Arachne论文和用户态线程库

1. 背景

- 标题：Arachne: Core-Aware Thread Management
- 会议：OSDI ‘18
- 作者：斯坦福大学 Henry Qin, Qian Li, et al

Arachne was a mortal weaver who challenged the goddess Athena to a weaving competition. Similarly, the Arachne user threading system attempts to challenge the current dominance of kernel threads in the C++ world.

- 摘要：本文是斯坦福大学Henry Qin小组的工作，实现了一个用户级线程调度方案（Arachne），它能依据负载对应用的线程进行CPU核心的调度，使得特定的应用程序（线程生存周期短）同时保持低延迟和高吞吐量。作者使用Memcached和RAMCould进行验证，使得Memcached提高了37%的吞吐量，同时降低了10倍的尾延迟，使得RAMCould增加了2.5倍以上的写吞吐量。项目代码开源在了Github上。

- 问题：传统的线程调度时，会为应用的每个请求创建一个内核线程，将应用进程在CPU上进行调度，这样导致系统资源利用率低下，而且会造成应用程序之间的资源的竞争。

- 方案：提出Arachne，用户级线程库的一个实现，让应用程序“知道”自己所需的CPU核心数，并独占核心，为每个核心创建的一个内核进程，消除与其他应用的资源竞争，最后对应用程序的线程依据负载在CPU核心上进行调度，以优化应用的性能。

- 效果：使得Memcached提高了37%的吞吐量，同时降低了10倍的尾延迟，使得RAMCould增加了2.5倍以上的写吞吐量。

2. 问题

传统应用在应对低延迟和高吞吐量时，很难权衡。大部分时候，应用程序无法告诉os他们需要多少core，也不知道哪些core是分配给自己使用的。因此应用程序无法调整其内部并行度以匹配可用的core，这可能导致部分core的利用率不足或者负载过高，而需要通过负载均衡来维持的话，将带来很多额外的开销，且延迟无法保证。

Arachne被设计用于那些需要处理大量短生命周期请求的服务，同时还要保证较低的延时，比如memcached、RAMCould。Memchached处理一个请求的时间大概是10us，正常情况下，内核线程的开销太大，无法为所有的请求创建单独的线程；所以memcached使用了线程池来处理这个问题，在程序启动时，创建一批线程池，当需要处理新的请求时，由一个分发线程将请求分派到某个工作线程来处理。但线程池的线程数量固定的，在运行时，当可用core比线程数少时，将造成多个线程共用同一个core，这会导致未被调度的线程增加很大的延时，最好的情况是，一个工作线程需要单独使用一个core。

此外，在低负载期间，每个工作线程都可能处于idle状态，这将导致部分core进入节能模式，而从节能模式被唤醒的开销又非常大，这也会增加延时。Arachne尝试在低负载期间在较少的core上高密集的的运行，这也能获取更好的表现。默认情况下，Arachne使用忙转的调度策略，这将使core的占用率达到100%。

3. 方案

- Arachne可以预估应用程序所需的core数量
- Arachne允许每个应用程序定义一个core policy，在运行时按照policy确定应用程序需要多少核心，以及如何在可用核心上放置线程
- Arachne旨在最小化Cache-miss，它使用了一种新的调度信息的新表示形式，为线程创建，调度和同步提供了低延迟和可扩展的机制
- Arachne为每个CPU核心创建一个内核线程，比传统的为每个请求创建一个内核线程更加灵活
- Arachne是用户态实现，不需要修改内核。Arachne应用程序可以与不使用Arachne的传统应用程序共存

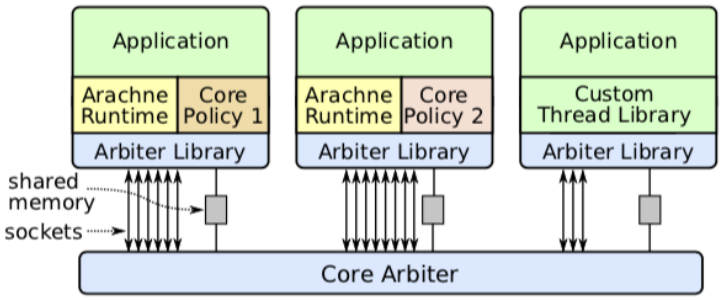


Figure 1: The Arachne architecture. The core arbiter communicates with each application using one socket for each kernel thread in the application, plus one page of shared memory.

