一种自动推断复杂系统层次结构任务模型的方法

高崇南 余宏亮 郑纬民
(清华大学计算机系 北京 100084)

摘 要 支撑 Internet 服务的复杂系统难于调试与分析.理解系统运行时行为是调试与分析这些复杂系统的关键.现有的技术将系统动态运行时行为用因果执行路径抽象描述,并在此基础上分析系统的行为.但是这些方法或者需要手动标注系统代码,或者需要使用者描述系统的执行结构,都需要使用者很多人工辅助.文中描述了一种自动推断复杂系统层次结构任务模型的方法.通过使用插装技术动态观察系统执行过程,文中的方法能够根据一组启发自动推断出系统运行时的任务模型,包括任务的边界和任务之间的因果依赖关系.通过使用聚类方法,能够进一步推断出任务模型的层次结构.通过在实际系统(Apache 和 PacificA)上应用推断方法,可以看出,使用得到的模型能够帮助理解系统的动态运行过程,并帮助分析解决系统的性能问题.

关键词 任务模型;系统分析;性能调试
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2010.00119

Methodology for Automatic Inference of Task Hierarchies in Complex Systems

GAO Chong-Nan YU Hong-Liang ZHENG Wei-Min
(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084)

Abstract Distributed systems are hard to debug and analysis. Understanding system runtime behavior is key to system debugging and analysis. Existing works describe system runtime behavior as causal paths, but these works requires either manual annotation or developer-provided execution structures. This paper describes methodology to automatically infer hierarchical task models for complex systems. By using instrumentation, it can automatically infer task models, including task boundaries and causal relations among tasks, based a set of general heuristics. By using clustering, it further infer hierarchical structures on generated task models. By applying the inference methodology on real systems (Apache and PacificA), it concludes that the hierarchical task models help both understanding system runtime behavior and debugging performance bugs.

Keywords task model; system analysis; performance debugging

1 引 言

Internet 服务越来越影响着人们生产生活的各个方面,它们的可靠性也变的越来越重要.系统设计与实现上的缺陷一直伴随着这些服务而生,导致服务性能下降,甚至彻底中断运行.在诸多的软件缺陷(bug)中,最难找到和解决的是让系统仍然运行,但是却偏离了期望行为的缺陷.这些缺陷隐藏在复杂甚至是混乱的应用逻辑中,寻找并分析这些缺陷异常困难.

支撑 Internet 服务的系统本质上非常复杂,这

收稿日期:2009-07-15;最终修改稿收到日期:2009-09-06.本课题得到国家自然科学基金(60603071)、国家"九七三"重点基础研究发展规划项目基金(2007CB311100)资助.**高崇南**,男,1981 年生,博士研究生,研究方向为分布式系统管理与性能调试.E-mail: gaochongnan@gmail.com.**余宏亮**,男,1976 年生,博士,副教授,研究方向为分布式系统.**郑纬民**,男,1946 年生,教授,博士生导师,研究领域为网格计算、集群计算、高性能存储系统和生物信息学.

些系统通常使用分层的体系结构，将其功能抽象表达为不同的层次结构. 其运行时具有高度的并发性. 系统在运行时，处理着许多用户层次的任务，例如用户请求，任务被分成许多阶段执行，不同的阶段被分布在多个机器、进程和线程上执行，使用事件或者异步消息作为通知机制. 验证单独每个任务的行为，是一件具有挑战性的问题，因为开发人员需要重构出任务的执行流，将任务执行过程中的各个阶段重新连接起来.

从概念上说，开发人员可以把任务执行过程用层次结构的任务模型表示，这与系统的分层体系结构设计一致. 任务模型中的任务表示在系统不同函数抽象层次的执行过程. 高层任务的执行被分为若干低层子任务. 以 PacificA[1] 为例，PacificA 是一个类似 BigTable[2] 的分布式存储系统. 图 1 显示了用户向 PacificA 提交数据任务的执行层次. 数据提交任务被分为两步(图的右边)，分别处理本地数据提交和远程数据提交. 每个任务分别由若干叶子任务构成，叶子任务的边界由同步点确定. 基于任务模型，开发人员可以更好地理解系统层次模块间的结构以及不同模块间的依赖关系，并验证处于不同层次任务的行为.

