ghOSt框架提供了一个丰富的API，**可以从用户空间接收进程的调度决策并将它们作为事务来执行**。ghOSt提供了一个高性能框架，用于将线程调度策略委托给用户空间进程，从而实现策略优化、无中断升级和故障隔离。

**概述**

ghOSt提供了通用的调度策略并托管给userspace进程。ghOSt提供状态封装、通信和action机制，允许在用户态代理中表达复杂策略。程序员可以使用任何语言来开发和优化策略。

**背景与目标**

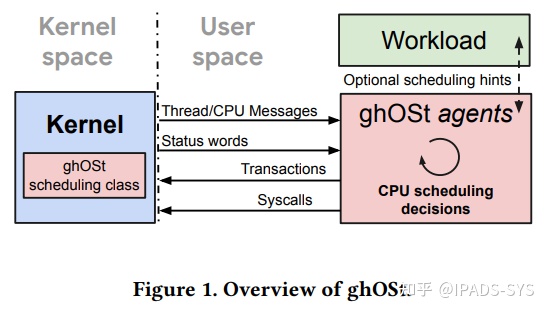
本文提出ghost系统的目标：

1. 调度策略应当实现和测试容易
2. 调度需要有效、广泛
3. 突破per-CPU调度模型的限制
4. 支持同时多个调度策略
5. 升级不需要重启host和错误隔离

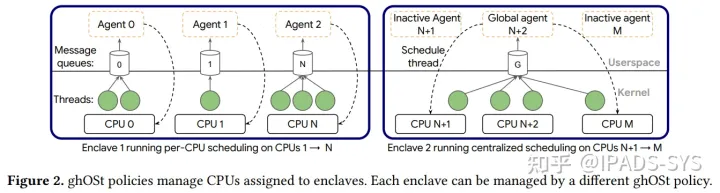
**设计与实现**

**ghost代码是如何实现的？**

如下图所示。userspace的agent做出调度决定并且指导kernel如何在CPU上调度native线程。ghost的kernel部分被实现为一个调度class，就类似常用的CFSclass。这个schedulingclass向userspace提供丰富的API来定义任意调度策略。为了帮助agents做出调度决定，kernel向agent导出线程的状态，通过messages和statuswords。agents然后通过transactions和systemcalls向kernel传递数据。



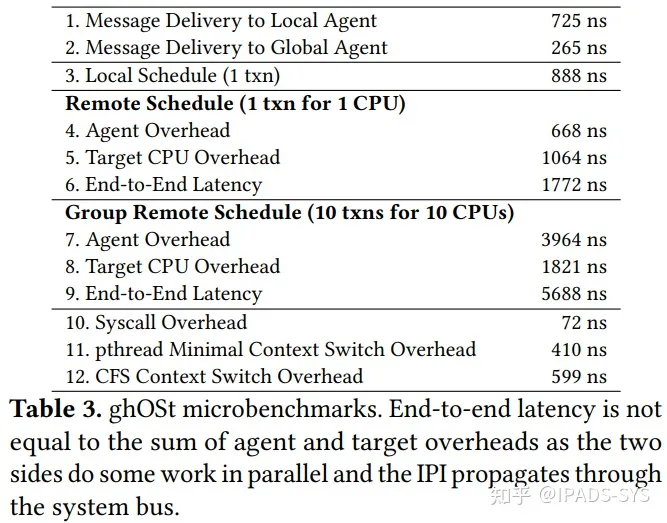
整个系统的调度架构如下图所示，ghost可以设置起多个enclave，每个enclave中可以运行不同的调度策略。ghost内核模块会将调度的决定逻辑托管给运行在每个物理CPU上的ghostagent。



如果该enclave采用per-CPU的调度策略，则ghost内核模块会向各个cpu上的ghostagent分别提供必要的线程信息。如果该enclave采用Centralized的调度策略，则该enclave中仅有一个ghostagent会生效，所有的调度信息也仅会提供给它。