Arachne由三部分组成：Arachne runtime、core policy、core arbiter。

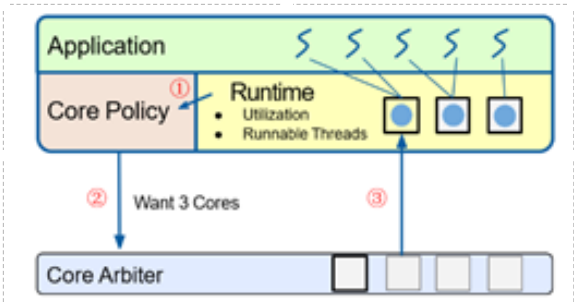
- Arachne runtime是连接到应用程序上的库，负责创建内核线程，应用程序性能的采集，以及应用线程的创建，删除等操作。
- core policy是连接到应用程序上的库，主要负责根据收集到的性能信息，作出资源预估，决定应用所需CPU核心数以及进程应该被分配到哪些核上
- core arbiter通过cpuset实现，主要负责CPU核的分配。应用进程与core arbiter通过socket和share memory进行通信。

Arachne大致流程：

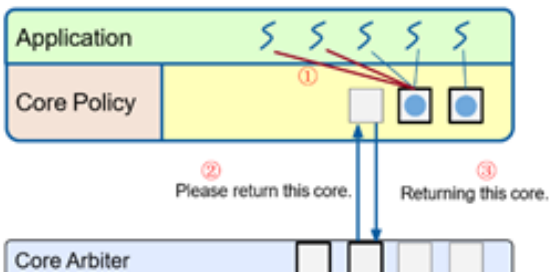
- core arbiter将CPU核分为managed cores（运行Arachne应用）和unmanaged cores（运行其他应用）
- 启动Arachne Runtime获取应用初始所需核心数，Arachne runtime为创建与请求核数相同的内核线程，并阻塞这些线程等待arbiter分配资源
- 当arbiter决定为这些线程分配资源时，它会通过cpuset绑定线程和CPU、通知并唤醒应用程序的内核线程，操作系统会负责将当前线程调度到绑定的CPU上
- Arachne 运行时中的内核线程随后会在调度器中为其他的用户线程分配合适的资源

Arachne分配调整：

- Runtime时刻监视系统性能，主要通过Arachne runtime测量两个指标：
  1）utilization，指的是每个核上执行用户进程的平均时间
  2）runnable threads，指的是每个CPU核心上用户进程的平均数量
- 根据Arachne Runtime测量的数据，core policy产生进行分配与回收CPU核心的策略，通过core arbiter进程对应的操作，这里注意，CPU核回收时并不是直接抢占CPU核心，而是在share memory page中放入等待变量，等待应用响应，若超时未响应，再强制回收
- CPU核心的分配条件是Runnable Threads大于某个阈值，回收条件是utilization小于上次分配核心时的值



CPU核心分配过程



CPU核心回收过程

保持低cache-miss的方法：
Arachne由于跨核通信，可能会产生大量cache-miss，cache-miss的高低是衡量Arachne性能的关键，文章使用以下方法降低cache-miss：

- 在应用线程创建阶段，将线程的许多操作并行执行，例如创建线程与分核操作。
- 在应用线程调度阶段不使用runnable queue，而是对当前核相关的所有线程进行扫描，直到找到一个可以运行的线程为止。

4. 细节

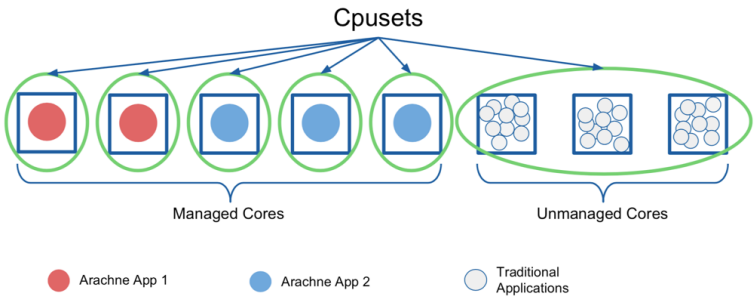
○ core arbiter

特性

核心仲裁者（Core Arbiter）是 Arachne 的核心组件，它是应用程序和操作系统的中间层，主要负责控制系统中的 CPU 并将这些资源合理地分配给不同的应用程序，该系统包含三个特点：

- 在用户态基于 Linux 的机制实现了 CPU 管理；
- 可以与不使用 Arachne 的应用程序同时存在；
- 使用协作式的方法管理内核，包括优先级管理和内核抢占；

仲裁者使用 Linux 的 cpuset 特性将内核分成管理的内核和未被管理的内核，未被管理的内核会分配给机器上的其他应用程序，而被管理的内存会分配给使用 Arachne 的进程。