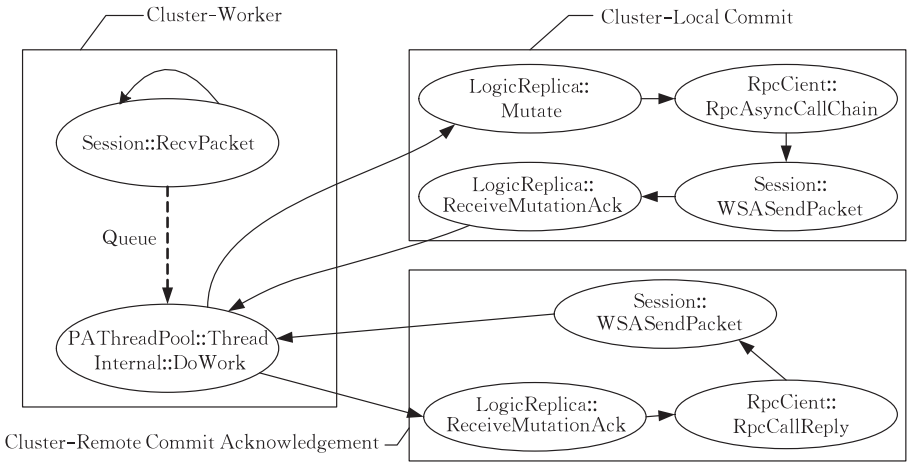

图 1 PacificA 的层次结构任务模型(包含 3 个高层任务. 图上的节点表示叶子任务，边表示叶子任务之间的因果依赖关系)

然而，目前的工具需要开发人员手动标注任务模型. 例如，Pip[3] 要求开发人员将系统预期行为，用“期望(expectations)”的形式表达，“期望”表达了系统正确执行时的任务模型，包括任务的执行顺序与对执行时资源使用的约束. 通过对比期望与实际执行的区别，可以验证系统运行时行为. 写出一个全面表达系统高层设计与底层实现的期望是非常困难的并容易出错，特别对那些正在快速演变中的系统. 基于执行路径的工具，例如 Magpie[4]，可以从包含运行时事件的 trace 推断每个请求的执行路径，但是它仅可以处理一组事前确定的事件，并且需要开发人员指定任务的边界与关联条件.

本文的目的是研究不需要人工帮助、自动推断层次结构任务模型的方法. 开发人员不需要手动标注源代码来指定任务边界，并且任务的层次结构也应该能够自动推断得到. 这是完成自动分析与诊断复杂系统目标的必要条件. 开发人员和系统管理员可以利用得到的任务模型，以可视化的形式表现系统设计和实现，也可以将任务模型作为输入，使用其它工具调试或验证系统设计.

设计一个自动推断任务模型的工具面临如下几个挑战性问题. 首先，应该能够确认合理的任务边界，这一过程应该只基于对系统执行过程的监视，而不需要开发人员显式地标注. 其次，必须能够正确地关联任务之间依赖关系. 特别是，必须能够辨别任务之间因为共享资源(例如共享队列、锁等)而产生的依赖关系. 最后，应该能够自动恢复任务的层次结构. 任务由许多或者顺序或者并行的子任务构成. 考虑到复杂系统中任务执行的并行性与非确定性，确定它们的依赖关系与层次结构并不容易.

在本文中，我们描述了如何自动推断复杂系统层次结构任务模型的方法，并实现了一个推断工具 Scalpel. 我们通过使用插装(instrument)技术来透明地观测系统运行，获取系统运行过程的 trace，包括应用层函数和系统同步函数的调用(例如 signal/wait). 推断方法使用 trace 作为输入，自下而上地推断系统层次结构任务模型，推断的过程分为 3 步，分别对应上面提到的 3 个挑战性问题.

首先，我们将运行过程分为一个个叶子任务，它们是层次结构任务模型中最基本的任务. 叶子任务的边界对应执行过程的同步点(synchronization point). 在同步点，线程或进程相互同步从而具有因果依赖关系，因此同步点是推断任务边界的一个合理启发(heuristic).

其次，我们按照运行时的因果依赖关系，将叶子任务用有向边连接，形成一个因果关系图. 叶子任务的因果依赖关系由任务运行时的 happened-before[5] 关系推断得到.

最后，我们在任务关系图上，推断出层次结构. 每一个层次，大致对应系统设计与实现上的一个层次. 我们使用聚类算法寻找任务因果关系图上重复出现的模式(频繁子图)，以此为依据确认高层任务与构成它的子任务. 通过递归地使用聚类算法，我们能够进一步寻找更高层次的任务.

我们分别在 Apache Web 服务器和 PacificA 上应用 Scalpel，结果显示，Scalpel 的推断方法能够得到具有合理意义的任务模型，并且完全不需要开发者的手工标注. 进一步，Scalpel 能够帮助开发者解决 PacificA 中的一个性能问题，该问题导致 PacificA 对网络带宽的利用率只有最大值的 70%.

本文第 2 节描述了推断方法设计；第 3 节叙述了推断方法的实现细节；第 4 节描述了对推断方法实现的优化；第 5 节我们描述了在实际系统上应用推断方法的实例；第 6 节叙述了相关工作；第 7 节为本文总结.

2　推断方法设计

本节给出自动推断系统层次结构任务模型的方法以及它的原型实现——Scalpel.

2.1　收集系统运行 trace

我们使用插装技术透明地观测系统运行，收集运行过程的函数调用 trace 以及调用参数，包括应用层函数和系统同步函数(例如 signal/wait)与系统 Socket 调用(例如 send/recv). 默认所有的函数调用都会被收集到 trace.