**测试评估**

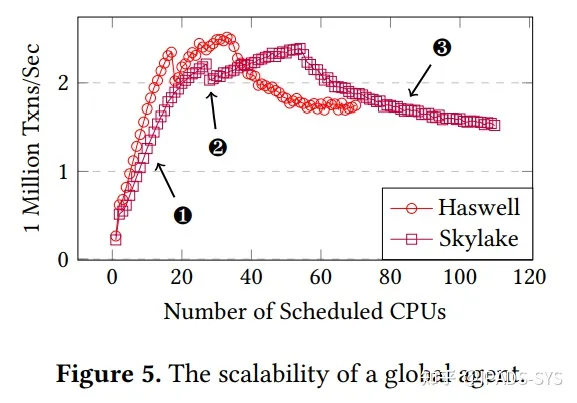
**ghost的额外开销：**首先思考下，哪些操作在传统调度器里是没有的，仅仅只存在于ghost？那么这些操作就是overhead，如下图所示：



**表3.ghOSt微基准。由于agent和target并行工作且IPI通过系统总线传播，端到端延迟不等于agent和target开销的总和。**

**Centralized场景下globalagent的可扩展性：**根据agent overhead可以计算出在这个intel机器上如果用global agent占用一个核心来管理的话，可以大约每秒在100个核心上各管理252000个任务，每个任务每次被调度后运行40us。

当然上面的只是理论上，本文实验了下global agent，实验机器配置为2\*socket，28core/socket，2logicalcore/core。



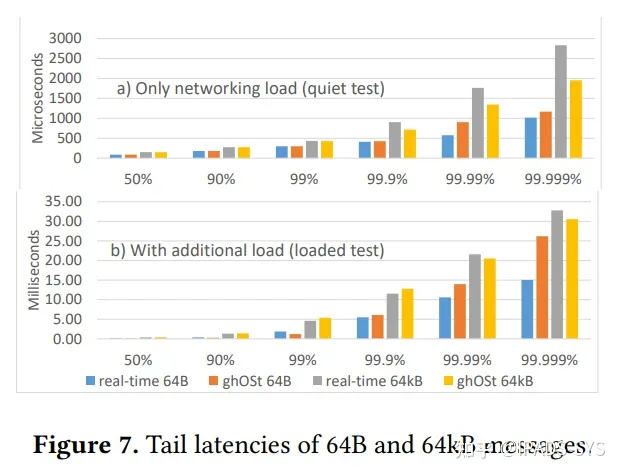
**图5.全局代理的可伸缩性**

第一段上升说明了越来越多cpu可以被用来调度

第二段下降了一下是因为此时已经是测试机单个socket的28核打开了超线程，允许global agent和别的ghost线程一起运行在同一个物理核上，所以global agent虽然受影响了，但是还能继续上升。

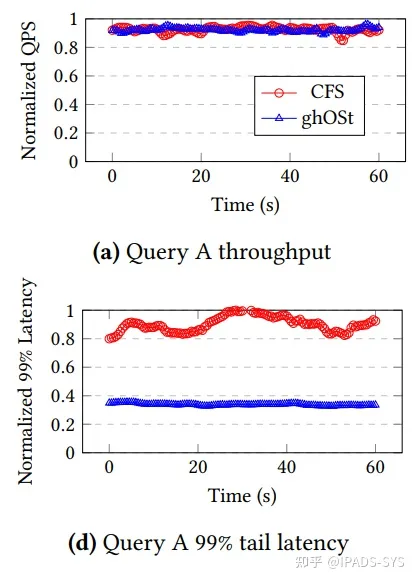
第三阶段，等到了56cpu左右，超线程也用完后开始跨socket进行调度，这时候受NUMA影响，agent对numa节点上的CPU进行调度时需要内存操作核发送跨numa sockets的interrupt，从而导致每次操作开销都更大，引起agent的吞吐量下降。

**与谷歌内部的MicroQuanto对比：**google把ghost和内部的kernel scheduler MicroQuanta进行对比，这个MicroQuanta是一个软实时的kernel scheduler用来管理snap框架，这个snap框架类似dpdk，需要维护一些polling线程来和网卡交互或是实现一些定制化的网络和安全协议。snap会根据网络负载来新增或减少polling的线程。