这种方案可以与现有的进程运行方式兼容，不需要修改内核，就可以直接利用 Linux 现有的特性管理 CPU 资源。

优先级管理

当应用程序启动时，它会调用 setRequestedCores: 方法通过 Socket 向仲裁者请求 CPU 资源，该方法的参数是一个数组，从左到右依次表示八种不同优先级的 CPU：
0 0 1 2 0 0 0 0
仲裁者使用了比较简单的优先级管理策略，高优先级的请求可以覆盖低优先级的请求，因为没有其他调度器的分时复用机制，并且调度基本都是协作式的，所以部分应用程序只要申请高优先级的资源就可以饿死其他进程，Arachne 中并没有解决这个问题，虽然它提出可以通过鉴权防止恶意资源申请，但是这仍然无法解决饥饿问题。

资源申请

当应用程序想要向仲裁者申请 CPU 资源时，会经历如下所示的几个步骤：

- 应用程序在发出 setRequestedCores: 后会即创建与请求核数相同的内核线程（Kernel Thread）；
- 调用 blockUntilCoreAvailable: 阻塞这些线程等待仲裁者分配资源；
- 当仲裁者决定为这些线程分配资源时，它会通过 cpuset 绑定线程和 CPU、通知并唤醒应用程序的内核线程，操作系统会负责将当前线程调度到绑定的 CPU 上；
- Arachne 运行时中的内核线程随后会在调度器中为其他的用户线程分配合适的资源；

每一个基于 Arachne 运行时的应用程序中都会启动对应数量的内核线程，运行时中的调度器会决定执行哪些用户态线程，该模型与 Go 语言的 G-M-P模型也比较相似，它们的初衷都是尽可能降低线程切换带来的额外开销，但是 Arachne 提供了对资源更细粒度的控制。
需要注意的是仲裁者和应用程序之间的通信机制是不同的，应用程序会通过 Socket 访问仲裁者，而仲裁者发出的请求都是通过共享内存。这是因为应用程序在发出请求后可以陷入休眠等待响应或者资源的分配，但是 Socket 是相对比较昂贵的操作，所以仲裁

者会通过共享内存请求应用程序。
Arachne 使用了协作的方式进行调度，当仲裁者需要回收资源时，应用程序可以延迟一段时间释放 CPU；如果应用程序超过 10ms 都没有释放资源，仲裁者就会通过 cpuset 将资源分配给其他程序，没有主动释放的程序会看到性能明显地下降。

Arachne runtime

Arachne 的运行时实现了用户态线程，但是这里的用户态线程针对特定的场景做出了优化，它主要支持细粒度的计算，其中包括大量生命周期极短的线程，例如常见的 HTTP/RPC 服务，它们会创建单独的线程处理外部的请求。
大多数的线程操作都需要在多个 CPU 之间通信，而跨核的通信会导致缓存缺失（Cahce Miss），缓存缺失大概需要 50 ~ 200 个 CPU 循环，这也是影响 Arachne 运行时性能的主要因素。为了解决缓存缺失带来的性能影响，运行时会在数据传输时并行执行其他的指令，优化 CPU 缓存以提升用户态线程的性能。线程的创建和调度是运行时的关键操作，优化这些操作的性能就可以降低延迟并提高吞吐量。

线程创建

多数的用户态线程管理器都会在同一个 CPU 上创建线程并使用工作窃取（Work-stealing）平衡多个 CPU 上的负载，但是工作窃取对于存在时间较短的线程来说是非常昂贵的，所以我们希望在线程创建时立刻触发负载均衡，将线程创建在其他的核心上；同时，为了减少程序中的缓存丢失，Arachne 将每一个线程的上下文都绑定了特定的 CPU 上，只有在仲裁者回收时才可能发生处理器迁移，大多数的线程在创建之后就不会触发迁移。当应用程序创建新的线程时：

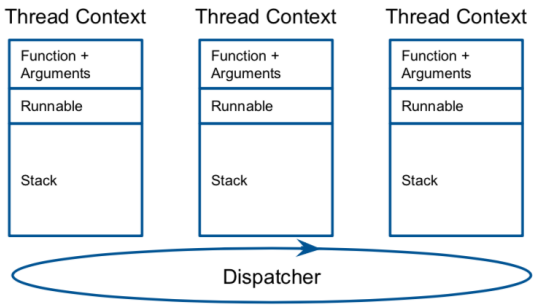
- 运行时会在多个 CPU 中随机选择两个；
- 在上述两个 CPU 中选择活跃线程数较少的 CPU；
- 将线程开始的方法地址和参数拷贝到上下文中；

每个 CPU 上都有对应的 maskAndCount 变量，其中存储着正在执行的线程和当前 CPU 上的线程数。为了减少线程的缓存丢失，我们使用单独的缓存块（Cache Line）存储线程所有信息，这样创建线程最少只需要触发四次缓存丢失，分别是读取 maskAndCount、传输函数地址、参数和调度信息，能够最大限度的减少开销。