2.2　确定叶子任务

在系统运行 trace 的基础上，我们首先需要确定叶子任务的边界. Scalpel 使用同步点作为启发定义任务边界. 同步点是两个线程同步相互执行，从而具有 happened-before 关系的地方. 在同步点，线程可能因为互斥或者相互协同而具有 happened-before 关系，前者的例子是线程等待另一个线程释放保护共享资源的锁，后者的例子是线程向另一个等待在 event 上线程发 signal 消息，通知对方继续执行.

我们定义两个相继的同步点之间的执行为一个叶子任务. 采用这个定义的原因是，首先两个同步点之间的一段执行是相对独立，并且自包含的，不与其它线程的执行相互依赖. 因此，它们合理地成为任务模型中最小粒度的一段执行，也是叶子任务的一个自然定义. 另外，因为采用了同步点定义叶子任务的边界，叶子任务之间的依赖关系也只发生在边界之间.

Scalpel 通过插装系统函数库中的同步原语(锁、信号、事件等)与 socket 操作① 记录同步点操作. 在实际系统中的应用经验表明，插装这些系统操作是足够的. 如果系统使用 spin-lock 或者 lock-free 的数据结构，则仅插装系统调用是不够的. 这时 Scalpel 会丢失一些同步点从而使任务模型粒度更粗. 手工标注可以解决这个问题，但是整个推断方法不再是全自动的. 鉴于多数实际系统还是采用系统调用进行同步，我们将这个问题留作将来的工作.

2.3　任务因果关系图

为了推断任务的层次结构，我们首先要连接叶子任务之间的因果关系. 我们使用有向图表示任务之间的关系. 这个图中，节点表示叶子任务，有向边表示叶子任务之间的因果依赖关系. 例如在 PacificA 中(图 1)，当存储的主节点收到用户提交数据的消息后，首先将数据在本地持久化存储(LogicReplica::Mutate)，之后将数据用异步 RPC 的方式发送给次节点(RpcClient::RpcAsyncCallChain〉). 因此这两个叶子任务之间用有向边连接.

Scalpel 使用 happened-before 关系推断任务的因果依赖关系，这包括同一线程内顺序执行的两个叶子任务以及因为同步而依次执行的叶子任务. 并不是所有因为同步而具有 happened-before 关系的叶子任务都存在因果关系. 我们考虑的因果依赖关系，是两个叶子任务相互确定性依赖. 一个“真”的因果依赖关系的例子是，一个从队列里取出并处理事件的任务，在因果关系上依赖于产生那个事件并将其加入队列的任务. 另一方面，如果两个线程使用互斥锁同步对共享资源(例如 I/O)的访问，虽然它们的行为构成 happened-before 关系，但实际上，任务

① 我们认为通信也是一种同步操作.

之间并不相互依赖,其执行的顺序由调度器随机决定.

Scalpel 使用若干启发区分这两种不同的关系.如果系统使用操作系统提供的队列(例如 I/O Completion Ports,IOCP),或者通知机制(例如 event),则可以使用同步操作使用的句柄(保存在插装得到的 trace 中,是同步操作的参数),将生产线程与消费线程联系起来.具体地,可以通过分析 trace 中对于同一同步对象句柄的操作,得到不同线程在同步对象上的通知——获取通知,或者 enqueue-dequeue 操作,区分生产线程与消费线程,得到它们的依赖关系.操作系统的互斥(mutex)与信号量(semaphore)对象,通常仅仅被用来同步线程对共享资源的访问,生产线程和消费线程之间通常会使用显式的通知机制(IOCP 等)协调数据共享,因此不认为它们构成因果依赖关系.

对于使用 TCP 通信产生的因果关系,由于 TCP 的数据流语意会让消息的边界无法分辨,因而目前只能依赖程序员提供额外信息(例如在消息中做标记)连接消息的发送者和接受者,否则 Scalpel 无法准确匹配.然而这会引入手工干预.一些网络协议逆向工程方面的工作[6-7]可以从数据流中自动推断消息的边界与格式,但是推断的结果存在错误的可能.本文工作暂时不考虑将 TCP 消息的发送者和接收者联系起来,但是任务的因果关系图和下面叙述的任务层次结构推断方法同样适用于包含 TCP 通信的情况.

2.4 推断任务层次结构

我们通过不断挖掘任务因果关系图上重复出现的模式,来推断任务的层次结构.控掘算法受到文献[8]的启发,文献[8]的工作通过搜寻"热点子路径"(就是函数调用路径上重复出现的字串),来推断上下文无关文法.类似地,我们的算法搜寻任务因果关系图上重复出现的子图.我们认为每一个子图都在逻辑上代表了一个更高层的任务,它由子图中的任务集合构成.通过递归调用挖掘算法,我们得到了层次结构的任务模型.

