向迁移线程模型演变的马赫 Bryan Ford Jay Lepreau

犹他大学

摘要

我们对Mach进行了修改，将跨域远程过程调用RPC视为一个单一的实体，而不是一连串的消息传递操作。 通过这样提升的RPC，我们通过改变线程模型来改善RPC期间的控制权转移。我们用迁移线程替换了Mach的静态线程，以试图隔离操作系统设计和实现的这一方面。我们设计的关键因素是将线程抽象解耦为执行上下文和由上下文链组成的可调度控制线程。新系统为线程操作和额外的控制操作提供了更精确的语义，允许调度和会计属性跟随线程，简化了内核代码，提高了RPC性能。我们的结论是，迁移线程模型优于静态模型，内核可见的RPC是这种改进的先决条件，以这种方式改进现有操作系统是可行的。

介绍和概述

我们首先定义并解释了四个概念，这些概念是本文的关键，它们是内核和用户线程远程过程调用静态线程和迁移线程。

在传统的Unix中，一个进程只包含一个内核提供的线程 Mach和其他许多现代操作系统支持每个进程和每个任务的多个线程，在Mach的术语中称为内核线程 它们与用户级线程包提供的用户线程不同，后者通过操纵来自用户空间的程序计数器和堆栈在内核提供的线程上实现多个控制线程。

在大多数操作系统中，线程所包括的内容远远超过了控制权。例如，在Mach中，线程还包括：i是具有优先权和调度策略属性的可调度实体；ii包含资源核算统计，如累积的CPU时间；iii包含计算的执行环境、寄存器的状态、程序计数器堆栈指针以及对包含任务和指定异常处理程序的引用；iv通过线程控制端口提供给用户程序可见的线程控制点。

这项研究得到了惠普研究资助计划和OSF研究所的部分赞助。

在Mach中，端口是一个内核实体，它是一种能力，一个通信通道和一个名称。

RPC 远程过程调用，顾名思义，是过程调用抽象的模型，但在不同的任务之间实施，控制权被暂时转移到另一个地方，被调用的过程随后返回到原点并继续进行 RPC 可以在远程计算节点之间使用，但通常在同一节点上的任务之间使用 本地RPC 本文只关注本地情况，在本文的其余部分，为了简洁起见，我们使用不完全的术语RPC，只指本地RPC，比通常的限制更多。当客户端任务中的一个线程需要另一个任务提供的服务时，比如打开一个LE，它就会捆绑一个数据包，其中包含服务提供者--服务器或LE 系统在这种情况下需要处理的所有内容，这就是所谓的 marshaling消息，然后客户端线程调用内核，将消息复制到服务器的地址空间，允许服务器开始处理它。

值得注意的是，尽管几乎所有的现代操作系统都在某种程度上提供了RPC抽象，但这种支持是连续的。 一些操作系统将RPC作为一个完全的内核可见实体来支持。
阿米巴(Amoeba) 有些提供了一个内核接口，并对RPC 中涉及的综合消息发送和接收进行了特别优化，但从根本上说，只理解单向消息传递的马赫，而有些只通过分层在其他进程间通信IPC 设施上的库支持RPC Unix

静态线程 静态线程和迁移线程之间的区别在于客户端处理和服务器处理之间的控制转移是如何实现的 在基于静态线程的RPC 中，涉及两个完全独立的线程，一个是客户端任务中的静态线程，另一个是服务器任务中的线程。这个线程被称为服务线程，是由服务器先前创建的，其唯一目的是等待RPC 请求并处理它们。 当服务线程处理完请求后，它调用内核，使服务器线程恢复睡眠，并再次唤醒客户线程。

在基于静态线程的RPC 中，服务器的计算资源（使用CPU 的权利）被用来向客户提供服务。 在面向对象的世界中，这被称为活动对象模型，因为服务器对象包含主动提供服务的线程。

迁移线程 如果RPC 对于内核来说是完全可见的，那么就可以实现另一种控制转移的模式 迁移线程允许线程从一个任务转移到另一个任务，作为其正常功能的一部分 在这种模式下，在RPC 期间，内核不会在客户线程的IPC 内核调用中阻塞它，而是安排它继续在服务器的代码中执行。而是安排它继续在服务器的代码中执行 没有服务线程需要被内核唤醒，相反，为了RPC 的目的，服务器只是一个被动的代码库，由客户线程来执行。
被动对象模型 只涉及部分上下文切换，内核切换地址空间和CPU 寄存器的一些子集，如用户堆栈指针，但不切换线程或优先级，也不涉及调度器 没有服务器线程状态寄存器堆栈可以恢复 客户端自己的计算资源对CPU 的权利被用来为自己提供服务 注意这个模型中的绑定可以与线程切换模型中的绑定非常相似，将在本节详细介绍。

尽管大多数操作系统使用静态线程模型支持RPC，无论内核是否可见，但重要的是要注意，这不是所有系统服务的情况。 所有使用进程模型执行自己的内核代码的系统，例如Unix Mach Chorus Amoeba

，实际上在内核调用期间将用户线程迁移到内核地址空间，没有上下文切换发生，只有堆栈和权限级别被改变，用户线程的资源被用来为自己提供服务。

人们可以对它所代表的对象进行操作

有时，对调度器的调用被大量优化并内联到IPC路径中，但它仍然存在。

进程模型与V操作系统的中断模型相反，V操作系统的内核代码必须在潜在的阻塞之前明确地保存状态。

静态线程与迁移线程 实际上，这两种模式之间有一个连续的过程。例如，在一些系统中，如QNX，某些客户线程属性（如优先级）可以传递给
或者服务线程在客户端调用之间可能不保留任何状态，只为执行提供资源，如Peregrine RPC系统。

在Mach上提供迁移线程

在我们的工作中，我们将线程抽象的这些语义方面解耦为两组，并增加了一个新的抽象--激活栈，它记录了RPC产生的客户服务器关系。
具有优先权和资源核算属性的可调度实体 激活代表 i 计算的执行环境，包括它正在执行的代码的任务、异常处理程序、寄存器和堆栈指针，以及 ii 用户可见的控制点。

