面向多租户 Web 应用的性能隔离方法

王伟^{①②*}, 黄涛^①, 魏峻^①, 钟华^①, 宋云奎^①

① 中国科学院软件研究所软件工程技术研究中心, 北京 100190

② 武汉大学软件工程国家重点实验室, 武汉 430072

* 通信作者. E-mail: wangwei@otcaix.iscas.ac.cn

收稿日期: 2012-10-26; 接受日期: 2012-11-20

国家自然科学基金 (批准号: 61100068)、国家高技术研究发展计划 (批准号: 2012AA011204)、国家重点基础研究发展计划 (批准号: 2009CB320704) 和武汉大学软件工程国家重点实验室开放基金 (批准号: SKLSE20100821) 资助项目

摘要 提出一种面向应用级共享的多租户 Web 应用性能隔离方法. 首先建立基于事务处理链的应用级资源管理模型, 基于信号量机制给出了模型的并发控制算法, 支持事务级、分阶段的多租户 Web 应用资源管理, 并在一次事务处理的各阶段实现线程复用, 避免因修改交互协议而产生的再工程代价. 基于上述工作, 给出多租户性能隔离算法及策略, 并利用 TPC-W 电子商务应用验证方法的系统开销及有效性. 实验结果表明, 该方法可以有效降低租户资源侵占行为的影响, 并避免系统过载.

关键词 多租户 性能管理 性能隔离 并发控制 Web 应用

1 引言

Internet 的开放性导致 Web 应用的用户负载规模难以预计, 80% 用户日常访问规模往往远低于峰值规模, 这些用户组成了数量庞大的“长尾用户”. 为满足“长尾用户”的服务需求而配备独立的基础设施, 造成了极大的能源、设施的浪费, 并带来巨大的维护成本, 因此产生了多租户共享的应用系统和服务模式. 多租户 (multi-tenancy) 使来自不同组织 (租户) 的并发用户共享同一基础设施资源, 有利于成本降低和收益提高¹⁾.

目前存在 4 种典型的多租户部署模式^[1] (如图 1 所示): 硬件级共享、操作系统级共享、中间件级共享以及应用级共享. 硬件级共享模式为每个租户配置一个虚拟机实例; 操作系统级共享模式在同一操作系统中为每个租户配置一个中间件服务器; 中间件级共享模式在同一中间件服务器上为每个租户配置一个应用实例; 应用级共享模式通过单一应用实例为多个租户提供服务. 其中, 应用级共享的多租户模式提供了更高层次的基础设施共享, 性能开销更低, 可以承载更多的租户, 因此是一种高效能的多租户模式. 然而, 开放环境下的租户难以完全信赖, 即使同一组织内部相互信赖的多个租户, 仍有可能由于误操作或过载而导致个别租户侵占大量资源, 进而影响其他租户的访问性能, 需要通过性能隔离 (performance isolation) 技术来避免此类性能干扰 (performance interference) 现象. 对于电子商务、在线支付等关键事务型 Web 应用, 性能隔离至关重要, 否则将导致客户损失和收益损失. 解决这一问题, 仍面临诸多技术挑战, 目前业内领先的中间件产品仍缺乏相关支持^[2].

1) <http://msdn.microsoft.com/en-us/library/aa479069.aspx>

图 1 典型的多租户部署方式

Figure 1 Typical multi-tenancy deployment approaches

多租户性能隔离取决于租户资源隔离, 应用级共享的多租户模式面临细粒度的资源管理需求和挑战. 传统资源管理模型通常利用代理通过黑盒方式进行准入控制和请求调度^[3,4], 但这种粗粒度、黑盒式的资源管理模型主要针对系统级信息 (客户端的 IP、请求目标 URL) 进行事务分类, 适用于硬件级共享、操作系统级共享以及中间件级共享等多租户模式, 但由于无法对准入的请求处理过程进一步控制, 因此性能隔离的效果无法保障, 无法适用于应用级共享的多租户模式. 目前, 典型的企业计算平台 JavaEE 已经在其最新的规范草案加入并发管理规范²⁾, 意图在中间件层实现细粒度、白盒式的资源管理, 即资源管理者可感知事务在中间件层的处理过程及其上下文环境. 在已有工作中, 一些基于事件驱动的资源管理模型被提出, 其中的代表工作是基于阶段的事件处理架构 SEDA^[5]. SEDA 将服务器端的事务处理过程分解为多个阶段, 每个阶段利用线程池控制并发量, 阶段间通过异步事件连接. 但是, 对于典型的多层架构的事务型应用^[6], 事务处理过程涉及的应用组件或服务调用仍以同步调用为主, 而此类采用事件驱动的架构需修改交互机制, 代价巨大. 例如, 开源应用服务器 Tomcat³⁾在 6.0 版本中对 HTTP 协议进行修改, 实现异步交互, 与 5.5 版本相比, 新增相关代码 1 万行以上.

