·体系结构与软件技术· 文章编号：1000—3428(2013)04—0058—04 文献标识码：A 中图分类号：TP311

基于缓存竞争优化的 Linux 进程调度策略

夏 厦，李 俊
(中国科学技术大学自动化系，合肥 230027)

摘 要：分析 Linux 经典内核版本 2.6.22 的进程调度算法，利用性能监测单元的监测信息，给出 3 个性能指标 CMR、CRR、OCIP 对进程的缓存竞争性强弱进行刻画，以此为依据，采用轮询算法优化 Linux 下的进程调度顺序，尽量避免在 CPU 上同时运行多个缓存竞争力强的进程，减小系统因缓存竞争产生的性能损耗。在 benchmark 上的测试结果表明，该方法能够提升系统在中、高负载下运行时的性能，在高负载下运行时的性能提升比例可达 6%左右。

关键词：Linux 内核；片上多处理器；进程调度；性能监测单元；进程行为；缓存竞争

Linux Process Scheduling Strategy Based on Cache Contention Optimization

XIA Sha, LI Jun
(Department of Automation, University of Science and Technology of China, Hefei 230027, China)

【Abstract】According to the analysis of the process scheduling algorithm of Linux kernel 2.6.22, through the information getting from Performance Monitor Unit(PMU), this paper puts forward three performance indexes CMR, CRR and OCIP to describe the process behavior of cache contention. On the basis of this, it uses polling algorithm to optimize the process scheduling in order to reduce the cache contention of Last Level Cache(LLC). Benchmark test results show that this algorithm can improve about 6% performance when the system load is high.

【Key words】Linux kernel; Chip Multi-processor(CMP); process scheduling; Performance Monitor Unit(PMU); process behavior; cache contention
DOI: 10.3969/j.issn.1000-3428.2013.04.014

1 概述

Linux 是当今世界上最流行的开源操作系统，其具有多用户、多线程、实时性好等多方位的优势。随着时间的推移，Linux 经历了多个版本的更新，其性能也在日益完善，最新发布的 Linux 内核版本为 2.6.24。由于对该版本的研究还有待加强，以及考虑到开发和运行的效率，本文将针对 Linux 比较经典的、也是目前研究的最多的版本 Linux2.6.22 的内核进程调度策略进行分析，并提出在多核处理器环境下的改进策略。

Linux 中的进程主要可分为实时进程和普通进程 2 种，实时进程要求响应的速度快且可靠性高，因此要比普通进程优先得到调度，在 Linux 中用优先级 0~99 代表实时进程，其优先级在 setscheduler()中设定，一经设定就不再改变。实时进程的调度策略主要有时间片轮转调度法(SCHED_RR)和先进先出调度法(SCHED_FIFO)这 2 种。普通进程主要采用 SCHED_NORMAL 策略，其优先级通过动态计算得到，优先级 100~139 代表普通进程[1]。

在 Linux 2.6.22 中，调度器为每一个 CPU 维护了 2 个进程队列数组，active 数组和 expire 数组，统称为就绪队列。其中 active 数组由时间片(time_slice)尚未用完的进程组成，而 expired 队列中存放着已用完时间片的进程。具有相同优先级 i 的进程都被插入到头指针为 queue[i]的链表中。由于系统共有 140 个不同优先级进程，因此 2 个数组大小都是 140。

当需要进行进程调度时，调度器可以通过 active 数组对应的 bitmap，选出 active 数组中优先级最高的队列中的第 1 个进程，作为候选进程 next，这种算法复杂度为 $O(1)$，因此，该调度器又称为 $O(1)$调度器。

Linux2.6 通过 $O(1)$调度器，给每个 CPU 设置了单独的

基金项目：中央高校基本科研业务费专项基金资助项目(Wk2100100006)
作者简介：夏 厦(1988－)，男，硕士研究生，主研方向：网络传播与控制，操作系统；李 俊，副教授、博士
收稿日期：2012-04-28 **修回日期：**2012-07-05 **E-mail：**xiasha@mail.ustc.edu.cn

