NUMA架构内多个节点间访存延时平衡的内存分配策略

李慧娟 栾钟治 王 辉 杨海龙 钱德沛

(北京航空航天大学计算机学院中德联合软件研究所 北京 100091)

摘 要 随着多核架构的发展和普及,NUMA多核架构凭借其本地访存低延时的优势,被各大商业数据中心以及科学计算集群广泛采用.NUMA架构通过增加多个内存控制器,缓解了多核架构下对同一个内存控制器的争用问题,但同时也增加了内存管理的负担.Linux的系统开发者为了实现充分利用NUMA本地访存低延时的特点,在为进程分配内存时,选择进程当前正在运行的NUMA节点作为分配内存的目标节点.这种分配会导致进/线程之间共享内存的不公平.例如,一个在当前本地节点被分配很多内存的进程,可能被调度到远端节点运行,这样会导致进程的性能波动.针对这一问题,该文设计了一种保证NUMA架构内各内存节点间访存延时平衡的内存分配策略,并在Linux系统中实现和验证.延时的获取方法依赖平台,但是系统内核的策略是通用的.实验结果表明,与Linux默认的内存分配策略相比,进/线程间的不公平性平均降低了16%(最多34%),并且各进/线程的性能没有较大抖动.

关键词 NUMA架构;内存分配策略;访存延时;访存延时感知;访存延时平衡

中图法分类号 TP302 **DOI号** 10.11897/SP.J.1016.2017.02111

A Memory Allocation Policy for the Balance of Access Latency Among Multiple Memory Nodes in NUMA Architecture

LI Hui-Juan LUAN Zhong-Zhi WANG Hui YANG Hai-Long QIAN De-Pei

(*Sino-German Joint Software Institute, School of Computer Science and Engineering, Beihang University, Beijing* 100091)

Abstract Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Modern data centers and the scientific computing clusters widely adapt the NUMA architecture due to its low latency of local memory accessing, which is achieved by partitioning the whole memory into multiple nodes and each memory node is connected to a processors' memory controller. However, this partitioning make the memory management much complicated. To achieve a low memory access latency, the default Linux memory policy chooses to allocate the physical memory pages from the memory node where the process is running. Unfortunately, this allocation policy may potentially result in the unfairness among the processes, e.g., due to the process scheduling, one process with many local memory access may execute on remote node, which leads to the performance variation. To solve this problem, this paper proposes a node access latency indicator. Based on the indicator, we design a memory allocation policy assuring the balance of access latency among memory nodes in NUMA architecture, and implement it in the Linux kernel. To

收稿日期:2015-12-24;在线出版日期:2016-10-16.本课题得到国家"八六三"高技术研究发展计划项目基金(2012AA01A302)、国家自然科学基金(61133004,61361126011,61502019,91530324)资助.李慧娟,女,1990年生,硕士,中国计算机学会(CCF)会员,主要研究方向为并行计算.E-mail:hj_space@163.com.栾钟治,男,1971年生,博士,副教授,中国计算机学会(CCF)高级会员,主要研究方向为分布式计算、高性能计算及并行计算等.王 辉,男,1987年生,博士,中国计算机学会(CCF)会员,主要研究方向为计算机体系结构和计算机系统等.杨海龙(通信作者),男,1985年生,博士,讲师,中国计算机学会(CCF)会员,主要研究方向为运行时系统、性能建模、服务质量及高能效计算机体系结构等.E-mail:hailong.yang@buaa.edu.cn.钱德沛,男,1952年生,硕士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为高性能计算和计算机体系结构等.

obtain the latency is platform dependent, but the policy in system kernel could be universal. The real system based evaluation results show that our proposal can reduce the unfairness up to 34% (with an average of 16%) while keeping a stable performance, compared to Linux's default memory allocation algorithm.

Keywords Non-Uniform Memory Access architecture; memory allocation policy; memory access latency; awareness of memory access latency; the balance of memory access latency

1 引 言

现代数据中心、科学计算集群以及云架构普遍采用多核架构作为基础平台设施. 多核架构凭借可扩展的计算资源提高了计算能力,然而内存一直是限制整体性能的瓶颈. 主要表现有:有限的内存容量、有限的内存带宽和内存控制器的争用. 虽然内存的容量已经足够大,但是相对于片上多核处理器的核数而言,内存资源依旧是有限的. 并且所有的进程访问内存都需要通过唯一的内存控制器,当多个进程同时需要访存时,将会导致多数进程处于等待内存控制器的状态. 非一致访存(Non-Uniform Memory Access,NUMA)多核架构通过把一块内存划分成多个内存节点并且每一个节点附加一个内存控制器的方式,缓解了内存控制器的争用. 在此基础上,所有的片上多核都可以通过高速片上互联和内存控制器访问每一个内存节点. 现在 NUMA 架构的服务器有很多,这些服务器至少含有 2 个片上多处理器,比如戴尔的 PowerEdge① 和惠普的 ProLiant②.

但是 NUMA 架构也使得内存管理更加复杂,尤其是片上高速互联和内存控制器的引入以及远程内存节点的可访问. 访问远端内存节点相较于访问本地节点会增加通过片上互联的额外延时. 现在已有一些研究以实现本地访存获得低延时来提高 NUMA 架构的整体性能. 在 NUMA 架构下,与访存相关的开销可以分为 4 种:最后一级缓存(Last Level Cache,LLC)的争用、内存控制器的拥塞、高速片上互联的拥塞和远端内存访问的延时[1]. 当前大多数研究集中在多级内存资源(LLC 和内存)的争用. 最常使用的解决方法是对应用进行分类学习,使相互干扰达到最小,以获取较高的整体性能[2-4]. 在这些研究中,一些集中在应用间的干扰识别和降低,另一些集中在实时调度策略[5-6]. 但是它们侧重于从调度策略入手提高应用运行过程中访存的命中率,进而提升系统的整体性能,而忽略了系统内存分配策略是影响应用运行时行为的直接因素,内存分配的结果直接影响应用的后期运行行为和系统调度. 因此本文从内存分配策略入手解决 NUMA 架构下共享内存资源时,各应用进程面临不公平的访存延时问题. 这里不公平的访存延时是指:NUMA 架构内各内存节点在分配时已具有的访存延时差异,这种内存节点的延时差异会使并行运行的应用进程遭受不公平的访存延时. 这种不公平性是内存分配策略在分配时未考虑内存节点已具有的访存延时引起的. 运行在这种内存分配策略下,各应用性能差异很大,这种不公平性在单物理机上很常见.

在多核架构下,多个进/线程公平地共享资源将会提高整个系统的性能,这点已有研究证实[7],并且对访存延时是性能瓶颈的应用而言,访存的公平性是整体性能制约的关键. 前人研究多集中在以调度的方式解决争用问题,而据我们研究发现,共享内存资源的各应用进程会由于操作系统的内存分配策略直接导致访存的不公平,进而影响系统整体性能. Linux 系统的内存分配策略仅考虑到尽可能访问本地内存以实现低延时访存,但是忽略了内存节点在分配时已具有的访存延时差异. 而这些访存延时差异直接影响应用的性能,尤其是访存密集型应用. 由于系统内进/线程的实时调度,进程会被调度在远端节点上运行,此时访存延时会在原有的基础上增加片间互联的传输开销. 最终导致应用的访存延时远远超过平均延时,进而影响整体性能. 因此,在NUMA 架构中,访存延时仍然是应用性能的主要制约因素,而在当前的内存分配策略下,它们还忍受着不公平的访存延时带来的性能抖动.

本文针对以上观察到的问题,提出一种内存分配策略,该策略可以感知每一个内存节点的访存延时,并且根据这些节点的访存延时是否平衡做出分配节点的选择,保证每次所选节点的访存延时最低. 为了使多个进程具有相对公平的访存延时,在分配

① Dell PowerEdge Server. http://www.dell.com/PowerEdge 2012

② HP Proliant Server. http://www.hp.com/go/proliant 2011

内存时实时分析各节点的访存延时并选出分配节点. 本文在 Linux 系统中对默认的内存分配策略进行修改，并增加底层微体系结构的支持，实现了访存延时平衡的内存分配策略. 实验结果表明：将 Linux 系统的原内存分配策略替换为本文提出的策略后，进程间共享内存的不公平度最多可由原来的 1.52 降低为 1.04，降低原来的 32%（平均降低 15%），同时可以保证各个进程的性能稳定.

本文主要的贡献如下：

(1) 提出一个可以衡量 NUMA 架构中内存节点访存延时的指标. 该指标可以反映出整体访存开销（详见 3.2 节 NUMA 架构中 GQ 队列所支持的硬件事件信息与所处位置以及 3.4.1 节的实现）.

(2) 提出针对 NUMA 架构的内存分配策略，该策略可以实时感知内存节点的访存延时，并且在保证访存延时平衡的基础上选出分配节点. 该策略用于选出访存延时最小节点并在该节点上分配物理页面，保证访存延时平衡和进程共享内存的公平性.

(3) 在 Linux 系统中实现并且用访存密集型应用验证了提出的策略. 实验结果表明进/线程间的不公平性被显著降低，同时保证了应用整体性能的稳定性.

本文第 2 节详细介绍 Linux 系统的内存分配策略，通过一组简单实验展示目前的分配效果，深入分析造成该分配效果的原因；第 3 节详细介绍本文提出策略的设计并引入 NUMA 架构的关键部件以及策略中各部分的实现和该策略的执行过程；第 4 节展示和分析实验结果；第 5 节总结本文.

2　相关概念及问题阐述

2.1　Linux 系统的内存分配策略

访问本地内存节点的延时要比访问远端节点低的多是 NUMA 架构的一个重要特征. 因此，Linux 操作系统为了充分利用这一特征，对物理内存的组织方式进行了改进. 现有 Linux 系统采用 3 层结构来管理内存，它们分别是：节点、内存区和内存页面，它们之间的关系如图 1 所示.

