

# 集群动态负载平衡系统的性能评价

唐 丹 金 海 张 永 坤

(华中科技大学集群与网格计算湖北省重点实验室 武汉 430074)

**摘要** 该文使用随机 Petri 网对集群动态负载平衡系统建立了一个抽象模型。通过细化模型中的节点本地处理部分对 5 种动态负载平衡算法的性能进行了分析，并讨论了集群负载特性对动态负载平衡系统性能的影响。最后得出的主要结论有：(1) 动态负载平衡算法可以取得比静态负载平衡算法更好的性能；(2) 与传统的只考虑 CPU 就绪队列的负载平衡算法相比，考虑了各种 I/O 请求队列的负载平衡算法可以取得更好的性能；(3) 即使在极端的集群负载特性中，集群动态负载平衡算法仍然能取得比较理想的性能，因此实现即使是十分简单的集群动态负载平衡系统也是很有必要的。

**关键词** 集群；动态负载平衡；性能分析；随机 Petri 网

**中图法分类号** TP303

## Performance Evaluation of Dynamic Load Balancing System for Clusters

TANG Dan JIN Hai ZHANG Yong-Kun

(Cluster and Grid Computing Laboratory, Huazhong University of Science and Technology, Wuhan 430070)

**Abstract** A cluster system consists of a collection of computing nodes that share resources. Cluster systems utilize load balancing technology to share processing power and other resources, and thus to improve system performance, by transparently transferring load between nodes. In this paper, the Stochastic Petri Nets are employed to model the dynamic load balancing systems for clusters. The model we represent can avoid the disadvantages such as the lack of flexibility and the obvious difference from practical systems of previous work. We detailed the local process part of the model in order to analyze the performance of five dynamic load balancing schemes. We also discussed the impact of some workload characters of clusters on the performance of dynamic load balancing systems. It is found that: (1) Dynamic load balancing has the better system performance over that obtained with static load balancing; (2) Compared to the traditional load balancing algorithms that consider only the running queue, the load balancing algorithms which consider the I/O request queue can get better performance; (3) Even with extreme workload characters, the dynamic load balancing system can also get ideal performance, therefore implementing even a very simply dynamic load balancing system is also necessary.

**Keywords** cluster; dynamic load balancing; performance evaluation; stochastic Petri net

## 1 引言

负载平衡是集群系统中的重要技术，它通过在

由高速网络连接起来的各个节点间平衡系统负载来提高集群系统的性能。已有的研究表明，在集群系统中采用负载平衡系统可以显著地提高集群系统的性能<sup>[1]</sup>。负载平衡的策略可以分为静态和动态两大类。

收稿日期：2003-08-05；修改稿收到日期：2004-03-11. 本课题得到国家“八六三”高技术研究发展计划项目基金(2002AA1Z2102)资助。  
唐 丹，男，1976 年生，硕士，研究方向为计算机体系结构、集群计算。金 海，男，1966 年生，博士，教授，博士生导师，主要研究方向为计算机体系结构、集群计算、网格计算、网络存储、网络安全。E-mail: hjin@hust.edu.cn。张 永 坤，男，1980 年生，硕士研究生，研究方向为计算机体系结构、集群计算。

静态负载平衡策略在各个节点上分配负载时只使用系统的静态信息,这种策略的好处是在数学上容易分析且易于实现,但这种策略不考虑系统节点的当前负载情况而进行负载平衡,其对集群系统利用率较低而性能较差。动态负载平衡系统根据集群的负载状态动态地做出负载分配的决定,相对来说能更大地提高系统的性能。因此,对集群动态负载平衡系统的性能评价在设计和实现集群动态负载平衡系统时显得尤为重要。

对集群动态负载平衡系统的性能评价已有很多研究工作<sup>[1~5]</sup>,这些工作多采用模型仿真(simulation)的方法<sup>[1~3]</sup>,使用特定的测试数据驱动所设计的模型,虽然有一定的代表性,但对性能的分析缺乏灵活性。也有一些方法基于数学分析<sup>[4,5]</sup>,但基本都采用排队系统理论进行模型描述,受限于排队系统理论对并行分布式系统的描述能力,他们的模型与实际系统相差较大,不能较为真实地描述集群动态负载平衡系统的实际行为特征。

本文通过应用随机 Petri 网理论<sup>[6,7]</sup>,首先给出一个集群动态负载平衡系统的抽象模型,然后通过细化模型中的节点本地处理部分,着重分析了几种负载平衡调度算法对集群动态负载平衡系统性能的影响。同时,也分析了集群负载的特征对动态负载平衡系统性能的影响。

本文应用的随机 Petri 网为随机回报网(stochastic reward net, SRN),是对随机 Petri 网的扩充,我们不对二者做出区分并都简称为 SPN。

本文剩下的部分组织如下,第 2 节给出本文描述的集群负载平衡系统模型及其 SPN 描述;第 3 节给出对本地节点处理的细化的 SPN 描述;第 4 节利用已给出的模型讨论几种不同的集群负载平衡策略的性能;第 5 节分析集群的负载特征对集群动态负载平衡系统性能的影响;第 6 节总结全文。

