The Design of a Practical System for Fault-Tolerant Virtual Machines

**一种实用的容错虚拟机系统的设计**

Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam

VMware, Inc

[{scales,mnelson,ganesh}@vmware.com](mailto:{scales,mnelson,ganesh}@vmware.com)

**摘要**

我们实现了一种商业的企业级系统，它提供容错虚拟机，基于通过在另一个服务器上的备份虚拟机复制主虚拟机（VM）执行的方法。我们在VMware vSphere 4.0上设计了一个完整的系统，它易于使用，运行在普通服务器上，并且通常减少实际应用的性能不到10%。此外，为了保持主和副虚拟机中几个实际应用步调一致，需要的数据带宽少于20Mbit/s，这使得在更长距离上实现容错成为可能。一个自动从失败之后恢复冗余的易于使用的商业系统，需要更多额外的组件，不仅仅是复制虚拟机的执行。我们设计并实现了这些额外的组件，在支持虚拟机运行商业应用的过程中，处理了很多实际问题。在这篇论文中，我们描述了我们的基本设计，讨论了不同设计和很多实现细节，并且提供了微基准测试和实际应用的性能报告。

# 简介

实现容错服务器的一种常见方法是主/备方法，如果主服务器故障，备份服务器总是可以接管。备份服务器的状态必须随时保持和主服务器几乎一致，这样在主服务器故障时，备份服务器才可以及时接管，这样故障对外部客户端是不可见的，并且没有数据丢失。在备份服务器上复制状态的一种方法是，把主服务器的所有状态，包括CPU、内存和I/O设备，全部连续不断的传递给备份服务器。然而，发送这个状态，特别是内存中的变化，需要非常大的带宽。

另外一种复制服务器的方法，使用更少的带宽，有时被称为状态机方法。这种方法的想法是把服务器当成确定的状态机，只要从相同的初始状态开始，保证它们以相同的顺序接收相同的输入请求，那么它们就会保持同步。因为很多服务器或者服务有一些操作不是确定的，需要一些额外的协调工作来使得主备保持同步。然而，保持主备同步的额外信息数量远小于主服务器上改变的状态数量（主要是内存更新）。

实现物理服务器上确定的执行的协调是困难的，尤其是处理器频率的提高。相反，运行在一个管理程序上的虚拟机（VM），是一个极好的实现状态机方法的平台。一个虚拟机可以被认为是一个定义良好的状态机，它的操作是被虚拟化的机器（包括所有的设备）的操作。跟物理服务器一样，虚拟机也有一些非确定性的操作（例如，读取当前时间或者传递一个中断），所以为了保持主备同步，额外的信息需要发送给备份虚拟机。因为管理程序完全控制虚拟机的执行，包括传递所有的输入，它可以捕获在主虚拟机上有关非确定性操作的所有必需信息，然后在备份虚拟机上正确的重放这些操作。

因此，状态机方法可以实现在运行普通硬件的虚拟机上，不需要修改任何硬件，对于最新的微处理器可以立即实现容错。另外，由于状态机方法需要较少的带宽，所以提供了大范围内主备的物理隔离可能性。例如，复制的虚拟机可以运行在分布于一个校园内的物理机器上，它比运行在同一建筑内的虚拟机更可靠。

在VMware vSphere 4.0平台中，我们使用主备方法实现了容错虚拟机，VMware vSphere 4.0以一种高效的方式运行完全虚拟化的x86虚拟机。因为VMware vSphere实现了一个完整的x86虚拟机，我们可以自动的对所有基于x86的操作系统和应用程序提供容错。能够记录主虚拟机的执行，并且确保在备份虚拟机上相同的执行，这种底层技术被称作确定性的重放。VMware vSphere容错（FT）基于确定性的重放技术，但是添加了创建一个完整的容错系统所必须的额外协议和功能。除了提供硬件的容错，我们的系统还自动在故障之后恢复冗余，通过在本地集群中任何可用的机器上开启一个新的备份虚拟机。到目前为止，确定性重放技术和VMware 容错仅支持单处理器的虚拟机。支持记录和重放多处理器虚拟机执行的工作目前还在进行中，目前遇到了严重的性能问题，因为几乎所有对共享内存的访问都是非确定性的操作。

Bressoud和Schneider [3] 描述了一个在HP PA-RISC平台上实现容错虚拟机的原型。我们的方法是类似的，但是因为性能的原因我们做了一些基础性的改变，并且研究了很多替代的设计。另外，为了创建一个完整的系统，它高效并且易于商业应用客户使用，我们必须设计和实现很多额外的系统组件，并处理一序列的实际问题。和很多其他被讨论的实际系统一样，我们仅尝试解决故障停止（fail-stop）故障[12]，它是一种服务器故障，在故障服务器引起一个外部可见的动作之前它是可被检测的。

论文的余下部分按照如下方式组织。首先，描述了我们的基本设计，详述了我们的基础协议，协议用于保证在主虚拟机故障之后备份虚拟机接管的过程中没有数据丢失。其次，我们详细介绍了，为了创建一个健壮的、完整的和自动化的系统，所需要被处理的很多实际问题。我们还描述了几个不同的用于实现容错虚拟机的设计选择，讨论了这些选择的折衷。然后，我们提供了基于我们的实现的一些基准和商业应用的性能测试结果。最后，我们描述了相关工作并总结。

# 基本容错（FT）设计

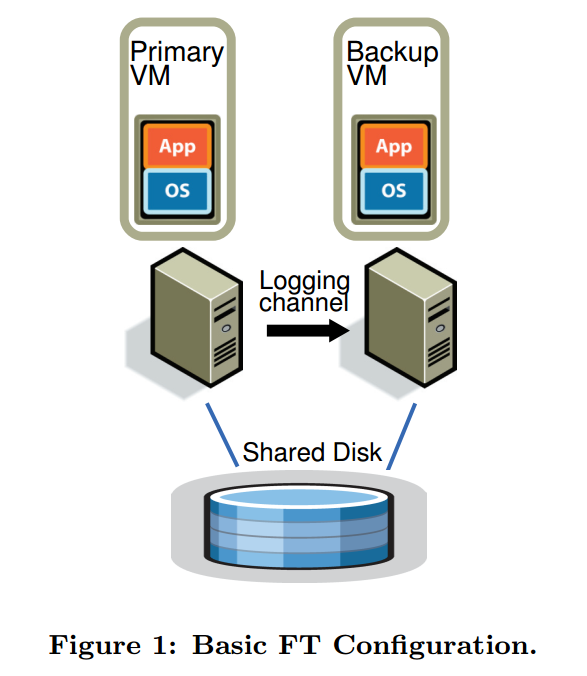


图1展示了我们的容错虚拟机系统的基本配置。对于一个给定的虚拟机，当我们想提供容错（主虚拟机），我们在另外一台物理服务器上运行一个备份虚拟机，它用于保持同步，并且执行和主虚拟机一样的操作，虽然有一个很小时间的延迟。我们称这两个虚拟机处于虚拟一致。这些虚拟机的虚拟磁盘都在共享存储设备上，因此输入输出都可以被主备虚拟机访问。（在4.1章节我们将会讨论一种采用分离不共享虚拟磁盘的主备虚拟机方案）只有主虚拟机在网络中广告它的存在，所以所有的网络输入都会到达主虚拟机。相似地，所有其他的输入（例如键盘和鼠标）也只到达主虚拟机。