具体地,推断算法首先枚举任务因果关系图上所有的连通子图,并将这些子图按照相似度(在下文中定义)聚类为多个模式.推断算法将每个出现次数超过设定阈值的模式定义为一个高层任务.接下来,得到的高层任务被替换为单个"超级"节点.通过递归应用算法,我们可以得到更高层的任务,直到没有新的高层任务产生.这时,因果关系图包括了若干超级节点以及一些未聚类的叶子节点,图可能是连通也可能是不连通的.Scalpel 将那些超级节点输出,每个节点都是一个最高层的任务模型.通过将超级节点展开为构成它们的子图,我们可以得到任务的整个层次结构.

算法中使用相似度将子图聚类为不同的模式,目前,我们使用精确匹配来定义两个子图相似,也就是两个子图完全同构.我们使用确定性的序列化算法,将子图编码,并使用编码的 Hash 值将子图聚类.这个聚类算法非常有效.当编码叶子任务时,我们使用任务两个边界上的函数调用栈作为任务的编码,忽略了任务执行过程中的函数调用.对于同一个类中的叶子任务,它们的函数调用栈完全相同.这个方法忽略了叶子任务一些不重要的区别,例如函数的参数与线程号.从经验来看,这个方法很好地区分了同一类别的叶子任务.

3 实 现

我们使用 R2[9] 在 Windows 平台上实现了 Scalpel.R2 是一个基于函数库的程序运行记录与回放工具.通过插装操作系统调用与应用函数,R2 记录了系统的运行过程,这包括对同步对象(互斥锁和信号等)的同步操作和 socket 操作.使用 R2,我们得到系统运行过程的 trace,并在 trace 基础上分析与推断得到系统的层次结构任务模型.

在被插装的系统调用与应用函数内,包含输出的函数名称与调用参数.这些信息用来推断叶子任务的边界,并给出任务边界的函数调用栈.利用函数调用栈,可以给叶子任务提供易于程序员理解的名字标签.

我们需要保证 trace 中记录的同步操作顺序和实际执行完全一致.由于操作系统调度会交叉执行不同线程的代码,因而直接在插装函数内输出 trace 并不能保证同步操作记录的顺序与实际执行顺序一致.由于线程被随机调度,可能存在图 2 显示的执行顺序.可以看出,输出 trace 的顺序,和 event 操作的顺序是相反的.为了解决这个问题,我们为每个同步对象创建一个 shadow lock,将 WaitForSingleObject 操作替换为 SignalObjectAndWait,保证了同步操作与输出 trace 是一个原子操作.

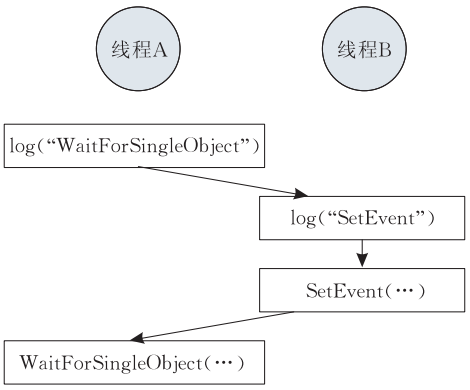

图 2　由于操作系统调度导致同步操作与输出 trace 执行顺序不一致的情况

4　性能优化

Scalpel 使用 R2 获取与记录函数调用事件. 对系统运行时的每一个线程，都创建一个单独的文件，记录这个线程的运行 trace. 在我们之前的工作中[10]，采用了基本的方法记录每个线程的运行时 trace. 我们使用字典对象(STL map)保存每个线程 trace 的文件句柄. 对每一次函数调用，首先通过系统调用得到当前线程号，并使用线程号为键值查询对应线程的 trace 文件句柄，将函数调用事件记录到 trace 中.

在多线程情况下，这个方法存在性能问题. 在每一次函数调用时，需要使用互斥锁(mutex)同步多个线程访问 trace 文件字典对象的过程. 由于实际系统运行时可能会频繁地发生函数调用，这样的实现会对性能产生很大的额外负载. 这个实现方法不是最优的，因为每个线程的 trace 文件都是相互独立的，并不交叉访问，因而存在改进的空间.

我们对之前的方法进行了优化，改进了性能. 具体地，我们使用线程局部存储(Thread Local Storage，TLS)技术保存每个线程的 trace 文件句柄，在记录函数调用事件时，直接从线程局部存储中得到 trace 文件句柄即可. 这样就不必使用字典对象统一保存所有线程的 trace 文件句柄，在记录函数调用事件时，也就不必要使用互斥锁同步线程的执行.

通过对比测试，我们能够看到优化后由 Scalpel 给应用带来的额外负载有显著降低.

在测试中，我们使用多个线程，并行执行一个执行体为空的函数(empty)，对比优化前后执行时间的差别. 由于 empty 函数的执行体是空，因而可以认为，相比原始代码的执行时间，应用 Scalpel 后增加的执行时间全部是因为输出 trace 所导致的，或者说通过代码可以测试得到输出 trace 产生额外负载的最大值. 在实际系统执行时，并不会发生如测试中"频繁"的函数调用，在函数调用之间还会执行其它代码，因而产生的额外负载远小于本测试的结果. 在第 5 节中我们将 Scalpel 应用于 Apache 和 PacificA 系统，并测试了 Scalpel 在这两个系统上产生的实际额外负载.