其中页面（Page）是一个逻辑结构并且与物理内存页框一一对应，页面大小一般是 4KB. 大量连续的页面组成一个内存管理区，也称内存区（Zone）. 内存区是为了有效地管理物理页面的使用而引入的. 内存区可被分为 3 种类型：DMA 内存区、NORMAL 内存区和 HIGHMEM 内存区. 它们按照内存的地址分类，DMA 内存区包含 16 MB 的低地址内存

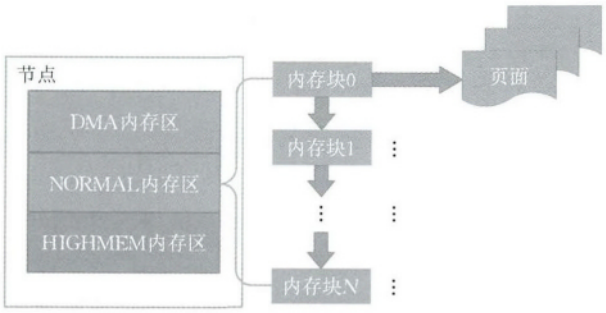


图 1　Linux 内存组织结构

页，用于硬盘 I/O；NORMAL 内存区包含地址高于 16 MB 的内存页，用于给一般的进程分配内存页面；HIGHMEM 内存区比较特殊，在 32 位系统中可能包含一些页面，但是在 64 位系统中，该区域不包含任何页面. 32 位操作系统的寻址能力是 4G，如果系统内存大于 4G，那么超出的内存页面就属于 HIGHMEN 内存区；但是对于 64 位操作系统，它的寻址能力对于当前的内存容量是足够的. 每一种类型的内存区都包含一些小的内存块，这些内存块以单链表的形式存在. 内存节点（Node）与 NUMA 架构中的物理内存节点一一对应，每一个节点内都包含上述的 3 类内存区（如果 HIGHMEM 内存区存在）.

为了实现 NUMA 架构内远端内存节点的可访问，内存的组织方式也做了调整，其中每个内存节点都包含两个单链表（如图 2 所示，以系统中包含两个内存节点为例），每个单链表上的结点包含一定数量内存页面的内存块. 单链表 0 包含整个系统中所有内存区的内存块，不同节点的内存区按照节点间距离的远近来排列（Linux 默认内存分配策略选择此

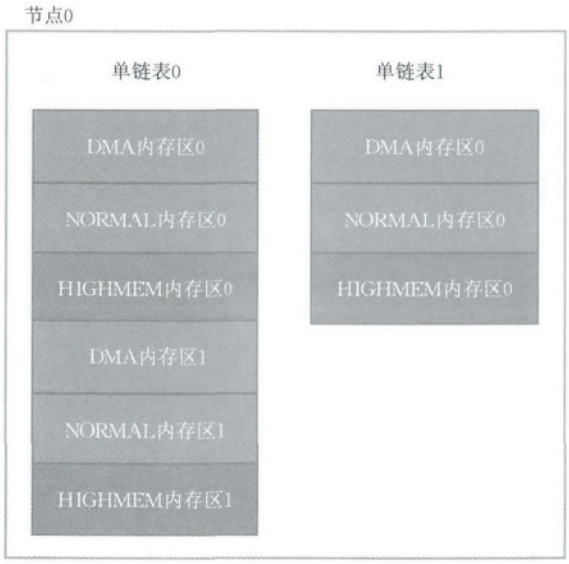


图 2　节点内存区组织形式

链表上的内存块进行内存分配，所选内存块可能分散在多个内存节点)；单链表 1 只包含属于该物理内存节点的内存块(以图 2 为例，节点 0 的单链表 1 仅仅包含属于它的 3 种内存区).

Linux 系统的默认内存分配策略从单链表 0 上选择合适的内存块，并将选定的内存块交给 Buddy 分配器进行具体的内存页面分配. 选择的合适内存块，是指该内存块上的空闲页面对于本次分配是足够的. 为了防止某个内存块的所有页面全部被分配，Linux 系统为每个内存块定义了 3 个“水位线”. 它们分别是 pages_high，pages_low 和 pages_min. 这些水位线用来表示内存块内剩余的空闲页面数量的多少，可以反映当前内存块经受的内存分配压力，图 3 中显示了内存块内可用空闲页面在系统运行过程中随着时间的变化情况.

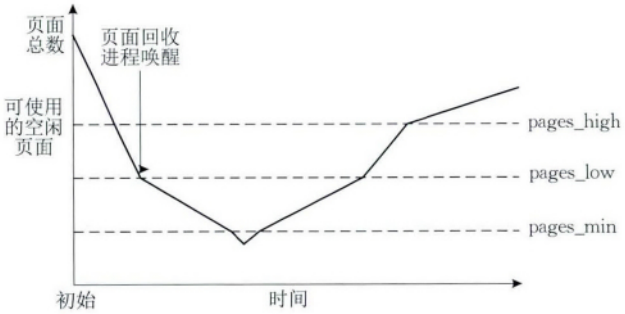


图 3 内存块水位线

在分配过程中，如果一个内存块中空闲页面数量已经降至 pages_low，表示该内存块的分配压力已经很大. 此时分配策略将会选择空闲页面数量在 pages_low 以上的内存块进行分配，这样可以保证每个内存承受的压力处在同一 level. 当所有内存块的空闲页面数量降至 pages_low，系统将会唤醒内存回收进程，为那些空闲页面数量将达到 pages_min 的内存块回收内存页面. 这样做可以保证，系统中每个内存块承受的分配压力相当，但是这种平衡并不能保证访存延时的平衡. 当两个内存块的分配压力一致，假设其中一个内存主要分配给访存密集型应用，而另一个分配给访存非密集型应用. 由于访存频率的极大差异，它们的访存延时很难一致，并且处于不同内存节点的多个内存块，需要经过不同的内存控制器被访问，因此在内存分配时，仅考虑内存块的分配压力并不充分，综合考虑内存块所处内存节点的访存延时是必要的.

当一个进程需要申请内存时，Linux 系统为其分配内存的过程如下：首先选择当前节点作为分配的目标内存节点. 然后选择节点的两个内存单链表之一：如果这个进程的所有分配行为已被用户指定为完全在当前节点，那么将会选择只含有本节点内存块的单链表 1；如果用户没有指定，则根据 Linux 默认分配策略选择单链表 0，并且优先选择属于本节点的内存块. 在单链表上选出合适的内存块后，交给 Buddy 内存分配器，分配物理页面并且将页面的地址返回给上层分配策略.

2.2 Linux 系统的内存分配效果

Linux 分配策略在选择内存块给进程分配内存时，不考虑所选择的内存块是否属于同一个内存节点，会造成进程分配到的内存分散在多个内存节点. 并且分配时不考虑各内存块已经具有的访存延时，会造成应用进程间访存延时的不平衡，即共享内存资源的不公平. 这两点造成的影响，从访存密集型应用的运行情况可以直观看到.

下面通过一组简单的实验来展示内存分配策略的内存分配结果和带给应用的不良影响. 选用 SPEC CPU 2006[8]. 测试集中的 leslie3d 测试程序在配置了 2 路 IntelXeon E5620 多核处理器①的 IBM 刀片服务器上运行. 在关闭处理器的预取和超线程机制的情况下，该服务器单次可并行运行 8 个进程. 配置 leslie3d 测试程序运行时的 8 个运行场景，每个运行场景下运行不同数量的并行运行副本，分别是 1～8. 本次实验选取运行时间作为性能评估标准，这些运行时间是多次运行的平均值.

图 4 所示为测试程序 leslie3d 在 Linux 系统内存分配策略下运行 8 个场景的性能. 其中场景 1 中只运行一个进程副本，该进程的运行时间将作为并行运行时性能下降的基准线. 从图中可以看出，平均运行时间随着进程数的增加而加长，但是运行时间并不是均匀增加. 在一些场景会出现激增，并且 leslie3d 在多次并行运行时整体性能抖动.

同一个场景中多个进程参差不齐的运行时间，

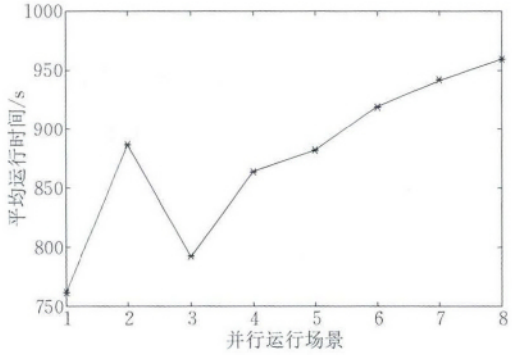


图 4 Leslie3d 在 8 个并行场景运行的性能

① Intel XeonProcessor E5620 Specifications. http://ark.intel.com/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI, 2010

反映了它们共享资源的不公平. 对于访存密集型应用，在计算资源充足的前提下，存储是主要的共享资源. 在本文关注的范围内，存储资源主要指内存，访存延时的不均衡是造成运行时间不稳定的主要原因，这也是引起性能抖动的主要原因. 为了量化这种不公平性，下面给出计算公式，该计算公式引用了 Ebrahimi 等人[7]提出的不公平度定义. 计算公式如下：

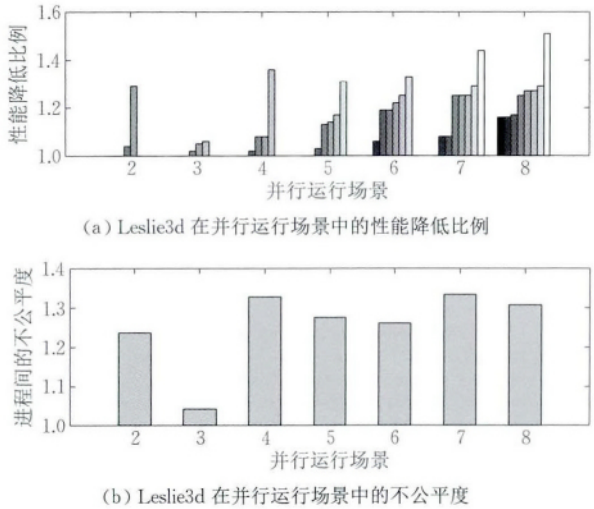
$$IS_i=\frac{T_{\text{ico-run}}}{T_{\text{alone}}},$$

$$Unfairness_{\text{solo}}=\frac{\text{MAX}\{IS_0,IS_1,\cdots,IS_{N-1}\}}{\text{MIN}\{IS_0,IS_1,\cdots,IS_{N-1}\}} \quad (1)$$

其中：IS_i 表示进程 i 的性能降低比例；$T_{\text{ico-run}}$ 表示进程 i 在并行运行时的运行时间；T_{alone} 表示单独运行一个进程的运行时间，则通过它们的比值得到任意进程并行运行时的性能降低比例. $\text{MAX}\{IS_0,IS_1,\cdots,IS_{N-1}\}$ 表示并行运行时性能相对最差进程的性能降低比例，$\text{MIN}\{IS_0,IS_1,\cdots,IS_{N-1}\}$ 表示并行运行时性能相对最好进程的性能降低比例，通过它们的比值得到该次并行运行时各进程间的不公平度[7].