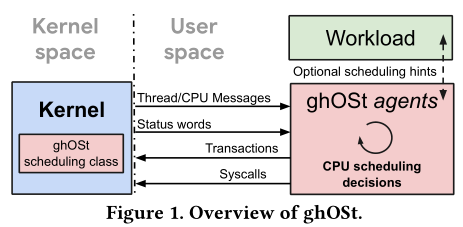
测试结果如上图7所示,在处理64B的网络包时ghost的时延会比MicroQuanta差10%是因为小数据包的处理时间很短所以ghost进出userspace的开销就会变得明显。处理64KB大小的网络包时，ghost的时延会更好则是因为ghost能跨cpu进行负载均衡。

**在谷歌搜索业务场景下的测试：**在谷歌搜索的workload上（数据库查询workload）进行测试，当前google search使用的是CFS，实验环境下换成ghost后经过优化，可将ghost的吞吐量达到与原本CFS相媲美的程度，但是能够在latency上有40-45%的提升，作者也将它归因于是**ghost的global agent能够带来比传统per-CPU调度更好的负载均衡**。



**——————————————————————————————————**

3设计



ghOSt概览。图1总结了ghOSt设计。用户空间代理做出调度决策，并**指示内核**如何在cpu上调度本机线程。ghOSt的内核端被实现为一个调度类，类似于常用的CFS类。这个调度类为用户空间代码提供了丰富的API来定义任意的调度策略。为了帮助代理做出调度决策，内核通过消息和状态词向代理公开线程状态(详见3.1)。然后，代理通过事务和系统调用(详见3.2)指示内核进行调度决策。

在本节中，我将使用两个激励人心的例子:per-CPU调度和集中式调度。经典的内核调度策略(如CFS)是per-CPU调度器。虽然这些策略通常使用负载平衡和工作窃取来平衡整个系统的负载，但它们仍然是从per-CPU的角度来操作的。

集中调度的例子类似于Shinjuku[25]、Shenango[1]和Caladan[21]中提出的模型。在这种情况下，有一个单一的全局实体不断地观察整个系统，并为其权限内的所有线程和cpu做出调度决策。

~~线程和cpu。ghOSt在cpu上调度本机线程。本节中提到的所有线程都是本地线程，与§2中提到的用户级线程相反。我们将逻辑执行单元称为cpu。例如，我们认为具有56个物理核和112个逻辑核(超线程)的机器有112个cpu。~~

对机器进行分区。ghOSt使用enclave在一台机器上支持多个并发策略。一个系统可以按CPU粒度划分为多个独立的飞地，每个飞地运行自己的策略，如图2所示。从调度的角度来看，这些enclave是隔离的。分区是有意义的，特别是当在一台机器上运行不同的工作负载时。通过机器拓扑设置这些enclencle的粒度通常很有用，例如per-NUMA-socket或per-AMD-CCX[40]。**enclave还有助于隔离错误，限制代理崩溃对它所属的enclave的伤害**(参见§3.4)。

ghOSt用户空间代理。为了实现我们的许多设计目标，调度策略逻辑是在用户空间代理中实现的。这些代理可以用任何语言编写，并通过标准工具进行调试，使它们更容易实现和测试。**为了实现容错和隔离，如果一个或几个代理崩溃，系统将退回到默认的调度器，比如CFS**。然后，在启动新的ghOSt用户空间代理时，机器仍然完全正常工作-要么是最后一个已知的稳定版本，要么是带有修复的更新版本。

多亏了崩溃恢复属性，更新调度策略相当于重新启动用户空间代理，而不必重新启动机器。这个属性允许针对各种硬件和工作负载进行试验和快速的策略定制。开发人员可以进行策略调整，并简单地重新启动代理。ghOSt策略的动态更新在§3.4中讨论。

不管调度模型是per-CPU调度还是集中式调度，ghOSt管理的每个CPU都有一个本地代理，如图2所示。在per-CPU情况下，每个代理负责自己CPU的线程调度决策。在集中式情况下，单个全局代理负责调度飞地内的所有cpu。所有其他本地代理都是不活动的。每个代理都在Linux pthread中实现，所有代理都属于同一个用户空间进程。

