关于 Eve 的一切：面向多核心服务器的执行-验证复制

All about Eve: Execute-Verify Replication for Multi-Core Servers

翻译: 孙锴(sunkaicn@gmail.com)

Manos Kaprisos*,Yang Wang* ,Vivien Quema†, Allen Clement‡, Lorenzo Alvisi*, Mike Dahlin*

*The University of Texas at Austin †Grenoble INP ‡MPI-SWS

摘要：本文提出了 Eve，一个可以将状态机复制（state machine replication）扩展到多核心服务器的新型执行-验证（execute-verify）结构。与传统的状态机复制所采用的一致-执行（agree-execute）结构不同，在 Eve 中副本（replica）首先并发地*执行*批量的请求，然后再*验证*副本状态的一致性；如果验证表明副本的状态不一致，那么副本将顺序地重新执行这些请求。通过采用与应用相关的准则，Eve 将收到的请求编成小组，使得被分在同一组的请求基本不存在相互干扰，从而使得 Eve 可以将副本出现不一致的可能性降到最低。我们的评测表明，Eve 所具有的独特的结合执行独立性（execution independence）与非确定性请求交错（non-deterministic interleaving of requests）的特性使得在具有广泛的容错能力（包括但不限于难以捉摸的并发错误）的前提下多核心服务器的高性能复制成为可能，

1. 引述

本文提出了 Eve，一个可以将状态机复制扩展到多核心服务器的新型执行-验证结构。

状态机复制（SMR）是一种强有力的容错技术 [26, 38]. 历史上，它的本质思想是让副本确定性地处理相同顺序的请求，从而使得所有正确的副本都会以同样的顺序经历同样的内部状态并产生同样的输出序列。

多核心服务器对 SMR 提出了挑战。为了充分地利用并行的硬件资源，现代的服务器并行地执行多条请求。然而，如果不同的服务器以不同的方式交替请求的指令，即使没有错误发生，正确运转的服务器的状态和输出也有可能出现不一致。因此，大多数 SMR 系统都要求服务器顺序地处理请求：在开始执行下一条请求前，副本需要完成当前请求的执行 [7, 27, 31, 39, 44, 50]。

初看起来，为了克服这一僵局，近来对增强确定性并行执行所做的努力似乎提供了一种有希望的解决方法。不幸的是，这些努力是有不足的。它的不足不仅仅源于当今系统实现中的实际限制（如高昂的系统开销 [2, 3, 5]），还有更为根本性的原因：为了获得更好的工作性能，许多现代复制技术的算法实际上并没有在每一个副本上以同样的顺序执行操作。（并且，有时甚至执行的都不是同一套操作）[7, 11, 21, 47]。我们将在下一节中更详细地说明这一论点。

为了避免这些问题，Eve 的复制结构不再要求每个副本以相同顺序执行请求。作为替代，Eve 将请求划分为若干批次，首先用轻量级的方法使得同一个批次内的请求基本不存在冲突，然后 Eve 允许不同的副本在每个批次内并行地执行请求。Eve“投机性地”期望这些并行执行的结果（即每个副本的系统的重要状态和输出）在足够多的副本中将会一致。

为了在不违反副本协调的安全要求下并行地执行请求，Eve 完全改变了既定的 SMR 结构。在传统技术中，确定性的副本首先在将被执行的请求的执行顺序上*达成一致*，然后再*执行*他们 [7, 26, 27, 31, 38, 50]; 在 Eve 中，副本首先“投机性地”并发地*执行*请求，然后再*验证*这些副本在正确状态和输出上的确达成了一致。如果过多的副本出现了不一致以至于正确的状态或输出无法被甄别，Eve 会顺序且确定性地重新执行这些请求从而保证安全性（safety）和正常运转（liveness）。