IS 的值越接近 1，表示并行运行的进/线程运行时间与单独运行时相差越小，也从侧面反映访存延时比较小，接近单独运行时的延时；如果 IS 的值比 1 大很多，这表示并行运行的进程的访存延时偏高，导致 IS 增大. 同一个场景多个进程的 IS 值接近，则表示该场景下运行的多个进程公平地共享内存，反之，则表示共享内存的不公平性很大. 由 $Unfairness$ 的计算公式可知，$Unfairness$ 的数值大于 1，并且并行运行多个进程得到的 IS 值差异越大，$Unfairness$ 的值越大，反之越小，趋近于 1.

如图 5 所示为 leslie3d 并行运行时，进程间的性能降低和不公平性（性能降低比例和不公平度分别由式(1)计算得到）. 其中图 5(a)是 leslie3d 在 7 个并行场景中每个进程的性能降低比例，图 5(b)是 leslie3d 在每个场景中进程间的不公平度. 从图 5(a)可以看出在各并行场景中，不同进程的性能降低比例差异很大. 对照图 5(b)可以发现，进程间性能降低比例差异较大的场景，其进程间的不公平度很明显. 同时对照图 4 发现应用平均运行时间激增发生在不公平显著的场景下.



(a) Leslie3d 在并行运行场景中的性能降低比例

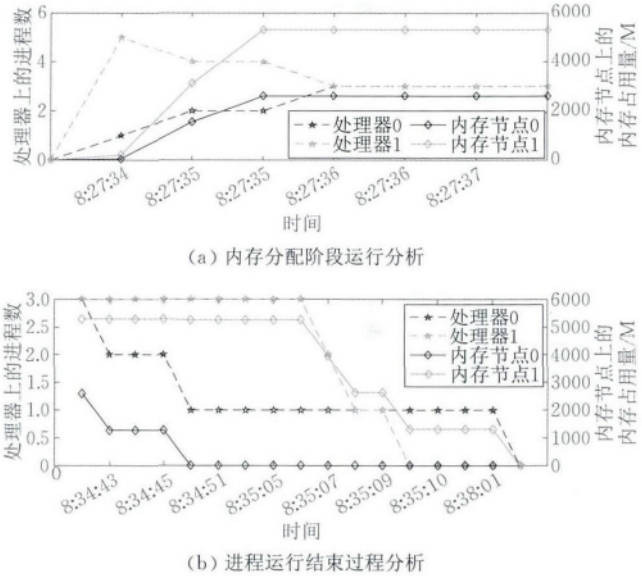
(b) Leslie3d 在并行运行场景中的不公平度

图 5　Leslie3d 在 Linux 策略下运行的效果

由此看来，Linux 系统内存分配策略在运行过程中会增大多个进程间共享内存的不公平，并最终导致应用性能不稳定. 这主要是由于运行过程中进程间较大的访存延时差异引起的.

2.3　应用运行效果的深入分析

为了查明形成性能抖动和进程访存不公平的原因，本节分析了应用运行过程中各进程的“进程(PID)-核-物理内存”三者的映射关系. 其中以并行运行 6 个进程的场景为例，如图 6 所示.



(a) 内存分配阶段运行分析

(b) 进程运行结束过程分析

图 6　并行运行 6 个进程时的运行过程分析

其中图 6(a)展示了 6 个进程在内存分配阶段进程所分配的内存与运行所在处理器的对应关系，图 6(b)展示了 6 个进程运行结束时的状态. 从图 6(a)中可以发现，进程在进入系统后的内存分配阶段，首先是有 5 个运行处理器 1 上，对应的内存节点 1 的内存分配量也很大；而同时只有一个进程运行在处理器 0 上，对应的内存节点 0 的内存分配量为内存节点 1 的一半. 由于系统的负载均衡的进/线程调度，运行过程中两个处理器上运行的进程数趋于平衡，在内存分配结束后，两个处理器上的进程数达到平衡状态. 从内存在各节点的分配比例和在各处理器上的进程数可以发现，有进程运行过程中处于远

端访存状态.各进程的内存分配结果直接影响了进程后期的运行状态,对比图 6(b)可以发现各进程结束的时间各不相同:首先是在处理器 0 上本地访存的进程结束(在 8:34:50,处理器 0 上运行的进程只剩一个,对应内存节点 0 上的已分配内存全部释放);其次是在处理器 1 上本地访存的进程结束(在 8:35:10,处理器 1 上的进程全部执行完,只剩下处理器 0 上远端访存的进程运行);最后是运行在处理器 0 上的远端访存进程运行结束.

经过以上分析可以看出应用进程的性能直接受内存分配阶段所分配的内存影响,不同的内存分配会导致进程遭受不同的访存延时,而访存延时也最终决定了应用的性能.内存分配策略分配在分配内存时不考虑当前节点已具有的访存延时,会使得进程所分配到的内存具有高访存延时,与访存延时较低的其它进程相比,遭受着不公平的访存延时,而且会导致后期运行的性能下降,最终会影响应用的整体性能.并且在分配内存块时不对当前分配内存块与进程已分配到的内存块进行是否属于同一内存节点的检查,这会造成应用多次运行时所分配内存的分布情况不同,因而多次运行的性能差异性较大,也就是说应用性能抖动.结合 Linux 的内存组织方式可以看出,Linux 内存分配策略需要从以下两点继续完善:

(1)分配策略选择单链表 0(包含所有内存节点内存块的链表)上的内存分配,分配过程中不进行严格的节点匹配(当前内存块是否与之前分配给进程的内存块属于同一个内存节点),可能会导致分配给一个进程的内存块分散在多个内存节点.

(2)分配内存时没有考虑节点间的访存延时是否平衡.

因此,本文的目标是设计一个可以感知多个 NUMA 节点间访存延时平衡的内存分配策略,分配时选择当前最小延时节点可以有效降低进程的访存延时,降低进程间的不公平性,同时保证性能的稳定.

3 访存延时平衡的内存分配策略

3.1 整体设计

实现访存延时平衡分配策略的整体设计如图 7 所示,该设计主要包括两个部分:延时感知部分和延时平衡的节点选择部分.

延时感知部分实时监测每个处理器的硬件事件计数器,并计算各节点的访存延时,然后对各节点的访存延时进行排序,得到访存延时的递增节点序列,

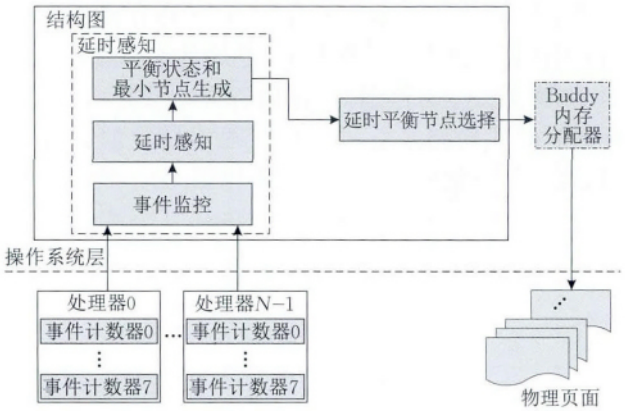


图 7 访存延时平衡内存分配策略的整体设计

最后根据有序序列选择延时最小节点和当前系统内各内存节点间的访存延时是否平衡,并且将这些信息发送给内存节点选择部分.延时平衡节点选择部分,根据延时感知部分提供的信息获取各内存节点的访存延时的平衡状态,并根据这些延时信息做出节点选择,并且对分配给同一进程的内存块所属内存节点的一致性进行验证.然后调用 Buddy 分配器分配物理内存页面.

3.2 延时感知

延时感知部分是基于 NUMA 架构中 Uncore[①] 子系统实现的,Uncore 子系统是 Intel 提出的概念,用来描述多核处理器上除去 core 以外的功能部件,该部分对系统的性能起关键作用[②].Uncore 子系统处于连接计算资源和内存资源的关键位置(详细位置如图 8 所示),并且访存延时可以用访存请求在 Uncore 子系统内的平均滞留时间来衡量.

图 8 以含 2 个节点的 Intel Westmare-EP 架构为例来展示详细的 NUMA 结构.虚线框圈出多核处理器,每个处理器都可以分成 Core 和 Uncore 两部分,Core 部分包含多个物理核,Uncore 部分包含 LLC,IMC 和 QPI[③].每一个物理核都含有一个队列 Super Queue(SQ),这个队列缓存了对应物理核上没有命中 L2 级 cache 的访存请求,这些请求会进入 Uncore 子系统等待内存服务.Global Queue(GQ)是 Uncore 子系统中缓存访存请求的队列,这些被缓存的请求最终被 LLC、本地内存控制器或者通过

① Uncore Wikipedia. http://en.wikipedia.org/wiki/Uncore 2014,12,29

② IntelPerformance Counter Monitor, Thomas, W. (Intel). https://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization 2012,8,16

③ An Introduction to the Intel QuickPath Interconnect (QPI). http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html 2009,1

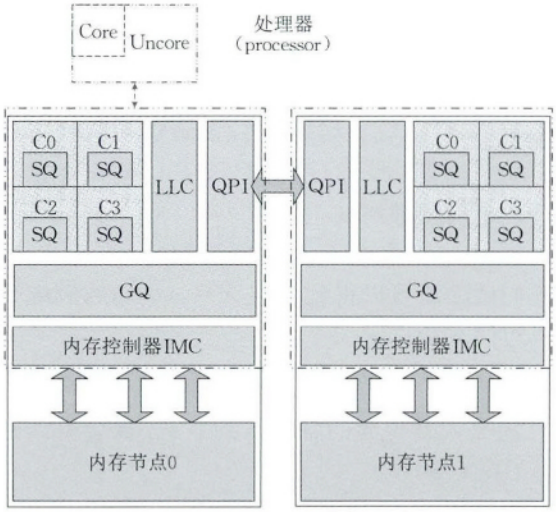


图 8　Intel 的 Westmare-EP NUMA 结构

QPI 的远端内存控制器服务. 而 LLC，IMC 和 QPI 是整个系统内的 3 个主要共享资源，它们看似独立却相互影响，正是这种影响使得 NUMA 架构的内存管理更加复杂但是 GQ 队列却能通过访存请求的平均滞留时间来感知这种影响. 针对访存密集型应用，这种 Uncore 子系统的数据阻塞会严重影响性能[9]，因为大部分请求都需要访问内存，因此通过借助 GQ 中访存请求的平均滞留时间可以得到访存密集型应用的访存延时(文中提到的访存延时严格来说指的是访存请求在 GQ 队列中的平均滞留时间).