对应于旧的线程抽象，输出给用户代码的抽象现在是我们内部所说的激活。 这不仅对用户程序的需求有意义，而且还提供了与原Mach的兼容性。

通过使RPC成为内核的一个单一的识别实体，并明确记录激活堆栈中各个激活之间的关系，我们已经将RPC提升为内核完全可见并支持的实体，而不是一连串的消息传递操作。 我们的线程抽象现在更接近于线程的原始概念基础，即控制的逻辑性。事实证明，提升线程和RPC的抽象也增强了可控性，因为内核现在可以对单个激活或整个线程采取更详细和精确的行动，例如，它可以沿着激活链传播信息警报。 另外，向内核引入任务间RPC概念的好处是，许多积极的IPC优化成为可能。

目标和益处概要

我们在这个项目中的最初目标有几个 我把马赫的线程模型改成了一个可迁移的模型
在设计和实现过程中，我们发现我们可以实现更多的东西 iv 基于普通消息的完全映射的RPC变得更快 v 线程可控性得到加强 vi 内核代码变得更简单 vii 实现了静态线程与迁移线程的苹果式比较 viii 以下讨论的其他几个优势变得很明显。

在本文的其余部分，我们详细描述了这项工作。我们首先讨论了相关工作，然后介绍了迁移线程模型的优势，描述了我们的内核实现和接口，包括对线程可控性问题的讨论，研究了RPC在新系统中的工作原理，并提到如何改变Unix服务器以更好地利用迁移线程。

相关工作

大多数操作系统使用静态线程模型，但也有一些例外 Sun的Spring操作系统支持与我们非常相似的迁移线程模型，尽管它使用了不同的术语 Spring的梭子对应于我们的线程，而他们的线程对应于我们的激活 Spring解决了可控性问题，但不需要关注向后兼容性 Alpha可能是第一个完全采用迁移线程的系统。在这两个系统中，线程可以在分布式环境中跨节点迁移，事实上，Alpha对迁移

线程的称谓是分布式线程 Psyche是一个支持迁移线程的单一地址空间系统。

迁移线程的控制传输是其设计的一个关键部分，但它专注于高性能的本地RPC，并包括额外的数据传输优化，这使得隔离改进的控制传输的好处变得很困难 面向对象的系统在传统上区分了主动和被动。

被动对象对应于静态和迁移线程模型 Clouds是被动对象迁移线程模型的例子，而Emerald和我们一样提供主动和被动对象，支持两种执行方式 Chorus只能在用户级任务之间使用线程切换，但在内核保护域中运行的任务之间，它有消息处理程序，在迁移线程模型中操作。

我们认为，许多这些迁移的线程系统没有完全解决随之而来的控制性问题，没有完全支持调试或需要提供Unix信号语义，例如

调度器激活是对用户级调度有特殊支持的内核线程 调度器激活主要关注内核线程在保护域内的行为，而我们的工作是处理跨保护域的线程行为，因此这些工作在很大程度上是正交的，理论上可以结合在同一个系统中，但我们在此不处理这个问题。

除了Taos上的LRPC，所有支持迁移线程的现有系统从一开始就是这样设计的，它们在线程模型之外的许多方面都与传统操作系统不同。

激励

迁移线程的方法有几个优点，在本节中进行了概述。大多数优点与RPC的使用有关，并在前面进行了描述。

远程程序调用

迁移线程的许多优势来自于它们与RPC的结合使用 迁移线程提供了一个更合适的底层抽象，与静态线程相比，可以在此基础上构建RPC接口 静态线程的许多问题来自于控制模型之间的语义差距，一个控制线程内的过程调用抽象和用于实现该模型的两个线程的机制。由于RPC的使用非常频繁，特别是在较新的基于微内核的操作系统中，大多数内部系统的交互都是基于RPC的，因此系统的这一方面对于决定整个系统的性能和功能非常重要。

呼叫的有效性

对于在静态线程模型中执行的RPC，每个任务中的两个线程必须在内核中进行同步。 在操作过程中，需要进行两个线程间的上下文切换，一个在调用时，一个在返回时。然而，在迁移线程模型中，整个RPC可以由一个线程执行，它临时进入服务器任务，执行请求的操作，然后带着结果返回到客户端任务。

线程迁移也允许进行优化，比如在LRPC和其他结构清晰或共享地址空间的系统中，如Lipto FLEX和Mach的内核服务器中，这些系统有一定程度的域间内存共享或保护放松，从而模糊了域的边界 由从一个域迁移到另一个域的线程实现的RPC可以利用这种边界模糊，在参数传递和堆栈

处理方面提供许多优化 在迁移线程模型中实现的极限RPC可以被专业化为简单的过程调用

这些优势一般适用于任何服务调用机制，而不仅仅是通过RPC调用的大型服务器。例如，在基于对象的环境中，如果所有对象都必须是主动的，那么对相对单一的对象进行调用就显得非常不方便。 对于被动对象，将类似的调用抽象应用于中等和中等规模的对象是比较可行的。

虽然基于静态线程的快速便捷的微内核的存在表明，在这种模式下，高性能是可能的，但这些系统通常会施加语义限制，使其实现方式向迁移线程模式转变。进程与进程之间的消息传递与优先级继承，这种设计使其成为事实上的迁移线程系统。 其他微内核，如L，保留了静态线程的全部语义，但为了实现高性能，必须对调度的灵活性和系统中与RPC不直接相关的其他方面施加严格限制。

线程属性和实时服务

在静态线程模型中，当客户端任务执行RPC时，控制权被转移到一个完全不同的线程，该线程有自己的调度参数，如执行优先级以及其他属性，如资源限制，除非采取特定的行动，否则服务器中线程的属性将与客户端线程的属性完全不相关。另一方面，如果客户线程迁移到服务器上执行操作，所有这些属性都可以得到适当的维护，而不需要额外的费用。