主虚拟机接收的所有输入，通过一个叫日志通道的网络连接，被发送给备份虚拟机。对于服务器工作负载，主要的输入流量是网络和磁盘。额外的信息，如在下面2.1章节讨论的，根据需要传输给备份虚拟机，以确保备份虚拟机和主虚拟机以相同的方式执行非确定性的操作。结果是备份虚拟机总是和主虚拟机执行一样。然而，管理程序丢弃备份虚拟机的输出，所以只有主虚拟机产生实际的输出返回给客户端。如2.2章节所述，主备虚拟机遵循一个特定协议，包括备份虚拟机的显式确认，为了确保在主虚拟机故障的情况下没有数据丢失。

为了检测主或者备虚拟机是否故障了，我们的系统使用了一个相关服务器之间的心跳和日志通道的流量监控的组合。另外，我们必须确保只有一个主或者备份虚拟机在接管执行，即使是在一个裂脑（split-brain）的处境下，其中主备服务器相互之间失去了通信。

在接下的章节中，我们提供了几个重要领域的更多细节。在2.1章节，我们介绍了一些有关确定性重放技术的细节，它通过日志通道传递的信息，确保了主备虚拟机保持同步。在2.2章节，我们描述了容错协议的一个基础规则，它确保在主虚拟机故障的情况下没有数据丢失。在2.3章节，我们描述了如何正确的故障检测和响应方法。

## 确定性重放的实现

正如我们已经提到的，复制服务器（或者虚拟机）的执行可以类比于一个确定性状态机的复制。如果两个确定性状态机从相同的初始条件开始，以相同的顺序提供完全相同的输入，那么它们将会走过相同序列的状态，产生相同的输出。虚拟机有非常广泛的输入，包括进来的网络包、读磁盘和来自键盘和鼠标的输入。非确定性事件（例如虚拟中断）和非确定性操作（例如读处理器的时钟周期计数）也会影响虚拟机的状态。这对于运行在任何操作系统和工作负载下的任何虚拟机执行的复制提出了三个挑战：（1）正确捕获所有输入和非确定性因素，以确保备份虚拟机的确定性执行。（2）把所有输入和非确定性因素正确地应用到备份虚拟机中。（3）这种方式不能降低性能。另外，x86微处理器中很多复杂的指令有未定义副作用，因此是非确定性的。捕获这些未定义的副作用，重放它们产生相同的状态，提出了一个额外的挑战。

VMware确定性重放 [15] 正好为VMware vSphere平台上的x86虚拟机，提供了这个功能。确定性重放记录，虚拟机的输入和跟虚拟机执行有关的所有可能的非确定性因素，把它们以日志条目流的形式写入到一个日志文件。随后，通过读取日志文件中日条目，虚拟机的整形可以被正确的重放出来。对于非确定性操作，足够的日志信息允许操作被重放，产生相同的状态变化和输出。对于非确定性事件，例如定时器或者IO完成中断，事件发生时的指令也被记录。在重放过程中，在指令流中相同的点，事件被传递出来。VMware确定性重放实现了一种有效的事件记录和事件传送的机制，它采用了多方面的技术，包括使用了和AMD[2]和Intel[8]联合开发的硬件性能计数器。

Bressoud 和Schneider [3]提到把虚拟机的执行划分成不同时期，其中非确定性事件，例如中断，仅在一个时期的结尾被传送。时期的概念意味着一个批量的机制，因为在指令确切发生的地方传送每一个中断非常昂贵。然而，我们的事件传送机制足够高效，所以VMware确定性重放没必要使用时期。每一个中断在它发生时都被记录，然后在重放时，在恰当的指令中被高效的传送出来。

## 容错（FT）协议

对于VMware容错，我们使用确定性重放产生必须的日志条目，用来记录主虚拟机的执行，但是不把日志条目写到磁盘，而是通过日志通道把它们发送给备份虚拟机。备份虚拟机实时的重放这些日志条目，因此和主虚拟机执行一样。然而，我们必须在日志通道上增加严格容错协议有关的日志条目，它的目的是为了确保我们能实现容错。我们的基本要求如下：

**输出要求**：如果备份虚拟机在主虚拟机故障之后接管，那么备份虚拟机将会以跟主虚拟机发送到外部世界的全部输出完全一致的方式继续执行。

注意到在一个失效备援发生之后（例如，备份虚拟机在主虚拟机故障之后接管），备份虚拟机很可能将会，以跟主虚拟机曾经非常不同的方式开始执行，因为在执行过程中很多非确定性事件的发生。然而，只要备份虚拟机满足输出要求，在失效备援到备份虚拟机的过程中，没有外部可见的状态或者数据丢失，那么客户端在它们的服务中，将不会注意到中断或者非一致性。

输出要求可以这样被确保：延迟任何外部输出（典型的是一个网络包），直到备份虚拟机接收到所有足以让它重放执行到输出操作点相关的信息。一个必要条件是备份虚拟机必须接收到输出操作之前产生的所有日志条目。这些日志条目将允许它执行到最后日志记录的点。然而，假设在主虚拟机执行了输出操作之后，立即发生了一个故障。备份虚拟机必须知道它必须重放到输出操作的点，并且只在那个点“上线”（停止重放，并且接管作为主虚拟机，如2.3章节所描述的）。如果备份虚拟机在输出操作之前的最后一个日志条目点上线，一些非确定性事件（例如，传送给虚拟机的定时器中断），在它执行输出操作之前，可能改变它的执行路径。

在以上给出的约束下，最简单的达到输出要求的方法是，对于每一个输出操作创建一条特殊的日志条目。然后，输出要求可以被下面的这条特殊规则达到：

**输出规则**：主虚拟机，直到备份虚拟机接收并确认了跟产生输出操作关联的日志条目之前，不会发送输出到外部世界。

如果备份虚拟机接收到所有日志条目，包括输出产生操作的那条日志条目，那么，备份虚拟机将能够准确的重新产生主虚拟机在输出点的状态，所以如果主虚拟机挂了，备份虚拟机将能够正确的达到跟输出一致的状态。相反，如果备份虚拟机没有接收到所有必须的日志条目就接管了，那么它的状态可能快速偏离到跟主虚拟机输出不一致。输出规则在某些方面类似于在[11]中描述的方法，其中一个“外部同步”IO可以被缓冲，只要在下一次外部通信之前确实写到磁盘中。

注意输出规则没有任何关于停止主虚拟机执行的说明。我们只需要延迟发送输出，至于虚拟机本身可以继续执行。因为操作系统是通过异步中断来标识非阻塞的网络和磁盘输出的完成，虚拟机可以轻松的继续执行，不会被输出的延迟立即影响。相反，以前的工作[3,9]明确指出虚拟机在输出之前必须完全停止执行，直到备份虚拟机确认了所有来自主虚拟机的必要信息。