延时感知部分由 3 个组件组成：(1) 事件监控组件，该组件周期性地监控每个处理器的 Uncore 事件计数器，并将采集到的信息，发送给上层延时感知组件，事件监控组件是延时感知部分实现的基础，该组件的实现细节见 3. 4. 1 节；(2) 延时感知组件，该组件首先缓存来自监控部件的信息，然后提取有效数据并且按照滑动窗口的思想处理数据以获得稳定的访存延时，最后将各内存节点的访存延时序列发送给上层的平衡状态延时最小节点生成组件，该组件实现的延时获取方法见 3. 4. 2 节；(3) 平衡状态与延时最小节点生成组件，该组件首先接收来自延时感知组件的延时序列，其次对该访存延时序列进行排序，然后根据有序的延时序列，决策多个节点间的访存延时是否平衡并且选择最低延时的节点，最后将节点间的延时平衡状态和最低延时节点发送节点选择模块，该组件的实现细节见 3. 4. 3 节.

为了避免分配选择延时高的内存节点，将延时平衡感知添加在原 Linux 分配策略的上层，并且基于延时感知选择内存节点.

3. 3　延时平衡的节点选择

延时平衡的节点选择部分首先接收来自延时感知部分的信息，并从信息中提取节点间的延时平衡状态和延时最小节点，然后根据分配进程的用户指定策略和系统平衡状态选择内存分配的目标节点. 选定目标节点后，将该节点信息写入进程策略中，并且在该节点的单链表 0 上选择内存块，在选择内存块时，除了进行节点空闲页面容量是否充足的判断外，也进行内存块所属内存节点的验证，验证通过后内存块才被最终选定，以供 Buddy 内存分配器分配物理页面.

延时平衡的节点选择部分是访存延时平衡的内存分配策略的核心，并且在内核发挥作用. 本节描述了该分配策略以及内存块的所属节点验证机制，其中分配策略的实现见算法 1 和 3. 4. 4 节.

算法 1.　延时平衡的节点选择策略.

输入：平衡状态：*GQ_balance*，最低延时节点：*GQ_node*

输出：目标内存分配节点.

```
1. If task memory policy==NULL THEN
2.    If GQ_balance==FALSE THEN
3.        memory allocation target node=GQ_node;
4.    ELSE
5.    memory allocation target node=current node;
6.    END IF
7.    END IF
8. Buddy allocator;
```

Linux 内存分配策略在选择内存块分配内存时，根据内存块中空闲页面的数量进行选择，但是这样可能会导致为同一个进程分配的内存页面分散在不同的内存节点上. 因此为了使其更加完善，本文在内存块选择机制中添加了所属内存节点验证机制. 具体实现见 3. 4. 5 节.

3. 4　访存延时平衡分配策略的实现

本节在内核版本是 2. 6. 32 的 Centos6. 5 系统中实现访存延时平衡的内存分配策略. 所用主机是有 2 个内存节点的 NUMA 服务器(结构见图 8)，该服务器的详细配置如表 1 所示. 本节主要讲述实现细节：(1) 监控硬件性能的工具；(2) NUMA 架构下访存延时的计算方法；(3) 延时分析处理方法；

表 1　服务器配置信息

项目	配置参数
机器型号	IBM HS22 刀片服务器
刀片参数	Intel Xeon CPU E5620，2. 4GHz，每个 CPU 含 4 个物理核
缓存系统	每个物理核含有 L1，L2 cache 两级私有缓存，每个 CPU 共享 L3 cache
内存	12G DDR2 内存
软件环境	Numactl
操作系统	CentOS 6. 5 版本的操作系统 内核版本：linux-2. 6. 32-431. el6

(4)延时平衡的节点选择.前3个在内核模块中实现,主要功能由附加在模块内的进程实现,该进程的生命周期从模块载入内核开始至模块移除为止.

3.4.1 硬件性能工具

该硬件性能工具是自己开发的工具,是基于MSR-tools中的读写方式在内核模块中实现的,是一个运行在Linux内核的可读写Uncore PMU事件的工具.该工具首先初始化Uncore事件的寄存器和计数器,然后周期性地读取和GQ队列相关的事件,这些事件列在表2中(所有这些事件来自于Intel的软件开发手册①,但是更详细的信息来自于Intel研发人员的讨论②).主要读取每个队列的*occupancy*和*alloc*事件,然后将通过收集的数据计算访存延时.

表2 硬件性能事件

硬件事件	事件描述
UNC_GQ_OCCUPANCY_READ_TRACKER	当前读队列中访存请求的占用周期数
UNC_GQ_ALLOC_READ_TRACKER	当前读队列中访存请求个数
UNC_GQ_OCCUPANCY_WRITE _TRACKER	当前写队列中访存请求的占用周期数
UNC_GQ_ALLOC_WRITE_TRACKER	当前写队列中访存请求个数
UNC_GQ_OCCUPANCY_PEER_PROBE _TRACKER	当前QPI队列中访存请求的占用周期数
UNC_GQ_ALLOC_PEER_PROBE_TRACKER	当前QPI队列中访存请求个数

3.4.2 访存延时的计算方法

为了对访存延时进行建模,本文提出一个延时指标.基于在3.2节中介绍的内容:GQ队列中访存请求的平均滞留时间可以计算访存延时,选择GQ队列中请求的平均滞留周期数*GQ_Cycles*作为指标.*GQ_Cycles*与*Uncore penalty*[5,7]有两点不同:首先计算方法不同,*GQ_cycles*完全通过性能计数器收集的数据计算得到;*Uncore penalty*通过估计不同资源的访问时间及比例计算得到.其次意义不同,*GQ_cycles*预测当前运行在同一个处理器上多个应用的整体访存性能;*Uncore penalty*估计运行在某个物理核上的进程性能.通过表2中事件,可以计算出每一个GQ内分队列的访存延时,然后用线性回归的方法得到*GQ_Cycles*.

计算各队列的访存延时:用X表示表中提到的3种队列,分别是读请求队列,写请求队列和通过QPI远端访存的队列.每个队列有两个事件,共6个事件可分为两类表示为:*GQ_OCCUPANCY_X*和*GQ_ALLOC_X*.其中*GQ_OCCUPANCY*事件的计数器记录了访存请求占用的周期数,该计数器会在每个周期按照队列中的请求个数增加周期数.*GQ_ALLOC*事件的计数器记录了当前队列的访存请求个数,该计数器会在请求进入队列时增加请求个数.例如,当前请求队列只放入一个元素,此时*GQ_ALLOC*计数器增加至1,该请求在队列中滞留了100个周期,那么*GQ_OCCUPANCY*计数器会在每一个周期都增加1,最终*GQ_OCCUPANCY*的值增加为100,而*GQ_ALLOC*的值一直是1.然后用它们做除法可以得到*GQ_X_Cycles*,然后统计得到下边的式子:

$$GQ\_Cycles=\sum \partial_x \times GQ\_X\_Cycles \qquad (2)$$

其中∂_x是统计得到的各分队列系数.最后得到延时指标*GQ_Cycles*.

3.4.3 延时分析处理方法

在得到*GQ_cycles*的前提下,用快排序的方法排列多个节点的访存延时.然后判断排序在第一位和最后一位的两个节点的延时是否处在同一个延时等级(在实际运行的系统中,同一延时等级是一个界限值,并且该界限值与系统状态和运行应用都有关系,本文设定的界限值是根据当前系统状态得到的).如果它们的绝对差值小于界限值,那么将返回TRUE,表示当前多个节点间的访存延时平衡;如果差值大于界限值,那么返回FALSE,表示当前节点间的访存延时不平衡.无论当前访存延时是否平衡,最后都会把平衡标志和最低延时节点传递给决策部件.

3.4.4 延时平衡的节点选择

该选择机制在内核中实现,细节见算法1.在分配内存时,首先判断当前任务内存策略是否为NULL,如果策略为空并且此时多节点间的延时不平衡,那么将选择当前的最低延时节点作为目标分配节点;如果策略为空,但是节点间的延时平衡,那么选择当前节点作为目标分配节点.当目标节点确定后,根据内存块选择机制,选定合适的内存块并调用底层Buddy分配器进行最后的物理页面分配.

访存延时平衡的内存分配策略可以实时感知访存延时并判断平衡,但是在Linux系统启动时,所感知的是"伪延时平衡".在内核初始化时,将平衡设置为TRUE,因此系统启动过程中的系统进程申请的内存都选择它们运行时的节点作为分配.进入系统后,模块会自动加载进入内核,只要模块处于加载状态,内存分配策略就可以真实地感知系统内多节点

① Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.cn/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, 2014, 9

② Intel forums: The uncore event about GQ. https://software.intel.com/en-us/forums/topic/540643 2015, 2, 5

间访存延时的平衡，并依照感知信息做出节点选择决策.

3.4.5　内存块所属节点的验证

该内存块所属节点的验证机制在内核中实现，在得到为进程分配的目标节点 *GQ_node* 之后，获取该 *GQ_node* 的节点掩码. 这样该进程在后续内存分配时将根据进程的目标内存节点选择内存块分配内存，可以尽可能保证该进程多次分配的页面处在同一个内存节点上.

在分配过程中，根据 *GQ_node* 的掩码和内存块包含空闲页面的数量来确定最终选择的内存块. 在当该 *GQ_node* 的节点上所有的内存块所包含的空闲页面都低于 page_low 的水位线时，会选择其他内存节点上的内存块分配内存以保证进程申请的内存被成功分配. 如果进程在分配过程中遇到这种情况，需要配合调度策略保证该应用进程的性能，但是本文目前的解决方案中，还没有实现基于访存延时平衡状态的调度策略，这也将是未来的研究方向.