一个相关的优势是在资源核算方面，由于在服务器上代表客户所做的工作可以自动归属，所以可以使资源核算更加准确。

RPC期间的中断

通常，由于异步条件的存在，人们希望中断一个客户端被暂时或永久阻塞的RPC，为了在静态线程模型中干净地做到这一点，仅仅中止消息发送和接收操作是不够的，因为服务器将继续处理请求，而没有任何迹象表明客户端不再希望完成它。此外，每个可能被访问的服务器都必须支持这些终止操作，这在实践中是很难保证的，特别是如果任何用户模式的任务可以将自己设置为服务器，并允许其他用户线程在Mach允许的情况下向它发出RPC。

服务器简化

在模拟单片机操作系统（如Unix或OS）的个性服务器中，我们希望迁移线程能够简化服务器，因为服务器所基于的原始操作系统可能已经使用了有限的迁移线程模型，其中线程迁移到单片机内核中进行系统调用 在个性服务器中保持这种模型应该能够实现更大的代码再利用，并简化对系统调用中断的处理 线程管理和控制机制，如Unix信号

我们还希望通过迁移线程来简化服务器中的RPC服务，因为预计激活池的管理比线程池更简单，如本节所述。

内核RPC路径简化

正如我们在后面的章节中所展示的那样，迁移线程极大地简化了内核RPC路径，基于迁移线程的RPC路径往往很短，而且很自然，而基于静态线程的优化RPC路径往往很长，而且包含无数的测试。

线程可控性和内核简易化

迁移线程的实现提供了与RPC无关的其他可控性和简化性。 在静态线程模型中，线程通常被认为是完全可控的资源。

能够在任何时候任意地停止一个线程并修改其状态 从概念上讲，线程只执行用户模式的指令，因此，从来没有任何时候，系统的完整性会被线程的操作所破坏。

不幸的是，这种最纯粹的模式在实际操作系统中是行不通的 线程必须能够调用内核级代码，以便与系统中的其他实体进行通信，如果它们要做的不仅仅是纯粹的计算，因为执行未知内核代码的线程可能不会被任意操纵，完全可控的模式必须在某种程度上被打破，必须能够在必要时推迟或拒绝线程控制操作。

例如，Mach提供了一个线程控制操作，该操作中止了目标线程被阻塞的系统调用，从而使该线程可以被操纵。然而，许多内核操作不能以透明的可重新启动的方式中止，因此试图控制该线程的实体可能不得不等待任意长度的时间或重试任意数量的次数，才能安全地做到这一点。

由于完整的可控性模型是不现实的，减少模型的范围以允许迁移线程，为线程操作提供了更精确的语义。事实上，通过强迫可控性的边界被明确定义，并记录跨任务的控制权，额外的线程控制机制，如跨域警报可以由内核提供。

内核实施

在这一节中，我们描述了我们在Mach微内核中迁移线程的实现的基本结构。我们使用的许多技术也可以类似地应用于其他传统的多线程操作系统，如单片机的Unix内核。

课题实施

从概念上讲， 传统的Mach用户线程开始在一个特定的任务中执行，并偶尔进入内核与外部实体进行通信，内核随后从系统调用中返回并恢复用户代码，线程的初始和正常位置是在用户空间，线程只是偶尔访问内核以请求服 务 。

在我们的迁移线程实现中，情况在某种意义上是相反的。线程开始时是作为一个纯粹的内核模式实体执行的，后来会向上调用到用户空间来运行用户代码。 概念上，内核是所有线程的大本营，用户级代码唯一被执行的时间是在临时进入任务的时候。

当内核中的线程现在可以上调到用户空间时，传统的内核用户界面仍然被保留下来。一旦线程在用户空间执行，它可以以陷阱和异常的形式回调到内核。

内核在概念上是一个保护域，很像一个用户级别的任务，线程可以在其中执行等待迁移，等等，它的主要区别是它有特殊的权限，提供基本的系统控制服务。

这是因为这两种类型的代码通常在监督者模式下执行，并且通常在一个二进制映像中链接在一起。然而，情况并不一定是这样的，例如在QNX中，K微内核基本上只由胶水代码组成，而内核本身则被放在一个有特殊权限的普通进程中。

控制抽象和机制

即使在静态线程模型中，在实践中也不能完全实现线程完全可控的目标，虽然线程在内核中的情况在某种程度上可以作为一种特殊情况来解决，但随着迁移线程的增加，可控性问题必须得到更仔细的考虑。例如，如果客户机中的一个线程迁移到服务器中进行RPC，那么同一客户机中的另一个线程在执行服务器代码时就不允许停止或操纵第一个线程的CPU状态。

为了同时提供可控性和保护，我们把线程的概念分成两部分，一部分是调度器使用的部分，另一部分是提供显式控制的部分。第一部分我们仍然称为线程，它在任务之间迁移，进入和离开内核。

每当一个线程迁移到一个任务中，包括线程创建时来自内核的初始上调，一个激活就会被添加到线程的激活栈的顶部。 当一个线程从迁移中返回时，相应的激活会从激活栈中跳出。

激活是在线程创建过程中隐式创建的，或者是由服务器显式创建的，期望能接收到进入的迁移线程。显式创建的激活在一个线程迁移到任务中并激活它之前是不被占用的。

在内核中，对激活的控制主要是通过异步过程调用或APC来实现的，类似于单片机内核中的异步陷阱AST，当从内核返回到激活时，胶水代码会检查连接到激活的APC，如果有的话就调用它们。以前，Mach将线程暂停作为调度器的一部分来处理，给其已经很复杂的状态机增加了更多的复杂性。

内核堆栈管理

由于激活链可以在任何时候中断，所以激活之间的所有联系信息都存储在激活本身中，而且整个线程只需要一个内核栈就够了。这种明确的状态保存一般被称为使用延续，尽管我们的实现与过去Mach中使用延续的方式非常不同。