## 2 系统模型

本文所描述的集群系统由高速网络连接起来的  $n$  个节点和一个专用的文件服务器组成(图 1 所示)。作业可以在除文件服务器以外的任一节点上提交。

要将图 1 所示的集群动态负载平衡系统的所有

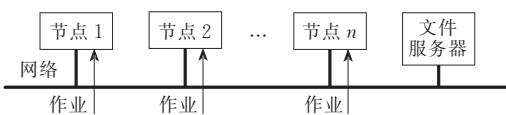


图 1 集群系统的结构模型

细节都描述清楚不太可能,因此我们根据要分析的重点做出适当的假设以简化模型的设计,然后在后面的分析过程中将其中的某些部分加以细化,以接近实际系统的行为特征。这些假设如下:

1. 所有节点为同构节点,即它们有完全相同的处理能力和使用完全相同的操作系统。

2. 使用专用的文件服务器。各个节点在文件系统上有一致的视图,因此各个节点对同一文件,读取所需要的时间完全一样。

3. 使用进程的远程执行作为负载平衡系统中的负载转移手段,使用拦截 execve 系统调用来实现进程的远程执行。

4. 本文的重点不在考察负载信息收集与传播部分,因此忽略负载信息收集和传播所需要的开销。当然也很容易将这部分加入到模型中来。

根据上面的假设,可以给出表示 2 个节点的集群动态负载平衡系统的 SPN 模型,如图 2 所示。

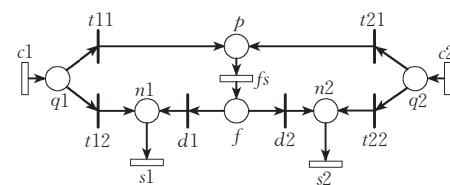


图 2 集群负载平衡系统 SPN 模型

对模型的描述如下(假设系统包含有  $n$  个节点):

变迁  $c_i$  代表在节点  $i$  上提交的作业。其到达为泊松过程,到达速率为  $\lambda$ ,可由该节点上 fork 系统调用的发生的速率表达。为防止系统状态空间的爆炸,系统中同时存在的最大作业数  $k$  必须加以限制,当  $k$  的值到达一个上限时,系统作业的到达过程中断。从实际计算结果来看,这个值设为 4 个以上即较为精确。这个限制可由  $c_i$  所关联的变迁可实施谓词(guard)函数来实现。图中由  $t_{i1}, t_{i2}$  组成的自由选择冲突表示由 fork 系统调用产生的进程是否调用 execve 系统调用改变其执行映像。如果调用了 execve 系统调用则会被负载平衡系统拦截经瞬时变迁  $t_{i1}$  进入负载平衡调度部分,否则经瞬时变迁  $t_{i2}$  直接进入由位置  $n_i$  表示的本地节点处理。二者的冲突选择结果由  $t_{i1}, t_{i2}$  所代表的随机开关的开关分布概率决定。这个冲突选择可以描述一般进程的行为特征,即一个进程由 fork 产生后可能继续共用父进程的执行映像和地址空间,也可能立即调用 execve 系统调用改变其执行映像,而居于二者之间的情况比较少见,这可以由 windows 操作系统中将类似

`fork` 和 `execve` 功能的函数整合在一起得到理解。采用拦截 `execve` 系统调用实现远程执行在保证进程迁移的效率下可以得到较好的负载平衡的调度粒度。

一旦进程调用了 `execve` 系统调用将会进入由位置  $p$  所代表的调度队列。在时间变迁  $fs$  中完成执行映像从文件服务器到本地节点的传送。我们假设执行映像的大小服从指数分布，因此在一定的网络传输速率下其传输时间亦服从指数分布。完成执行映像传送后的作业进入表示负载平衡调度判断的位置  $f$  中，判断的过程由  $n$  个瞬时变迁  $(d_1, d_2, d_3, \dots, d_n)$  代表的自由选择冲突表示。位置  $f$  瞬时保留到来的任务，根据  $n$  个瞬时变迁  $(d_1, d_2, d_3, \dots, d_n)$  联系的变迁可实施谓词和随机开关来决定将到来的作业放入哪个节点进行处理。一旦作业被调度到某个节点，则该作业在该节点上执行直到退出。这是使用远程执行和进程迁移作为负载转移手段的负载平衡系统的一个重要区别。

以上的 SPN 模型给出了对集群动态负载平衡系统的基本描述。这个模型对集群动态负载平衡系统的描述比较粗略。比如模型中仅用一个时间变迁表示了本地节点的处理，显然不能满足我们对集群动态负载平衡系统性能分析的要求。下一节我们对节点本地处理做出进一步细化。

### 3 节点本地处理细化

对本地节点处理的细化，主要在于对节点操作系统的描述。在文献[8]中给出了一个基于微内核的动态优先级通用操作系统的 SPN 模型，鉴于纯微内核操作系统应用较少，因此我们对该模型加以改造，得到一个宏内核的操作系统的 SPN 模型。

在宏内核的操作系统中，进程的执行分为内核态和用户态 2 种方式。这 2 种执行态的转换通过操作系统提供的系统服务完成。进程在用户态执行时通过请求系统调用服务而进入内核态，进程在内核态执行时可以提交 I/O 服务请求，I/O 服务完成后以中断形式通知操作系统并返回系统调用执行。由此得到表示宏内核操作系统的框图，如图 3 所示。

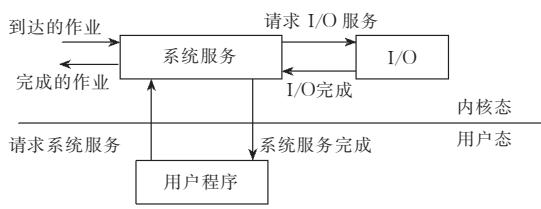


图 3 结点本地处理结构框图

在这个框图中，作业只能由系统调用部分进入系统，也只能从系统调用部分离开系统。系统调用中可以选择进行 I/O 操作，也可以选择进入用户态执行。I/O 操作和用户态执行完成后都进入系统调用部分。