作为一个例子，在图2中我们以图的形式展示了容错协议的要求。这个图展示了在主和备虚拟机中事件的时间轴。从主线到备份线的箭头反应了日志条目的转移，从备份线到主线从的箭头反应了日志条目的确认。关于异步事件、输入和输出操作相关的信息必须确保以日志条目的形式发送到备份虚拟机。如图中所示，一个外部世界的输出被延迟，直到主虚拟机收到备份虚拟机对输出操作关联的日志条目的确认。如果输出规则被遵循，备份虚拟机将能够以跟主虚拟机最后一个输出相一致的状态接管。

我们不能保证在一个失效备援的情况下，所有的输出恰好只被产生一次。在不使用两阶段提交事务的前提下，当主虚拟机打算发送一个输出时，备份虚拟机没有办法知道，主虚拟机是在发送最后一个输出之前或者之后故障了。幸运的是，网络基础设施（包括常用的TCP）被设计成可以处理丢包和相同（重复）包。注意到在主虚拟机故障时，输入到主虚拟机的网络包可能丢失，因此也不会传送到备份虚拟机。然而，输入的网络可能因为各种跟服务器故障无关的原因被丢失，所以网络基础设施、操作系统和应用程序都被编码成它们可以处理网络包丢失。

## 故障检测与响应

如以上所提到的，如果其他虚拟机好像故障了，主和备份虚拟机必须快速响应。如果备份虚拟机故障了，主虚拟机将继续在线 --- 意味着，退出记录模式（因此停止在日志通道中发送日志条目），开始正常执行。如果主虚拟机故障了，备份虚拟机应该以类似的方式在线，但是处理过程有一些复杂。因为执行的延迟，备份虚拟机可能由很多接收并确认的日志条目并没有消费掉，因为备份虚拟机还没有达到恰当的执行点。备份虚拟机必须继续重放日志条目的执行，直到它消费了最后一个日志条目。这个时候，备份虚拟机将会停止重放模式，然后像一个正常虚拟机一样开始执行。本质上，备份虚拟机被提升为主虚拟机（现在它缺少一个备份虚拟机）。因为它不再是备份虚拟机，当客户操作系统需要输出操作时，新的主虚拟机将会对外部世界产生输出。在向正常模式转移的过程中，可能需要一些设备专用的操作来让输出恰当的发生。特别是，为了网络传输的目的，VMware容错会自动在网络中把新主虚拟机的网卡地址广而告之，这样物理网络交换机将会知道新主虚拟机的地址。另外，新提升的主虚拟机可能需要重新发出一些磁盘IO（如3.4章节所述）。

有很多方法可以拿来尝试检测主和备份虚拟机的故障。VMware容错使用运行容错虚拟机服务器之间的UDP心跳，来检测是否由服务器故障了。另外，VMware容错监控主虚拟机发送到备份虚拟机的日志消息和备份虚拟机发送到主虚拟机的确认消息。因为定期的定时器中断，日志流量必须是定期的，并且对于一个运行的客户操作系统来说从来不会停止。因此，一个日志条目或者确认消息流的停止就标识了虚拟机的一个故障。如果心跳或者日志流量停止了超过一定时间（以几秒钟的顺序），则故障被声明发生了。

然而，任何这样的故障检测方法都容易受到分裂脑问题的影响。如果备份服务器停止收到来自主服务器的心跳，那可能标志着主服务器可能故障了，或者可能意味着任然在运行的服务器之间的全部网络连接都丢失了。如果在主虚拟机实际上还在运行的情况下，备份虚拟机上线，这可能会导致数据损坏或者客户端和虚拟机之间的通信问题。因此，当故障被检测到时，我们必须确保主或者备份虚拟机中只有一个能上线。为了避免分裂脑问题，我们使用了存储虚拟机磁盘的共享存储设备。当主或者备份虚拟机将要上线时，它会在共享存储设备上执行一个test-and-set操作。如果操作成功了，虚拟机就会被允许上线。如果操作失败了，其他虚拟机已经上线了，所以当前虚拟机就停止自己（“提交终止自己”）。当虚拟机尝试做这个原子操作时不能访问共享存储设备，就继续等待直到能访问。注意到，如果因为存储网络的一些故障导致共享存储不能被访问，那么虚拟机就可能做不了任何有意义的工作，因为虚拟磁盘是放在这些共享存储上。因此，采用共享存储来解决分裂脑问题，不会引入任何额外的不可用性。

设计的最后一个方面是一旦故障发生了，其中一个虚拟机上线了，VMware容错会自动通过在另外一个主机上开启一个新的备份虚拟机来恢复冗余。虽然这个过程在以前很多工作中并没有被涉及到，但是对于使容错虚拟机有用是基本条件，并且它需要仔细的设计。更多的细节在3.1章节中给出。

# 容错（FT）的实际实现

第2章描述了容错的基础设计和协议。然而，为了创建可用的、健壮的自动化系统，还必须设计和实现很多其他的组件。

## 容错虚拟机（FT VM）的启动和重启

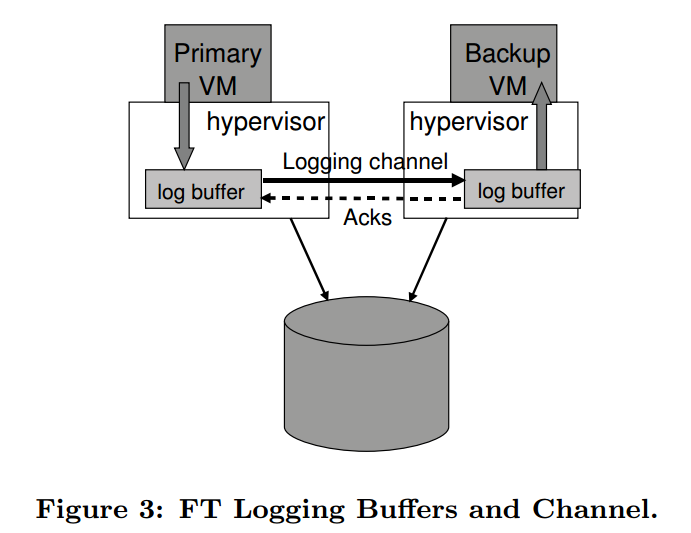
必选设计的最大的附加组件之一是，启动一个跟主虚拟机状态相同的备份虚拟机的机制。这个机制也被用于在故障发生之后重新启动一个备份虚拟机。因此，对于一个处于任意状态（也就是不仅仅是启动）的运行中的主虚拟机，这个机制必须可用。另外，我们更希望这个机制不会显著的破坏主虚拟机的执行，因为那样会影响虚拟机当时所有的客户端。