运行队列，尽量避免进程在不同核上的频繁迁移，有效平衡各个 CPU 的负载，同时兼顾了 Cache 的连续性，因此能较好的支持多核系统。但是对于 CMP 结构而言，Linux 还有所不足，例如在文献[2]中提到的 Linux 内核不能够识别有关联的进程，从而将它们分配到一个核上，提高 Cache 的命中率。

如何提高 Cache 的命中率，一直是研究的热点。文献[3]从 Cache 替换算法角度分析，提出了优化 Cache 命中率的方法；文献[4]中提出 3 种预测 CMP 上线程间竞争的模型，但这些模型都需要离线统计信息，难以应用于实时操作系统中。本文针对 CMP 架构下进程之间的缓存竞争问题，通过在线获取的进程参数，对进程的行为特征进行分析，并以此为指导，优化 Linux 的进程调度顺序，降低进程之间因共享缓存的竞争带来的性能下降。

2　改进算法

2.1　缓存资源的竞争

CMP 架构是将多个相对简单的超标量处理器核集成到一个芯片上，从而提高吞吐量，典型代表是 Intel Core 微架构处理器，每个核都拥有独立的 L1 Cache，并且共享 2 MB 或 4 MB 的 L2 Cache，图 1 是典型的 Intel Core 微处理器的结构示意图。

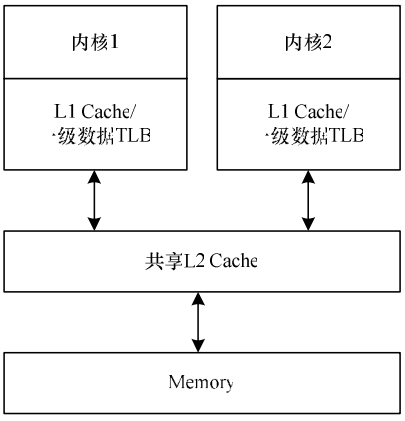

图 1　Intel Core 微处理器结构

当核 1 和核 2 上同时有进程运行时，就会向共享缓存中读/写入数据，从而形成对 Cache 的竞争。文献[5-6]详细地分析了共享 Cache 竞争对于系统性能的影响。根据进程共享 Cache 资源的占用程度，可以将进程划分为 Cache 竞争性强、中、弱这 3 类。当核 1 和核 2 同时运行 Cache 竞争性强的进程时，将会导致 L2 Cache Miss 数急剧上升，影响 CPU 的性能。

2.2　进程特性的刻画

2.2.1　事件监控单元

在现代处理器中，一般都会在芯片上集成一种硬件事件的监测单元(Performance Monitor Unit, PMU)，能对底层的硬件的各种指标进行监测[7]，表 1 为性能计数器可以监测的事件(部分)。

表 1　PMU 单元功能

周期	时钟周期
Load instructions	装载内存指令数
Store instructions	写入内存指令数
L1 misses	L1 Cache 缺失数
L2 misses	L2 Cache 缺失数
L2 cache reference	L2 Cache 访问数
Total cache misses	内部 Cache 的未命中数
Instructions	已完成的指令数

2.2.2　进程特征的选取

利用 PMU 单元读取的数据，可以对进程的一些行为特征做出判断，例如文献[8]利用 PMU 计数单元获取的数据，提出了对于进程发热量的刻画标准。采用类似思想，可以利用 PMU 单元的在线统计功能，分析进程对共享 Cache 资源竞争性的强弱。下面将从理论上提出 3 个性能指标，对进程进行刻画。

指标 1 CMR(L2 Cache Miss Rate)

$$C_{\mathrm{CMR}}=\frac{\text{L2 Cache Misses}}{Instructions} \tag{1}$$

当处理器访问数据的时候，会首先从 Cache 中寻求数据，如果 L2 Cache 中没有则从内存中寻找。L2 Cache Miss 越多，说明进程将从内存读取更多的数据，替换 Cache 中原有的数据(现代 Cache 数据替换常见的是采用 LRU 算法)，从而有可能污染另一个核上进程在 Cache 的数据。CMR 越大，表示对共享 Cache 竞争性越强。