针对上述问题, 本文首先依据多租户 Web 应用的上下文信息构造事务处理链, 对并发的事务处理过程进行租户区分、事务类型区分和阶段划分, 提出一种面向多租户 Web 应用的应用级资源管理模型 (application-level resource management model, ARMM), 并基于信号量机制给出了 ARMM 模型的并发控制及容量动态调整算法, 算法在一次事务处理的各阶段实现线程复用, 避免已有方法因修改交互协议而产生的再工程代价. 基于上述工作给出多租户性能隔离算法及策略, 并通过 TPC-W 实验案例分析方法的系统开销及性能隔离效果. 实验结果表明, ARMM 提供的细粒度资源管理机制能够提高性能隔离方法的敏捷性, 有效降低租户性能干扰影响, 同时避免了系统再工程代价.

后文组织如下: 第 2 节分析应用级资源管理模型的需求及挑战, 第 3 节围绕 Web 应用的事务处理过程, 定义 ARMM 模型的组成和结构, 并给出了 ARMM 模型的并发控制算法、容量动态调整算法以及租户性能隔离算法的设计, 第 4 节针对 TPC-W 电子商务应用验证方法的系统开销及性能隔离效果, 第 5 节比较相关研究工作, 最后是全文结论.

2) <http://jcp.org/en/jsr/detail?id=236>

3) <http://tomcat.apache.org>.

图 2 典型的 Web 应用事务处理过程

Figure 2 A typical transaction handling process

图 3 请求区分粒度

Figure 3 Request differentiation spectrum

2 应用级资源管理模型 (ARMM) 需求及挑战

事务是多层架构 Web 应用的主要研究对象. 客户端请求发起后, 多个处理任务结合在一起完成一个业务要求, 称为事务^[7]. 服务器端的事务处理从接受到客户端请求开始, 到返回请求响应结束. 如图 2 所示, 在事务的生命周期中, 包含一至多次应用组件操作, 同时, 可能包含一至多次数据库操作. 在事务处理过程中, ARMM 模型需要依据多租户 Web 应用的上下文信息对不同租户的并发请求进行区分和管理:

- 面向租户的请求区分: 在应用级共享的多租户模式下, 可依据租户 (请求发起者) 的属性和特征进行请求区分. 租户标识往往保存在 HTTP cookie、会话 (session) 或者安全上下文信息中⁴⁾. ARMM 模型需要支持基于上述信息的请求区分.
- 面向事务类型的请求区分: 不同类型的事务对资源的需求存在很大差异, 仅区分租户难以有效实现性能隔离^[8], ARMM 模型还需要支持事务类型的区分.
- 面向事务处理过程的阶段划分: 在多层架构的多租户 Web 应用中, 不同层次 (如图 2 所示的中间件层和数据库层) 均易出现租户性能干扰^[9]. 为了进行针对性的管理, ARMM 模型需要根据请求目标 (如图 3 所示, 如组件操作、数据库操作等) 将事务处理过程进行阶段划分.

4) <http://www.ibm.com/developerworks/cloud/library/cl-multitenantsaas/index.html>

上述模型需求可以总结为事务处理过程的租户区分、事务类型区分和阶段划分, 可以借鉴基于阶段的事件处理架构 SEDA 实现. SEDA 采用白盒式的资源管理架构, 将事务处理过程分解为多个阶段, 每个阶段利用线程池控制并发量, 阶段间通过异步事件连接, 符合模型的需求. 但是, Web 应用的事务处理过程涉及的应用组件、服务以及数据库操作的调用主要是同步调用, 例如企业计算中广泛使用的 JSP/Servlet^[10]、JDBC^[11] 等, 使用 SEDA 架构需要修改这些组件、服务以及数据库操作的交互协议. 开源应用服务器 Tomcat 在 6.0 版本中支持异步 HTTP 协议, 从而实现 Servlet 的异步访问, 但与 5.5 版本相比, 相关代码新增 1 万行以上, 再工程代价巨大.

3 ARMM 模型设计及算法

3.1 事务处理链

针对应用级共享的多租户模式的细粒度资源管理需求, 依据多租户 Web 应用的上下文信息构造事务处理链, 对事务处理过程进行租户区分、事务类型区分以及阶段划分.

定义 1 事务处理链 (transaction-processing chain, TC) 完成事务的请求处理过程, 由一组阶段序列组成, 每个阶段执行一种特定的事务处理任务. 如图 4 所示, 事务处理链的每个阶段与一个资源管理器绑定.

定义 2 资源管理器 (resource manager, RM) 是一种资源控制元素, 包含一组任务处理器 (task processor, TP)、一个容量阈值 (TPMax) 以及一个任务等待队列. 其中, TP 执行事务处理任务, TPMax 用于约束事务处理任务的并发量, 当事务处理任务的并发量大于 TPMax 时, 新到来的任务将进入等待队列.