根据这个框图，很容易得到如图 4 所示的 SPN 模型。

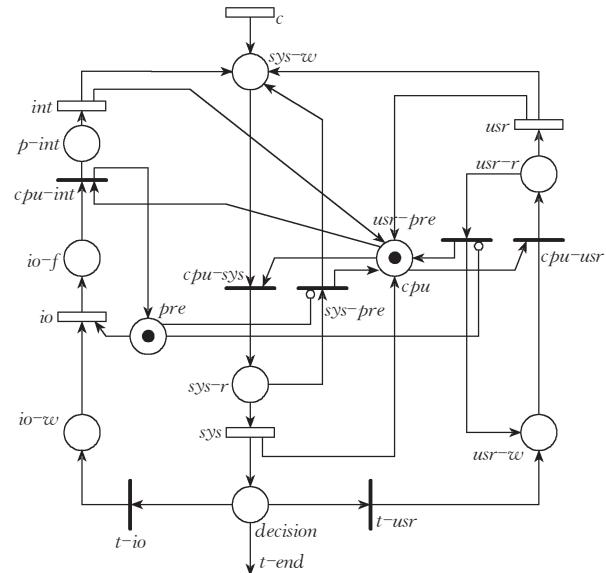


图 4 结点本地处理 SPN 模型

在这个模型中，到达节点的作业由时间变迁  $c$  表示，其到达过程服从泊松分布。位置  $cpu$  中的标记表示 CPU 是否空闲。所有运行状态的转换都需要 CPU 来参与。这个过程由瞬时变迁  $cpu-sys$ ,  $cpu-usr$ ,  $cpu-int$  表示。当一个作业被放置到节点上执行时，它首先进入系统服务中执行，时间变迁  $sys$  表示了这个处理过程。

当作业从系统服务中离开时，表示 CPU 的标记被放回到位置  $cpu$  中表示 CPU 空闲，在位置  $decision$  处，作业可以选择以下几种执行方式：

(1) 进入 I/O，由瞬时变迁  $t-io$  表示。由于 I/O 可以独立于 CPU 工作，因此 I/O 处理过程中不需要占用 CPU。当 I/O 完成后，以中断方式通知 CPU。此时无论 CPU 在哪种服务状态 ( $usr$  或者  $sys$ ) 占用都会被抢占，转入 I/O 的中断处理。这个过程由位置  $pre$  和其到瞬时变迁  $sys-pre$  和  $usr-pre$  的抑制弧表示。当 I/O 完成后，即时间变迁  $io$  实施时，位置  $pre$  为空，使得瞬时变迁  $usr-pre$  和  $sys-pre$  可以实施，这 2 个瞬时变迁实施的结果导致在位置  $usr-r$  或  $sys-r$  占有 CPU 并接受服务的作业被抢占。表示被抢占的作业的标记被移回到位置  $usr-w$  或

*sys* 中. 同时, 表示 CPU 的标记也被移回到位置 *cpu* 处, 表明 CPU 空闲. I/O 中断处理程序的执行需要占有 CPU, 由瞬时变迁 *cpu-int* 表示, 瞬时变迁 *cpu-int* 与 *cpu/usr*, *cpu/sys* 相比具较高的优先级, 因此当这三者均有机会占有 CPU 时, 会由 *cpu-int* 占有, 保证了操作系统中中断处理的最高优先级. 当中断处理完毕后(由时间变迁 *int* 表示), 服务完毕的作业返回位置 *sys-w*, 再次进入系统服务继续执行. 同时, 表示 CPU 的标记返回位置 *cpu*, 表示 CPU 空闲.

(2) 进入用户空间处理, 由瞬时变迁 *t/usr* 表示. 首先进入由位置 *usr-w* 表示的等待队列, 当得到 CPU 后(由瞬时变迁 *cpu/usr* 表示)进入表示运行态的位置 *usr-r*, 得到由时间变迁 *usr* 表示的服务. 在服务的过程中如果有 I/O 服务完成而发生中断, 则目前正在被服务的作业将被抢占, 由瞬时变迁 *usr-pre* 返回等待队列, 并将表示 CPU 的标记放入位置 *cpu* 中, 表示 CPU 空闲. 如果在服务过程中没有中断发生, 则该作业服务完毕后返回位置 *sys-w* 进入系统服务, 同时将表示 CPU 的标记放入位置 *cpu* 中.

(3) 离开这个系统, 因为这个作业执行完毕, 由瞬时变迁 *t-end* 表示.

表 1 为利用 SPNP 软件包<sup>[9]</sup>计算得到的本地节点处理的结果.