线程调度

传统的调度模型会在运行时中引入准备队列（Ready Queue）保存可执行的线程，例如：Linux 和 Go 语言的调度器，但是如果准备队列是跨 CPU 的，那么增加或者删除任务时都需要修改多个变量以及获取共享锁，这都会触发缓存丢失进而影响性能。
为了减少程序中的缓存丢失，Arachne 的调度器不会使用准备队列，它会持续检查当前 CPU 上的所有活跃线程直到发现可以运行的线程，因为以下的两个原因，这个看起来非常简单的轮训机制实际上非常高效：

- 在同一时间，单个 CPU 上应该只会包含少数几个线程上下文；
- 当前线程一定会由其他核心唤醒，因为跨核的传输会带来缓存丢失，而在触发缓存丢失时可以并行扫描全部上下文不会带来过大的开销；



因为线程可能处于阻塞状态等待特定的执行条件满足，所以上下文中会包含 wakeupTime，即线程多久后需要被唤醒；线程可以通过 block(time) 和 signal(thread) 两个方法改变 wakeupTime 通知调度器当前线程的状态和唤醒时间。

core policy

为了能够让应用程序对CPU有更细粒度的控制，Arachne不会在运行时中指定CPU的使用策略，如何使用CPU都是由独立的策略模块确定的，Arachne中的一些默认核心策略如果不能满足应用程序的需求，它们还可以实现一些自定义的策略满足特定的需求。
与 Linux调度器中的调度类比较相似，当应用程序创建新的用户线程时，它需要指定该线程的类，核心策略会根据调度类选择 CPU 执行。默认的核心策略中包含两个线程类，分别是独占的（Exclusive）和正常的（Normal），前者会为线程保留整个CPU 资源，而后者会在多线程之间分享资源。

资源预估

默认的核心策略中包含动态的 CPU 资源预估功能，它会使用以下的三个参数根据过去一段时间的负载和 CPU 使用情况调整当前应用程序申请的资源：

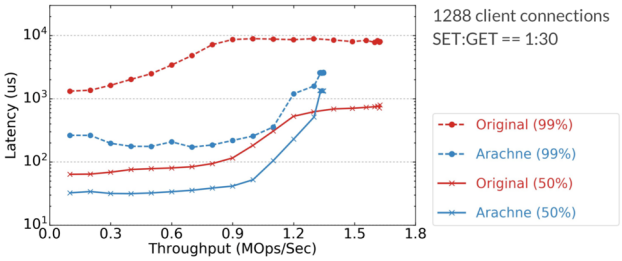
- 负载因子（Load Factor） — 当运行正常线程的所有 CPU 负载达到特定的阈值时，向仲裁者申请额外的 CPU；
- 时间间隔（Interval）— 用于预估占用资源的CPU采样区间，默认为过去的 50ms，这可以决定策略对负载变换的敏感程度；
- 滞后因子（hysteresis Factor） — 记录每次提升负载前的资源利用率，并根据该利用率减少申请的 CPU 资源；

作为 Arachne 中的默认策略，它提供的一定是相对简单的、普适的模型，我们可以根据自己的需求调整策略中的三个参数，也可以实现其他的策略。

5. 测试结果

- **Configuration (CloudLab m510)**
  - 8-Core (16 HT) Xeon D-1548 @ 2.0 Ghz
  - 64 GB DDR4-2133 @ 2400 Mhz
  - Dual-port Mellanox ConnectX-3 10 Gb
  - HPE Moonshot-45XGc
- **Experiments**
  - Threading primitives
  - Latency vs Throughput
  - Changing Load and Background Applications

Operation	Arachne	Go	uThreads	std::thread
Thread Creation	320 ns	444 ns	6132 ns	13329 ns
Condition Variable Notify	272 ns	483 ns	4976 ns	4962 ns



6. 思考🤔

Arachne 与其他线程管理器使用了完全不同的设计思路，因为它的目的是充分利用 CPU 资源，所以应用程序需要对资源有着更细粒度的掌握和控制，其他的线程管理器都会尽可能地屏蔽底层的实现细节，让上层只关注内部的一些逻辑。

从软件工程的角度也不能说谁对谁错，但是在真正追求极致的性能时，一定清楚对下一层甚至下两层的信息，在线程调度器这个场景下就是 CPU 资源的详细使用情况。

- 只是对线程CPU核心进行调度，并没考虑到其他系统资源，如LLC。
- 本文没有考虑机器的异构性，和应用程序的多
- arachne很多方面做的很粗糙，都是采用最简单的策略，

参考

1. https://draveness.me/papers-arachne/#fn:5
2. https://draveness.me/system-design-scheduler/