如图 4 所示, 事务处理链的基本设计思想是: 对于一个包含 L 个租户、 M 种事务类型的多租户 Web 应用, 首先按照分隔 (partition) 设计模式, 将所有事务处理过程按租户标识和事务类型进行区分, 得到 $L * M$ 种事务处理过程:

$$\begin{bmatrix} T_1^1 & T_2^1 & \dots & T_i^1 & \dots & T_M^1 \\ T_1^2 & T_2^2 & \dots & T_i^2 & \dots & T_M^2 \\ \vdots & & & & & \\ T_1^k & T_2^k & \dots & T_i^k & \dots & T_M^k \\ \vdots & & & & & \\ T_1^L & T_2^L & \dots & T_i^L & \dots & T_M^L \end{bmatrix},$$

其中, T_i^k 表示第 k 个租户的第 i 种事务, 其中, $1 \leq k \leq L, 1 \leq i \leq M$. 然后, 对 T_i^k 的事务处理过程利用流水线 (pipeline) 设计模式进行阶段划分, 得到事务处理链 $TC_i^k = \langle P_{i,1}^k, \dots, P_{i,j}^k, \dots, P_{i,PN_i}^k \rangle$, 其中, $P_{i,j}^k$ 表示事务 T_i^k 的第 j 个阶段, PN_i 表示事务处理过程中的阶段数量. 阶段 $P_{i,j}^k$ 与资源管理器 $RM_{i,j}^k$ 绑定, 利用容量阈值 $TPMax_{i,j}^k$ 控制该阶段的任务并发量.

3.2 ARMM 模型并发控制及容量动态调整算法

我们借鉴信号量机制, 设计事务处理链的并发控制机制. 信号量机制在操作系统中用于解决共享资源的同步和互斥问题, 通过设置一个非负整数值的信号量来表示临界区内的可用共享资源数量, 以

图 4 事务处理链

Figure 4 Transaction processing chain

图 5 基于信号量的并发控制

Figure 5 Semaphore-based concurrency control

及设计共享资源的请求行为 (P 操作原语) 和释放行为 (V 操作原语). 基于信号量的事务处理链保持了事务处理各阶段的同步交互方式, 从而避免修改交互协议, 并且, 在阶段内利用信号量进行并发控制, 化解死锁与饿死隐患.

如图 5 所示, 与 SEDA 架构不同, 事务处理链的每个阶段不提供单独的线程池, 而是利用一个阶段信号量控制阶段的任务并发量. 在每个阶段中, 任务处理器使用原有系统线程池提供的线程. 阶段信号量的初始值表示阶段容量. 在一次事务处理过程中, 各阶段复用同一线程. 在进入每个阶段时, 执行信号量的 P 操作, 并根据当前信号量状态, 决定是否挂起线程, 若需要挂起, 则将一个执行唤醒该线程的任务放入等待队列 (FIFO 队列). 在离开每个阶段时, 执行信号量的 V 操作, 并根据当前信号量状态, 决定是否唤醒等待队列中被挂起的线程.

基于信号量的事务处理链并发控制机制通过两个关键算法实现: 1) ARMM 并发控制算法; 2) ARMM 容量调整算法.

3.2.1 并发控制算法

ARMM 并发控制算法基于信号量机制对阶段内的事务处理任务进行并发控制, 算法使用的数据结构如表 1 所示. 阶段 $P_{i,j}^k$ 内的并发量由容量阈值 $TPMax_{i,j}^k$ 限制, 受信号量 $TPCount_{i,j}^k$ 控制,

表 1 ARMM 算法使用的数据结构

Table 1 Data structure used in the ARMM algorithms

$P_{i,j}^k$	j th phase in transaction T_i of tenant k
$\text{TPCount}_{i,j}^k$	Semaphore of the j th phase in transaction T_i of tenant k
$\text{TPMax}_{i,j}^k$	Capacity threshold of the j th phase in transaction T_i of tenant k
$\text{Queue}_{i,j}^k$	FIFO waiting queue of the j th phase in transaction T_i of tenant k
$\text{Adj}_{i,j}^k$	Adjustment value of the capacity threshold of the j th phase in transaction T_i of tenant k
$\text{ToDec}_{i,j}^k$	Decrease in the capacity threshold of the j th phase in transaction T_i of tenant k

Algorithm 1 Concurrency control in phase

```

1. semPOperation( $P_{i,j}^k$ ){
2.  $\text{TPCount}_{i,j}^k - -$ ;
3. if  $\text{TPCount}_{i,j}^k < 0$ 
4.   create a signal work for this task;
5.   insert the wakeup task into the  $\text{Queue}_{i,j}^k$ ;
6.   suspend this thread;
7. end if
8. }
9. semVOperation( $P_{i,j}^k$ ){
10. if  $\text{ToDec}_{i,j}^k > 0$ 
11.    $\text{ToDec}_{i,j}^k - -$ ;  $\text{TPMax}_{i,j}^k - -$ ;
12.   return;
13. end if
14.  $\text{TPCount}_{i,j}^k + +$ ;
15. if  $\text{TPCount}_{i,j}^k \leq 0$ 
16.   remove a wakeup task from the  $\text{Queue}_{i,j}^k$ ;
17.   wake up the suspended thread;
18. end if
19. }

```

图 6 ARMM 模型并发控制算法

Figure 6 ARMM concurrency control algorithm

$\text{TPCount}_{i,j}^k$ 的初始值为 $\text{TPMax}_{i,j}^k$. 事务处理在进入和离开每个阶段时, 会执行算法的 PV 操作. 算法伪代码如图 6 所示.