可控性语义接口和实现

在这一节中，我们描述了线程控制操作的语义、这些操作的接口以及实现的一些方面。我们相信我们的方法也可以类似地应用于其他传统的多线程操作系统。

线程控制接口

在最初的Mach内核中，线程是以线程控制端口的形式输出给用户模式程序的，通过这些端口可以调用控制操作，在我们的系统中，当线程仍然存在的时候，控制端口就会出现。

我们通过让激活控制端口直接替代二进制级别的线程端口来保持与现有Mach代码的兼容性，所有以前期望或返回线程端口的系统调用现在都使用激活端口。

警报

在我们的迁移线程实现中，我们提供了Mach的线程中止调用的功能，它中止了一个正在进行的内核操作，并将控制权返回给用户代码。它们本身并不提供对线程的控制，因为它们没有强制力，警报只是请求而不是要求。

默认情况下，添加到线程堆栈的新激活的警报被阻止，以防止干扰不明真相的服务器的运作。内核已经能够尊重大多数警报，为与迁移线程一起工作而编写的新服务器也可以被设计为尊重它们。 在这方面，我们提供了一个通用的中断请求机制，它对迁移的RPC和内核调用都是统一的。

我们还提供了另一个操作，它首先在目标激活处生成一个警报，然后中断链，立即将控制权返回给客户端。

悬浮液

在马赫线程语义中，暂停一个线程的基本目的是防止它执行任何更多的用户模式指令，直到它被恢复。例如，当线程暂停时，可以允许显式设备I O继续进行，但内核的copyin和copyout操作不能隐含在任务的地址空间中。

我们目前的实现允许这样的内核活动继续进行，我们不希望这在实践中成为一个问题，但在任何情况下，它应该作为相关工作的副产品来解决。 在这项工作中，我们进一步将内核代码与前面描述的胶水代码分开。

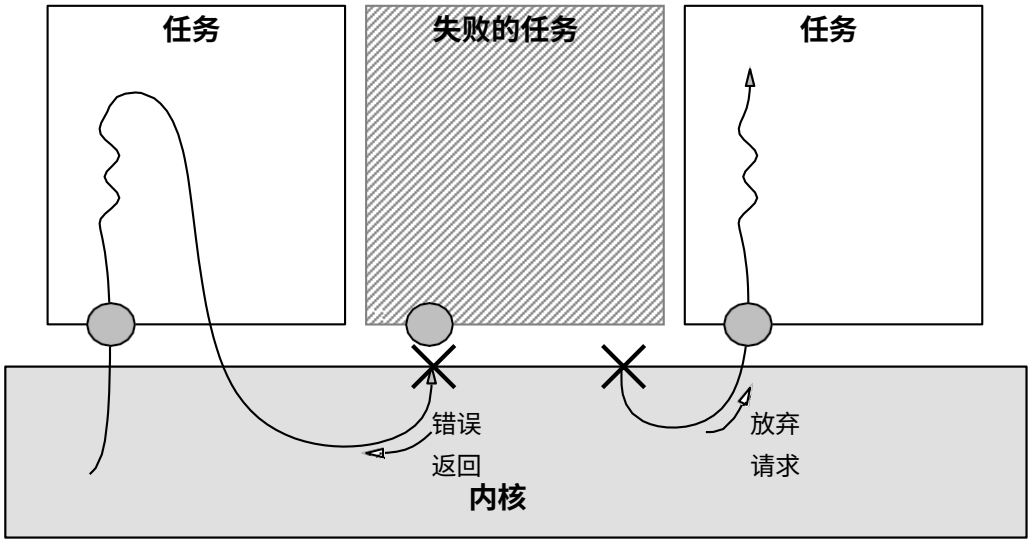
为了在这个模型中保持正确的悬浮语义，内核系统调用对调用者任务的隐式引用必须与胶水代码相联系。这在Mach中相对容易做到，因为大多数内核调用被实现为对内核对象的通用RPC，只有低级的RPC代码需要遵守控制语义，而不是实现内核调用的实际代码。

终止

在我们的实现中，终止机制如图所示 如果一个不在线程激活栈顶部的激活被终止，或者如果线程当时处于内核调用中，那么线程就会分裂成两个具有相同调度参数的独立线程 一个线程被留在激活栈的顶部部分，另一个线程被赋予底部部分 处于上部部分的线程继续在最顶部的服务器中不间断地执行，确保线程终止不违反保护。

请注意，传统的Unix暂停语义比Mach暂停语义更强，它们意味着在线程暂停之前，内核操作实际上已经完成或中止了。 在Mach上，无论是静态线程还是迁移线程，实现Unix暂停语义除了简单的Mach暂停之外，还涉及其他Mach控制操作。

图 激活 终止



这个线程提供了一个提示，即正在进行的工作可能不再有价值了 给予激活栈底层的线程返回到它现在最顶层的激活，并有一个适当的错误代码 这与Spring 中使用的终止机制基本相同

在我们的系统中，不仅当线程以用户模式运行在比被终止的激活更近的激活中时，而且当线程代表被终止的激活在内核中执行时，都可以被分割。在新的模型中，如果被终止的激活恰好在调用内核，那么它就会被留下来完成它正在执行的任何操作，并在之后安静地自我毁灭。

我们的终止机制需要对内核的数据结构和锁定机制进行仔细的规划，特别是内核和胶水代码之间的界限必须被精确的定义。 然而，这种技术不仅增加了额外的可控性，而且大大简化了内核中控制机制的实现，正如我们在章节中所展示的那样。

CPU状态

最初的Mach设计提供了线程操作，在理想的完全可控性模型中，允许线程的整个CPU状态在任何时候被保存、恢复和修改。然而，由于完整的可控性模型的问题，许多通常被认为有用的CPU状态控制机制实际上无法在Mach中以有限的时间可靠地实现。例如，为检查点或运输到另一个节点的任务状态的封装可能需要控制线程等待一个任意长的时间，否则需要中止内核操作，产生潜在的不准确状态。

由于现有的CPU状态控制操作已经存在问题，要实现对它们的完全向后兼容是很困难的，我们选择在我们的迁移线程实现中对这些操作进行结构化，以适应当前对这些操作的使用，特别是那些由Unix服务器进行的操作。