表 1 结点本地处理细化计算结果

PLACE	Pr[非空]	Av[tokens]
<i>sys-w</i>	2.637945937015e-02	2.733398652833e-02
<i>sys-r</i>	1.543815747079e-01	1.543815747079e-01
<i>decision</i>	0.000000000000e+00	0.000000000000e+00
<i>p-int</i>	2.223095795920e-02	2.223095795920e-02
<i>io-f</i>	4.756082522207e-04	4.756082522207e-04
<i>io-w</i>	7.41381182475e-01	2.788246443222e+00
<i>usr-r</i>	8.233683048509e-01	8.233683048509e-01
<i>usr-w</i>	9.999531972942e-01	2.701811721344e+01
<i>pre</i>	9.995243917478e-01	9.995243917478e-01
<i>cpu</i>	1.916248200897e-05	1.916248200897e-05
TRANSITION	Pr[已启动的]	Av[吞吐量]
<i>c</i>	1.235052602585e-01	2.470105205170e+00
<i>fs</i>	2.470105206779e-02	2.470105206779e+00
<i>sys</i>	1.543815747079e-01	1.235052597663e+02
<i>usr</i>	8.233683048509e-01	9.880419658210e+01
<i>io</i>	7.410319318558e-01	2.223095795567e+01
<i>int</i>	2.223095795920e-02	2.223095795920e+01

注:(1)各指数时间变迁实施速率:  $\mu_c = 20$ ,  $\mu_{fs} = 100$ ,  $\mu_{sys} = 800$ ,  $\mu_{usr} = 120$ ,  $\mu_{io} = 30$ ,  $\mu_{int} = 1000$  单位都是 tasks/s.

(2) *decision* 处随机开关的分布概率为:  $pro\_usr = 0.8$ ,  $pro\_io = 0.18$ ,  $pro\_end = 0.02$ .

(3) 最大作业数  $max\_task = 30$ .

从表 1 中的结果可以看出, *sys*, *io-f* 非空的概率和平均 token 数与 *io-w*, *usr-w* 相比较都非常小(分别在  $10^{-2}$  和  $10^{-4}$  量级), 这是因为系统服务和中断处理的服务速度很快, 作业无需排队即可得到处

理. 表明系统的主要等待队列为 CPU 的就绪队列和在各种 I/O 中的阻塞队列. 这 2 个队列对整个系统的吞吐量和平均响应时间有较大的影响, 因此在下面的动态负载平衡系统调度算法的分析过程中, 我们只考虑这 2 个系统中最主要的等待队列.

## 4 集群动态负载平衡调度策略性能分析

上面讨论了集群负载平衡系统和单个节点本地处理的模型. 将这 2 者结合起来可以得到细化本地节点处理后包含 2 个结点的负载平衡系统模型. 如图 5 所示.

这个模型中负载平衡调度算法, 可以由图中的  $n$  个瞬时变迁  $(d_1, d_2, d_3, \dots, d_n)$  (图 5 中为  $d_1, d_2$ ) 所联系的变迁可实施函数和随机开关来描述. 位置 *f* 瞬时保留由负载平衡系统拦截的作业, 根据瞬时变迁  $d_i$  所联系的随机开关和变迁可实施函数来决定将作业放入哪个队列来执行, 由此达到负载平衡的目的. 有关多服务器多队列系统调度的 SPN 描述见文献[10].

集群负载平衡系统负载向量的确定已有很多的研究, 文献[11, 12]的结论表明, CPU 就绪队列长度是比较理想的负载向量描述. 下面给出几种根据不同节点负载向量得到的负载平衡调度策略的描述. 我们用标记  $M(P)$  表示位置  $P$  的 token 个数, 用符号  $\mu_t$  表示指数时间变迁  $t$  的实施速率.

### (1) 随机调度策略(Random Scheduling, RS)

这个调度策略, 不根据调度时集群系统当时的负载信息做出调度决定, 而是随机等概率地将作业调度到每个节点上运行. 我们可以把这种调度策略看成是一种静态调度策略.

变迁  $d_i$  可实施谓词: 无,

变迁  $d_i$  的实施概率  $P_i$  为:  $P_i(M) = 1/n$ .

### (2) 最短 CPU 队列长度调度(Shortest CPU Queue Scheduling, SCQS)

这个调度策略将作业调度到 CPU 就绪队列长度最短的节点上运行. 如果有  $k$  个节点同时有最短的 CPU 就绪队列长度, 则以  $1/k$  的概率调度到这  $k$  个节点上执行.

变迁  $d_i$  可实施谓词:  $M(usr\_wi) = \min\{M(usr\_w1), M(usr\_w2), \dots, M(usr\_wn)\}$ ,

变迁  $d_i$  的实施概率  $P_i$  为  $P_i(M) = \frac{1}{\|SCQS(M)\|}$ , 若  $k \in SCQS(M)$ .

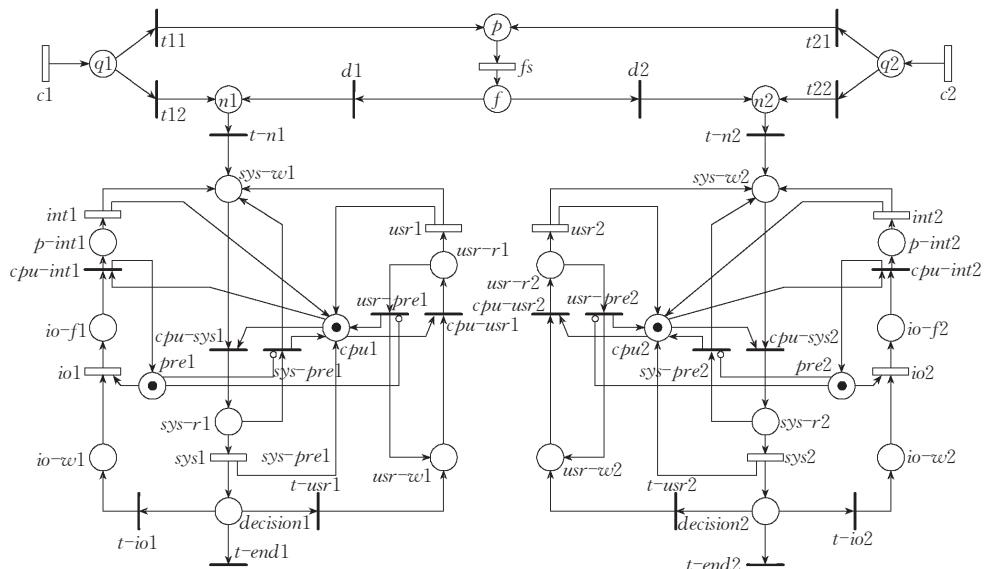


图 5 细化后包括 2 个结点的集群负载平衡系统 SPN 模型

其中  $SCQS(M) = \{k | M(usr\_wk) = \min(M(usr\_w1), M(usr\_w2), \dots, M(usr\_wn))\}$ .

