算法（比如 HMP）协同工作,对 HMP 的任务类型适配性进行优化,有一定的效果.但是在完全异构的工作负载场景下,影响 S-Bridged 的异构适配效果.

5.3 讨论

本部分围绕以下三个方面进行讨论,一是 CPU 因子设置（实验中采用了经验值）对 S-Bridge 效果的影响;二是微架构差异（实验中采用频率设置差异）对于 S-Bridge 效果的影响;三是 S-Bridge 的系统开销.

5.3.1　CPU 因子设置对 S-Bridge 效果的影响

在固定频率的情况下,CPU 因子的设置是否能反映不同核之间的处理能力差异对于 S-Bridge 的效果影响至关重要.图 10 表示,在大核和小核频率为 1.608G-0.72G 的时候,S-Bridge 在 CPU 因子设置为 1.5 的情况下性能最优.而 CPU 因子为 2 和 2.5 的时候,由于对于 CPU 处理能力差异性评估的不合理,会导致大核负载过重而影响性能,而且会造成任务在大核和小核之间的迁移颠簸.因此,不合理的 CPU 因子会严重影响 S-Bridge 的效果,甚至起到负效果.

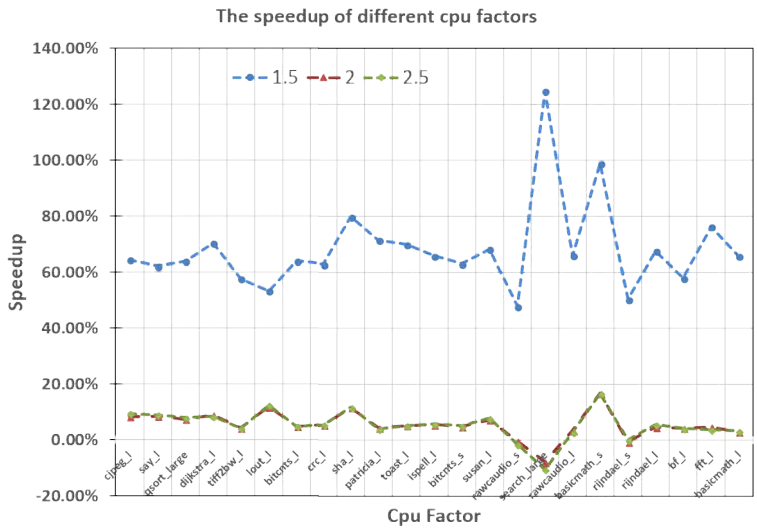

Fig.10　The speedup of the workload on the ARM platform with 0.72-1.608G under different CPU factor which means the ratio of core capacity between the big core and the small core. The appropriate cpu factor like 1.5 here which reflects the real difference of the core capacity will give the most effective performance improvement.

图 10　ARM 平台上当大小核的频率固定为 1.608G-0.72G 时,不同 CPU 因子下工作负载的加速比对比.本实验中当 CPU 因子为 1.5 性能提升最明显,说明 1.5 合理反映了大小核处理能力的差异.

5.3.2　微架构差异对于 S-Bridge 效果的影响

本部分通过设置大小核不同的时钟频率进行实验,讨论微架构差异对于 S-Bridge 的影响.图 11 表示 ARM

平台上大核和小核之间不同的时钟频率设置下所有测试程序加速比的分布,大核设置固定频率为 1.608G,小核的频率设置范围为 0.48G-1.104G.结果表示 S-Bridge 在大核和小核频率为 1.608G-0.72G 的时候性能最优,所有程序的平均性能约 65%,有 50%的程序性能提升超过 70%.而在大核和小核频率为 1.608G-0.48G 效果相对较差,平均约有 10%的性能提升.由于两个核的处理能力相差最大,在负载均衡的时候,更多的任务被迁移到大核上执行,会出现大核在忙,而小核出现空闲的情况,这也是 S-Bridge 后续要继续考虑改进的情况.