我们正在考虑提供替代的终止语义，它完全保留了RPC的同步性。被终止的激活将仅仅被接出，因此只有在链中较晚的激活退出后，控制才会返回到较早的激活。

仿真器和应用程序创建自己的线程，并以直接的方式控制它们 马赫要求在检查或设置线程的状态之前调用线程中止。

除非线程刚刚被创建，否则状态操作可能会在过时的信息下工作。

在迁移线程时，在操作其状态之前中止激活并不是严格要求的，如果不这样做，CPU状态操作会耐心等待，直到线程处于目标激活状态，而不干扰其运作。 因此，我们放松了对这些操作的限制，同时保持与它们在原始Mach中的唯一有效用法的向后兼容性。

调度参数

必须映射到激活操作的Mach线程控制操作是那些管理线程调度参数的操作，如优先级调度策略和CPU使用统计与上述操作不同，这些操作在概念上仍然是对线程而不是激活进行的。

在我们目前的实现中，由于每个活动的激活都精确地附着在一个线程上，我们将线程操作导出为对激活的操作，并在内核中将操作重定向到附着的线程。在我们最初的实现中，这在实践中并不是一个问题，因为Unix服务器是被所有客户信任的。

任务控制界面

大多数任务控制操作在迁移线程下的工作方式与在原始Mach de sign中的工作方式相同，其他的则以直接的方式进行修改，以匹配新的线程模型。特别是任务线程调用现在返回一个任务的激活端口列表，而不是线程端口。

迁移RPC

一旦支持迁移线程的基本内核机制到位，就需要证明它对RPC性能和复杂性的影响。因为我们目前的重点主要是在RPC过程中的控制传输，我们最初的实现保留了基于用户模式存根的marshaling和unmarshaling的原始Mach数据传输接口。

客户端

从客户的角度来看，包括绑定在内的RPC语义是不存在的。 现有的二进制文件中的正常机器味精调用是被支持的，内核检查消息选项以确保它们指定了一个真正的RPC，并检查目标端口以确保服务器有能力处理迁移的RPCs。

服务器端

初始化服务器以支持迁移RPC的方式与只支持线程切换RPC的服务器几乎相同 在完成绑定的主要部分时，服务器完全像以前一样向客户输出发送权限 此外，服务器必须创建一个或多个未被占用的激活，每个激活包含一个指向其自身地址空间中的堆栈的指针，在绑定的最后部分是入口点

向内核提供这些信息可以被封装在一个函数中，例如在cthreads包中。

传统的静态线程RPC仍然被自动支持 不再需要大量的服务器线程池，但至少必须有一个线程来处理偶尔的异步消息，因为在目前的实现中，当不能使用迁移RPC时，这被用作一种后备机制。

当一个迁移的RPC进入服务器时，内核从服务器池中分配一个未被占用的激活，将传入的消息复制到服务器堆栈中，并向上调用服务器任务中的调度例程，这个MIG生成的例程与用于调度传统消息的例程相同，只是它通过一个特殊的内核入口返回。

如果一个迁移的RPC被尝试，而内核发现目前没有可用的激活，在我们最初的实现中，内核会退回到正常的消息路径，导致一个正常的消息被排到端口上，这并不理想，我们计划在最后一个可用的激活即将被用于一个迁移的RPC时进行检测，而不是立即让所请求的RPC暂时
在这一点上，如果服务器认为有必要，它可以创建更多的激活，如果它这样做了，它将这些激活返回给内核，原来的RPC可以继续进行，否则它立即返回，RPC被阻塞，直到堆栈被释放。

当这一点被实现时，服务器对激活池的管理应该比对线程池的管理要简单得多，原因如下：资源将由客户线程自己按需分配，而不是由服务器提前预测资源需求。与线程池相比，服务器被内核墙与客户端请求隔开，如果服务器线程的数量不足，客户端消息就会在服务器的队列中堆积，而服务器并不知情，因此服务器的工作更加复杂，因为它试图保持等待请求的线程数量等于或略大于处理器的数量，它必须跟踪在服务器中运行的等待消息的线程数量，并在传出的RPC或内核调用中被阻塞。如果不这样做，就会导致死锁 最后，大量的复杂性是由于在内核线程上复用cthreads，而这是迁移线程无法做到的。然而，如果发现有必要限制特定服务器中执行线程的总数，以避免由于过度的内核上下文切换而导致饱和，那么这种简化就不会出现了。

用户层面的线程问题

迁移RPC最重要的问题是，在Mach上最广泛使用的用户级线程和同步包cthreads在迁移RPC的情况下有很大的限制。

服务器线程管理 cthreads库给迁移RPC的服务器带来了很大的问题 服务器使用cthreads将用户线程复用到内核线程之上，尽可能用更快的用户级上下文切换取代内核模式上下文切换。用户级线程包的一个主要假设是，它所运行的所有内核线程的用户级线程都是可以互换的，即一个内核线程可以和另一个内核线程一样用于某个操作。

在迁移线程模型中，从客户端迁移进来的内核线程是不能互换的，它们可能有不同的优先级和其他属性，即使忽略这一点，在RPC被处理后返回内核时，也必须在RPC进来的同一内核线程上进行。

当像通常那样对RPC存根生成器接口进行编程时，这不是对绑定语义的改变。

怀疑论者应该检查OSF服务器的UX服务器循环和相关代码，在这些代码中，所列出的复杂性被认为对性能和安

全是必要的。

实体的激活堆栈代表了一个正在进行的特定工作，即一个完整的逻辑控制线程。

然而，由于服务器中的同步操作有时需要内核级上下文切换，而不是用户级上下文切换，因此在消除用户级线程复用时，会损失一些速度。 我们认为，在典型的RPC服务器中，用户级上下文切换的速度优势并不像在计算密集型应用中那样显著，后者是线程实施的传统基准。我们指出，在许多基于微内核的商业系统中，包括QNX Chorus和KeyKOS操作系统，服务器通常不会在多个内核线程上复用用户级线程。然而，在对使用迁移RPC的服务器进行更广泛的性能分析之前，在服务RPC时丢失用户级线程仍然是一个令人担忧的问题。