**3.1内核到代理的通信**

将线程状态暴露给代理。为了让代理为其权限范围内的线程做出调度决策，内核必须向代理公开线程状态。一种方法是将现有的内核数据结构内存映射到用户空间，比如task\_structs，这样代理就可以检查它们来推断线程状态。然而，这些数据结构的可用性和格式因内核和内核版本而异，用户空间策略实现与内核版本紧密耦合。另一种方法

是通过sysfs文件公开线程状态，在/proc/pid/…时尚。然而，文件系统api对于快速路径操作来说效率很低，很难得到支持 s-scale策略:open/read/fseek，最初是为块设备设计的，太慢且太复杂(例如，需要错误处理和数据解析)。

最终，我们需要一个既快速又不依赖底层内核线程实现的内核用户空间API。受到分布式系统的启发，我们使用消息作为一种高效而简单的解决方案。

**ghOSt信息**。在ghOSt中，内核使用表1中列出的消息通知用户空间代理**线程状态的更改**。例如，如果一个线程被阻塞，现在准备运行，**内核会发布一条THREAD\_WAKEUP消息**。此外，内核用TIMER\_TICK消息通知代理计时器滴答。为了帮助代理验证他们正在根据最新的状态做出决策，消息也有序列号，后面会解释。

**消息队列**。消息通过消息队列传递给座席。在ghOSt下调度的每个线程都被分配到一个队列，关于该线程状态变化的所有消息都被发送到该队列。在每CPU示例中，每个线程被分配到一个队列，该队列与它打算运行的CPU对应(图2，左)。在集中式的例子中，所有线程都被分配到全局队列(图2，右)。CPU事件(如TIMER\_TICK)的消息被路由到与CPU相关的代理线程的队列。

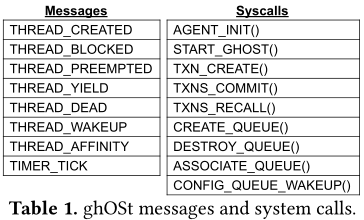
虽然实现队列的方法有很多，但我们选择**在共享内存中使用自定义队列**来高效地处理代理唤醒(下文将进行解释)。我们认为现有的队列机制对于ghOSt是不够的，因为它们只存在于特定的内核版本中。例如，**BPF系统通过BPF环缓冲区将BPF事件传递给用户空间**[65]，**最近版本的Linux也通过io\_urings将异步I/O消息传递给用户空间**[66]。这些都是同步消费者/生产者访问的快速无锁环缓冲区。然而，旧的Linux内核和其他操作系统不支持它们。

**Thread-to-queue协会**。ghOSt飞地初始化后，飞地中只有一个默认队列。代理进程可以使用create/DESTROY\_QUEUE()API创建/销毁队列。添加到ghOSt的线程被隐式分配给将消息发送到默认队列。代理可以通过ASSOCIATE\_QUEUE()来改变这个分配。

**Queue-to-agent协会**。队列可以有选择地配置为当消息产生到队列中时唤醒一个或多个代理。代理可以通过CONFIG\_QUEUE\_WAKEUP()配置唤醒行为。在每CPU示例中，每个队列只与一个CPU关联，并被配置为唤醒相应的代理。在集中式的例子中，队列由全局代理连续轮询，因此唤醒是冗余的，因此没有配置。产生消息到队列中并在代理中观察它的延迟在§4.1中讨论了。

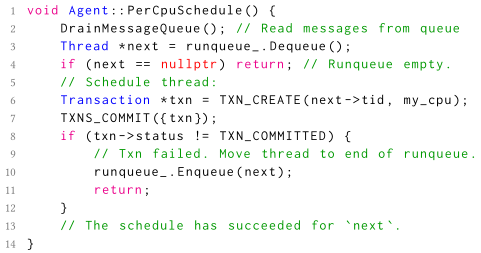
代理唤醒使用标准的内核机制来唤醒阻塞的线程。这包括识别要唤醒的代理线程，将其标记为可运行的，可选地向目标CPU发送中断以触发重新调度，以及执行到代理线程的上下文切换。