对于VMware容错，我们适配了VMware vSphere中的VMotion功能。VMware VMotion可以把一个运行中的虚拟机从一个服务器迁移到另外一个服务器，影响非常小 --- 虚拟机停止的时间通常少于一秒钟。我们创建了VMotion 的一个修改形式，它可以在远程服务器上拷贝一个运行的虚拟机，并且不会破坏本地服务器上的虚拟机。那就是，我们修改的容错VMotion在远程服务器上拷贝一个虚拟机，而不是迁移它。容错VMotion也会建立一个日志通道，使得源虚拟机进入日志模式，充当主虚拟机，而目标虚拟机进入重放模式，充当备份虚拟机。就像正常的VMotion，容错VMotion中断主虚拟机的执行少于一秒钟。因此，对于运行的虚拟机，启用容错是一个简单的，并且非破坏性的操作。

启动一个备份虚拟机的另外一个方面是如何选择一个服务器。容错虚拟机运行在共享存储的服务器集群中，所以所有虚拟机可以运行在集群中任何一台服务器上。这个灵活性允许VMware vSphere在一台或者多台服务器故障的时候，恢复容错冗余。VMware vSphere实现了一个维护管理和资源信息的集群服务。当故障发生时，主虚拟机需要一个新备份虚拟机来重新建立冗余时，主虚拟机通知集群服务它需要一个新备份虚拟机。集群服务依据资源的使用和其他限制情况，选择最优的适合运行备份虚拟机的服务器，然后调用一个容错VMotion来创建新的备份虚拟机。结果是在服务器故障几分钟之内，VMware容错就能够重新建立虚拟机冗余，所有这些都不需要明确中断容错虚拟机的执行。

## 日志通道的管理

在管理日志通道的流量中由很多有趣的实现细节。在我们的实现中，管理程序为主和备份虚拟机维护了一个很大的日志条目缓冲区。当主虚拟机执行时，它向日志缓冲区中生产日志条目，类似的，备份虚拟机从日志缓冲去中消费日志条目。主虚拟机日志缓冲的内容会尽可能快地刷到日志通道中，到达日志通道中的日志条目会被立即读取到备份虚拟机的日志缓冲中。当备份虚拟机每次从网络中读取到一些日志条目到自己的缓冲中，它会向主虚拟机返回确认。这些确认使得VMware容错可以决定，根据输出规则延迟的输出可以被发送出去。图3展示了这个过程。



如果备份虚拟机读取下一条日志条目时，遇到了一个空日志缓冲，它会停止执行，直到出现一条新的日志条目。因为备份虚拟机不跟外部通信，所以这个暂停不会影响到虚拟机的任何客户端。类似的，当主虚拟机需要写一条日志条目时，遇到了一个满日志缓冲，它必须停止执行，直到日志条目可以被刷出。这个停止执行是一个自然的流程控制机制，当主虚拟机以过快的速度生产日志条目时，它减慢了主虚拟机。然而，这个暂停会影响到虚拟机的客户端，因为主虚拟机将会完全停止并且无反应，直到它可以写日志条目为止才继续执行。因此，我们的实现必须被设计成最小化主虚拟机日志缓冲填满的可能性。

当主虚拟机日志缓冲填满时，其中一个原因是备份虚拟机执行得太慢，因此消费日志条目太慢。一般而言，备份虚拟机必须能够以跟主虚拟机记录执行大致相同的速度来重放执行过程。幸运的是，在VMware确定性重放中记录和重放的开销大致上是相同的。然而，如果备份虚拟机所在的服务器加载了过多其他的虚拟机（因此就过度使用资源），备份虚拟机就有可能因为拿不到足够的CPU和内存资源，导致不能像主虚拟机执行一样快，尽管管理程序的虚拟机调度器做了最大的努力。

避免当日志缓冲填满时出现意外停顿，存在另外一个原因导致我们不希望执行延迟变得太大。当主虚拟机故障时，在备份虚拟机上线和开始与外部世界通信之前，它必须通过重放已经确认过的日志条目来“赶上”。在故障发生点完成重放的事件基本上就是执行延迟，所以备份虚拟机上线的事件大致上等于故障检测时间加上当前的执行延迟时间。因此，我们不希望执行延迟时间太大（超过1秒），因为那样会让故障转移时间增加更多。

因此，我们有一个额外的机制来减慢主虚拟机，防止备份虚拟机落后太多。在我们发送和确定日志条目的协议中，我们会发送额外的信息，用来确定主和备份虚拟机之间实时的执行延迟。通常执行延迟少于100毫秒。如果备份虚拟机开始有一个显著的执行延迟（例如，超过1秒），VMware容错就开始通知调度器少给一些CPU（最初只有百分之几），来减慢主虚拟机。我们使用一个慢反馈机制，它会尝试逐渐的精确控制CPU以允许备份虚拟机来匹配它的执行。如果备份虚拟机继续落后，我们继续逐渐的减少主虚拟机的CPU。相反的，如果备份虚拟机赶上了，我们会逐渐的增加主虚拟机的CPU，直到备份虚拟机回到只有轻微的延迟为止。

注意到这样对主虚拟机的减速是非常少的，通常只会发生在系统在极端压力下。地5章节所有的性能数据都包括了这样的减速。

## 容错虚拟机（FT VM）的操作

另外一个实际的事情是处理各种会被应用到主虚拟机的控制操作。例如，如果主虚拟机被明确的关机了，备份虚拟机也应该被停止，并且不能企图上线。另外一个例子，主虚拟机上任何资源管理的改变（例如增强的CPU共享）也应该被应用到备份虚拟机上。对于这些操作，特别的日志条目会通过日志通道从主虚拟机发送到备份虚拟机，为了在备份虚拟机上实现恰当的操作。

通常来说，虚拟机大部分操作应该只在主虚拟机上开始，然后VMware容错会发送任何必须的控制条目，以引起备份虚拟机上恰当的改变。在主和备份虚拟机上独立完成的操作只有VMotion。那是，主和备份虚拟机可以独立的被迁移到其他主机上。注意VMware容错会保证没有虚拟机会被迁移到其他虚拟机所在的服务器上，因为那样会导致不再提供容错。

主虚拟机的迁移比正常的迁移增加了一些复杂性，因为备份虚拟机必须和源主虚拟机断开，然后在恰当的时间重新连接到目标主虚拟机。备份虚拟机的迁移有一个类似的问题，但是增加了额外的复杂性。对于一个正常的虚拟机迁移，我们要求所有未决的磁盘IO静默（也就是说完成的）就像在迁移发生时最后的切换一样。对于一个主虚拟机，这种静默是易于处理的，只要等到物理IO完成并且通知到虚拟机为止。然而，对于一个备份虚拟机，没有简单的方法可以让所有的IO在必须的点上完成，因为备份虚拟机必须重放主虚拟机的执行，然后在相同点上完成IO。主虚拟机可能运行一个在正常执行时总是有磁盘IO的工作负载。VMware容错有一个唯一的方法来解决这个问题。当备份虚拟机处于迁移的最后切换点，它通过日志通道请求主虚拟机临时及静默所有IO。备份虚拟机的IO自然就会静默，就像单个执行点一样，因为它会重放主虚拟机的静默操作。