请注意，只有从其他任务迁移过来的客体线程才会出现用户级线程复用的问题，服务器上的原生线程仍然可以使用某种用户级线程系统，甚至是专门的复用机制，如调度器激活。

一个更合适的同步系统 由于cthreads不能再在服务器中的内核线程上复用用户级线程，它应该被一个更好地优化的同步库所取代，以提供对内核线程的同步 同样，内核可见的同步对于完全实现优先级继承也是必要的，我们在一篇较长的论文中讨论过。

Unix服务器

为了在新内核上使用传统的RPC功能，OSF单一服务器和仿真器以及它们使用的库都不需要改变。现有的对服务器线程池的复杂管理，虽然基本上不再使用，但还是保留了下来。 我们没有对模拟器进行修改，因为我们为线程操作提供了向后兼容的语义，不需要对现有的处理Unix信号的复杂代码进行修改。

即使我们的实现被调整了，我们也没有预计到单个服务器的性能会有很大程度的提高，因为它是在假设RPC非常昂贵的情况下编写的，因此服务器尽可能地避免RPC，而采用其他方法，如共享内存页，其性能不会因为迁移RPC而提高。然而我们最初的目标主要不是显示性能的提高，而是证明迁移线程所带来的简单性和清洁性的提高，以及如何在现有操作系统中以向后兼容的方式实现迁移的线程。

我们预计，通过两个利用迁移线程的修改，可以使Unix服务器变得更简单。一个是模拟Mach下的Unix信号，在另一个是由于Mach没有提供将中止请求传播到RPC的标准方法，Unix服务器必须手动处理所有的Unix系统调用中断，比如那些由待定信号引起的中断。它应该通过利用现在由内核提供的传播中止操作来大大简化。这也将使中断语义自然地扩展到系统中的其他服务器，如在Unix下运行的Mach speci c应用程序所安装的服务器。

结果

状况

我们已经完成了本文所描述的系统的内核实现 一个基于仿真器的Unix服务器在新的微内核上正常运行，使用传统的线程切换RPC 一个服务器被修改以提供激活，因此它使用迁移RPC运行多用户Unix信号，包括C和Z正在工作

实验环境

所有的时间都是在一台有MB RAM的HP上收集的，这台机器有一个Mhz的PA RISC处理器， K o chip Icache K o chip Dcache entry ITLB 和 entry DTLB，页面大小为K。缓存是直接映射和虚拟寻址的，缓存失误成本约为周期。 RPC测试时间是通过读取PA的时钟寄存器收集的，该寄存器每周期递增，可以在用户模式下读取。

系统软件是我们对Mach内核版本NMK和基于仿真器的OSF 单服务器版本b 的移植，编译器是GCC u，具有完全优化功能。

RPC路径分解和分析

为了分析下一节中介绍的空RPC的加速次数，我们沿着内核的空RPC路径手工计算了指令数。

内核不再需要在每个RPC上保存和恢复服务器的寄存器，这来自于内核对RPC的第一手知识，它不再需要创建翻译和消耗回复端口，以便将回复与它的请求相匹配。从迁移线程 的优化控制传输来看，切换激活比切换线程要简单得多，这种改进是由于更简单的数据管理，特别是消除了在内核的地址空间中维护一个临时消息库的需要，因为可以直接从源点复制到目的地。

有趣的是，现在有将近一半的RPC迁移成本停留在内核客户端边界上
因此，进一步改进内核RPC路径的其他部分，可能只会导致最小的整体速度提升。

表 Null RPC路径的细分和改进

舞台	石 建筑物计数		改进 说明		改进 负载商店
	开关	迁移	孽子 镁	燕 镁 子	燕 镁 子
内核进入出口 客户端 内核进入退出 服务器端 港口翻译 螺纹激活开关 信息副本					
整个内核路径					

表的最后两栏显示了每个阶段的改进情况，以加载存储操作的数量来衡量，由于内存子系统的原因，这些操作对总周期的贡献不成比例。

乍一看，这似乎是完全正交的，但是在切换路径上，信息的复制是在最开始，而复制到目的地是在最后，中间有数百行复杂的代码。

成本 我们观察到，RPC的每个阶段在指令数和内存操作方面的改进百分比几乎是相等的，这表明指令数是衡量每个阶段对整体性能增益的相对贡献的有效标准。

对旧的优化RPC路径中的上下文切换代码的检查解释了它的大部分成本，内核基本上执行了调度器的一部分，特别是手工编码的内联。 众多的约束条件必须得到满足，新旧线程必须处于正确的状态，运行和等待队列必须正确维护端口上的锁。线程的IPC空间和其他数据结构必须以正确的顺序获取和释放，以避免死锁 计时器被操纵 中断级别被改变 沿途获得的资源必须被仔细跟踪，以确保在计算因某种原因落入优化路径时有可能解开一切

表中显示了每个路径的指令组合，分为三类 总指令 负载 存储和分支 迁移路径的负载和存储的比例更高一些

这可能是由于IPC寄存器保存和恢复、内存复制和数据结构遍历等基本内存密集型方面较少被计算开销所掩盖。

表Null RPC路径指令混合

舞台	开关			迁移		
	全部	装载存储	分公司	全部	装载存储	分公司
内核进入出口 客户端 内核进入退出 服务器端 港口翻译 螺纹激活开关 信息副本						
整个内核路径						

我们希望我们在RPC路径上的结果一般会扩展到除PA RISC之外的其他架构上，尽管有时会出现问题，但大多数架构可以在日期连续时实现单一的直接拷贝，例如通过临时映射改变中断优先级IPL，由于调度器的参与，在切换路径上要做四次，但在迁移路径上不需要。虽然IPL的改变在PA RISC上很便宜，但在其他一些架构上却非常昂贵，因此迁移线程在这些架构上特别重要。地址空间切换的不可避免的成本在一些架构上要高得多，这将导致较低的改进率，但即使如此，我们预计其好处也是相当大的。