图 3 显示了优化前后额外负载的变化. 测试是在如下机器配置上进行的：2.0GHz Xeon 双核 CPU，4 GB 内存，运行 Windows Server 2003 SP2 操作系统. 分别测试了采用与不采用优化时，不同数量线程同时执行 hello 函数的时间消耗平均值. 可以看出，优化带来的性能提升是明显的. 在单线程时，两种方法并没有明显性能差异. 在多线程时，优化后性能提升很明显，在 5、10、20 个并行线程情况下，执行时间分别提高了 8.8、7.0 和 7.7 倍.

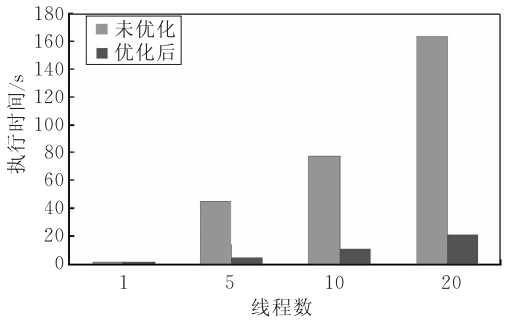

图 3　优化前后输出 trace 的性能额外负载情况

5　应用实例

在本节，我们使用 Scalpel 分析一些复杂系统，以评测任务模型是否有效. 可以从两个方面来评测任务模型. 首先，任务模型应该能够表达出系统设计的含义. 这一点很难用数字测量，因此，我们需要程序员去将推断的任务模型与实际系统设计对比，判断任务模型是否表达了系统设计的某些方面. 其次，任务模型也是验证系统正确性或者性能的方法，所以，我们也可以评测任务模型在帮助调试系统时的有效性.

我们同时使用这两种方式评测 Scalpel，分别将 Scalpel 应用在 Apache 和 PacificA[1] 上. Apache 和 PacificA 都使用了典型的配置. 我们在 Apache 上增加了一个 Subversion(SVN)模块，并使用客户端执行 10 次 Checkout，每次 Checkout 的文件数目是

377 个，文件大小总和是 1583KBbytes. 我们使用两台机器配置了一个 PacificA 的复制组(Replication Group)，一台机器是主节点(primary)，一台是从节点(backup). 我们运行一个测试程序，在 Pacifica 中创建一个表，并提交 15 条随机数据. 我们使用 Scalpel 追踪每个系统的运行，并推断任务模型. 两个实验都是在如下机器配置上进行的：2.0GHz Xeon双核 CPU，4GB 内存，运行 Windows Server 2003 SP2 操作系统，机器间通过 1 Gb 网络连接.

5.1 Apache

我们首先人工验证 Scalpel 在 Apache 上推断的任务模型. 图 4 显示了得到的模型. 在图上，每个节点都是一个叶子任务，连接任务的实线与虚线表示任务之间的因果关系，实线表示线程内部执行顺序因果关系，虚线表示线程间因果关系. 叶子任务上标记了任务起始点的调用栈. 为了易于理解，在本文中，我们只显示了调用栈尾部调用同步函数的那个函数名称.

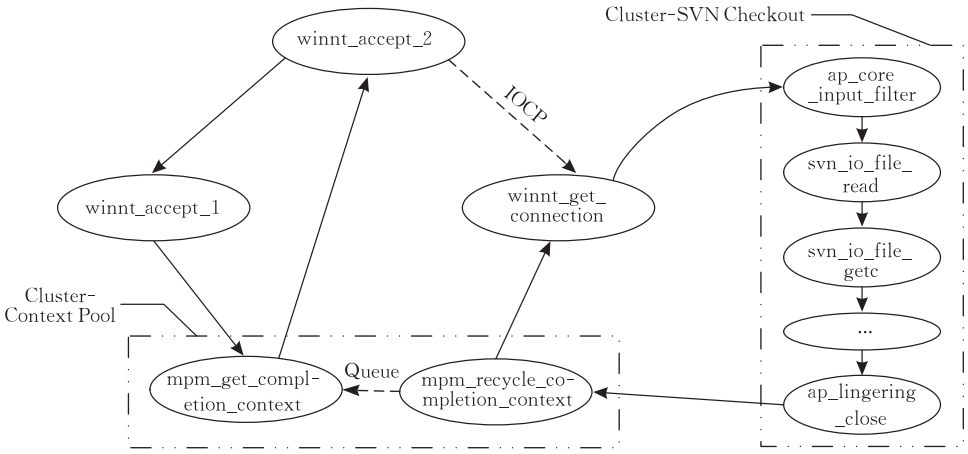

图 4 执行 SVN Checkout 得到的 Apache 任务模型

通过仔细阅读理解 Apache 和 SVN 的代码，我们确定推断的模型就是 Apache 服务 SVN Checkout 操作时的工作流程. Apache 接受外来的请求，并将请求交给线程池中的一个 worker 线程去完成. Scalpel 推断出 5 个 Apache 核心执行的叶子任务(图 4 的左边部分). 每个任务表示 Apache 接受并服务请求的主要步骤. SVN Checkout 过程中的一些叶子任务未在图上完全显示，但是它们都是处理请求过程中的关键步骤. 因此，这也是使用同步点为启发推断叶子任务边界有效性的一个例证.