执行 $P_{i,j}^k$ 的 P 操作时, 信号量 $\text{TPCount}_{i,j}^k$ 值减 1. 此时, 如果 $\text{TPCount}_{i,j}^k \geq 0$, 表示当前线程可以继续执行, P 操作结束; 如果 $\text{TPCount}_{i,j}^k < 0$, 表示当前线程需要挂起, 并将一个执行唤醒该线程的任务放入 $P_{i,j}^k$ 阶段的等待队列 $\text{Queue}_{i,j}^k$ 中. 执行 $P_{i,j}^k$ 的 V 操作时, 信号量 $\text{TPCount}_{i,j}^k$ 值加 1. 此时, 如果 $\text{TPCount}_{i,j}^k \leq 0$, 则执行等待队列 $\text{Queue}_{i,j}^k$ 中队首的唤醒任务以唤醒对应的挂起线程, V 操作结束.

Algorithm 2 Runtime capacity adjusting in phase

```

1. adjustCapacity( $P_{i,j}^k, Adj_{i,j}^k$ ){
2.   if  $Adj_{i,j}^k > 0$ 
3.     while  $Adj_{i,j}^k > 0$ 
4.        $TPCount_{i,j}^k ++$ ;  $TPMax_{i,j}^k ++$ ;  $Adj_{i,j}^k --$ ;
5.       if  $TPCount_{i,j}^k \leq 0$ 
6.         remove a wakeup task form the  $Queue_{i,j}^k$ ;
7.         wake up the suspended thread;
8.       end if
9.     end while
10.    return;
11.  end if
12.  if  $Adj_{i,j}^k < 0$ 
13.    if  $TPMax_{i,j}^k > |Adj_{i,j}^k|$ 
14.      while  $|Adj_{i,j}^k| > 0$ 
15.        if  $TPCount_{i,j}^k \leq 0$ 
16.           $ToDec_{i,j}^k = |Adj_{i,j}^k|$ ;
17.          return;
18.        end if
19.         $TPCount_{i,j}^k --$ ;  $TPMax_{i,j}^k --$ ;  $Adj_{i,j}^k ++$ ;
20.      end while
21.    end if
22.  end if
23. }
```

图 7 ARMM 模型运行时容量调整算法**Figure 7** ARMM runtime capacity adjusting algorithm**3.2.2 容量动态调整算法**

容量动态调整算法为资源管理提供基本操作, 可根据资源状态和资源需求, 利用容量动态调整算法调整不同租户的事务处理链各阶段容量, 是进行性能隔离的基础. 算法使用的数据结构如表 1 所示, 算法伪代码如图 7 所示.

算法首先判断容量调整值 $Adj_{i,j}^k$, 如果 $Adj_{i,j}^k > 0$, 则增加 $TPCount_{i,j}^k$ 和 $TPMax_{i,j}^k$ 的值, 每次加 1, 同时减小 $Adj_{i,j}^k$ 的值, 每次减 1. 在此过程中, 如果 $TPCount_{i,j}^k \leq 0$, 则执行等待队列 $Queue_{i,j}^k$ 中队的唤醒任务以唤醒对应的挂起线程. 重复上述过程直到 $Adj_{i,j}^k$ 等于 0 时, 算法结束. 如果 $Adj_{i,j}^k < 0$, 则减少 $TPCount_{i,j}^k$ 和 $TPMax_{i,j}^k$ 的值, 每次减 1, 同时增加 $Adj_{i,j}^k$ 的值, 每次加 1, 直到 $Adj_{i,j}^k$ 等于 0 时, 算法结束. 在此过程中, 如果 $TPCount_{i,j}^k \leq 0$, 即该阶段的等待队列不为空时, 使用容量待减值 $ToDec_{i,j}^k$ 记录剩余调整量, 留待执行并发控制算法的 V 操作时再减小容量阈值 $TPMax_{i,j}^k$, 算法结束. 在 3.2.1 小节中并发控制算法执行信号量 V 操作时, 首先判断容量待减值 $ToDec_{i,j}^k$ (如图 6 所示算法

表 2 性能隔离算法参数设置

Table 2 Parameters used in the performance isolation algorithm

T_{control}	Control interval size	10 s
a	Smoothing constant	0.7
$U_{\text{threshold}}$	Resource utilization threshold	85%
adj_d	Multiplicative rate decrease	0.9
adj_i	Additive rate increase	1

第 10 行). 如果 $\text{ToDec}_{i,j}^k > 0$, 则 $\text{TPMax}_{i,j}^k$ 和 $\text{ToDec}_{i,j}^k$ 的值分别减 1, 并结束 V 操作, 此时, 仅是调整容量阈值, 而信号量 $\text{TPCount}_{i,j}^k$ 没有增加, 也没有唤醒一个挂起的线程.