## 磁盘IO的实现问题

关于磁盘IO，存在一序列微妙的实现问题。首先，由于磁盘操作是非阻塞的，所以可以并行执行，同时访问磁盘相同位置的磁盘操作会导致非确定性。而且，我们磁盘IO的实现是在虚拟机的内存中直接使用DMA，所以相同的磁盘操作即是访问相同内存页也会导致非确定性。我们的解决方案是检测这样的磁盘IO竞争（这些很少发生），然后强制性的让这些竞争磁盘操作在主和备份虚拟机中以相同的方式顺序执行。

其次，虚拟机中磁盘操作也会和应用程序（或者操作系统）的内存访问产生竞争，因为磁盘操作是通过DMA直接访问虚拟机的内存。例如，当虚拟机的应用或者操作系统读取一个内存块，同时一个磁盘读取也发生在同样的内存块上，这会产生一个非确定性的结果。这种情况很少发生，但是我们必须检测它，并在它发生时处理它。一个解决方案是对于磁盘操作的内存页临时设置页保护。当虚拟机碰巧访问一个未决的磁盘操作的内存页时，页保护会导致一个陷阱，这时虚拟机可以暂停，直到磁盘操作完成。因为改变这些内存页上的MMU保护是一个昂贵的操作，所以我们选择使用弹性缓冲区。一个弹性缓冲区是一个跟磁盘操作内存大小相同的临时缓冲区。一个磁盘读取操作被修改成读取弹性缓冲区中特定的数据 ，只有在IO完成之后数据才会被拷贝到客户内存中。类似的，对于一个磁盘写入操作，发送的数据首先被写入弹性缓冲区，磁盘写入操作被修改成从弹性缓冲区写入数据。弹性缓冲区的使用会减慢磁盘操作，但是我们没有发现任何显著的性能损失。

第三，当主虚拟机故障，备份虚拟机接管时，有一些问题是跟主虚拟机中未决的（也就是说未完成的）磁盘IO有关。没有办法让新提升的主虚拟机明确这些磁盘IO被发到磁盘或者成功完成。另外，因为在备份虚拟机上这些磁盘IO没有对外发布，当新提升的主虚拟机继续执行时将不会有显式的IO完成，它有可能引起虚拟机中客户操作系统开始一个退出或者重置的程序。我们可以发送一个错误的完成来标志这些IO失败了，因为即使IO成功完成返回错误也是可接受的。然而，客户操作系统从它的本地磁盘可能不会正确的响应这些错误。相反，在备份虚拟机上线的过程中，我们重新发布这些未决的IO。因为我们消除了所有的竞争，并且所有的IO都直接制定了访问那块内存和磁盘块，这些磁盘操作可以被重新发布，即使它们已经成功完成（也即是说他们是幂等的）。

## 网络IO的实现问题

VMware vSphere针对虚拟机网络提供了很多性能优化。一些优化是基于管理程序异步更新虚拟机网络设备的状态。例如，在虚拟机执行时，管理程序可以直接更新接收缓冲区。不幸运的是，这些对虚拟机状态的异步更新增加了不确定性。除非我们可以保证，所有这些更新在主和备份虚拟机中指令流相同点上发生，要不然备份虚拟机的执行就会偏离主虚拟机。

为了实现容错网络仿真代码中最大的改变是，异步网络优化的禁用。异步更新虚拟机接收网络包的环形缓冲区的代码被修改了，强制让客户端陷入管理程序的陷阱，这可以让它记录更新然后应用到虚拟机。类似的，异步从传输队列中获取数据包的代码在容错时也被禁用，相反传输是通过管理程序的陷阱来完成的（除非像下面说明之外）。

消除网络设备的异步更新，加上2.2章节所描述的延迟发送数据包，这些会导致网络的一些性能改变。我们采取了两个方法，来提高在运行容错时虚拟机的网络性能。首先，我们实现了集群优化来减少虚拟机的陷阱和中断。当虚拟机非常高比特率的流式传输数据，管理程序可以为每一组数据包只产生一个传输陷阱，最好是没有陷阱，因为它可以在接收新包的同时传输这些数据包。同样的，管理程序可以通过一组数据包只触发一个中断来减少虚拟机的中断数量。

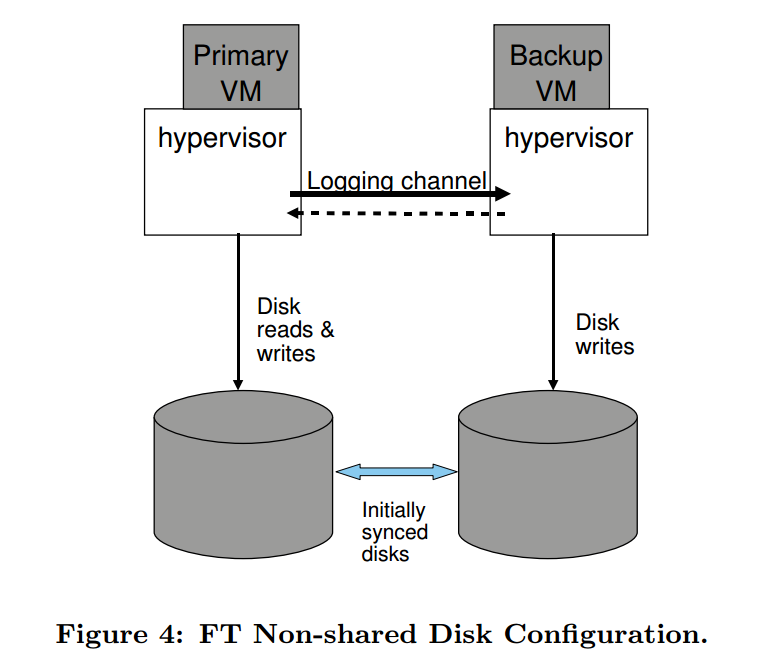
我们的第二个针对网络的性能优化牵涉到降低传输数据包的延迟。如前所述，管理程序必须延迟所有传输的数据包，直到它收到备份虚拟机对于特定日志条目的确认。降低传输延迟的关键是，减少发送日志消息到备份虚拟机和接收确认的时间。针对这个地方，我们的主要优化，涉及到保证发送日志条目和接收确认能够在没有任何上下文切换的情况下完成。VMware vSphere管理程序允许在TCP栈中注册函数，当接收到TCP数据时，可以在一个延迟执行的上下文（类似Linux中的一个小任务）中被调用。这使得我们可以，在没有任何线程上下文切换的前提下，快速处理，备份虚拟机上接收的任何日志消息和主虚拟机上接收的任何确认消息。另外，当主虚拟机把一个数据包放入传输队列，通过调度延迟执行上下文，我们立即强制发起一个输出有关的日志条目（如2.2章节所述）的刷出。