微观和宏观基准结果

表中列出了跨任务迁移和传统切换RPC的成本测量结果。左边的一列只包括RPC的内核成本，而右边的一列包括在另一组运行中获得的内核和用户调集成本。迁移线程的速度随参数大小而变化，从空RPC的系数到K数据的系数，再到长线marshaled数据的系数，这个系数是由于数据在切换路径中被复制了三次，一次是marshaling，两次是在内核中，但在完整的迁移路径中只有两次。

一个有趣的现象是，迁移路径上的每条指令的周期数CPI比原始路径上的要差得多。我们认为，这部分或全部是由于两个因素造成的，一是上文所述的负载存储指令的比例较高，二是手工编

码的迁移路径上的指令没有像C语言编译器对大部分旧路径那样进行仔细的调度和优化。

表RPC的周期时间

测试	内核时间			内核时间用户管理		
	切换	迁移	比率	切换	迁移	比率
空的RPC 在 K 在 K 在						

对K迁移RPC的内核时间的测量显示了惠普直接映射的高速缓存的严重副作用。 上面是我们的原始测量，一个可疑的低时间导致了相当令人难以置信的速度改进。

作为对整体性能影响的初步测试，我们测量了气体汇编器的制作时间 在迁移线程的情况下，耗时从秒到秒，提高了大约几秒钟 链接阶段花了大约几秒钟 一个较大程序的链接，惠普链接器本身从秒到秒，提高了几秒钟 我们相信这个更大的改进是由于ld的系统调用与计算的比例更高

我们放慢速度的一个领域是对内核的RPC，这些目前还没有迁移，因为我们还没有改变内核，以便在其端口上提供激活，我们预计这样做不会有什么问题。

我们希望经过调整的实现能够实现更多的整体加速，如果通过迁移线程而实现的其他RPC优化，则会有明显的加速。

内核代码简化

可控性使线程独立于任务，并且在用户模式之外不可控制，这在很多方面大大降低了代码的复杂性。 在新的K源代码中，包含大多数线程控制操作的源代码从K到K减少了一半以上。原来的Mach线程控制机制必须对特殊情况进行大量的测试，例如一个线程操纵自己或两个线程试图同时控制对方，可能会导致内核死锁。 现在，可控性被控制在明确的边界内，因为它必须支持迁移线程，这些棘手的情况永远不会发生，因为内核代码总是在边界之外。

由于类似的原因，任务管理代码从K个减少到K个， 更清洁的模型简化了锁定，并消除了许多特殊情况，如线程终止自己的任务的情况。

迁移RPC 在最初的切换路径上，端口转换和上下文切换代码大多是用独立于机器的C代码编写的，而其他类别的代码则是PA规格的汇编语言。由优化的RPC路径组成的整行复杂的C代码加上大约手工编码的汇编语言指令被替换成了大约汇编语言指令，由此产生的逻辑复杂度简化了9倍，这一点从本节的图中可以看出。

其余的用于分配和释放激活和其他与控制操作无关的管理。

虽然我们认为汇编编码带来了一定的速度提升，但由于迁移路径的简单性，这只是可行的。

内存使用

在最初的微内核中，由于延续机制的存在，每个处理器只需要几个内核栈K。在这个项目开始的时候，为了简化我们的工作，我们禁用了延续机制，这立即将内核内存的使用提高到每个线程一个内核栈。当运行我们的多用户基准时，我们观察到内核物理内存的使用比原来的系统最多增加了K，但是我们正在新的模型下重新引入连续性，所以即使增加也应该是暂时的。

关于服务器虚拟内存的使用，在写这篇文章时，我们静态地分配了大量的激活量
当我们移除线程池时，服务器的虚拟机使用量应该与旧系统中的差不多，因为在大多数情况下，每个服务器线程用户栈都变成了一个服务器激活用户栈。

未来的工作

这项工作使Mach和Mach服务器得到了许多进一步的改进，同时也提出了进一步研究的领域 为cthreads同步原语提供一个适当的替代品是很重要的，以便对依赖内核级上下文切换的影响进行公平的评估 我们早期关于将可信服务器移入内核的保护域和地址空间的工作INKS使用了临时的线程迁移 通过从头开始重新进行线程抽象，我们的新系统解决了所有遇到的问题

NORMA NO远程内存访问版本的Mach允许在微内核中实现的分布式内存多处理器的不同节点之间进行IPC，应该将迁移线程系统扩展到包括节点之间的RPC，所涉及的问题已经在Alpha中进行了深入探讨。

在不同的方向上，我们的工作允许改进Mach对实时系统的支持 在实现层面上，我们已经在很大程度上将线程抽象的两个部分解耦，可调度实体的优先级调度策略等与控制线程的激活链解耦，这使得将它们完全解耦，使优先级继承的全面实现变得可行。

迁移线程模型使其他设计能够提供更高的性能，如LRPC。在保护域被合并的情况下，许多复制可以被避免。

迁移RPC机制也可以用于线程异常处理，这将允许无仿真器服务器（如OSF MK）进行更方便的参数复制。 我们相信迁移RPC也可以通过使Mach寻呼机接口与服务其自身页面故障的线程同步来加以利用，这需要在涉及不受信任的寻呼机时明确提供安全性。

OSF研究所正在采用我们的代码，并计划在英特尔基础上进行上述许多改进。

总结

我们从我们的工作中得出了三个主要的结论 首先，通过改变现有操作系统的线程模型并对两个版本进行评估，我们发现迁移的线程模型优于静态模型 迁移的线程提供了卓越的功能性能和代码简化 在功能方面，线程迁移i为线程操作和额外的控制操作提供了更精确的语义 ii允许调度和其他属性跟随线程，这一点对实时系统尤其重要在性能方面，线程迁移一是提高了普通RPC的性能，

二是实现了多种积极的RPC优化，特别是在目前研究的提供跨域内存或地址空间共享的系统中。然而，线程迁移确实有潜在的性能缺点，即不允许服务于RPC的用户级线程在多个内核线程上进行复用。