(3) 最短预期 CPU 队列等待时间调度 (Shortest Expected CPU Queue Delay Scheduling, SECQDS)

这个调度策略将作业调度到 CPU 就绪队列等待时间最短的节点上运行,如果有  $k$  个节点同时有最短 CPU 就绪队列等待时间,则以  $1/k$  的概率调度到这  $k$  个节点上运行。

变迁  $di$  可实施谓词:  $\frac{M(usr\_wi)}{\mu\_usr} = \min\left(\frac{M(usr\_w1)}{\mu\_usr1}, \frac{M(usr\_w2)}{\mu\_usr2}, \dots, \frac{M(usr\_wn)}{\mu\_usrn}\right)$ ,

变迁  $di$  的实施概率  $Pi$  为:  $Pi(M) = \frac{1}{\|SECQDS(M)\|}$ , 若  $k \in SECQDS(M)$ ,

其中  $SECQDS(M) = \left\{ k \left| \frac{M(usr\_wk)}{\mu\_usrk} = \min\left(\frac{M(usr\_w1)}{\mu\_usr1}, \frac{M(usr\_w2)}{\mu\_usr2}, \dots, \frac{M(usr\_wn)}{\mu\_usrn}\right) \right. \right\}$ .

(4) 最短 CPU I/O 队列长度调度 (Shortest CPU & I/O Queue Scheduling, SCIOQS)

这个调度策略把作业调度到 CPU 就绪队列长度和 I/O 等待队列长度之和最小的节点上运行,如果有  $k$  个节点同时具有最小的 CPU 就绪队列长度和 I/O 等待队列长度之和,则以  $1/k$  的概率调度到这  $k$  个节点上运行。

变迁  $di$  可实施谓词:

$$\begin{aligned} M(usr\_wi) + M(io\_wi) = \\ \min \{ M(usr\_w1) + M(io\_w1), \\ M(usr\_w2) + M(io\_w2), \dots, \\ M(usr\_wn) + M(io\_wn) \}, \end{aligned}$$

变迁  $di$  的实施概率  $Pi$  为:

$$Pi(M) = \frac{1}{\|SCIOQS(M)\|}, \text{ 若 } k \in SCIOQS(M),$$

其中

$$\begin{aligned} SCIOQS(M) = \{ k | M(usr\_wk) + M(io\_wk) = \\ \min(M(usr\_w1) + M(io\_w1), \\ M(usr\_w2) + M(io\_w2), \dots, \\ M(usr\_wn) + M(io\_wn)) \}. \end{aligned}$$

(5) 最短预期 CPU I/O 队列等待时间调度 (Shortest Expect CPU I/O Queue Delay Scheduling, SECIOQDS)

这个调度算法把作业调度到预期 CPU 就绪队列等待时间和预期 I/O 队列等待时间最短的节点上运行,如果有  $k$  个节点同时具有这个最短等待时间,则以  $1/k$  的概率调度到这  $k$  个节点上运行。

变迁  $di$  可实施谓词:

$$\begin{aligned} \frac{M(usr\_wi)}{\mu\_usri} + \frac{M(io\_wi)}{\mu\_ioi} = \\ \min \left\{ \frac{M(usr\_w1)}{\mu\_usr1} + \frac{M(io\_w1)}{\mu\_io1}, \right. \\ \frac{M(usr\_w2)}{\mu\_usr2} + \frac{M(io\_w2)}{\mu\_io2}, \dots, \\ \left. \frac{M(usr\_wn)}{\mu\_usrn} + \frac{M(io\_wn)}{\mu\_io2} \right\}. \end{aligned}$$

变迁  $di$  的实施概率  $P_i$  为:  $P_i(M) =$

$$\frac{1}{\|SECIOQDS(M)\|}, \text{若 } k \in SECIOQDS(M),$$

其中

$$SECIOQDS(M) = \left\{ k \left| \frac{M(usr-wk)}{\mu_{usrk}} + \frac{M(io-wk)}{\mu_{iok}} = \min \left( \frac{M(usr-w1)}{\mu_{usr1}} + \frac{M(io-w1)}{\mu_{iol}}, \frac{M(usr-w2)}{\mu_{usr2}} + \frac{M(io-w2)}{\mu_{io2}}, \dots, \frac{M(usr-wn)}{\mu_{usrn}} + \frac{M(io-wn)}{\mu_{ion}} \right) \right. \right\}.$$

这 5 种调度策略分别考虑系统中不同的负载向量描述, 考察这 5 种调度策略可以了解实际系统中采用哪种负载向量描述方法会得到比较好的性能, 从而可以指导集群动态负载平衡系统的调度算法设计。