# 替代设计

在我们实现VMware容错的过程中，我们探索了很多其他有趣的替代设计。在这章中，我们套索其中的一些替代设计。

## 共享 vs. 非同享磁盘

在我们默认的设计中，主和备份虚拟机共享相同的虚拟磁盘。因此，故障发生时，共享磁盘的内容自然是正确并且可用的。本质上来说，共享磁盘被认为在主和备份虚拟机之外，所以任何对共享磁盘的写入被认为是一次对外部世界的通信。因此，只有主虚拟机才会真正写到磁盘，并且共享磁盘的写入必须延迟以保持跟输出规则一致。



一个替代设计是主和备份虚拟机使用分离（非共享）的虚拟磁盘。在这种设计中，备份虚拟机执行所有磁盘写入到它自己的虚拟磁盘，这样做，它自然就让自己的虚拟磁盘内容和主虚拟机的虚拟磁盘内容保持一致。图4展示了这种配置。在非共享磁盘情况下，虚拟磁盘本质上被认为是每个虚拟机内部状态的一部分。因此，主虚拟机的磁盘写入根据输出规则并不需要被延迟。非共享的设计在主和备份虚拟机不能访问共享存储的情况下非常有用。这可能是这样的，因为共享存储不可用或者太贵了，或者因为运行主和备份虚拟机的服务器离得很远（“长距离容错”）。非共享设计的一个缺点是，当容错首次启用时，虚拟磁盘的两份拷贝必须以某种方式显式的被同步。另外，故障之后磁盘可能不同步，所以当备份虚拟机在故障之后重启时，它们必须被显式的再次同步。那就是，容错虚拟机迁移不仅同步主和备份虚拟机的运行状态，而且同步它们的磁盘状态。

在非共享磁盘配置中，可能没有共享存储来用于处理分裂脑情况。在这种情况下，系统可以使用一些其他的外部控制器，例如一个可以被主备都能访问的第三方服务器。如果服务器是一个超过2个节点的集群的一部分，系统也可以使用一个基于集群成员的多数派算法。在这种情况下，只有当它运行在一个包含大多数原始节点的连通子集群的其中一台服务器上，虚拟机才被允许上线。

## 在备份虚拟机上执行磁盘读取

在我们的默认设计中，备份虚拟机从不会从它自己的虚拟磁盘中读取（无论是共享还是非共享）。因为读磁盘被认为是一个输入，自然就应该通过日志通道来把磁盘读取的结果发送到备份虚拟机。

一个替代设计是让备份虚拟机执行磁盘读取，因此就消除了磁盘读取数据的日志。这种方法对于由很多磁盘读取的工作负载来说，可以极大的减少日志通道的流量。然而，这种方法有一些微妙的问题。它可能减慢备份虚拟机的执行，因为备份虚拟机必须执行所有的磁盘读取，然后，当它到达主虚拟机已经完成的点上，如果物理上没有完成就必须等待。

而且，对于失败的磁盘操作，一些额外的工作必须被完成。如果主虚拟机中一个磁盘读取成功了，但是相应的磁盘读取在备份虚拟机上失败了，然后这个备份虚拟机上的磁盘读取必须重试直到成功，因为备份虚拟机必须在内存中获取到跟主虚拟机一样的数据。相反，如果主虚拟机中一个磁盘读取失败了，然后目标内存的内容必须通过日志通道发送到备份虚拟机，因为内存的内容将会不确定，并且由于备份虚拟机上一次成功磁盘读取导致的不必要的复制。

最后，当这个替代的磁盘读取和共享磁盘配置一起使用时，有一个微妙的问题。如果主虚拟机读取一个特定的磁盘位置，然后立即写入相同的磁盘位置，那么这个磁盘写入就必须延迟直到备份虚拟机已经执行了第一个磁盘读取。这种依赖可以被正确的检测和处理，但是增加了实现的额外复杂性。

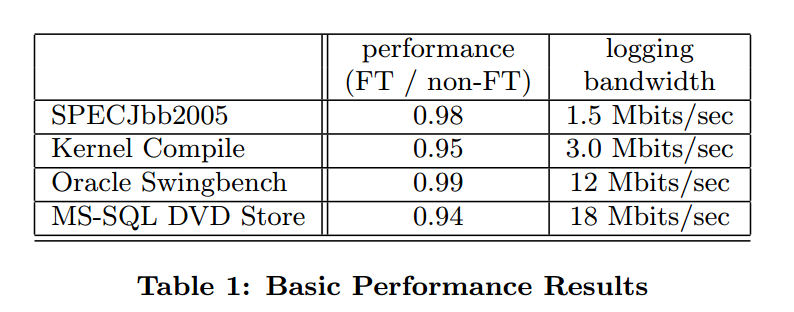
在5.1章节，我们给出了一些性能结果表明，对于实际应用，在备份虚拟机上执行磁盘读取会引起一些轻微的吞吐量的降低（1-4%），但是也会显著的减少日志带框。因此，在备份虚拟机上执行磁盘读取，对于日志通道带宽相当有限的情况，会非常有用。

# 性能评估

在这个章节中，对于许多应用工作负载和网络基准，我们做了一个VMware容错性能的基本评估。对于这些结果，我们在相同的服务器上运行主和备份虚拟机，各自拥有8核Intel Xeon 2.8Ghz的处理器和8GB内存。这些服务器通过10G/s的交换机连接起来，虽然在所有测试情况中只要不到1G/s的网络带宽被使用了。两个服务器访问的虚拟磁盘是一个通过标准的4G/s光钎通道网络连接起来的EMC磁盘阵列。驱动一些工作负载的客户端是通过1G/s的网络连接到服务器。

在我们的性能结果中评估的应用如下。SPECJbb2005是一个行业标准的Java应用基准测试程序，它是CPU和内存密集型的，做非常少的IO。内核编译是一个运行Linux内核编译的工作负载。这个工作负载做了一些磁盘读写，是CPU和内存密集型的，因为很多编译进程的创建和销毁。Oracle Swingbench是一个由Swingbench OLTP（在线事务处理）驱动的Oracle 11g数据库工作负载。这个工作负载做了大量的磁盘和网络IO，并且同时有8个数据库连接。MS-SQL DVD Store是一个由DVD存储基准测试驱动的微软SQL-Server 2005数据库工作负载，它同时有16个客户端。

## 基本性能结果



表格1给出了基本的性能结果。对于每一个列出的应用来说，第2列给出了同一个虚拟机上容错启用对于不启用的性能比率。计算性能的比率，所以小于1的值说明容错的工作负载慢一下。很显然，在这些代表性的工作负载中启用容错的开销少于10%。SPECJbb2005完全是计算型的，没有闲置时间，但是运行很好，因为它除了定时器中断之外有最少非确定性事件。其他工作负载做磁盘IO，有一些闲置时间，所以一部分容错开销可能因为容错虚拟机有更少的闲置时间而隐藏了。然而，通常的结论是VMware容错有能力以一个相当低地性能损失支持容错虚拟机。