在队列/CPU之间移动线程。在我们的每个CPU示例中，为了在CPU之间实现负载平衡和工作窃取，代理可以更改消息的路由



通过ASSOCIATE\_QUEUE()从线程到队列。这取决于代理实现(在用户空间中)，以适当协调跨队列到代理的消息路由。如果一个线程将其关联从一个队列更改为另一个队列，而原始队列中有挂起的消息，则关联操作将失败。在这种情况下，代理必须在重新发出ASSOCIATE\_QUEUE()之前清空原始队列。将代理与内核同步。代理操作通过消息观察到的系统状态。然而，当代理正在做出调度决策时，新的消息可能会到达队列，从而改变该决策。对于每个CPU示例和集中式调度示例，这个挑战略有不同(见§3.2和§3.3)。无论哪种方式，我们都用代理/线程序列号来解决这个挑战:每个代理都有一个序列号， ，每当消息被发布到与该代理关联的队列时，该序列号都会递增。的使用 对于§3.2中的每CPU示例。每个线程 都有一个序列号，𝑇 ，每当该线程发布一个新的状态改变消息时，该数字就会递增， 。当一个代理弹出队列时，它会收到一条消息和它对应的序列号:(𝑇, )。我们解释如何使用 对于§3.3中的集中式调度示例。通过共享内存公开序列号。幽灵使代理能够通过状态字有效地轮询有关线程和CPU状态的辅助信息，这些信息映射到代理的地址空间中。为了简洁起见，我们只讨论我们使用状态词来暴露序列号， 和𝑇 ，送给代理商。当内核更新线程或代理的序列号时，它也会更新相应的状态词。然后，**代理可以从共享映射中的状态词中读取序列号**。

**3.2Agent-to-Kernel通信**



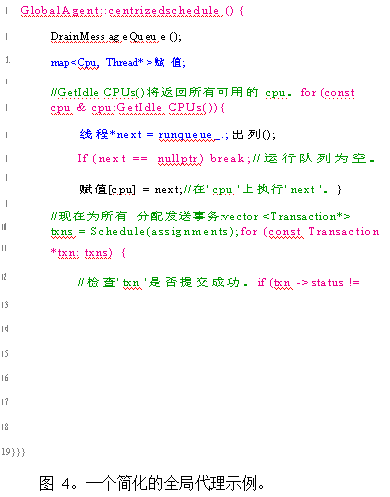
我们**现在描述代理如何指示内核下一个调度哪个线程**。

通过事务发送调度决策。代理通过提交事务向内核发送调度决策。代理必须能够调度其本地CPU(每CPU情况)和其他远程CPU(集中式情况)。提交机制必须能够快速支持s-scale策略和可扩展到数百个核。对于每CPU的示例，理论上一个系统调用接口就足够了。对于集中式调度，代理需要高效地将调度请求发送到多个CPU和然后检查这些请求是否成功。因此，共享内存接口更合适。作为题外话，在共享内存中使用事务作为调度接口将允许在未来将调度决策卸载给能够访问该内存的外部设备。

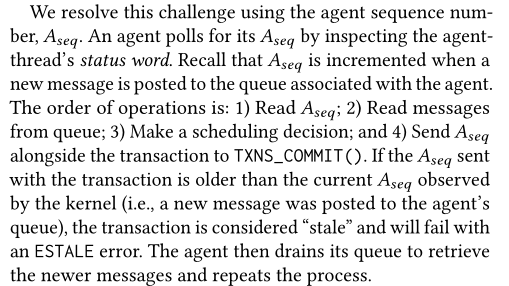
受到事务内存[67]和数据库[68]系统的启发，我们设计了自己的事务API，通过共享内存实现。这些系统支持具有原子语义的快速、分布式提交操作，并且可以同时针对同一个远程节点进行多个提交。ghOSt代理需要类似的属性。代理程序使用TXN\_CREATE()helper函数在共享内存中打开一个新事务。代理写入要调度的线程的TID以及要调度线程的CPUID。在每CPU示例中，每个代理只调度自己的CPU。当事务被填充时，代理通过TXNS\_COMMIT()系统调用将其提交给内核，这将启动提交过程并触发内核启动上下文切换。图3展示了一个简化的例子。