3.4.6　小　结

本节主要描述了访存延时平衡内存分配策略的设计与实现，图 9 是该策略在内存分配过程中的具体流程图. 访存延时平衡内存分配策略的执行过程如下：进程申请内存，触发系统调用内存分配函数，此时进入访存延时平衡分配策略的入口. (1) 延时感知模块在后台实时运行，周期性地获取各内存节点的访存延时以及节点间的访存延时平衡状态；(2) 延时平衡策略读取进程的 *memory_policy* 并判断是否为 NULL，若不为空，则按照 *memory_policy* 的内容分配；若为空，则置 *memory_policy* 为访存延时平衡的内存分配策略，同时读取 *GQ_balance*，此时延时感知模块会返回 *GQ_balance* 和 *GQ_node* 给延时平衡策略；(3) 延时平衡策略根据 *GQ_balance* 判断当前多个内存节点间的延时平衡状态，如果 *GQ_balance* 为 TRUE，则选择当前 node 作为目标内存分配节点并且记录为 *memory_policy* 的优先选择节点；若 *GQ_balance* 为 FALSE，则选择最小延时节点 *GQ_node* 同时在 *memory_policy* 中记录节点信息；(4) 最后，将所选择的节点上合适的内存块交给 Buddy 进行物理页面分配.

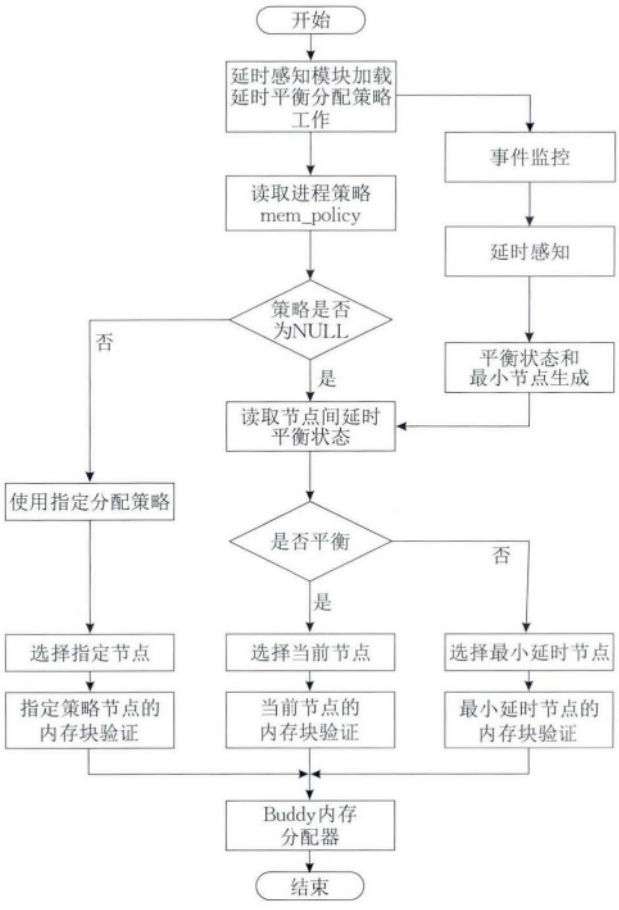


图 9　访存延时平衡内存分配策略的内存分配流程图

延时感知模块未加载时，分配策略感知的伪平衡状态不会影响进程的内存分配. 因为当延时平衡内存分配策略判断节点间延时平衡时，会按照默认操作系统策略的分配方式选择当前节点分配内存，唯一的区别是，延时平衡策略会选择同一内存节点的内存块为同一进程分配内存，以保证进程的内存在同一个内存节点保证进程访存的公平性.

4　实验结果分析与评价

本节通过对比分析实验结果来评价访存延时平衡内存分配策略(在本节后续小节中称为延时平衡策略)的效果. 主要从宏观上延时平衡策略的分配效果和微观上各内存节点之间的访存延时两个方面进行评价. 在 4.1 节选用 SPEC CPU 2006，NPB[①] 和 stream[②] 3 种测试集应用进行实现效果的对比分析. 首先通过 2.2 节描述的实验场景对比分析延时平衡策略和 Linux 系统内存分配策略(后续小节称为原 Linux 策略)的宏观实验效果，在不公平性对比分析中加入手动调优策略辅助对比分析，其中手动调优策略是指根据 numactl 设置的本地访存(localalloc)配置，以实现所有进程的本地访存. 然后在 4.2 节对各测试应用在延时平衡策略和原 Linux 策略下运行

① NPB. http://www.nas.nasa.gov/publications/npb.html
② McCalpin J D. STREAM: Sustainable memory bandwidthin high performance computers. http://www.cs.virginia.edu/stream/1995

时各内存节点的访存延时进行了对比分析. 最后在4.3节通过多个访存密集型应用混合在原Linux策略,延时平衡策略以及手动调整最优的3种策略下执行的实验结果对比,验证延时平衡策略的有效性.

4.1 同一应用多个进程实验的对比分析

本节对比分析3个方面:并行运行时应用的整体性能(图10),并行运行时的各应用进程的性能降低(图11～图18),并行运行时应用进程间的不公平(图12).

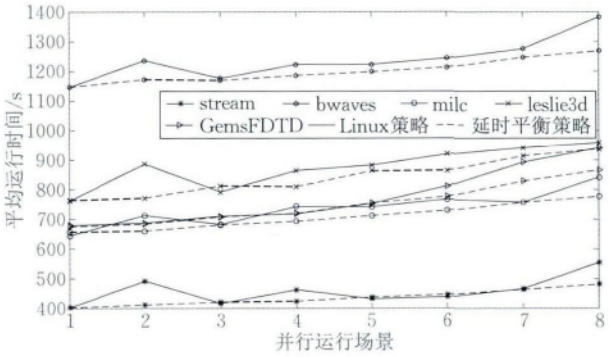


图10 两种策略下的应用性能对比图

4.1.1 应用的整体性能

图10中显示的是应用在延时平衡策略和原Linux策略下运行的性能对比图. 为了清晰地展示图中信息,选用了SPC CPU 2006中的bwaves,milc,leslie3d,GemsFDTD和STREAM的5个应用. 其中实线是在原Linux策略下运行的结果,虚线是采用本文策略运行的结果. 从图中可以看出随着并行进程数的增加,实线图的运行时间不平稳地增长,并且有凸起;而虚线图的运行时间几乎是线性增长的,并且增长幅度整体低于实线. 由此可见,延时平衡策略能够保证同一应用的多个进程并行时性能稳定. 为了进一步验证,在图11～图18给出并行运行时,每一个进程的性能降低比例.

4.1.2 同一应用多个进程并行运行时的性能降低

本小节显示的是在两种策略下并行运行时,每个进程的性能降低的对比. 图11(a)是leslie3d在原Linux策略下运行的实验结果,图11(b)是leslie3d在延时平衡策略下运行的实验结果. 图11(a)与(b)对比很明显,在延时平衡策略下运行的并行场景中,同时运行的多个进程由共享内存带来的性能降低比例趋于一致,而在原Linux策略下,并行的多个进程由共享内存带来的性能降低比例参差不齐,这种参差不齐反映出它们共享内存的不公平. 同时对比图11中leslie3d在各场景的性能情况,延时平衡策略的平稳性很好,因此减小进程间性能降低比例的差异可以有用地稳定整体性能.

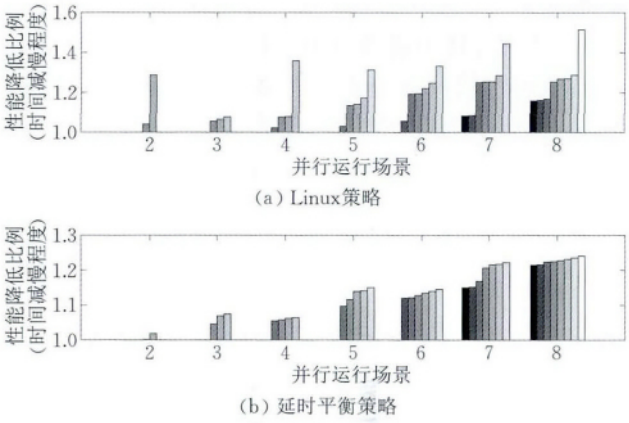


图11 Leslie3d在两种策略下的性能降低对比

图12中显示了stream应用在延时平衡策略和原Linux策略下并行运行时,每个进程的性能降低比例对比. 其中图12(a)是stream在原Linux策略下运行的实验结果,图12(b)是stream在延时平衡策略下运行的实验结果. 图12(a)与(b)对比很明显,从纵坐标的数值范围可以发现,Linux策略下运行的各并行场景中,性能减慢比例在8个进程并行运行时最大达到了1.70,而在延时感知策略下运行时最大仅为1.24. 同时发现在延时平衡策略下,2～6个进程运行时各进程的性能减慢比例在1.10左右,在7～8个进程运行时进程性能减慢比增大,但是都小于Linux策略导致的性能降低比例.

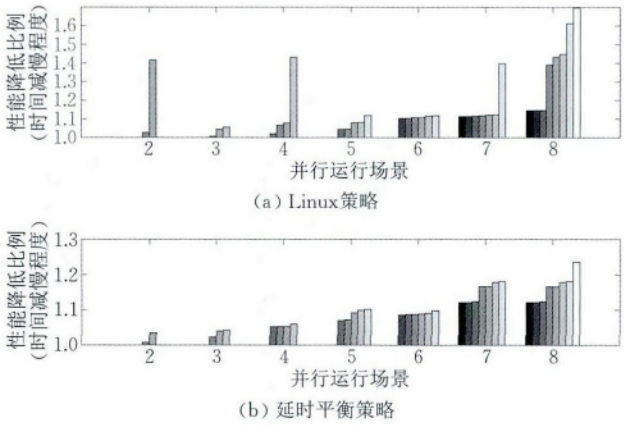


图12 Stream在两种策略下的性能降低对比

图13显示了应用bwaves在延时平衡策略和原Linux策略下并行运行时,每个进程的性能降低比例对比. 其中图13(a)是bwaves在原Linux策略下运行的实验结果,图13(b)是bwaves在延时平衡策略下运行的实验结果. 图13(a)与(b)对比很明显,bwaves应用在延时平衡分配策略下运行时,7个并行场景中各进程的性能降低比例保持在1.12以下,

即使在 8 个进程满负载运行时也同样保持，并且每个场景下各进程的性能降低比例趋于一致；但是在 Linux 策略下运行的各并行场景中，性能降低比例在 8 个进程并行运行时最大达到了 1.32，最小 1.1 以下，且在每个并行场景中各进程的性能降低比例差异性比较大. 在 Linux 策略下场景 3 运行时，各进程的性能降低比例维持在 1.05 以下，说明 Linux 策略在分配时的节点选择随机性直接影响了并行运行各进程的性能降低比例. 延时平衡策略的节点确定选择机制保证了策略分配时各进程性能的稳定性.