在表格的第三列中，我们给出了当这些应用运行时日志通道中数据发送的平均带宽。对于这些应用，日志带宽是合理的，并且很容易由1G/s网络满足。实际上，低带宽的要求意味着，在没有任何负面性能影响的情况下，多个容错工作负载可以共享同一个1G/s网络。

对于运行普通客户操作系统想Linux和Windows的虚拟机，我们发现当客户操作系统空闲时典型的日志带宽是0.5-1.5M/s。“空闲“带宽大部分是记录定时器中断的结果。对于一个处于活动状态的虚拟机，日志带宽主要是必须发送到备份虚拟机的网络和磁盘的输入 - 接收的网络包和从磁盘中读取的磁盘块。因此，日志带宽，对于有非常高的网络接收或者磁盘读取的应用来说，就会比表1测试的有更高的带宽。对于这些应用，日志带库可能是个瓶颈，特别是有其他使用日志通道。

对于很多实际应用，当采取长距离配置使用非共享磁盘时，日志通道中相对低的带宽要求，使得基于重放的容错非常有吸引力。对于主和备份虚拟机可能相隔1-100千米这种长距离配置，光钎在少于10毫秒的延迟下可以轻松支持100-1000M/s。对于表1中的应用，一个100-1000M/s的带宽对于好性能是足够的。注意，然而，主和备份虚拟机之间往返的延迟可能引起网络和磁盘的输出被延迟最高20毫秒。长距离配置仅适用于那些客户端能够忍受每次请求这样一个额外的延迟的应用。

对于两个磁盘密集型的应用，我们分别测试了在备份虚拟机上执行磁盘读取（如4.2章节所述）和通过日志通道发送磁盘读取数据对性能的影响。对于Oracle Swingbench，在备份虚拟机上执行磁盘读取时吞吐量要低4%。对于MS-SQL DVD Store，吞吐量要低1%。同时，对于Oracle Swingbench日志带宽从12M/s减少到3M/s，对于MS-SQL DVD Store从18M/s减少到8M/s。明显地，对于有更多磁盘读取的应用，带宽会节省的更多。如4.2章节所提到的，可以预期的是当在备份虚拟机上执行磁盘读取，性能某种程度上会更差。然而，对于日志带宽是有限的情况（例如，长距离配置），在备份虚拟机上执行磁盘读取可能更有用。

## 网络基准测试

因为许多原因，网络基准测试对于我们的系统来说相当有挑战性。首先，高速网络会有非常高的中断率，它要求异步事件的日志和重放非常高速。其次，高速接收数据包的基准测试会导致高速的日志流量，因为所有这些数据包必须通过日志通道发送给备份虚拟机。第三，发送数据包的基准测试必须遵循输出规则，这会延迟网络包的发送直到接收到备份虚拟机恰当的确认。这个延迟会增加客户端的测试延迟。这个延迟也会降低到客户端的网络带宽，因为网络协议（例如TCP）会因为往返延迟增加而降低传输率。

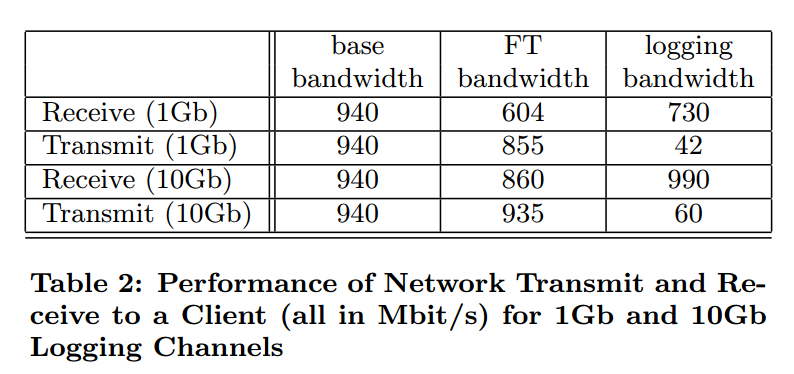


表2给出了一些基于netperf基准测试所测量的结果。在所有这些测试中，客户虚拟机和主虚拟机通过1G/s网络起连接起来。前2行给出当主和备份主机通过1G/s日志通道连接时发送和接收的性能。第3和4行给出了当主和备份服务器通过10G/s连接时发送和接收的性能，它不仅有更高的带宽，而且也比1G/s网络延迟更低。粗略测试表明，1G/s连接的管理程序之间通信时间大约150微秒，而10G/s连接的大约是90微秒。

当容错被禁用，主虚拟机传输和接收可以达到接近（940M/s）1G/s的上限速度。当容错在接收工作负载是被启用，日志带宽非常大，因为所有输入的网络包都必须通过日志通道发送。日志通道因此变成了瓶颈，正如1G/s日志网络下的结果所显示的。对于10G/s日志网络来说影响就少多了。当容错在传送工作服在被启用时，传输的数据包没有被记录日志，但是网络中断必须被记录日志。日志带宽更低，所以可实现的网络传输带宽就高于网络接收带宽。总体来说，容错在非常高传输和接收率下会显著限制带宽，但是高绝对速率任然是可达到的。

# 相关工作

Bressoud 和Schneider [3]最早描述了有关在管理程序层的软件来实现虚拟机的容错。他们通过基于HP PA-RISC处理器的服务器，演示了备份虚拟机和主虚拟机保持同步可行性的一个原型。然而，由于PA-RISC架构的限制，他们不能实现完全安全隔离的虚拟机。而且，他们也没有实现任何故障检测的方法，或者尝试指出在第3章中所描述的实际问题。更重要的是，他们在他们的容错协议上强加了许多不必要的限制。首先，他们强加了一个时间周期的概念，其中异步事件被延迟直到设定时间间隔的末尾。时间周期的概念是不需要的 - 他们可能是因为不能足够高效的重放独立的异步事件才强加了这个。其次，他们要求主虚拟机停止执行直到备份虚拟机接收并确认所有之前的日志条目。然而，只有输出本身（例如一个网络包）需要延迟 - 主虚拟机本身可以继续执行。

Bressoud [4]描述了一个在操作系统（Unix类操作系统）中实现容错的系统，因此对所有运行在那个系统上的应用都提供了容错。系统调用接口成为了必须确定复制的操作集合。这个工作和基于管理程序的工作有类似的限制和设计选择。

Napper et al. [9] 、Friedman 和 Kama [7] 描述了容错的Java虚拟机的实现。他们采取了跟我们相似的设计，通过日志通道发送有关输入和非确定性操作的信息。就像Bressoud，他们似乎不关注故障检测和故障之后的重新建立容错。另外，他们的实现只限于对运行在Java虚拟机上的应用提供容错。这些系统尝试处理多线程的Java应用，但是要求，或者所有的数据被锁正确的保护，或者强制对共享内存必须顺序访问。

Dunlap et al. [6] 描述了一个在超虚拟化系统上用于调试应用软件的确定性重放的实现。我们的工作支持运行于虚拟机中的任何操作系统，并且对这些虚拟机提供容错支持，它需要更高的稳定性和性能。