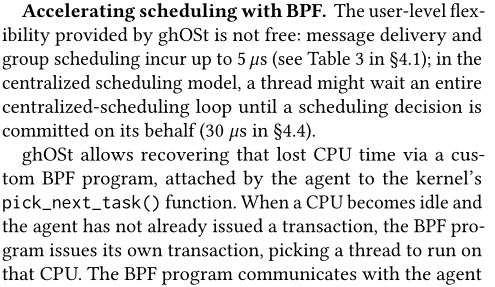
小组提交。在集中式调度示例中，为了允许ghOSt扩展到每秒数百个cpu和数十万个事务，我们必须降低昂贵的系统调用成本。我们通过引入组提交来摊销事务的成本。组提交也减少了发送到其他cpu的中断数量，类似于Caladan[21]。代理通过将所有事务都传递给TXNS\_COMMIT()系统调用来提交多个事务。这个系统调用将昂贵的开销摊销到几个事务上。最重要的是，它通过使用大多数处理器中存在的批处理中断功能来平摊发送中断的开销。内核没有发送多个中断(每个事务发送一个中断)，而是向远程cpu发送单个批处理中断，从而节省了大量开销。

序列号和事务。在每CPU示例中，提交事务的代理将其CPU放弃给它正在调度的目标线程。在代理运行时发布到队列的消息不会引起唤醒，因为代理已经在运行。然而，队列中的新消息可能来自一个更高优先级的线程，



并且会影响调度决策，如果代理意识到这一点。代理只有在下一次唤醒时才有机会检查该消息，这就太晚了。现在，我们将解释如何通过每CPU示例的序列号来解决这一挑战。我们将在§3.3中解释集中式调度的略有不同的情况。



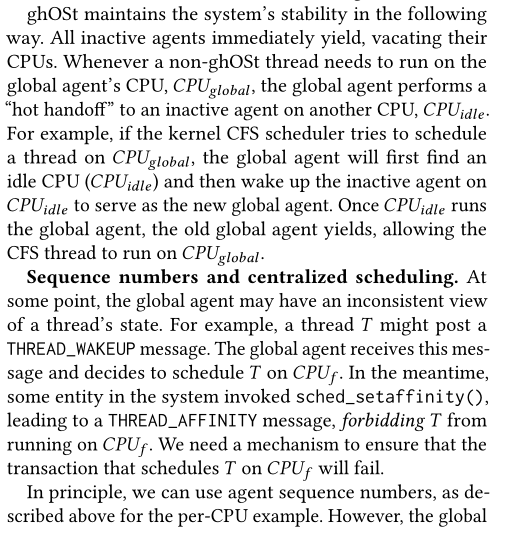


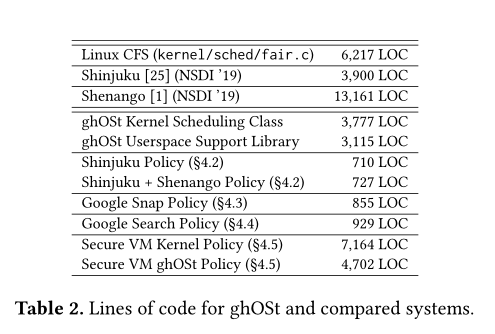
通过共享内存窗口进入代理的地址空间。代理如何使用BPF基础设施在cpu上调度线程的细节是调度策略的一部分。鬼BPF程序本质上是代理本身的扩展，因此BPF字节码被嵌入到代理二进制中，使用libbpf[69]。