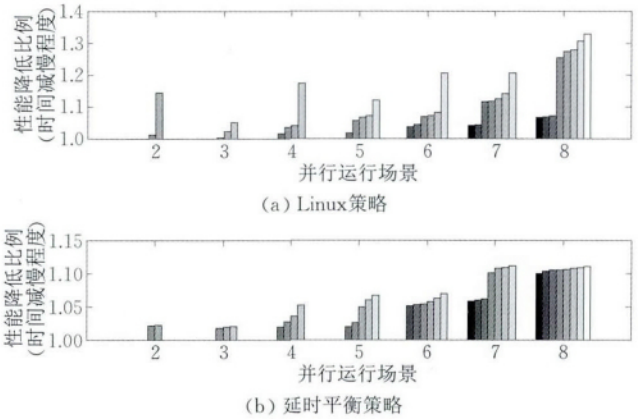


图 13　Bwaves 在两种策略下的性能降低对比

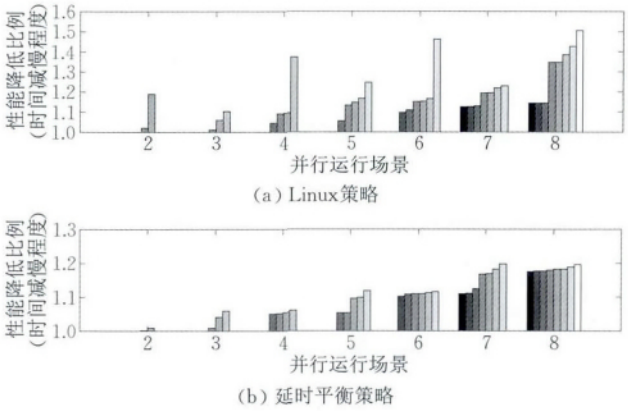


图 14　Milc 在两种策略下的性能降低对比

图 15 中显示了应用 milc 在延时平衡策略和原 Linux 策略下并行运行时，每个进程的性能降低比例对比. 其中上子图是 milc 在原 Linux 策略下运行的实验结果，下子图是 milc 在延时平衡策略下运行的实验结果. 两副子图对比很明显，milc 应用在延时平衡分配策略下运行时，每个并行场景下各进程的性能降低比例基本一致，并维持在 1.20 以下. 而在 Linux 策略下运行时，各进程的性能降低比例不一致，体现出进程间访存的不公平.

图 15 显示了应用 GemsFDTD 在延时平衡策略和原 Linux 策略下并行运行时，每个进程的性能降

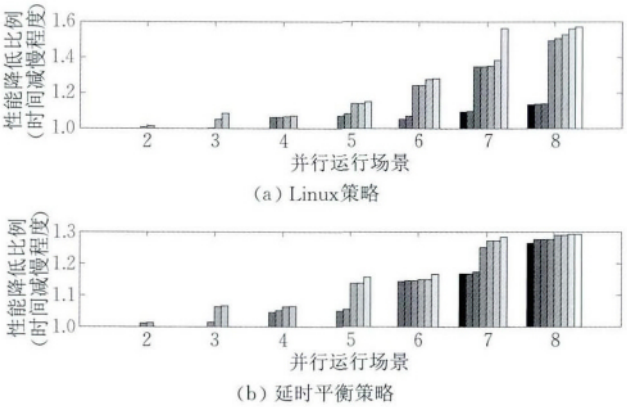


图 15　GemsFDTD 在两种策略下的性能降低对比

低比例对比. 其中上子图是 GemsFDTD 在原 Linux 策略下运行的实验结果，下子图是 GemsFDTD 在延时平衡策略下运行的实验结果. 两副子图对比很明显，GemsFDTD 应用在延时平衡策略下运行时，每个并行场景下各进程的性能降低比例都一致，但是相对其它应用，仅维持在 1.30 以下. 即使如此各进程的性能降低比例也都小于 Linux 策略下运行的情况.

图 16 中显示了应用 IS 在延时平衡策略和原 Linux 策略下并行运行时，每个进程的性能降低比例对比. 其中上子图是 IS 在原 Linux 策略下运行的实验结果，下子图是 IS 在延时平衡策略下运行的实验结果. IS 应用在延时平衡策略下运行时，各进程的性能降低比例基本一致，但是相对其它应用，IS 在并行运行时由于并行带来的性能降低明显，延时平衡策略也仅能将降低比例仅维持在 3.15 以下.

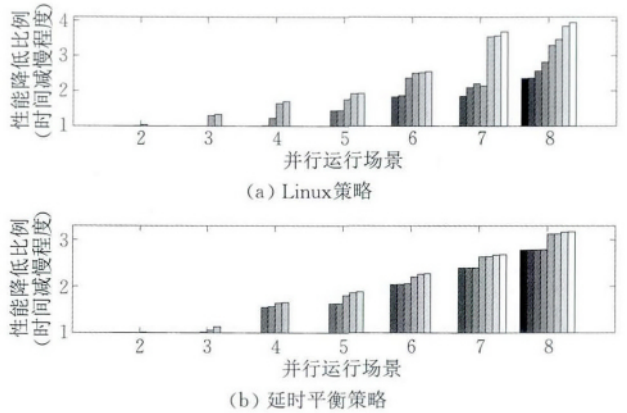


图 16　IS 在两种策略下的性能降低对比

图 17 中显示了应用 EP 在延时平衡策略和原 Linux 策略下并行运行时，每个进程的性能降低比例对比. 其中上子图是 EP 在原 Linux 策略下运行的实验结果，下子图是 EP 在延时平衡策略下运行的实验结果. 两副子图对比很明显，EP 应用在延时

平衡策略下运行时，各进程的性能降低比例都一致，且在进程数小于 8 的场景中性能降低比例接近 1，EP 并行运行时由并行带来的性能降低不明显，延时平衡策略将其性能降低比例维持在 1.2 以下.

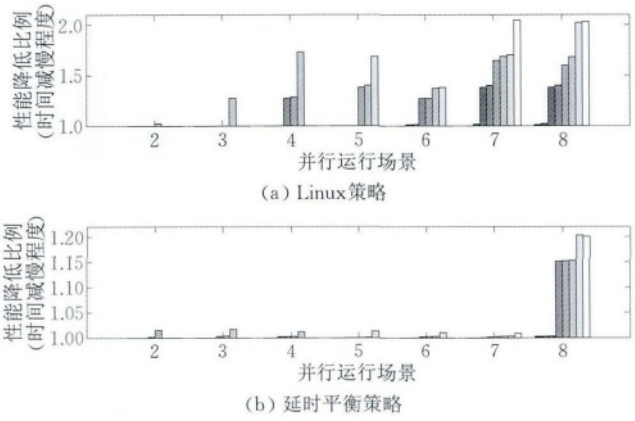


图 17 EP 在两种策略下的性能降低对比

图 18 中显示了应用 LU 在延时平衡策略和原 Linux 策略下并行运行时，每个进程的性能降低比例对比. 其中上子图是 LU 在原 Linux 策略下运行的实验结果，下子图是 LU 在延时平衡策略下运行的实验结果. 两副子图对比很明显，LU 应用在两种策略下运行时，每个并行场景下各进程的性能降低比例基本一致，随着进程数的增加，性能降低比例的差异越明显. 延时平衡策略仅将性能降低比例维持在 1.24 以下，略微优于 Linux 策略.

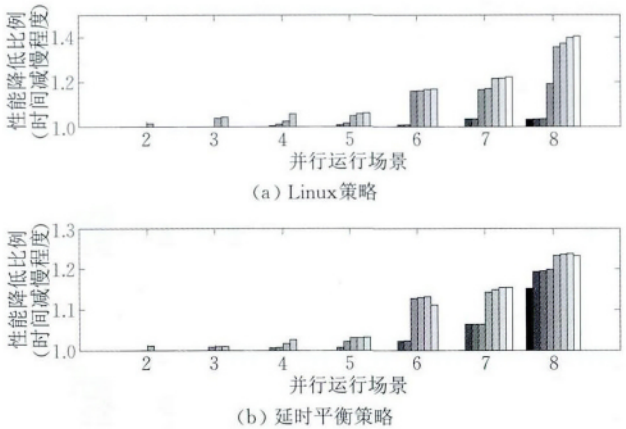


图 18 LU 在两种策略下的性能降低对比

SPEC CPU 测试集中的应用主要是偏向 CPU 计算的应用集，NPB 测试集中的应用包含了整型和浮点型运算，而 stream 测试集是经典的访存密集型应用. 综合以上分析可得出结论：访存延时平衡内存分配策略缓解了并行进程运行时各进程的性能降低比例，稳定了应用运行的整体性能.

4.1.3 同一应用多个进程并行执行时的不公平

图 19 显示的是 8 个应用在原 Linux 策略，延时

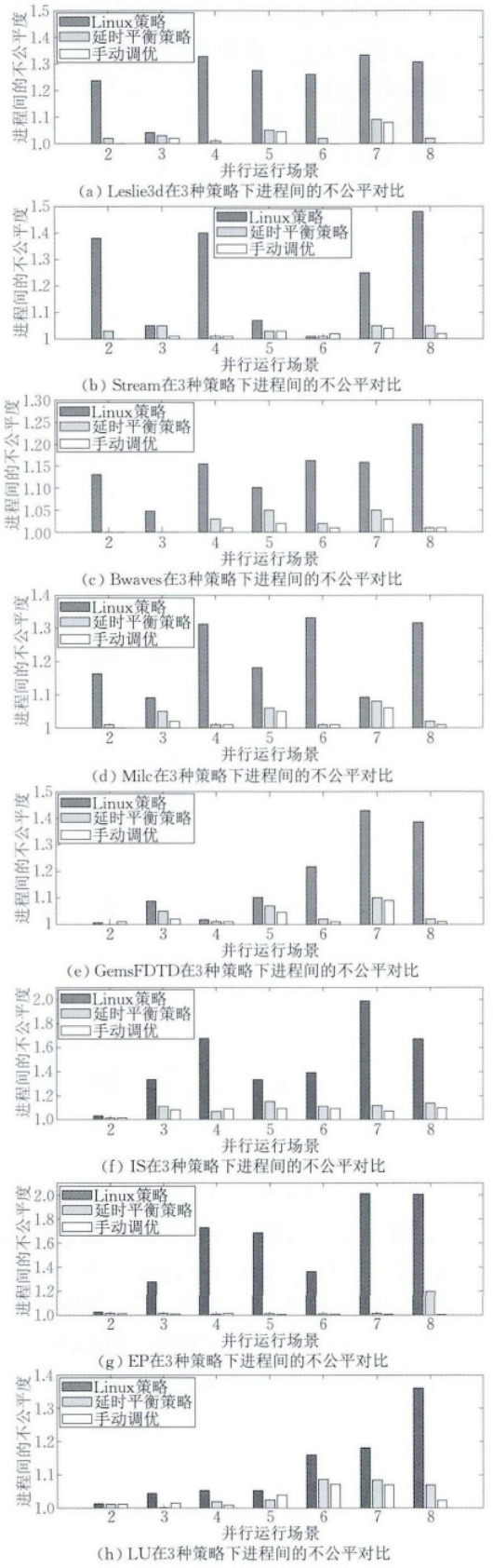


图 19 3 种策略下进程间的不公平对比

平衡策略以及手动调优的 3 种策略下分别运行时，同一应用的多个进程间访存延时的不公平.