指标 2 CRR(L2 Cache Reference Rate)

$$C_{\mathrm{CRR}}=\frac{\text{L2 Cache Reference}}{Instructions} \tag{2}$$

CRR 反映了一个进程对 L2 Cache 访问的频繁程度。CRR 越高，表示进程对 L2 Cache 的访问越频繁，从而消耗更多的 L2 Cache 资源，因此 CRR 越大，该进程的共享 Cache 竞争性越强。

指标 3 OCIP(Off-chip Instruction Proportion)

$$O_{\mathrm{OCIP}}=\frac{I_{\mathrm{load}}+I_{\mathrm{store}}}{Instructions} \tag{3}$$

其中，I_{load}+I_{store}表示装载和写入内存指令数之和。OCIP 反映缓存与内存交互的频繁程度。I_{load}越大，表示由内存写入缓存的数据越多，I_{store}越大，表示由 CPU 写回缓存的数据越多，从而占用更多的共享缓存空间，因此，OCIP 越大，表示该进程越倾向于 Cache 竞争性强的进程。

2.2.3　实验验证

为证实上述理论，采用 SPEC CPU2006 基准测试程序集[9]，测试各程序对于 L2 Cache 竞争性的强弱与上述 3 个指标之间的关系。

测试环境：ubuntu7.10、Intel Core2 Duo E4600 CPU(主频 2.4 GHz，L1 cache 256 KB，L2 cache 2 MB)、DDR2 1 GB 内存、Linux 内核版本 2.6.22。

实验方法如下：

(1)在核 1 上运行基准测试程序 mcf，测试它的运行时

间。之所以选择 mcf 作为基准测试程序，是因为 mcf 运行速度与 L2 cache 有很大的关系，能直观地反映另一个核上的测试程序对共享 cache 竞争性的强弱。

(2)在核 2 上分别运行 bzip、gamess 等 15 个测试程序，同时测试核 1 上 mcf 的运行时间。通过核 1 的性能下降程度(以 mcf 运行时间为标准)，作为度量 15 个测试程序的缓存竞争性强弱的标准。

(3)通过 PMU 监测单元信息，统计平均每执行 100 万条指令 L2 cache misses、L2 cache reference、$I_{\text{load}}+I_{\text{store}}$ 的值。实验结果如图 2 所示。

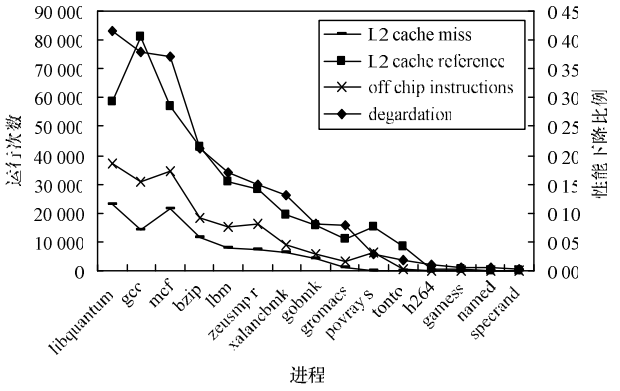

图 2 3 个进程特征值对性能的影响

图中左边的主坐标轴表示 100 万条指令中 L2 cache miss、L2 cache reference、off chip instructions 的数量，右边的副坐标轴表示测试进程 mcf 的性能下降幅度。

2.3 目标函数的提出和实现

从图 2 可以看出，以上 3 个进程特征与缓存竞争性的强弱有明显的相关性，为了能更准确地刻画进程对共享缓存的影响，以测试进程 mcf 的性能下降程度作为目标函数：

$$E=\lambda_1\times C_{\text{CMR}}+\lambda_2\times C_{\text{CRR}}+\lambda_3\times O_{\text{OPIP}} \tag{4}$$

为了确定 λ_1、λ_2、λ_3 的最佳取值，本文根据 15 个测试进程的结果，选取当 $\sum_{n=1}^{15}|\lambda_1\times CMR_n+\lambda_2\times CRR_n+\lambda_3\times OPIP_n-E_n|$ 取最小值时 λ_1、λ_2、λ_3 的值，同时考虑到运算的效率(尽量减少浮点数运算)，最终取 $\lambda_1=10$，$\lambda_2=2$，$\lambda_3=2$。图 3 反映了目标函数 E 的值与实际值的对比。