进一步，Scalpel 还成功地找到了两个有实际含义的高层任务，见图 4 中方框表示的部分，在图上我们按照两个任务的工作给它们标了名字. 第一个高层任务(Context Pool)包括了连接上下文被取出用来传递用户请求又被回收的过程. 另一个(SVN Checkout)包含了 SVN Checkout 被处理的过程. 因为这两个任务都被频繁执行，因此 Scalpel 可以精确地将他们确认为高层任务.

5.2 PacificA

图 1 显示了 PacificA 的任务模型. PacificA 开发者确认了这个模型准确表达了用户提交数据在主节点上执行的过程. 模型包含了 3 个高层任务，分别是提交数据执行过程的 3 个主要步骤：一个 worker 任务、一个本地提交任务和一个确认远程提交的任务. 在 worker 任务中，一个 socket 监听线程循环的等待用户请求(Session::RecvPacket)，接受来自用户的消息，并将用户请求通过 IOCP 交给线程池中的 worker 线程去执行(PAThreadPool::ThreadInternal::DoWork). Worker 线程取到用户请求后，立即执行本地提交任务. 它首先将数据在本地存储(LogicReplica::Mutate)，并将提交请求使用异步 RPC 发送给从节点(RpcClient::RpcAsyncCallChain)，RPC 被序列化后通过网络层发送出去(Session::SendPacket)，在主节点确认自己已经持久化存储用户提交数据后(LogicReplica::ReceiveMutationAck)，本地提交任务结束. 从节点在收到提交请求后将数据存储，并给主节点发确认命令(acknowledgement). 确认远程提交任务就是主节点处理从节点确认命令的. 它首先处理从节点的确认消息(LogicReplica::ReceiveMutationAck)，之后向用户回复数据提交请求的结果(RpcClient::RpcCallReply)，RPC 被序列化后通过网络层发送出去(Session::SendPacket).

值得注意的是，Scalpel 不仅仅能够推断出用户

提交数据的过程可以分为 3 个高层任务，同时能够将 worker 线程执行的本地提交和确认远程提交两个任务区分开来. 这不是因为 Scalpel 理解了任务执行的语意，而是因为，这两个任务分别被两次 DoWork 任务触发执行，因此，挖掘算法能够将每个任务正确地区分开来.

接下来，我们评测使用 Scalpel 得到的任务模型对帮助调试系统是否有效. PacificA 的开发者希望我们能够使用 Scalpel 帮助他们解决系统中的一个性能问题. 开发者注意到了这个问题，并曾经试图使用函数性能概要分析(profiling)的办法，但是没有解决问题. 我们使用图 1 的模型分析 PacificA 的性能. 对每一个叶子任务，我们使用 Scalpel 记录它的执行时间、网络带宽消耗、CPU 使用等信息. 对高层任务，我们将它的子任务性能参数聚集得到它的性能参数信息.

通过在压力测试中执行数据提交操作，我们很快发现了一个性能问题：提交任务很难使用所有的网络带宽，同时它的 CPU 使用率也很低，远小于 100%(为 70%). 我们通过自上而下的办法找到这个问题的症结，从最高层任务向下，找到执行时间最长的那个任务. 我们发现，当以很高频率发送数据时，发送线程(典型的配置是 4 个并行线程同时发送)会在某一时刻阻塞大约 1s，其原因是，在 PacificA 的网络层代码，会对数据发送做流量控制，如果发送消息的缓冲区满了的话，流量控制会引起所有发送线程同步 Sleep 1s，如图 5 所示. 这样，4 个并发的发送线程就会表现出几乎同步的行为来，如图 6 所示.

```
int Session::WSASendPacket(NetworkStream * pkt) {
  CAutoLock guard(_send_lock);
  while (_send_size >(64<<20)) //64 MB
      Sleep(1000);
  ...
  int rt=WSASend(_socket, buf, buf_num, &bytes,
      0, (OVERLAPPED* )ce, 0);
  ...
}
```