在这些领域中，我们的实施和测量已经证明了第一种好处，而第二种好处似乎很明显，但还没有通过在我们的系统中全面实施来实现。

其次，由于迁移线程需要内核将本地RPC视为一个可识别的语义实体，我们得出结论，操作系统内核应该直接支持RPC抽象。

我们的第三个主要结论是，通过将线程模型从静态改为迁移，至少改善一些现有的操作系统是可行的，即使在Mach的情况下，它有一个异常丰富的线程操作接口，我们表明，这种深远的变化可以在保留向后兼容性的同时，只需要适度的实现e ort。

鸣谢

我们特别感谢Mike Hibler在实现方面的专业帮助，以及对可控性和信号问题的讨论。我们感谢Douglas Orr提供的各种帮助，以及Greg Minshall对早期草案的仔细审查。

参考文献

Thomas E Anderson Brian N Bershad Edward D Lazowska and Henry M Levy Scheduler acti vations E ective kernel support for user level management of parallelism ACM Transactions on Computer Systems February

J S Barrera A fast Mach network IPC implementation In Proc of Second USENIX Mach Symposium pages

Brian N Bershad Thomas E Anderson Edward D Lazowska and Henry M Levy 轻量级远程过程调用 ACM Transactions on Computer Systems February

Andrew D Birrell 用线程编程的介绍 技术报告 SRC DEC系统研究中心 一月

A P Black N Huchinson E Jul H Levy and L Carter 在Emerald的分布和抽象类型 IEEE Trans on Software Engineering SE

Alan C Bomberger and Norman Hardy The KeyKOS nanokernel architecture In Proc of USENIX Workshop on Micro kernels and Other Kernel Architectures pages Seattle WA April

John B Carter Bryan Ford Mike Hibler Ravindra Kuramkote Je rey Law Jay Lepreau Douglas B Orr Leigh Stoller and Mark Swanson FLEX 构建高效和可行系统的工具 In Proc Fourth Workshop on Workstation Operating Systems October

D R Cheriton The V distributed system Communications of the ACM March

Roger S Chin和Samuel T Chanson基于分布式对象的编程系统ACM Com puting Surveys 三月

David D Clark 使用向上调用的系统结构 在ACM操作系统原理研讨会上的发言 华盛顿州Orcas岛 12月

Raymond K Clark E Douglas Jensen and Franklin D Reynolds An architectural overview of Alpha real time distributed kernel In Proc of USENIX Workshop on Micro kernels and Other Kernel Architectures pages Seattle WA April

Michael Condict 个人通信 11月

Sadegh Davari和Lui Sha 实时系统中无限制的优先级倒置的来源和可能的解决方案的比较研究 ACM Operating Systems Review April

Richard P Draves Brian N Bershad Richard F Rashid and Randall W Dean Using continuations to implement thread management and communication in operating systems In Proc of th ACM Symposium on Operating Systems Principles Asilomar CA October

Peter Druschel Larry L Peterson and Norman C Hutchinson Beyond micro kernel design Decou pling modularity and protection in Lipto In Proc of th International Conference on Distributed Computing Systems pages Yokohama Japan J u n e

Partha Dasgupta等人 《云计算分布式操作系统的设计与实现》计算系统冬 季 展

Bryan Ford Mike Hibler和Jay Lepreau 关于Mach中线程模型的说明 技术报告 UUCS 犹他大学 计算机科学系 4月

Bryan Ford和Jay Lepreau 发展Mach以使用迁移线程技术报告UUCS 犹他大学11月

Graham Hamilton and Panos Kougiouris The Spring nucleus a microkernel for objects In Proc of Summer USENIX Conference pages Cincinnati OH June

Dan Hildebrand QNX 的架构概述 USENIX 微内核和其他内核架构研讨会论文 华盛顿州西雅图，4 月。

D B Johnson和W Zwaenepoel The Peregrine high performance RPC system Software Practice and Experience February

Jay Lepreau Mike Hibler Bryan Ford和Je Law 在内核服务器上的Mach实施和性能 在Proc of Third USENIX Mach Symposium pages April

Jochen Liedtke 通过内核设计改进IPC 在ACM操作系统原理研讨会上的发言 北卡罗来纳州阿什维尔 12月

开放系统基金会和卡耐基梅隆大学MACH内核接口

Simon Patience 在Mach中重定向系统调用 仿真器的替代方案 In Proc of the Third USENIX Mach Symposium pages Santa Fe NM April

M Rozier V Abrossimov F Armand I Boule M Gien M Guillemont F Herrmann C Kaiser S Langlois P L ！ eonard and W Neuhauser The Chorus distributed operating system Computing Systems December

Michael L Scott Thomas J LeBlanc和Brian D Marsh Psyche的设计原理 一个通用的多处理器操作系统 在Proc of International Conference on Parallel Processing pages A u g u s t

Daniel Stodolsky J Bradley Chen and Brian N Bershad Fast interrupt priority management in operating system kernels In Proc of Second USENIX Workshop on Micro kernels and Other Kernel Architectures San Diego CA September

Vrije Universiteit Amsterdam NL The Amoeba Reference Manual Programming Guide rpc manual page ftp cs vu nl amoeba manuals pro ps Z

作者信息

Bryan Ford是犹他大学计算机科学系的本科生，他目前的主要研究兴趣是改进Mach，但他也追求其他的兴趣，包括数据压缩语言图形和音乐。他是几个广泛使用的Amiga软件包的作者，包括XPK压缩包和MultiPlayer音乐程序，Bryan是Mach迁移线程的设计师和主要实现者。

Jay Lepreau是软件科学中心的助理主任，该中心是犹他州计算机科学系的一个研究小组，从事系统软件的许多方面的工作。他从Unix开始工作，曾担任USENIX会议的联合主席和许多其他USENIX计划委员会的成员。他的小组对BSD和GNU软件发行做出了重大贡献。

作者地址：软件科学中心 犹他大学计算机科学系

他们可以通过电子方式联系到baford lepreaug cs utah edu。

Unix是USL的商标OSF是开放软件基金会的商标OS是IBM的商标