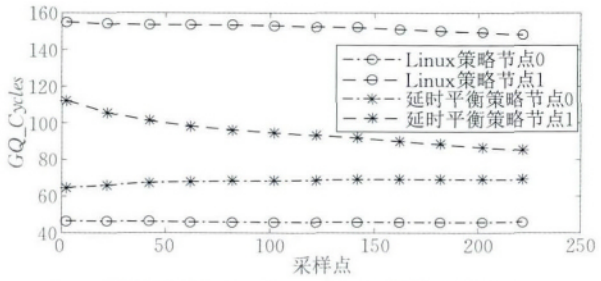
图 19(a)展示了 leslie3d 的多个进程运行不公平的实验对比. 延时平衡策略维持进程间的不公平在 1.1 以下，并且接近于手动调优结果. 与原 Linux 策略相比极大地降低了进程间的不公平，同时提高了整体的性能. 在 leslie3d 的实验对比中，延时平衡策略可以将原 Linux 策略带来的不公平至多降低了 24%，平均降低 17%. 其它子图给出的是剩余 4 个应用在这 3 种策略下运行时，进程间共享内存不公平的展示.

以上对比结果，表明延时平衡策略极大地降低了进程间的不公平性，并且与手动调优结果接近. 其中 stream 在场景 8 的对比结果显示，延时平衡策略将原 Linux 策略带来的不公平性降低了 35%.

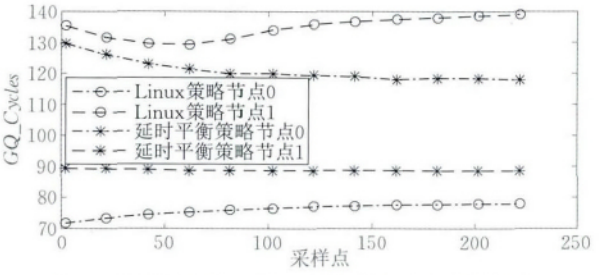
4.2　并行运行时各内存节点访存延时的对比分析

为了分析访存延时平衡策略对各内存节点访存延时的影响，本节针对应用在原 Linux 策略和延时平衡策略下，并行运行时各内存节点所具有的访存延时进行了对比分析. 其中访存延时通过访存请求在 GQ 队列中的平均滞留时间 *GQ_cycles* 来衡量. 选择 LU 在场景 6 和场景 7 以及 EP 应用在场景 7 运行的实验数据进行分析.

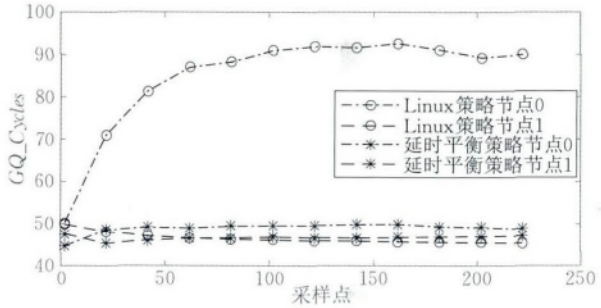
图 20 中显示了应用在两种策略下运行时的各内存节点访存延时随着采样的状态图，其中图 20(a)是两种策略下 LU 在并行场景 6 中运行时两内存节点访存延时的状态；图 20(b)是两种策略下 LU 在并行场景 7 中运行时两内存节点访存延时的状态，采样周期为 0.5 ms 一次. 对比图 20(a)和(b)可以发现，在延时平衡策略下运行时两内存节点的访存延时随着运行时间逐渐靠近，尤其是在场景 6 进程数为偶数时，每个内存节点提供访存服务的进程数是一致的，因此访存延时也趋于一致. 在场景 7 进程数为奇数时，会出现某个内存节点的访存延时高于另一个，这是因为该内存节点提供的访存服务的进程数多一个，这是由算法思想决定的. 但是在原 Linux 策略下，无论哪个场景两内存节点的访存延时都相差甚远，是由 Linux 策略在分配内存时为同一进程分配的内存分散在多个节点引起的. 图 20(c)是两种策略下 EP 在并行场景 7 中运行时两内存节点访存延时的状态. 两策略的对比结果明显，EP 在延时平衡策略下运行时，两内存节点的访存延时基本一致. 而在 Linux 策略下运行时，其中一个节点访存延时是另一节点的两倍. 对比结果从微观角度验证了



(a) LU在两种策略下运行6个进程时节点间的访存延时



(b) LU在两种策略下运行7个进程时节点间的访存延时



(c) EP在两种策略下运行7个进程时节点间的访存延时

图 20　两种策略下内存节点间的访存延时

延时平衡策略对各内存节点间访存延时平衡的保证. 但是关于远端内存访问的比例与 *GQ_cycles*，不公平度以及应用性能下降之间的内在联系，是否有明确的定量关系，是我们后续的研究点之一.

4.3　混合实验

为了验证访存延时平衡内存分配策略的有效性，本文还设计了多个应用并行执行的混合实验. 文中分别设置了混合 2 种、3 种和 4 种访存密集型应用同时运行的实验，实现效果都很明显. 在混合 2 个应用的实验中，延时平衡策略最大降低了原来不公平的 32%，平均 18%. 在混合 3 个应用的实验中，延时平衡策略最大降低了原来不公平的 24%. 下面详细给出混合运行 4 个应用的实验配置和实验结果.

混合 4 个应用的实验有两个场景，场景 1 是每个应用分别运行 1 个进程，总共 4 个进程并行执行，场景 2 是每个应用运行 2 个进程，共 8 个.

4.3.1　应用的整体性能

图 21 中展示 4 个应用：stream，leslie3d，milc 和

GemsFDTD 同时运行在两个并行场景下的运行时间图. 在场景 1 中,原 Linux 策略下的性能与延时平衡策略相当,这是因为此时进程总数只是系统满负载的一半,造成的数据阻塞还不明显. 但在场景 2 中,结果明显很多,平均运行时间减少 50 s,性能提高 5%.

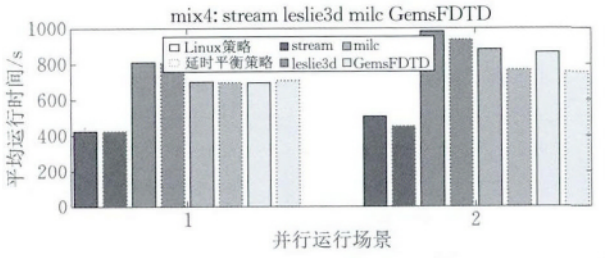


图 21 两种策略下 4 个应用混合运行时性能对比

4.3.2 多应用多个进程并行运行时的性能降低

在前文的不公平度计算公式适用于在单应用多进程并行运行模式下的性能降低比例和不公平度的计算,对于多应用多进程并行运行模式通过下边的公式计算得到

$$IS_{(i,j)}=\frac{T_{(i,j)\text{co-run}}}{T_{\text{ialone}}},$$

$$Unfairness_{\text{mix}}=\frac{\text{MAX}\{IS_{(0,0)},\cdots,IS_{(i,j)},\cdots,IS_{(M-1,N-1)}\}}{\text{MIN}\{IS_{(0,0)},\cdots,IS_{(i,j)},\cdots,IS_{(M-1,N-1)}\}} \tag{2}$$

其中:i 表示同一场景中的某个应用,j 表示该场景中某个进程,(i,j)表示该场景中属于 i 应用的进程 j;M 表示该场景中的应用总数;N 表示该场景中每一应用可运行的进程个数. 其中,$IS_{(i,j)}$ 表示进程 i 的性能降低比例,其中 $T_{(i,j)\text{co-run}}$ 表示在并行运行时应用 i 的进程 j 的运行时间,T_{ialone} 表示应用 i 单独运行一个进程的运行时间,通过它们的比值得到应用 i 的进程 j 在多应用混合并行运行时的性能降低比例. 通过并行运行时性能相对最差进程与相对最好进程的性能降低比例的比值,得到该次并行运行时各进程运行的不公平度.

图 22 显示了在两种策略下 4 种应用混合运行实验的性能降低对比. 图 22(a)它们在原 Linux 策略下运行的实验结果,图 22(b)是在延时平衡策略下运行的实验结果. 两副子图实验结果对比很明显,在延时平衡策略下相同场景多个进程的性能降低基本一致且都低于 1.2,而在原 Linux 策略它们参差不齐,尤其是在场景 2 中,它们最小低于 1.1,最大高于 1.6. 同时对比图 21 应用在场景 2 下的性能情况和图 22 进程间的不公平,在场景 2 时,延时平衡策略有效地降低了进程间的不公平同时性能稳定并得以提升.