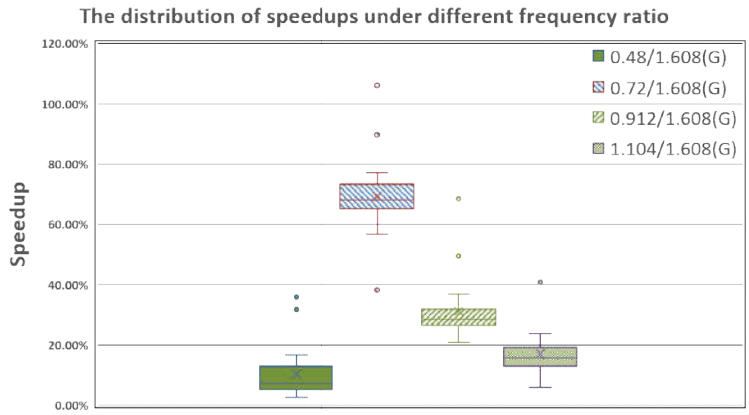

Fig.11 The distribution of speedup under different frequencies on the ARM big.LITTLE platform. The effect of the 0.72-1.608G and 0.912-1.608G is more significant than the others.

图 11 在 ARM 平台上不同时钟频率设置下的测试程序加速比分布.大小核的频率为 1.608G/0.72G 时效果最明显.

5.3.3 S-Bridge 的系统开销

如 4.3 所述,S-Bridge 在原有 Linux 系统上的增强实现主要包括与调度器交互接口和与硬件性能事件寄存器（PMC）交互的部分.调度器在进行上下文切换（调度周期到达、新建任务、唤醒任务、任务迁移）时,进行任务性能信息的统计与分析,生成扩展的负载均衡规则.伴随着所调度任务数量的增加及任务大小的增加,任务上下文切换次数增加,因此 S-Bridge 实现的时间复杂度与调度算法时间复杂度相同,以 CFS 调度算法为例,复杂度为 O(n),本身与调度器交互并不会带来太大的开销.另外,与硬件交互的部分主要是访问 PMCs 和特定的内核数据结构用来保存每个新线程的性能信息.文献[34]实验表明与硬件交互的开销一般少于 1000 个时钟周期（对于程序执行来说时间的影响微乎其微）,这对于内核对调度处理的开销来说是非常小的影响.

本文根据上面的分析进行实验,主要针对原始的 CFS 算法和 S-Bridge 使能情况下的 CFS 算法的性能进行对比,跟 5.2 实验不同的是,S-Bridge 的各个组件都在工作,但是无效 S-Bridge 生成的规则,使其不对调度决策产生任何影响,因此原始 CFS 算法与 S-Bridge 生效的 CFS 算法由于调度对程序执行的影响应该没有差异.选取的测试工作负载是并发执行 10000 次的空函数,在频率固定的核上运行,在原始 CFS 算法的场景下,工作负载完成的时间 690.07S,,在 S-Bridge 使能的情况下为 692.35S,由于单次执行空函数的时间少于 0.008S,由于系统进程的影响,在工作负载执行阶段所发生的上下文切换将大于几十万次,由此验证 S-Bridge 对于每次切换所造成的开销影响是比较小的.

6 结论

本文针对异构多核处理器环境中传统负载均衡问题,提出了一种新的负载均衡代理机制 S-Bridge.该方法的核心思想是在系统层面提出异构感知的接口和参数,在不修改具体调度算法的前提下,协助已有调度器进行异构环境的适配,提高所有可替换调度器在异构处理器环境的决策正确性;同时,为调度器开发提供异构接口支持.本文对 S-Bridge 在不同内核版本的 ARM 和 X86 平台上进行实现和验证.实验表明,在适配未针对异构处理