Cully et al. [5]描述了另外一种支持容错虚拟机的方法，它的实现在一个叫做Remus项目中。用这种方法，主虚拟机的状态在执行过程中被重复的设置检查点，然后发送给备份虚拟机，它收集检查点的信息。这些检查点必须被非常频繁的执行（每秒钟很多次），因为外部输出必须被延迟，直到随后的检查点被发送并确认。这种方法的优点是，它对于单处理器和多处理器虚拟机应用的一样好。这个方法的主要问题是它要求非常高的网络带宽，用于发送每一个检查点中内存的增量改变。在[5]中的结果显示了，当Remus尝试每秒做40个检查点，使用1G/s网络连接来传输内存状态的改变，Remus对于内核编译和SPECweb基准测试会减慢100%到225%。可能有许多有用的优化可以减低带宽的要求，但是很显然不能在1G/s连接的情况下实现合理的性能。相反，对于几个实际应用，我们基于确定性重返的方法可以达到少于10%的开销，只使用主和备份主机之间不到20M/s的带宽。

# 结论与未来工作

在VMware vSphere中，我们设计并实现了一个高效完整的系统，它可以对运行在集群服务器上的虚拟机提供容错（FT）。我们的设计是基于在另外一个主机上的备份虚拟机使用VMware确定性重放技术来复制主虚拟机的执行。如果主虚拟机所在服务器故障了，备份虚拟机立即接管，没有任何中断或者数据丢失。

总体来说，普通硬件上容错虚拟机的性能非常好，对于一些典型应用它显示了少于10%的开销。VMware容错的大部分性能成本来自使用VMware确定性重放来保持主和备份虚拟机同步。VMware容错的低开销来源于VMware确定性重放的高效率。另外，保持主和备份虚拟机同步所需的日志带框通常相当小，时常少于20M/s。因为在大多数情况下，日志带宽相当小，所以实现主和备份虚拟机长距离（1-100千米）配置是可行的。因此，VMware容错可以使用在那些防止整个场所故障的灾难的场景。值得指出的是，日志流通常是可以压缩的，简单的压缩技术可以额外的一小部分CPU开销来显著的减少日志带宽。

我们VMware容错的结果显示了，容错虚拟机的一个有效实现可以基于确定性重放创建。这样一个系统能够，以最小开销，透明的为运行任何操作系统和应用的虚拟机提供容错。然而，对于客户有用的容错虚拟机系统来说，它也必须是健壮的、易于使用的和高度自动化的。一个可用的系统除了复制状态机执行之外还需要很多其他组件。特别是，VMware容错自动在故障之后恢复冗余，它在本地集群中先找到恰当的服务器，然后创建一个新的备份虚拟机。通过指出这些所有必需的问题，我们演示了一个可以用于客户数据中心实际应用的系统。

通过确定性重放实现容错的一个折衷是，目前的确定性重放只能在单处理器虚拟机高效实现。然而，单处理器虚拟机对于很多应用已经足够了，特别是因为物理处理器会持续变得更强。另外，很多工作负载可以通过使用很多单处理器的虚拟机来扩展，而不是通过一个大的多处理器虚拟机。多处理器虚拟机的高性能重放目前正在研究中，有可能通过微处理器中一些额外的硬件支持来启用。一个有趣的方向是通过扩展事务内存模型来促进多处理器重放。

未来，我们也有兴趣扩展我们的系统来处理部分硬件故障。对于部分硬件故障，我们指的是部分功能的缺少或者不会引起损坏或者数据丢失的服务器冗余。一个例子是丢失所有跟虚拟机的网络连接，或者一个物理服务器上冗余电源的丢失。当运行主虚拟机的服务器上发生部分硬件故障时，很多时候（但不是所有）立即故障迁移到备份虚拟机是有益的。这种故障转移可以立即恢复一个关键虚拟机的完整服务，并且保证了快速去掉潜在不可靠服务器上的虚拟机。

**致谢**

We would like to thank Krishna Raja, who generated many of the performance results. There were numerous people involved in the implementation of VMware FT. Core im-plementors of deterministic replay, (including support for a variety of virtual devices) and the base FT functional-ity included Lan Huang, Eric Lowe, Slava Malyugin, Alex Mirgorodskiy, Kaustubh Patil, Boris Weissman, Petr Van-drovec, and Min Xu. In addition, there are many other people involved in the higher-level management of FT in VMware vCenter. Karyn Ritter did an excellent job man-aging much of the work.

# 参考

[1]Alsberg, P., and Day, J. A Principle for Resilient Sharing of Distributed Resources. In Proceedings of the Second International Conference on Software Engineering (1976), pp. 627–644.

[2]AMD Corporation. AMD64 Architecture Programmer’s Manual. Sunnyvale, CA.

[3]Bressoud, T., and Schneider, F. Hypervisor-based Fault Tolerance. In Proceedings of SOSP 15 (Dec. 1995).

[4]Bressoud, T. C. TFT: A Software System for Application-Transparent Fault Tolerance. In Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerance Computing (June 1998), pp. 128–137.

[5]Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchison, N., and Warfield, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In Proceedings of the Fifth USENIX Symposium on Networked Systems Design and Implementation (Apr. 2008), pp. 161–174.

[6]Dunlap, G. W., King, S. T., Cinar, S., Basrai, M., and Chen, P. M. ReVirt: Enabling Intrusion Analysis through Virtual Machine Logging and Replay. In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (Dec. 2002).

[7]Friedman, R., and Kama, A. Transparent Fault-Tolerant Java Virtual Machine. In Proceedings of Reliable Distributed System (Oct. 2003), pp. 319–328.

[8] Intel Corporation. IntelA R 64 and IA-32 Architectures Software Developer’s Manuals. Santa Clara, CA.

[9] Napper, J., Alvisi, L., and Vin, H. A Fault-Tolerant Java Virtual Machine. In Proceedings of the International Conference on Dependable Systems and Networks (June 2002), pp. 425–434.

[10] Nelson, M., Lim, B.-H., and Hutchins, G. Fast Transparent Migration for Virtual Machines. In Proceedings of the 2005 Annual USENIX Technical Conference (Apr. 2005).

[11] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. In Proceedings of the 2006 Symposium on Operating Systems Design and Implementation (Nov. 2006).

[12] Schlicting, R., and Schneider, F. B. Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems. ACM Computing Surveys 1, 3 (Aug. 1983), 222–238.

[13] Schneider, F. B. Implementing fault-tolerance services using the state machine approach: A tutorial. ACM Computing Surveys 22, 4 (Dec. 1990), 299–319.

[14] Stratus Technologies. Benefit from Stratus Continuing Processing Technology: Automatic

99.999% Uptime for Microsoft Windows Server Environments. At http://www.stratus.com/˜/media/-Stratus/Files/Resources/WhitePapers/continuous-processing-for-windows.pdf, June

2009.

[15] Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., and Weissman, B. ReTrace: Collecting Execution Traces with Virtual Machine Deterministic Replay. In Proceedings of the 2007 Workshop on Modeling, Benchmarking, and Simulation (June 2007).