当节点数增加时, 系统的 SPN 模型的状态空间将以指数增长而急剧增大, 计算的规模将会导致问题不可求解, 因此我们只考虑 2 个节点时的 5 种调度策略的性能对比, 同时也给出了没有使用负载平衡算法的 2 个节点的性能作为参照。在这个数值比较中, 我们计算了在不同的输入速率情况下系统的吞吐量和平均响应时间。

表 2 给出了计算过程中使用的各个参数的值。

$\mu_{c1}, \mu_{c2}, \mu_{sys1}, \mu_{sys2}, \mu_{usr1}, \mu_{usr2}, \mu_{io1}, \mu_{io2}, \mu_{int1}, \mu_{int2}, \mu_{fs}$  分别表示 2 个节点上指数时间变迁  $c1, c2, sys1, sys2, usr1, usr2, io1, io2, int1, int2, fs$  的实施速率, 单位为 tasks/s.  $pro\_usr1, pro\_io1, pro\_end1, pro\_usr2, pro\_io2, pro\_end2$  表示每个节点在位置  $decision$  处的随机开关的分布概率.  $pro\_exec1, pro\_exec2$  分别代表每个节点上调用 execve 系统调用的进程占总进程个数的比例.  $max\_task$  表示整个系统中同时存在的作业最大数。

表 2 计算过程使用参数列表

参数	值	参数	值	参数	值
$\mu_{c1}$	?	$pro\_io1$	0.18	$\mu_{fs}$	100
$\mu_{c2}$	$1.2 * \mu_{c1}$	$pro\_io2$	0.23	$\mu_{io1}$	30
$pro\_usr1$	0.8	$\mu_{int2}$	1000	$\mu_{io2}$	20
$pro\_usr2$	0.75	$\mu_{usr1}$	80	$pro\_exec1$	0.95
$\mu_{int1}$	1000	$\mu_{usr2}$	120	$pro\_exec2$	0.95
$\mu_{sys1}$	800	$pro\_end1$	0.02	$max\_task$	8
$\mu_{sys2}$	800	$pro\_end2$	0.02		

图 6, 图 7 为 5 种调度策略及未使用负载平衡的集群系统吞吐量和平均响应时间的对比。从图 6 和图 7 中可以看出, 在任何输入的情况下, 无论使用

哪种调度算法与不使用负载平衡系统相比在吞吐量上都有较大提高, 同时系统平均响应时间也有减小, 即使是使用类似静态调度算法的随机调度策略性能也有所提高。其中最短 CPU I/O 队列长度调度 (SCIOQS) 策略具有最大的吞吐量和最小的平均响应时间, 最短预期 CPU I/O 队列等待时间调度 (SECIOQDS), 最短 CPU 队列长度调度 (SCQS), 最短预期 CPU 队列等待时间调度 (SECQDS) 次之, 随机调度策略 (RS) 最差。

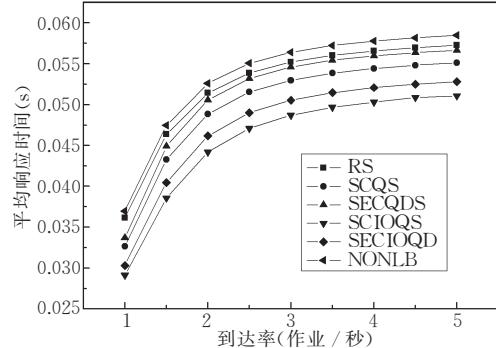


图 6 不同调度策略平均响应时间对比

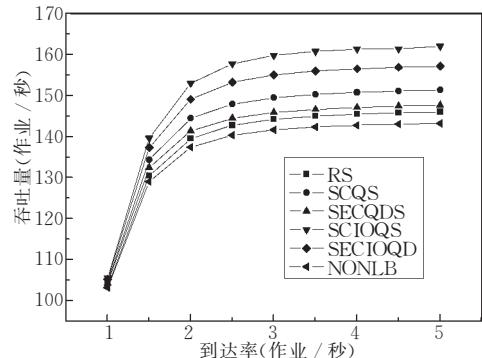


图 7 不同调度策略系统吞吐量对比

根据这个结果, 我们从 2 个不同的方面对这 5 个不同动态负载平衡调度策略进行性能分析。在分析过程中我们不考虑采用静态调度策略的随机调度算法和不采用负载平衡的集群系统。它们只作为性能对比的参考。

首先, 我们对这 5 种调度算法从是否考虑 I/O 队列长度作为节点负载向量这个方面来分析性能。显然最短 CPU 队列长度调度 (SCQS)、最短预期 CPU 队列等待时间调度 (SECQDS) 这 2 种调度策略没有考虑节点各种 I/O 请求队列。从图 6 和图 7 中可以看出, 这 2 种调度算法无论是吞吐量还是平均响应时间都比剩下的 3 种考虑了 I/O 队列的调度算法要差。这个现象可以充分说明, 只考虑 CPU 就绪队列的负载平衡调度算法, 虽然在很大程度上提

高了系统的性能,但与考虑了 I/O 等待队列的动态负载平衡调度算法相比,它们的性能较差。

产生这个结果的原因是,在节点中,CPU 就绪队列是系统中最重要的队列,这是因为 CPU 是系统中最重要的资源。从我们的计算结果(表 1)可以看到,CPU 就绪队列长度与其它队列相比是最长的。I/O 队列虽然没有 CPU 就绪队列长,但与 CPU 的处理速度相比,I/O 的处理速度很慢,导致作业花费在 I/O 上的时间相对较长,使得 I/O 处理的时间对节点的吞吐量和平均响应时间产生了较大的影响。为进一步了解考虑 I/O 和不考虑 I/O 的调度算法在不同的集群 I/O 负载情况下的性能差别,我们固定系统的输入速率在 3.0,在其它参数不发生变化时,变化每个节点 decision 处进入 I/O 的随机开关的概率,比较了最短 CPU 队列长度调度(SCQS)和最短 CPU I/O 队列长度调度(SCIOQS)这 2 种调度算法的平均响应时间,如图 8 所示。