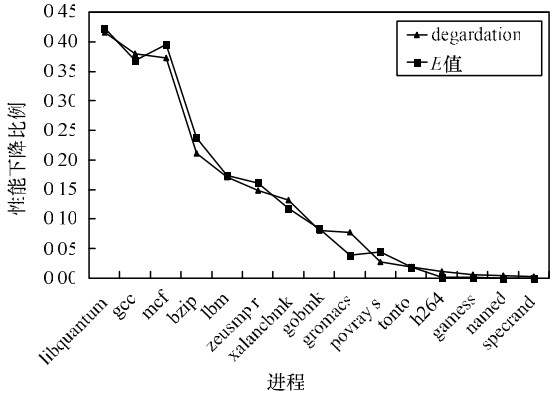

图 3 目标函数值与实际性能下降之间的关系

可以看到选取的进程目标函数与该进程对测试程序 mcf 造成的实际影响吻合得非常好，从而能准确地反映该进程对于共享缓存竞争性的强弱。

为了实现上述功能，在 schedule()函数中添加如下模块：

```
Void_sched schedule(void)
 {…
   if (likely(prev!=next))
  {   if(stat_cmr_switch==1)
          cmr_stat_end(prev);//统计 cmr
      if(stat_crr_switch==1)
          crr_stat_end(prev)//统计 crr
      if(stat_ocip_switch==1)
          ocip_stat_end(prev)//统计 ocip
prev->character=task_io_charactrer(cmr_count,crr_count,  ocip_
count)
//计算特征函数
prev=context_switch(rq,prev,next) //进程切换
barrier();
if(stat_cmr_switch==1)
          cmr_stat_start();//打开 cmr 统计
            …//打开 crr 统计
            …//打开 ocip 统计
finish_task_switch(this_rq(),prev);
}else
      spin_unlock_irq(&rq->lock)
…}
```

即在进程发生切换时对旧任务的任务特征值进行现场保存，计算任务特征函数，并在新任务开始时打开任务特征值的统计。

2.4 改进算法

通过上述分析，可以在线统计进程对共享 Cache 资源竞争性的强弱。2 个核上运行进程的 Cache 竞争性组合可能为(强，弱)、(中，弱)、(弱，弱)、(中，中)、(中，强)、(强，强)这 6 种情况，其中，前 4 种不会对处理器的性能产生大的影响，而后 2 种情况则应该尽量避免。界定特征值 $E\in(0,0.1)$ 的进程为 Cache 竞争性为“弱”；$E\in(0.1,0.2)$ 的进程 Cache 竞争性为“中”；$E>0.2$ 的进程 Cache 竞争性为“强”。基于以上思想，对 schedule()函数做出修改，增加 task_choose_from_queue()单元：

```
Void task_choose_from_queue()
       { sched_get_info(p,cpu_id)
//发生进程切换时，获取另一个核上当前运行的进程信息；
   if (p->character >0.2)
//另一个核上运行的进程的共享 cache 竞争性“强”；
next=search_min_character(queue[idx].next,task_t,run_list)
//搜寻当前最高优先级队列下特征函数值最小的进程，即
//else if(p->character <0.1)
```

```
//另一个核上运行的是 cache 竞争性弱的进程；
next=list_entry(queue[idx].next,task_t,run_list)
//遵循 Linux 原有的调度策略；
else  if(p->character > 0.1&& p->character < 0.2)
//cache 竞争性为“中”；
{if(next=search_fit_charactor(queue[idx].next,task_t,run_list))==NULL
//搜寻队列中第 1 个 cache 竞争性非“强”的进程；
next=search_min_character(queue[idx].next,task_t, run_list)
//如果没有则取队列中特征值最小的进程；
}
}
```

3　调度器改进后的测试

测试环境：ubuntu7.10、Intel Core2 Duo E4600CPU(主频 2.4 GHz，L1 cache 256 KB，L2 cache 2 MB)、DDR2 1 GB 内存、Linux 内核版本 2.6.22。