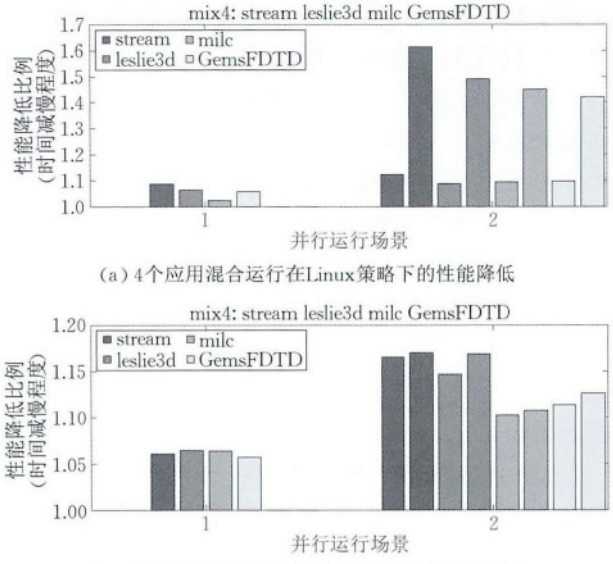


图 22 两种策略下 4 个应用混合运行时性能降低对比

4.3.3 多应用多个进程并行运行时的不公平

图 23 显示了 4 个应用在原 Linux 策略,延时平衡策略以及手动调优的 3 种策略下混合运行时进程间的不公平. 可以看出在 8 进程并行时实验结果对比很明显. 显著降低了原来不公平的 28%,平均值 17%,并且接近手动调优.

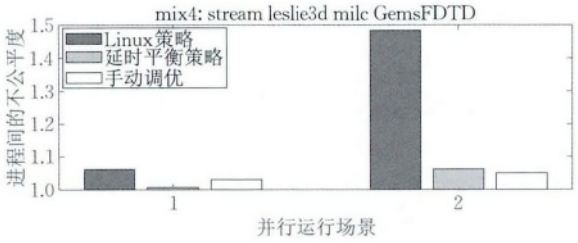


图 23 3 种策略下 4 个应用混合运行时的不公平对比

总之,延时平衡策略能够有效地降低进程间共享内存资源的不公平性,进而稳定应用的整体性能. 实验效果显示最大降低了原来不公平的 34%,平均降低了 16%. 单一应用和混合应用实验都充分证明了延时平衡分配策略的有效性和稳定性.

4.4 整体开销

整体开销从理论和实验两个角度分析:(1)本文策略实现的主要开销来自于两个方面:在内核中判断节点间访存延时的平衡和硬件性能工具读取计数器. 首先,判断平衡的快排序算法,由于 NUMA 节点的有限性,快排序算法的开销可以忽略. 其次,读取计数器仅使用极少的系统资源,因此读取事件计数器的开销也可以忽略;(2)实验数据表明:在进程的内存分配阶段,该策略与原 Linux 策略所耗时间是一致的;在后期执行过程中,原 Linux 策略所引

起的进程调度频繁，后期执行时间比本文提出策略长. 综上两点，访存延时平衡内存分配策略的开销可以忽略.

实验所用应用是多次配置的 stream 测试集，使其中数组占用的实际大小分别为 1GB、2GB、4GB 和 8GB，运行过程中实际分配内存为 1.5GB、3.1GB、5.8GB 和 11.3GB. Linux 策略和延时平衡策略为以上 4 个配置分配内存所占用的时间对比如图 24 所示. 延时平衡策略在内存分配阶段的耗时与 Linux 策略基本相同(±0.02s).

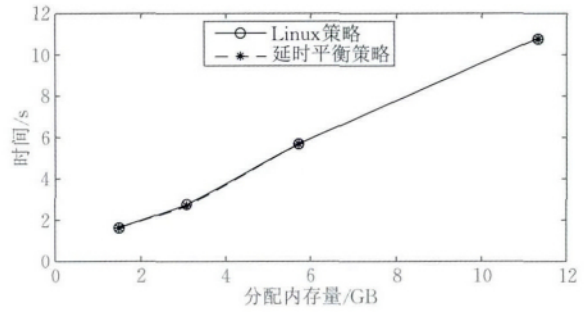


图 24　内存分配阶段两策略消耗时间对比

5　相关工作

缓解因访存争用导致的应用间内存抢占和整体性能下降的研究方法，侧重于从调度角度入手提高应用运行过程中访存的命中率，进而提升系统的整体性能，忽略了系统内存分配策略是影响应用运行时行为的直接因素，内存分配的结果直接影响应用的后期运行行为和系统调度.

已有针对 NUMA 架构内存管理的研究，主要集中在进程内存分配阶段结束后的管理和调度，文献[10]首次设计了追踪 NUMA 中进程与其运行时访问路径的 MemProf 工具，借助该工具可以估计进程的访存模式，可在此基础上实现进程调度. 文献[11]针对多核处理器设计实现了内存带宽预留系统 MEMGuard，在该系统中预留的内存带宽可以暂时隔离应用间的访存性能. 文献[12]设计实现了测试 NUMA 架构内存带宽的测试程序，该测试程序能够反映 NUMA 架构的内存结构特点. SchedNUMA 实现了应用进程数据放置在本地内存节点，充分利用了 NUMA 多核架构的本地访存低延时的特点①. Auto-NUMA-Balancing 实现了线程调度与内存资源迁移的结合②. 以上研究通过调度方式实现了进/线程与其内存的动态映射，而通过内存分配实现内存管理的研究很少. 文献[13]设计了针对 NUMA 架构的内存条分配算法，其设计思想为最大化分配本地内存节点上的内存条，避免使用交换区等内存资源，提高 RAM 资源的利用率[13].

6　总结与展望

多核系统内并行运行多个应用共享内存时的有效性和公平性一直是研究的焦点，在 NUMA 架构引入多内存节点之后，这一问题的复杂程度又显著增加. 针对这个问题，本文提出了一个节点访存延时指标，并在该指标的基础上提出节点间的访存延时平衡内存分配策略. 该策略通过感知节点间访存延时的平衡和选择访存延时最低的内存节点分配内存使得多个应用可以高效公平地共享内存资源. 该策略的实现是针对特定平台的，因为延时获取的方法与底层硬件紧密相关，但是策略本身是可以通用于 NUMA 架构的，针对不同的平台延时感知模块实现不同，但是内核策略是通用的. 在 Linux 默认的内存分配策略的外层加入访存延时平衡感知实现了该策略，并且在 Linux 系统中对比访存延时平衡分配策略与原 Linux 内存分配策略的性能：综合单应用多进程和多应用多进程的并行实验，实验结果表明访存延时平衡的内存分配策略降低了原 Linux 内存分配策略引起的进程间共享内存资源不公平的 16%，最大降低了 34%. 该策略实现对 Linux 内核改动很小，仅在内存分配部分做了改进，在未来的研究工作中会考虑结合基于访存延时平衡的调度策略.

参　考　文　献

[1] Ming L, Tao L. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads //Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA). Piscataway, USA, 2014: 325-336

[2] Pusukuri K K, et al. ADAPT: A framework for coscheduling multithreaded programs. ACM Transactions on Architecture and Code Optimization, 2013: 9(4), Article 45

[3] Zhuravlev S, et al. Addressing shared resource contention in multicore processors via scheduling//Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

① Sched NUMA. Linux Kernel Mailing List, Sched NUMA Rewrite. https://lkml.org/lkml/2012/5/9/312/, May 2012

② RedHat. Auto-NUMA-Balancing. https://access.redhat.com/docu mentation/en-US/Red_Hat_Enterprise_Linux/7/html/Virtualization_Tuning_and_Optimization_Guide/sect-Virtualization_Tuning_Optimization_Guide-NUMA-Auto_NUMA_Balancing.html

Pittsburgh, USA, 2010: 129-142

[4] Aravind A, Manickam V. A flexible simulation framework for multicore schedulers: Work in progress paper//Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, 2013: 355-360

[5] Jia R, et al. Optimizing virtual machine scheduling in NUMA multicore systems//Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA). Shenzhen, China, 2013: 306-317

[6] Jia R, Zhou X. Towards fair and efficient SMP virtual machine scheduling//Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). Orlando, USA, 2014: 273-286

[7] Ebrahimi E, et al. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems//Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Pittsburgh, USA, 2010: 335-346

[8] Henning J L. SPEC CPU2006 memory footprint. ACM SIGARCH Computer Architecture News, 2007, 35(1): 84-89

[9] Dashti M, et al. Traffic management: A holistic approach to memory placement on NUMA systems//Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Houston, Texas, USA, 2013: 381-394

[10] Renaud L, Baptiste L, Vivien Q. MemProf: A memory profiler for NUMA multicore systems//Proceedings of the 2012 USENIX Annual Technical Conference(USENIX ATC 12). Boston, USA, 2012: 53-64

[11] Caccamo M, Pellizzoni R, Sha L, et al. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms//Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). Philadelphia, USA, 2013: 55-64

[12] Molka D, Hackenberg D, Schone R, et al. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system//Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques. Raleigh, USA, 2009: 261-270

[13] Farina M, Zoni D, Fornaciari W. A control-inspired iterative algorithm for memory management in NUMA multicores//Proceedings of the 19th World Congress of the International Federation of Automatic Control (IFAC-2014). Cape Town, South Africa, 2014: 24-29

LI Hui-Juan, born in 1990, M. S. Her main research interest is parallel computing.

LUAN Zhong-Zhi, born in 1971, Ph. D., associate professor. His research interests include tributed computing, high performance computing and parallel computing.

WANG Hui, born in 1987, Ph. D. His research interests include computer architecture and computer systems.

YANG Hai-Long, born in 1985, Ph. D., lecturer. His research interests include runtime system, performance modeling, QoS and energy-efficient computer architecture.

QIAN De-Pei, born in 1952, M. S., professor, Ph. D. supervisor. His research interests include high performance computing and computer architecture.

Background

Non-uniform memory access (NUMA) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users.

Modern data centers and the scientific computing clusters widely adapt the NUMA architecture due to its low latency of local memory accessing, which is achieved by partitioning the whole memory into multiple nodes and each memory node is connected to a processors' memory controller. To achieve a low memory access latency, the default Linux memory policy chooses to allocate the physical memory pages from the memory node where the process is running. Unfortunately, this allocation policy may potentially result in the unfairness among the processes, which leads to the performance variation. To solve this problem, this paper designs a memory allocation policy assuring the balance of access latency among memory nodes in NUMA architecture, and implements it in the Linux kernel. The real system based evaluation results show that our approach can apparently reduce the unfairness among applications while keeping a stable performance, compared to Linux's default memory allocation algorithm.

This work is supported by the High-Tech Research and Development Program (863 Program) of China under Grant No. 2012AA01A302 and the National Natural Science Foundation of China under Grant Nos. 61133004, 61361126011, 61502019 and 91530324.