器优化的调度算法时,S-Bridge 具有明显效果,平均性能提升超过 15%,部分情况下可超 65%.而且,S-Bridge 与 HMP 适配时仍继承了 HMP 本身的优化效果,并且在此基础上进行不同任务类型的适配.但是与 HMP 这种专用调度器相比,在完全同构负载的情况下 S-Bridge 效果并不明显,所以如何同时发挥 S-Bridge 平台特性以及 HMP 等异构环境专用调度器的优势,获得进一步调度优化,以及结合动态电源管理技术(比如 DVFS)向能效的扩展是本研究未来的工作方向.而且,伴随着人工智能,边缘、近似计算等技术的兴起,数据融合的时代已经到来,即使在移动终端也要进行媒体信息识别和处理,对于系统的能效要求越来越高.除了单一指令集(Single ISA)异构系统,也出现通用处理器(CPU)和加速协处理器(GPU、DSP、媒体处理器等)协同的异构指令集(Heterogeneous ISA)系统[35],协处理器主要满足特定需求和目标;同时由于应用执行阶段特性的不同,对于指令级的亲和度也不同,有另外一种将通用 CPU 指令集混成发挥各自优势的研究思路,比如 ARM 和 X86[36],ARM 和精简的专用 ARM 指令集[37]等以满足不同需求.总之,异构系统在向着多样化和"术业有专攻"的方向发展,这也为操作系统、编译器、运行环境等基础软件提出了更多挑战,也是本文异构调度优化扩展延伸的方向.

References:

[1] Mittal S. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. ACM Computing Surveys, 2016,48(3):1-38.

[2] Kumar R, Farkas K I, Jouppi N P, et al. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003:81-92

[3] Kumar R, Tullsen D M, Ranganathan P, et al. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. ACM SIGARCH Computer Architecture News, 2004:64.

[4] Greenhalgh P. Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7. ARM, 2011:1-8.

[5] Shiu E, Prakash S. System challenges and hardware requirements for future consumer devices: From wearable to ChromeBooks and devices in-between. IEEE 2015 Symposium on VISI Technology, 2015:1-5.

[6] Lv Fang, Cui Huimin, Huo Wei, Feng Xiaobing. Survey of Scheduling Policies for Co-Run Degradation. Journal of Computer Research and Development. Journal of Computer Research and Development. 2014,51 (1):17-30.

[7] Koufaty D, Reddy D, Hahn S. Bias scheduling in heterogeneous multi-core architectures. Proceedings of the 5th European conference on Computer systems, 2010:125-138.

[8] Srinivasan S, Kurella N, Koren I, et al. Exploring Heterogeneity within a Core for Improved Power Efficiency. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(4):1057-1069.

[9] Paul McKenny. A big.LITTLE scheduler update. https://lwn.net/Articles/501501/, 2012.

[10] PH Tseng, PC Hsiu, CC Pan, TW Kuo. User-Centric Energy-Efficient Scheduling on Multi-Core Mobile Devices. DAC'14 Proceedings of the 51st Annual Design Automation Conference, 2014:1-6.

[11] Van Craeynest K, Jaleel A, Eeckhout L, et al. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). ACM SIGARCH Computer Architecture News, 2012:213-224.

[12] Chen J, Nair A A, John L K. Predictive Heterogeneity-Aware Application Scheduling for Chip Multiprocessors. IEEE Transactions on Computers, 2014, 63(2):435-447.

[13] Nie P, Duan Z. Efficient and scalable scheduling for performance heterogeneous multicore systems. Journal of Parallel and Distributed Computing, 2012, 72(3):353-361.

[14] Saez J C, Prieto M, Fedorova A, Blagodurov S. A Comprehensive Scheduler for Asymmetric Multicore Systems. New York: Assoc Computing Machinery, 2010:139-152.

[15] Wang T, An H, Sun T, Gao XC, Zhang HB, Cheng YC, Peng Y. Fair scheduling on dynamic heterogeneous chip multiprocessor. Ruan Jian Xue Bao/Journal of Software, 2014,25(Suppl.(2)):80−89 (in Chinese).

[16] Rehman M, Asfand-E-Yar M. Scheduling on Heterogeneous Multi-core Processors Using Stable Matching Algorithm[J]. International Journal of Advanced Computer Science and Applications, 2016, 7(6).