图 5　PacificA 网络层流量控制代码

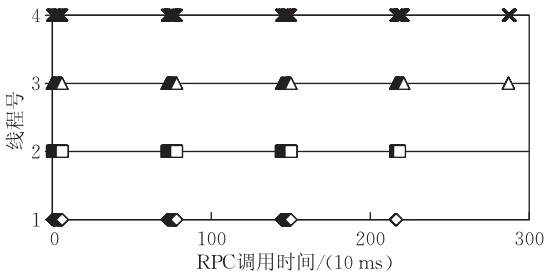

图 6　4 个线程异步调用 RPC 执行过程

进一步对代码审查让我们对问题的根本原因有了深刻理解. 当使用异步方式调用 RPC 时，RPC 层并没有流量控制机制. 每个发送线程都会以非阻塞方式独立发送 RPC 消息. 网络层的消息缓冲区很快就会被填满，导致发送线程被网络层阻塞. 这就让发送线程表现出了同步的行为，无论使用多少 RPC 发送线程，它们都会在网路层被同步阻塞. 引起这个问题的根本原因是 RPC 层和网络层没有一致的控制逻辑. 通过使用层次结构任务模型，我们很快能够找到性能问题，并理解造成它的根本原因.

我们进一步尝试解决这个性能问题. 我们将异步发送 RPC 消息改为同步发送方式，也就是每个线程发送 RPC 之后，等待 RPC 结果返回，然后再发送下一个 RPC. 使用相同的压力测试参数，我们看到这一次网络带宽的利用率接近 100%(97%左右)，线程发送 RPC 的行为模式如图 7 所示.

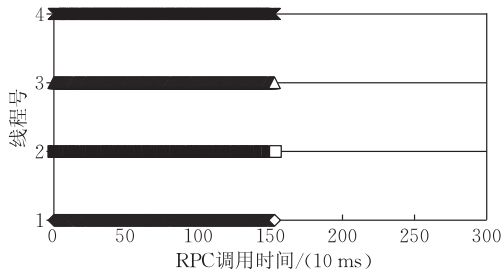

图 7　4 个线程同步调用 RPC 执行过程

这个结果与人们的直观预期不太一致. 通常，人们会期待异步通信会比同步通信的性能优良. 针对这个例子，我们认为，造成同步通信的性能更好的原因是，同步发送 RPC 不会触发 PacificA 网路层的流量控制. 使用同步 RPC，系统中正在发送的 RPC 数目最多等于发送线程数(本例是 4 个)，因而不会因为发送消息缓冲被填满而导致所有线程 Sleep 1s 的情况.

将 RPC 通信模式从异步改为同步并没有完全地解决问题. 如果网络发生拥塞导致同步 RPC 调用不能很快返回，则系统处理用户请求的速度会受影响，因而需要重新设计 PacificA 中 RPC 层和网路层代码，改变两个函数抽象层的交互方式，才能从根本上提高系统性能.

5.3　性能评测

Scalpel 使用了插装在系统函数和应用函数调用时输出 trace，这会对系统性能造成一定影响，我们测试了 Scalpel 对系统带来的额外负载. 具体说，我们测试了相同环境与配置下，应用与不应用 Scalpel 对系统性能造成的影响. 对 PacificA，我们测

试了两种情况下执行一次提交数据花费的时间.对Apache实验,我们测试了执行一次SVN Checkout花费的时间.每组测试都运行5次,取平均值作为最终结果.结果显示在表1中.

表1 Scalpel对Apache和PacificA的性能影响

	执行时间(原始)/s	执行时间(Scalpel)/s	额外开销/%
Apache	1.564	2.054	31.33
PacificA	20.792	28.362	36.41

可以看出,对于PacificA和Apache+SVN,由于插装输出trace对性能的影响在可以接受的范围以内.

6 相关工作

已有若干工作使用trace构建因果路径来分析多层结构分布式系统的运行时行为,trace的来源可能是操作系统、网络或应用层.Magpie[4]通过使用用户提供的模式(schema)将系统事件相关联为一组因果路径.Pinpoint[11]通过对代码加标注传播路径的唯一标识,路径的标识和用户请求一一对应.X-trace[12]在网络协议扩展上加入了路径相关的元数据,从而维持了网络请求在网络层处理的因果路径.这些系统都需要开发者提供应用相关的任务结构,包括任务边界、关联标示符和传播规则.本文的方法能够使用一些一般的启发式方法自动推断任务模型,而不需要标注.Project 5[13]和Sherlock[14]将系统看做黑盒,通过分析系统组件之间的网络通信,自动推断系统组件之间的依赖关系.我们的方法也使用了统计推断方法,但是并不限于网络组建.

Whodunit[15]和Data Flow Tomography[16]的工作通过使用虚拟机追踪函数调用引起的数据流依赖关系.这些工作能够给出精确的因果依赖关系,但是监测细粒度的内存操作导致了很大的性能损耗.相对比的,我们的方法仅仅追踪了系统的同步操作,并使用启发推断因果依赖关系.因此相比更加轻量.

Pip[3]和D3S[17]检查用户提供的谓词来发现软件缺陷(bug).Scalpel推断得到任务模型可以应用在这些系统之上,从而更加容易给出与检查谓词.