3.3 租户性能隔离算法及策略

在 ARMM 并发控制和容量调整算法基础上, 根据不同的性能管理决策模型^[12], 可以设计多种性能隔离算法. 本文的重点不是讨论哪种决策模型更有效, 因此选取相对简单的基于观测的决策模型, 给出一种简单的启发式性能隔离算法及策略, 其基本思路如下: 在一组性能隔离算法的控制周期中, 在线获得的各个租户、事务的资源状态以及应用服务器和数据库服务器各自的资源总体状态; 同时, 采用如下指数平滑函数避免资源状态的瞬时突变情况: $U'_{\text{cur_interval}} = a \cdot U_{\text{last_interval}} + (1 - a) \cdot U'_{\text{cur_interval}}$. 其中, $U'_{\text{cur_interval}}$ 和 $U_{\text{cur_interval}}$ 分别为平滑处理前和处理后的当前控制周期内的资源状态 (租户、事务或服务器总体资源状态), $U_{\text{last_interval}}$ 为上一周期的资源状态; 当应用服务器或数据库服务器的资源总体利用率大于预先设定的过载阈值 $U_{\text{threshold}}$ 时, 算法检测占用资源最多的租户以及该租户占用资源最多的事务; 接着, 算法在上一控制周期的平均请求处理并发量基础上, 乘以并发量调整系数 adj_d , 得到并发量调整值, 然后利用 ARMM 容量调整算法在租户粒度或事务粒度逐步降低并发量, 直到资源总体利用率接近过载阈值; 当应用服务器或数据库服务器的资源总体利用率小于预设过载阈值时, 并且 ARMM 模型中存在由于前期并发限制而出现的等待队列时, 则利用 ARMM 容量调整算法, 依据递增系数 adj_i , 逐步提高租户粒度或事务粒度的并发量, 直到资源总体利用率接近过载阈值. 上述参数设置见表 2, 其中, 资源过载阈值根据已有经验设置为 85%. 最优参数取值不在本文讨论范畴.

依据上述算法, 我们可以设计黑盒式的租户级性能隔离策略 (tenant-level performance isolation, TenPI) 和白盒式的事务级性能隔离策略 (transaction-level performance isolation, TraPI). 两种不同粒度的性能隔离策略的区别在于, 前者 (TenPI) 依据每个租户的资源占用情况, 计算相应租户的并发量调整值, 进行粗粒度的性能隔离, 后者 (TraPI) 则依据每个租户中不同类型事务的处理过程各阶段资源占用情况, 得到相应事务处理任务并发量调整值, 进行细粒度的性能隔离.

4 方法实现与实验分析

本节主要介绍方法的原型系统实现和在其基础上展开的实验分析和验证.

4.1 方法实现及实验环境

如图 8 所示, 我们根据本文前述方法实现了一个多租户性能隔离系统原型, 主要包括 ARMM 框架、资源评估器以及性能隔离控制器三部分. 首先, 通过面向方面的编程 (AOP) 方式, 将 ARMM 框

图 8 实验系统架构概览

Figure 8 An overview of experimental system architecture

架分别注入到 Tomcat 应用服务器的 Servlet 容器和 JDBC 连接管理两个模块中, 实现了对典型两层架构 Web 应用的事务处理链构造及其包含的 Servlet 请求处理和 SQL 语句执行两个阶段的划分, 并且没有修改原系统的交互协议, 仅增加了 Tomcat 的启动参数; 然后, 资源评估器在线评估资源状态信息, 评估方法采用一种统计分析方法^[13], 避免常用的直接监测方法导致较高的系统开销; 最后, 性能控制器检测资源侵占型租户及事务, 并依据黑盒式的租户级性能隔离策略 TenPI 和白盒式的事务级性能隔离策略 TraPI, 通过 ARMM 框架进行租户性能隔离。

我们针对 TPC-W 电子商务应用⁵⁾, 设计了一系列实验, 对 ARMM 框架的系统开销以及性能隔离策略的有效性进行实验分析. TPC-W 是工业级的电子商务测试基准, 定义了 14 种事务操作, 包括网上书店的浏览、查询以及订单等事务操作. 这 14 种事务可根据不同的混合比例定义为 3 种负载模式 (浏览、购物、订单). 实验使用中国科学院软件研究所软件工程技术中心研发的符合 TPC-W 基准规范的 Bench4Q 测试套件⁶⁾, 可模拟复杂、可控的多租户访问负载. 数据库大小取决于电子商务网站的商品数量和客户量, 在本文实验中, 我们使用 TPC-W 的默认数据库设置, 商品数量为 10000, 客户数量为 1440000. 如图 8 所示, 实验环境是一个典型的两层架构系统, 包括: 1 个 Tomcat 服务器, 1 个 MySQL 服务器以及 1 个租户负载生成服务器. 所有机器通过 1 Gbps 以太网互联. 服务器机均运行 Windows Server 2003 操作系统.