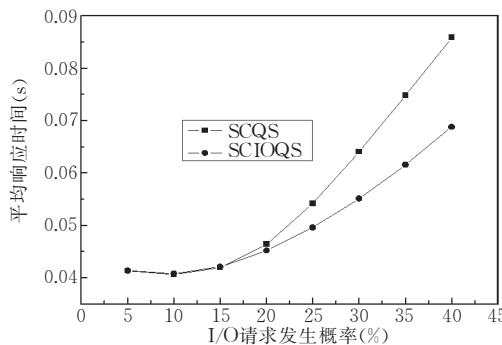


图 8 SCQS 与 SCIOQS 在不同 I/O 请求概率下平均响应时间对比

从图 8 中可以看出,在 I/O 请求概率小于 15% 时,这 2 种算法的平均响应时间基本一致,但当 I/O 的请求概率继续上升,从 20% 开始,二者平均响应时间差距明显增大。因此,在集群动态负载平衡算法中考虑 I/O 队列是非常有必要的。

更进一步,我们在上面讨论的基础上从是否考虑各个队列的预期等待时间这个方面来进行性能分析,显然最短预期 CPU 队列等待时间调度(SC-QDS)和最短预期 CPU I/O 队列等待时间调度(SCIOQDS)都考虑到了 CPU 就绪队列和 I/O 等待队列的等待时间,它们从节点中挑选等待时间最短的节点进行调度。这 2 种调度策略都使用了对系统的先验知识(priori knowledge)来进行调度,但从计算结果来看,与只考虑相应的队列长度的最短 CPU 队列长度调度(SCQS)和最短 CPU I/O 队列长度调度(SCIOQS)的策略相比,他们并没有得到较好的

性能,反而较差。这种现象可以解释为,作业在 CPU 就绪队列或(和)I/O 等待队列中等待时,这个等待时间并不能代表作业在系统中的所有运行时间,因此不能用这个等待时间来反应节点的负载情况。

## 5 负载执行特性对集群动态负载平衡系统的影响

大部分操作系统中,进程是系统执行的最小单位。对于负载转移的实现方法来说,负载转移的粒度越接近进程则负载转移发生的时机越多,从理论上来说能取得更好的性能。

本文描述的负载平衡系统采用进程的远程执行作为负载转移的手段,并且采用拦截 execve 系统调用实现。这种实现方法比采用拦截 shell 实现远程执行的负载转移的粒度要细,但负载转移的粒度仍达不到进程的粒度。这是因为,在很多应用中,比如很多网络应用程序对每个到来的连接 fork 一个进程处理这个连接的请求,但这些由 fork 产生的进程并不调用 execve 系统调用改变自己的执行映象,而是继续共用父进程的执行映象。因此,在这样的应用环境中,虽然产生了很多进程,而且这些进程对系统的负载产生了较大的影响,但是由于这些进程不调用 execve 系统调用,它们并不能被采用进程的远程执行作为负载转移手段的集群动态负载平衡系统所拦截。在图 2 所示的模型中,它们会直接进入本地节点处理,导致在这样的应用环境中发生负载转移的时机急剧减少。我们考察了一些这样的应用程序,统计得到的结果表明,在极端的情况下,调用了 execve 系统调用的进程的比例只占整个进程的 65% 左右。因此分析在这些应用环境下的采用拦截 execve 系统调用实现负载转移的集群动态负载平衡系统的性能是很有必要的。为此,我们计算了最短 CPU I/O 队列长度调度(SCIOQS)策略在调用了 execve 系统调用的进程在所有进程中的比例从 5%~95% 间变化时的性能变化情况。同时,我们以不采用负载平衡的集群系统和调用了 execve 系统调用的比例为 95% 时的最短 CPU I/O 队列长度调度(SCIOQS)算法的性能作为参照。其中使用的其它参数同表 2。计算的结果如图 9,图 10 所示。

如图所示的结果令人鼓舞,在调用了 execve 系统调用的进程的比例从 95% 降低到 60% 时,系统的性能只下降了约 10%,即使这个比例下降到 50% 整个系统性能的下降比例也是可以接受的,大约下降