比较对象：Linux 内核 2.6.22、改进的 Linux 2.6.22。

测试程序：从 Mibench[10]和 SPEC CPU2006 中选取了 20 个基准测试程序(benchmark)，通过 sched_setaffinity()函数，可以将其分配到指定的核上运行(主要是防止 Linux 中的负载均衡函数将进程迁移到其他核上，对试验结果造成影响)。

分别使用改进后的 Linux 内核和原版对比，测试 CPU 在低、中、高负载下(低负载时每个核上运行 2 个测试程序，中负载时每个核上运行 4 个测试程序，高负载时每个核上运行 8 个测试程序)，每周期执行指令数值(Instructions Per Cycle, IPC)的变化，定义加速比 $speedup=\frac{\mathrm{IPC}_{改进}}{\mathrm{IPC}_{原}}-1$，得到结果如图 4 所示。

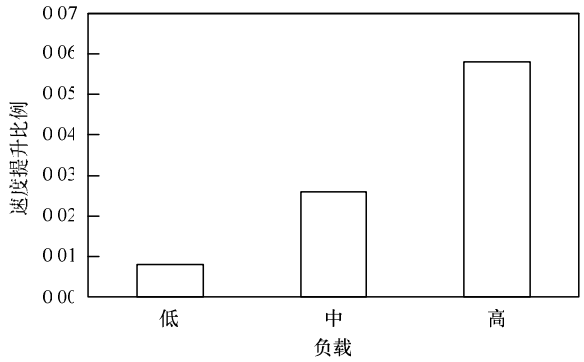

图 4　改进后的内核性能测试

从分析结果可以看出，本文的算法在低负载是提升效果并不明显，原因是相同优先级队列中可供选择的进程较少，进程的调度方式相较于原来的内核差异不大；在中、高负载时，尤其是在高负载下，性能提升较为明显。

4　结束语

本文提出一种利用 PMU 在线监测数据对进程的缓存竞争性强弱进行判定的方法，利用该方法，对 Linux 2.6 内核的进程调度顺序做出调整。实验结果表明，经过修改后的内核比原内核的性能有所提升。由于现在的内核普遍采用完全公平调度器(CFS)，因此今后主要的研究方向是如何将缓存竞争优化策略与完全公平调度器相结合，提升系统性能。

参考文献

[1] Bovet D P. 深入了解 Linux 内核[M]. 陈莉君, 张琼声, 张宏伟, 译. 北京: 中国电力出版社, 2008.

[2] 覃　中. 基于多核系统的线程调度[D]. 成都: 电子科技大学, 2009.

[3] 林晓敏, 桂　婷, 乔福明. 多核系统中共享 Cache 的冒泡替换算法[J]. 微电子学与计算机, 2011, 28(4): 118-121.

[4] Chandra D, Guo F, Kim S. Predicting Inter-thread Cache Contention on a Chip Multi-processor Architecture[C]// Proceedings of the 11th International Symposium on High-performance Computer Architecture. San Francisco, USA: [s. n.]. 2005: 340-351.

[5] Zhang Xiao, Dwarkadas S, Shen K. Towards Practical Page Coloring-based Multi-core Cache Management[C]// Proceedings of the 4th ACM European Conference on Computer Systems. [S. 1.]: ACM Press, 2009: 89-102.

[6] Jaleel A, Hasenplaugh W, Qureshi M, et al. Adaptive Insertion Policies for Managing Shared Caches on CMPs[C]//Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. Toronto, Canada: [s. n.], 2008: 208-219.

[7] 夏　亮. 温度感知的调度算法研究与实现[D]. 上海: 上海交通大学, 2009.

[8] 王中飞. 系统级处理器热量控制的研究和设计[D]. 合肥: 中国科学技术大学, 2011.

[9] Spec[EB/OL]. (2012-04-20). http://www.spec.org/cpu2006/.

[10] Mibench[EB/OL]. (2012-04-20). http://www.eecs.umich.edu/mibench/.

编辑　索书志