4.2 实验方法、结果及分析

4.2.1 性能开销分析

本节在不同规模的负载 (50, 100, 150, 200, 300, 400, 500) 情况下, 对使用 ARMM 框架与未使用 ARMM 框架的情况进行比较, 分析 ARMM 框架的性能开销. 实验采用 TPC-W 浏览负载模式. 图 9 (a) 和 (b) 分别显示在无并发限制和有并发限制情况下 ARMM 框架的性能开销; 其中 X 轴表示负载规模, Y 轴表示吞吐率; 图 9(a) 中曲线 “No ARMM” 表示没有集成 ARMM 框架时的系统吞吐率在不同规模负载下的变化情况, 曲线 “ARMM” 表示集成 ARMM 框架后的系统吞吐率变化情况; 图 9(b) 中曲线 “No ARMM-20-20” 表示没有集成 ARMM 框架时, Tomcat 线程池大小设置为 20、JDBC 连接池设置为 20 情况下的系统吞吐率变化, 曲线 “ARMM-20-20” 集成 ARMM 框架后、Tomcat 并发

5) <http://www.tpc.org/tpcw>

6) <http://www.trustie.net/projects/files/list/Bench4Q>

图 9 ARMM 框架性能开销

Figure 9 Performance overhead induced by ARMM. (a) No concurrency limit; (b) concurrency limit of 20 Tomcat threads and 20 MySQL connections

图 10 租户负载

Figure 10 Tenants workloads

限制为 20、JDBC 连接限制为 20 情况下的系统吞吐率变化. 如图 9(a) 中数据显示, 使用 ARMM 框架后吞吐率较未使用时有所降低, 但性能开销最大在 10% 左右 (并发用户为 100 时), 最小低于 1% (并发用户为 150, 500 时). 图 9(b) 中的数据具有相似的特征.

4.2.2 性能隔离效果验证

本小节验证基于 ARMM 模型及其算法的两种租户性能隔离策略的有效性, 受篇幅所限, 下面仅对包含 2 个租户 (Ten1, Ten2) 的实验场景进行阐述. 图 10 表示所生成的 2 个租户的负载, X 轴表示性能隔离算法的控制周期, Y 轴表示访问系统的并发用户数 (EB). 图 11 表示在使用 TenPI 和 TraPI 两种策略时每个租户的吞吐率变化情况, X 轴表示控制周期, Y 轴表示租户吞吐率 (WIPS). 图 12(a) 和 (b) 分别表示两种策略下各租户在应用服务器端的 CPU 利用率变化情况, 图 13(a) 和 (b) 分别表示两种策略下各租户在数据库服务器端的 CPU 利用率变化情况, 图中 X 轴表示控制周期, Y 轴表示 CPU 利用率 (%).

如图 10 所示, 我们利用负载生成服务器模拟租户 Ten1 和 Ten2 在购物负载模式下的持续负载 (100 EB), 同时, 模拟了租户 Ten1 和 Ten2 的资源侵占行为, 包括增加租户 Ten1 请求 Admin 相关事

图 11 租户吞吐量

Figure 11 Tenants throughputs

务的负载规模以增加应用服务器端 CPU 资源的使用, 以及增加租户 Ten2 请求 Best Seller 事务的负载规模以增加对数据库服务器端 CPU 资源的使用^[14]. 可以看到, 租户 Ten1 和 Ten2 的资源侵占行为分别出现在实验过程的前半段和后半段. 分析实验结果, 可得到以下观察:

- 粗粒度的 TenPI 策略对受控租户的吞吐率产生较大影响.

由图 11 可以看出, TenPI 策略对租户 Ten1 的吞吐率产生较大影响, 第 11 至第 30 控制周期内的平均吞吐率比第 1 至第 10 控制周期下降近 53%. 相应的, TraPI 策略对租户 Ten1 的吞吐率产生的影响较小, 与第 1 至第 10 控制周期内的平均吞吐率相比, 第 11 至第 30 控制周期下降 21%, 第 20 至第 30 控制周期仅下降 8%. 类似的现象也出现在第 41 至第 60 控制周期, 当租户 Ten2 的 Best Seller 事务负载增加后, TenPI 策略导致租户 Ten2 在第 41 至第 60 控制周期的平均吞吐率比第 1 至第 10 控制周期下降 46%, 而 TraPI 策略在第 41 至第 60 控制周期仅产生 7% 的影响, 在第 50 至第 60 控制周期内仅下降不到 4%. 产生上述现象的原因在于, 粗粒度的 TenPI 策略需要对租户 Ten1 (或 Ten2) 的并发请求进行更大的并发限制才能降低 Admin 相关事务 (或 Best Seller 事务) 的并发量, 由此对受控租户的吞吐率产生较大影响.

- 细粒度的 TraPI 策略可以更快地响应系统过载, 避免性能干扰.

如图 12(a) 所示, 从第 11 个控制周期开始, 当租户 Ten1 的 Admin 相关事务的负载增加时, 租户 Ten1 出现资源侵占行为, 导致租户 Ten1 在应用服务器端的 CPU 利用率突然升高, 最高达到 83%, 进而导致总 CPU 利用率超过阈值, 最高达到 95%. 性能隔离算法启动后, TenPI 策略在 20 个控制周期内 (第 11 至第 30 控制周期), 并没有有效控制租户 Ten1 的资源侵占行为, 总 CPU 利用率始终高于 85%. 由图 12(b) 可知, TraPI 策略在 6~8 个控制周期内将相应租户的 CPU 利用率控制到合理范围, 总 CPU 利用率也维持在 85% 附近. 这表明, 粗粒度的 TenPI 策略对租户 Ten1 的资源侵占行为反应迟缓, 没有及时避免系统过载, 而细粒度的 TraPI 策略则可以更快地响应系统过载, 避免性能干扰. 如图 13(a) 和 (b) 所示, 类似的现象也发生在租户 Ten2 出现数据库 CPU 资源侵占行为的第 41 至第 60 控制周期.