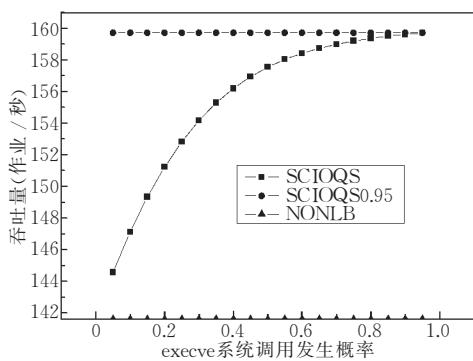


图 9 execve 发生比例对 SCIOQS 吞吐量的影响

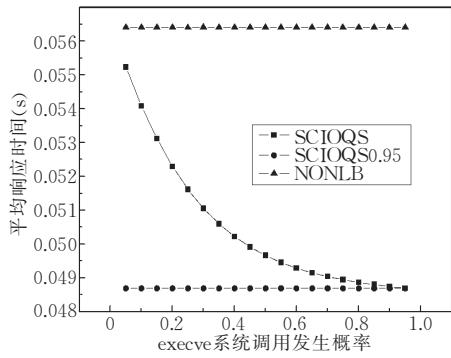


图 10 execve 系统调用发生概率对 SCIOQS 调度策略平均响应时间的影响

了约 15%。随后性能下降极为迅速。进一步考虑到在系统中调用了 execve 系统调用的进程一般比未调用 execve 系统调用而共用父进程的执行映象的进程的执行时间要长, 对节点负载的影响更大, 因此, 即使在一些极端的应用环境下采用进程的远程执行作为负载转移手段的集群动态负载平衡系统也可以得到比较理想的性能。因此, 实现较简单的(与采用进程迁移的负载平衡系统相比)采用进程的远程执行的集群动态负载平衡系统是非常有意义的。

## 6 结 论

本文利用随机 Petri 网理论给出了一个动态负载平衡系统的抽象模型, 通过细化其中的本地节点处理部分对集群动态负载平衡的调度策略和集群负载进程执行特征进行了性能分析, 得到一些对实际系统的设计起到指导作用的结论。这些结论为: (1)无论使用静态还是动态的负载平衡系统都可以提高集群系统的性能, 动态负载平衡系统会得到更好的性能。(2)在动态负载平衡算法中除了要考虑系统中最重要的等待队列——CPU 就绪队列外, 还要考虑各种 I/O 请求队列。(3)把节点的总的预期等待时

间作为节点的负载向量时, 不能简单地将节点的所有队列的预期等待时间之和当做节点的总的预期等待时间。(4)即使在一些极端的应用环境下, 采用通过拦截 execve 系统调用实现的进程远程执行的动态负载平衡系统仍然能够取得比较好的性能, 因此实现即使是很简单的动态负载平衡系统也是很有必要的。

## 参 考 文 献

- Koyama K., Shimizu K., Ashihara H., Zhang Y., Kameda H.. Performance evaluation of adaptive load balancing policies in distributed systems. In: Proceedings of Singapore International Conference on Networks/International Conference on Information Engineering'93, Singapore, 1993, 606~611
  - Ferrari D., Zhou S.. A trace driven simulation study of dynamic load balancing. IEEE Transactions on Software Engineering, 1988, 14(9): 1327~1341
  - Shivaratri N. G., Krueger P., Singhal M.. Load distributing for locally distributed systems. Computer, 1992, 25(12): 33~44
  - Eager D. L., Lazowska E. D., Zahorjan J.. The limited performance benefits of migrating active processes for load sharing. In: Proceedings of SIGMETRICS, New Mexico, 1988, 662~675
  - Livny M., Melman M.. Load balancing in homogeneous broadcast distributed systems. In: Proceedings of ACM Computer Network Performance Symposium, New York, 1982, 47~55
  - Ciardo G., Blakemore A., Chimento P. F., Muppala J. K., Trivedi K. S.. Automated generation and analysis of Markov reward models using stochastic rewards nets. In: Meyer C., Plemmons R. J. ed.. IMA volumes in Mathematics and its Applications, Heidelberg, Germany, 1992, 48, 145~191
  - Molloy M. K.. Performance analysis using stochastic Petri nets. IEEE Transactions on Computers, 1982, C-31(9): 913~917
  - Greiner S., Puliafito A., Bolch G., Trivedi K. S.. Performance evaluation of dynamic priority operating systems. In: Proceedings of IEEE International Workshop on Petri Nets and Performance Models (PNPM'95), Durham, NC, 1995, 241~250
  - Ciardo G., Muppala J. K., Trivedi K. S.. SPNP: Stochastic Petri net package. In: Proceedings of the 3rd International Workshop on Petri Nets and Performance Models, Kyoto, Japan, 1989, 142~151
  - Lin Chuang, Yang Shi-Qiang. Performance analysis of scheduling schemes in multiserver multiqueue systems. Acta Electronica Sinica, 2000, 28(5): 17~20 (in Chinese)
- (林 阖, 杨士强. 多服务器多队列系统调度方案的性能分析.)

电子学报,2000,28(5):17~20)

- 11 Ferrari D., Zhou S.. A load index for dynamic load balancing. In: Proceedings of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, 1986,

684~690

- 12 Kunz T.. The influence of different workload descriptions on a heuristic load balancing scheme. IEEE Transactions on Software Engineering, 1991, 17(7):725~730



**TANG DAN**, born in 1976, master.

His research interests include computer architecture, cluster computing and operating systems.

**JIN HAI**, born in 1966, Ph. D., professor and Ph. D. supervisor. His research interests include computer architecture, cluster computing, grid computing, network storage systems and network security.

**ZHANG YONG-KUN**, born in 1980, master. His research interests include computer architecture, cluster computing.

## Background

This work is part of our project, which is named the designation and implementation of single process space for clusters. The goal of this project is to study and develop a single system image (SSI) and dynamic load balancing system for Linux clusters at process level. With the project beginning in 2002, we focus on such fields as single PID over cluster, dynamic load balancing algorithms, load index, process migration and socket migration and have made a lot of novel works. We have published five research papers in domestic

and overseas publication or conference proceedings. Moreover, some research result, such as process migration which supporting socket migration and a new load index which uses the total processes remaining time have been successfully applied into the prototype of our dynamic load balancing system. This paper is mainly focused on the model of dynamic load balancing system for clusters which uses remote execution technology.