[17] Liu G S, Park J, Marculescu D. Dynamic Thread Mapping for High-Performance, Power-Efficient Heterogeneous Many-core Systems[J]. IEEE 31st International Conference on Computer Design (Iccd), 2013:54-61.

[18] Shelepov D, Saez Alcaide J C, Jeffery S, et al. HASS: a scheduler for heterogeneous multicore systems[J]. ACM SIGOPS Operating Systems Review, 2009, 43(2):66-75.

[19] Becchi M, Crowley P. Dynamic thread assignment on heterogeneous multiprocessor architectures[J]. Proceedings of the 3rd conference on Computing frotiers, 2006:29.

[20] Li T, Baumberger D, Koufaty D A, et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures[C]. Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007:53.

[21] S Balakrishnan, R Rajwar, M Upton, K Lai. The impact of performance asymmetry in emerging multicore architectures. Proceedings of the 32nd International Symposium on Computer Architecture, 2015:506-517.

[22] JC Saez, A Pousa, AED Giusti, M Prieto-Matias. On the Interplay Between Throughput, Fairness and Energy Efficiency on Asymmetric Multicore Processors. Computer Journal, 2018,61(1):74-94.

[23] YG Kim, M Kim, SW Chung. Enhancing energy efficiency of multimedia applications in heterogeneous mobile multicore processors[J]. IEEE Transactions on Computers, 2017, P(99):1-1.

[24] MK Tavana, MH Hajkazemi, D Pathak, I Savidis, H Homayoun. ElasticCore: A Dynamic Heterogeneous Platform With Joint Core and Voltage/Frequency Scaling. IEEE Transactions on Very Large Scale Integration systems, 2018,PP(99):1-13.

[25] Kim M, Kim K, Geraci J R, Hong S. Utilization-aware Load Balancing for the Energy Efficient Operation of the big.LITTLE Processor[J]. Design, Automation & Test in Europe Conference & Exihibition, 2014.

[26] Petrucci V, Loques O, Mossé D, et al. Energy-Efficient Thread Assignment Optimization for Heterogeneous Multicore Systems[J]. ACM Transactions on Embedded Computing Systems, 2015, 14(1):1-26.

[27] Saez J C, Pousa A, Castro F, et al. ACFS: A Completely Fair Scheduler for Asymmetric Single-ISA Multicore Systems[J]. 30th Annual Acm Symposium on Applied Computing, Vols I and Ii, 2015:2027-2032.

[28] Jonathan Corbet. Per-entity load tracking. https://lwn.net/Articles/531853/, 2013.

[29] Intel I. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part, 2013, 1:64

[30] Eyerman S, Eeckhout L, Karkhanis T, et al. A top-down approach to architecting CPI component performance counters. IEEE Micro, 2007, 27(1):84-93.

[31] Henning, John L. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 2006, 34(4).

[32] Guthaus M, Ringenberg J, Ernst D, et al. MiBench: A free, commercially representative embedded benchmark suite. WWC '01 Proceedings of the Workload Characterization, 2001:3-14.

[33] Brodowski D, Wysocki R J, Kumar V. CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/ Documentation/cpu-freq/governors.txt.

[34] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. IEEE Micro, 2008.

[35] Kunio U, Fumio A, Hironori K, Tohru N, et al. Heterogeneous Multicore Processor Technologies for Embedded Systems. Springer Science+Business Media New York, 2012.

[36] Venkat A, Tullsen D M. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor[J]. Acm Sigarch Computer Architecture News, 2014:121-132.

[37] Lee W, Sunwoo D, Emmons C D, Gerstlauer A, John L K. Exploring Heterogeneous-ISA Core Architectures for High-Performance and Energy-Efficient Mobile SoCs. GLSVLSI, 2017.

附中文参考文献:

[6] 吕方, 崔慧敏, 霍玮, 冯晓兵. 面向并发性能下降的调度策略的综述. 计算机研究与发展, 2014, 51 (1):17-30.

[15] 王涛,安虹,孙涛,高晓川,张海博,程亦超,彭毅. 面向动态异构多核处理器的公平调度算法.软件学报, 2014, 25(S2):80-89.