7 总 结

本文研究了自动推断系统层次结构任务模型的方法.这种方法使用插装技术,透明地监测系统运行,得到系统运行trace.在运行trace基础上,我们能够自动推断出合理的、表达系统设计含义的层次结构任务模型来.在一些复杂系统上的应用实例表明,使用得到的任务模型,能够帮助理解系统设计,解决系统已有的性能问题.

参 考 文 献

[1] Lin W, Yang M, Zhang L, Zhou L. PacificA: Replication in log-based distributed storage systems. Microsoft Research, Technical Report TR-2008-25, 2008

[2] Chang F, Dean J, Ghemawat S, Hsieh W C, Wallach D A, Burrows M, Chandra T, Fikes A, Gruber R E. Bigtable: A distributed storage system for structured data//Proceedings of the OSDI. Seattle, WA, 2006: 205-218

[3] Reynolds P, Killian C, Wiener J L, Mogul J C, Shah M A, Vahdat A. Pip: Detecting the unexpected in distributed systems//Proceedings of the NSDI. San Jose, CA, 2006: 115-128

[4] Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling//Proceedings of the OSDI. San Francisco, CA, 2004: 259-272

[5] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558-565

[6] Caballero J, Yin H, Liang Z, Song D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis//Proceedings of the CCS. Alexandria, VA, 2007: 317-329

[7] Cui W, Kannan J, Wang H J. Discoverer: Automatic protocol reverse engineering from network traces//Proceedings of the USENIX Security Symposium. Boston, MA, 2007: 199-212

[8] Larus J R. Whole program paths//Proceedings of the PLDI. Atlanta, GA, 1999: 259-269

[9] Guo Z, Wang X, Tang J, Liu X, Xu Z, Wu M, Kaashoek M F, Zhang Z. R2: An application-level kernel for record and replay//Proceedings of the OSDI. San Diego, CA, 2008: 193-208

[10] Mai H, Gao C, Liu X, Wang X, Voelker G M. Towards automatic inference of task hierarchies in complex systems//Proceedings of the Hot Topics in System Dependability. San Diego, CA, 2008

[11] Chen M Y, Accardi A, Kiciman E, Lloyd J, Patterson D, Fox A, Brewer E. Path-based failure and evolution management//Proceedings of the NSDI. San Francisco, CA, 2004: 309-322

[12] Fonseca R, Porter G, Katz R H, Shenker S, Stoica I. Xtrace: A pervasive network tracing framework//Proceedings of the NSDI. Cambridge MA: USENIX Association, 2007: 271-284

[13] Aguilera M K, Mogul J C, Wiener J L, Reynolds P, Muthitacharoen A. Performance debugging for distributed systems of black boxes//Proceedings of the SOSP. New York, 2003: 74-89

[14] Bahl P, Chandra R, Greenberg A, Kandula S, Maltz D A, Zhang M. Towards highly reliable enterprise network services via inference of multi-level dependencies//Proceedings of the SIGCOMM. Kyoto, Japan, 2007: 13-24

[15] Chanda A, Cox A L, Zwaenepoel W. Whodunit: Transactional profiling for multi-tier applications//Proceedings of the EuroSys. Lisbon Portugal, 2007: 17-30

[16] Mysore S, Mazloom B, Agrawal B, Sherwood T. Understanding and visualizing full systems with data flow tomography//Proceedings of the ASPLOS. Seattle, WA, 2008: 211-221

[17] Liu X, Guo Z, Wang X, Chen F, Lian X, Tang J, Wu M, Kaashoek M F, Zhang Z. D3S: Debugging deployed distributed systems//Proceedings of the NSDI. San Francisco, CA, 2008: 423-437

GAO Chong-Nan, born in 1981, Ph. D. candidate. His research interests include distributed system management and performance debugging.

YU Hong-Liang, born in 1976, Ph. D., associate professor. His research interests focus on distributed system.

ZHENG Wei-Min, born in 1946, professor, Ph. D. supervisor. His research interests include grid and cluster computing, high performance storage system and biology computing.

Background

The complex systems which support Internet services are hard to debug and analysis. The systems usually involve multiple tiers, a hierarchy of functional abstractions, and high degrees of concurrency. The systems execute user-level tasks, e. g., processing client requests, in a series of stages, which can be distributed across multiple machines, processes, and threads, using events and asynchronous messages as notification mechanisms. Verifying the behavior of such individual tasks therefore becomes a very challenging problem since developers have to reconstruct the task flow by linking together pieces of its execution throughout the system.

Understanding the runtime behavior are key to debug and analyze the systems. However, existing tools require developers to manually specify task models. They use the models to extract system runtime as causal paths and analyze system runtime for correctness and performance bugs.

The authors propose a methodology for automatic inferring task hierarchies in complex systems. Based on instrumentation to monitor system runtime transparently, we can infer leaf task boundaries and associate causal dependencies among them using predefined heuristics. By using clustering algorithm, we can further infer the task hierarchies. The experiences of applying the inference algorithm on real systems show that the inferred task models help both understanding system runtime behavior and performance debugging.