对比图 12(a) 和图 13(a), 可以发现在图 12(a) 中的第 11 至第 30 控制周期, 应用服务器 CPU 有缓慢下降的趋势, 但是在图 13(a) 中的第 41 至第 60 控制周期, 数据库服务器 CPU 持续维持在 95% 以上, 过载情况更为严重. 这是因为粗粒度的 Ten 策略没有对事务处理过程进行阶段划分, 仅对前端应用服务器有一定效果, 而无法有效控制事务处理过程的全生命周期. 在图 12(b) 和图 13(b) 中, TraPI

图 12 应用服务器端租户 CPU 利用率

Figure 12 Tenants CPU utilization on application servers. (a) TenPI policy; (b) TraPI policy

图 13 数据库服务器端租户 CPU 利用率

Figure 13 Tenants CPU utilization on database servers. (a) TenPI policy; (b) TraPI policy

策略并没有出现类似的现象。

5 相关工作

应用级共享的多租户 Web 应用性能隔离需要建立一个应用级资源管理模型, 定义资源、资源使用者以及资源管理者间的关系. 其中, 资源使用者的分类以及与资源管理者的绑定关系是实施细粒度资源管理策略的关键。

在资源使用者的分类机制方面, 已有的资源管理模型主要针对系统级信息进行事务分类. 例如 Abdelzaher^[4], Zhang^[15] 等工作通过客户端的 IP、请求目标 URL 等系统级信息进行事务分类. 上述事务分类机制通常适用于硬件级共享、操作系统级共享以及中间件级共享等多租户模式, 但无法满足应用级共享的多租户模式的细粒度资源管理需求. 在本文中, ARMM 模型及框架可以依据多租户 Web 应用的上下文信息构造事务处理链, 对并发的处理过程进行租户区分、事务类型区分和阶段划分。

在资源使用者与资源管理者绑定方面, 包括黑盒式和白盒式两类工作. 黑盒式的资源管理方法, 其优点是不必修改应用、中间件、数据库等系统, 但是由于事务请求还没有解析, 所以资源管理者只能

基于系统级目标 (如 URL) 进行分类, 但资源管理能力有限^[16]. Elnikety 等^[17]的工作是此类工作代表, 他们利用代理机制进行请求准入控制, 防止系统过载. 白盒式的资源管理方法可感知事务处理过程, 其代表工作是基于阶段的事件处理架构 SEDA^[5]. SEDA 将服务器端的请求处理过程分解为多个阶段, 每个阶段利用线程池控制该阶段内的任务并发量, 阶段间通过异步事件连接. SEDA 是一种细粒度的资源管理模型, 但是, 由于阶段间采用基于事件的异步交互方式, SEDA 架构难以支持应用组件间的同步交互, 因此难以应用到主流的组件运行环境, 其成功案例局限于不包含组件交互的 HTTP server 或基于消息中间件的 ESB 等. 在本文工作中 ARMM 模型基于信号量机制进行细粒度的事务并发控制, 在一次事务处理的各阶段实现线程复用, 避免修改交互协议以及共享资源并发访问时易产生的死锁和饿死问题.

6 结束语

针对应用级共享的多租户应用面临的性能隔离技术挑战, 本文提出一种基于事务处理链的应用级资源管理模型 ARMM, 在实现细粒度的并发事务管理的同时, 避免了已有方法由于需要修改交互协议而产生的再工程代价, 同时, 基于本文工作设计了两种不同粒度的并发控制策略, 并针对 TPC-W 电子商务应用设计了一系列实验, 对本文方法的系统开销和性能隔离策略的有效性进行实验分析. 我们发现细粒度的性能隔离策略在受控租户的吞吐率和 CPU 过载控制的响应性等方面都要优于粗粒度的并发控制策略.

目前的 ARMM 框架仅支持在组件或服务粒度对事务处理过程进行阶段划分和并发控制, 如果组件的内部逻辑包含大量的资源密集操作, 则即使设置并发量为 1, 仍无法避免出现系统过载和性能干扰. 未来工作将考虑对组件内的业务逻辑进行抢占调度, 这一工作的难点在于需要依赖编程语言和操作系统的支持, 并且需要考虑可能出现的死锁问题.

致谢 在此, 我们向对本文的工作给予支持和建议的各位评审专家、编辑和在 NASAC 2012 会议上提出宝贵修改意见的陈仪香教授表示感谢.