**3.3集中式调度器(CentralizedScheduler)**

我们现在解释构建集中式调度鬼影策略所需的其他实现细节。

具有单个队列的全局代理。对于集中式调度，一个全局代理轮询一个消息队列，并为鬼下管理的所有cpu做出调度决策。如果指定的CPU已经运行了一个鬼线程，事务将抢占前一个线程，以优先于新线程。图4描述了一个简化的调度器代码示例。直观地看，集中式策略似乎无法支持s-scale调度，虽然我们在§4中展示了鬼在我们的生产工作负载上有相当或更好的整体性能。避免抢占全局代理。支持s-scale调度，全局代理必须持续运行，因为任何抢占都会直接导致调度延迟。为了防止更高优先级的内核调度类抢占全局代理，鬼给所有代理分配一个高的内核优先级，类似于实时调度。换句话说，机器中的任何其他线程，无论是鬼还是非鬼，都不能抢占代理线程。然而，这种优先级分配，除非小心处理，否则会破坏系统的稳定。例如，大多数系统都有必须在其指定的cpu上运行的每cpu守护进程工作线程。





Agent必须支持成千上万个不断向全局队列发布消息的线程，这使得耗尽队列非常耗时。与每cpu示例中的本地代理不同，全局代理不会放弃自己的CPU。全局代理必须只验证它相对于线程是最新的正在被调度。我们用线程序列号来解决这个问题。回想一下每个排队的消息是否标记了线程序列号

(𝑇,)。当代理为线程𝑇提交事务时，它将事务连同最近的序列号一起发送它意识到:。当内核接收到事务时，它会验证𝑇相对于事务中的线程来说是最新的。否则，事务失败，报错ESTALE。

**3.4故障隔离和动态升级**

与其他内核调度类的交互。ghOSt的设计目标之一是使现有系统易于采用。因此，即使ghOSt策略出错，我们仍然希望鬼管理的线程能够与系统中的其他线程很好地交互。我们希望避免ghOSt线程对其他线程造成意想不到的后果，例如饥饿、优先级反转、死锁等。

我们通过在内核的调度类层次结构中为ghOSt的内核调度器类分配一个低于默认调度器类(通常是CFS)的优先级(§2)来实现这一目标。结果是系统中的大多数线程将抢占ghOSt线程。ghOSt的抢占会导致创建THREAD\_PREEMPT消息，从而触发相关代理(在不同的高优先级调度类中运行)做出调度决策。代理会进一步决定如何处理抢占。

动态升级和回滚。ghOSt支持快速部署，因为更新了调度策略(即:代理程序)不需要重新启动内核或应用程序。许多生产服务可能需要几分钟到几个小时才能启动，特别是填充内存缓存。同样，我们希望尽量减少对客户端虚拟机的中断。这些长期运行的应用程序在计划中的代理更新或计划外的代理崩溃期间继续正确运行。ghOSt通过(a)替换代理在保持飞地基础设施完整的同时，或者(b)摧毁飞地并从头开始。

替换特工，摧毁飞地。ghOSt支持在不破坏飞地的情况下“就地”更新代理。用户空间代码可以查询代理是否附属于飞地，并对其进行epoll。为了升级代理，我们同时运行旧代理和新代理;新代理会阻塞，直到旧代理崩溃或退出，不再连接。新代理从内核中提取enclave中所有线程的状态，并恢复调度。如果这个过程失败，内核或用户空间代码都可以破坏这个enclave。摧毁飞地杀死飞地中的所有代理，保持系统中其他飞地的完整，并自动将被摧毁飞地中的所有线程移回CFS。此时，线程仍然正常工作，但由CFS而不是ghOSt调度。

ghOSt看门狗。ghOSt或任何其他内核调度器中的调度错误都会对整个系统产生影响。例如，一个ghOSt线程可能会在持有一个内核互斥时被抢占，如果它没有被调度太长时间，它可能会传递地暂停其他线程，包括那些在CFS或其他ghOSt飞地中的线程。类似地，如果垃圾回收器和I/O轮询器等关键线程没有被调度，机器就会慢慢停止。作为一种安全机制，《ghOSt》会自动摧毁行为不端的飞地。例如，当内核检测到代理在用户可配置的毫秒数内没有调度可运行线程时，就会销毁一个enclave。

**——————————————————————————————————**

# 本发明技术方案带来的有益效果：