参考文献

- 1 Natis Y V. Reference Architecture for Multitenancy: Enterprise Computing in the Cloud. Gartner, 2008
- 2 Pezzini M, Yefim V N, Iijima K, et al. Magic Quadrant for Enterprise Application Servers. Gartner, 2011
- 3 Elnikety S, Nahum E, Tracey J, et al. A method for transparent admission control and request scheduling in E-commerce Web sites. In: Proceedings of the 13th International Conference on World Wide Web. New York: ACM, 2004. 276–286
- 4 Abdelzaher T, Shin K G, Bhatti N. Performance guarantees for Web server end-systems: A control-theoretical approach. IEEE Trans Parallel Distrib Syst, 2002, 13: 80–96
- 5 Welsh M, Culler D, Brewer E. SEDA: An architecture for well-conditioned, scalable Internet services. In: Proceedings of the 18th Symposium on Operating Systems Principles. Banff: ACM, 2001. 230–243
- 6 Jiménez-Peris R, Patiño-Martínez M, Kemme B. Enterprise grids: challenges ahead. J Grid Comput, 2007, 5: 283–294
- 7 Cherkasova L, Fu Y, Tang W, et al. Measuring and characterizing end-to-end Internet service performance. ACM/IEEE Trans Internet Technol, 2003, 3: 347–391

- 8 Barham P, Donnelly A, Isaacs R, et al. Using Magpie for request extraction and workload modeling. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation. Berkeley: ACM, 2004. 18
- 9 Lin H L, Sun K, Zhao S, et al. Feedback-control-based performance regulation for multi-tenant applications. In: Proceedings International Conference on Parallel and Distributed Systems. Shenzhen: IEEE, 2009. 134–141
- 10 Murray G. JavaTM Servlet Specification, Version 2.5. SUN Microsystems, Inc., 2005
- 11 Reese G. Database Programming with JDBC and Java, 2nd Edition. O'Reilly, 2000
- 12 Guitart J, Torres J, Ayguad E. A survey on performance management for internet applications. *Concurr Comput Pract Exp*, 2010, 22: 68–106
- 13 Wang W, Huang X, Zhang W B, et al. Dynamically estimating approach for CPU consumption of multi-tenancy web applications. *Chin J Comput*, 2011, 34: 2292–2304 [王伟, 黄翔, 张文博, 等. 多租户应用 CPU 资源动态评估方法. *计算机学报*, 2011, 34: 2292–2304]
- 14 Amza C, Cecchet E, Chanda A, et al. Bottleneck characterization of dynamic web site benchmarks. Rice University Technical Report TR02-391. 2002
- 15 Zhang R, Diao Y, Parekh S, et al. Control of fair queueing: modeling, implementation, and experiences. In: Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management. Nice: IEEE, 2005. 177–190
- 16 Welsh M, Culler W. Adaptive overload control for busy Internet servers. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems. Berkeley: ACM, 2003. 4
- 17 Elnikety S, Nahum E, Tracey J, et al. A method for transparent admission control and request scheduling in E-commerce Web sites. In: Proceedings of the 13th International Conference on World Wide Web. New York: ACM, 2004. 276–286

Performance isolation approach for multi-tenancy web applications

WANG Wei^{1,2*}, HUANG Tao¹, WEI Jun¹, ZHONG Hua¹ & SONG YunKui¹

¹ *Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China;*

² *State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China*

*E-mail: wangwei@otcaix.iscas.ac.cn

Abstract Multi-tenancy enables the sharing of resources and costs across a large pool of users, thus allowing for centralization of infrastructure in locations with lower costs. Performance isolation is a key requirement for multi-tenant Web application hosting environments. This paper proposes a performance isolation approach for application-level sharing multi-tenancy Web applications. A fine-grained application-level resource management model based on the transaction-processing chain is first defined, as well as its concurrency control algorithm based on the semaphore mechanism. This model helps to manage resources usage in a transaction-level and phase-targeted way. It reuses one thread within a transaction process, and therefore, avoids the reengineering cost generated by system reconstruction. The end result is an approach that extends the current Web application hosting environments to multi-tenancy aware hosting environments. A detailed set of experiments were performed under different settings for changing workloads in the controlled TPC-W application to evaluate the effectiveness and performance overhead of our approach. Compared to the traditional coarse-grained strategy, experimental results show that the fine-grained strategy based on our approach is more efficient and agile in avoiding performance interference between multiple hosted tenants and CPU overload. Our results also show that

the performance overhead of our approach is small.

Keywords multi-tenancy, performance management, performance isolation, concurrency control, Web application

WANG Wei received his M.S. and Ph.D. degrees from Institute of Software, Chinese Academy of Sciences in 2007 and 2010 respectively. Currently he is an Associate Professor in the Institute of Software, Chinese Academy of Sciences. His research interests include distributed computing and performance management.

HUANG Tao received his M.S. and Ph.D. degrees from Department of Computer Science, University of Science and Technology of China in 1991 and 1994 respectively. Currently he is a Professor and Ph.D. Supervisor in the Institute of Software, Chinese Academy of Sciences, and a CCF member. His research interests include service oriented computing, middleware, software component technology, distributed object technology.

WEI Jun received his B.S. and Ph.D. degrees in computer science from Wuhan University, Hubei, in 1992 and 1997, respectively. He was a post-doctoral researcher at the Hong Kong University of Science and Technology, China. He is currently a Professor in the Institute of Software, Chinese Academy of Sciences, Beijing, also a CCF member. His research interests include service oriented computing, middleware, mobile computing, software engineering.

ZHONG Hua received his M.S. and Ph.D. degrees from Institute of Software, Chinese Academy of Sciences in 1996 and 1999 respectively. Currently he is a Professor and Ph.D. Supervisor in the Institute of Software, Chinese Academy of Sciences, and a CCF member. His research interests include distributed computing and software engineering.