尽管允许并行执行引入了不确定性，但是副本很少会出现不一致，并且即使不一致确实出现了，它也会被有效的检测并解决掉。对 Eve 工作性能影响至关重要的正是上述这样的一套机制。Eve 中有一个混合器（mixer）阶段，这一阶段产生若干请求小组，与应用相关的准则在这一阶段的应用使得产生的每个请求小组中的请求基本不存在相互干扰。通过混合器阶段，Eve 将副本出现不一致的可能性降到了最低。我们注意到：给定一个程序，如果它在未采用复制的并行执行中是正确的，那么在采用复制的情形中，我们将一致性检查推迟到执行完毕后，同时在一致性检查后，如果必要我们可以顺序地重新执行它，那么即使混合器允许同一请求小组中存在相互干扰的请求，我们仍可以保证副本安全和正常运转。

Eve 的执行-验证结构是通用的，它既可以用在崩溃容错系统（crash tolerant system），也可用在拜占庭容错系统（Byzantine tolerant system）。特别的，当 Eve 被配置在前者上时，它还可以额外地提供显著的对于并发错误的保护。因此，相比拜占庭容错 Eve 可以获得更多的设计上的空间，同时相比标准的崩溃容错 Eve 提供了更可靠的保障。Eve 的鲁棒性源于两点。其一，通过尝试仅仅并行地执行那些基本不可能存在冲突的请求，Eve 的*混合器*降低了引起潜在并发错误的可能。其二，Eve 的*执行-验证*结构使得它可以检测出由并发导致的执行不一致问题，同时予以恢复，并且实际上 Eve 这种对不一致的检测能力并不局限于由并发导致的错误，实际上 Eve 也可以检测出由于不同的正确副本做出了不同的合法动作而产生的不一致。

实质上，Eve 对传统的 SMR 的实现所做的基本假设做了进一步的精炼。在一致-执行结构中，“正确的副本*在状态与输出上保持一致*”这一安全性要求被降低为“保证确定性的副本处理相同的命令序列”（即，*“输入相同”*）。Eve 在保证要求副本保持确定性的基础上，对此做了进一步精炼，

Eve 不再要求副本处理相同的命令序列：Eve 不再依赖输入的一致性，转而选择了一种弱化版本的原始安全性要求——副本在状态和输出上保持一致。

在这种假设下的结果是，在 Eve 中，正确的副本同时拥有两点特性：*请求的非确定性交替*和*独立的执行*。而这两点特性在先前的副本协调协议中被视为从根本上相斥。事实上，正是通过对这两点特性的组合，Eve 提高了多核心服务器复制的水平：

(1) *请求的非确定性交替使得 Eve 能够为多核心服务器提供高性能的复制*。通过避免由确定性产生的额外系统开销，Eve 获得了性能的提升。例如，我们在 TPC-W 基准测试的实验中，原始的不采用复制的服务器相比顺序执行的服务器复制可获得 7.5 倍的速度提升，而 Eve 相比顺序执行的服务器复制，可获得了 6.5 倍的速度提升，这与原始的不采用复制的服务器的速度是接近的。另外，在同样的基准测试中，Eve 通过利用它独一无二的允许独立的副本非确定性地交替处理请求，相比 Remus 主-备（primary-backup）系统，Eve 取得了 4.7 倍的速度提升。

(2) *执行的独立使得 Eve 具有宽泛的容错能力*。如果没有独立执行的副本，那么一般来说是不可能容忍任意的故障的。独立性使得 Eve 的结构非常的通用，例如我们的原型支持对容错能力的调节，保持了传统的 SMR 可被配置为崩溃容错，缺失容错，或拜占庭容错的能力。值得注意的是，我们发现，执行独立还有提供了额外的优点，即使 Eve 被配置为只容忍崩溃故障或缺失故障，执行独立性可以屏蔽一部分并发错误。尽管我们不能肯定我们的实验结果具有一般性，我们发现它们是有前景的：对于在 H2 数据库上运行的 TPC-W 基准测试，一个过去未诊断出的并发错误在不采用复制的服务器上并行地处理 75 万个执行请求时发生了 73 次。在 Eve 下，我们的混合器将这个错误全部地消除了。另外，当我们修改混合器，使得它在部分时刻允许有冲突的请求并行执行，由此导致的错误被 Eve 发现并纠正了 82%，这是因为 Eve 的执行独立性允许错误在不同的副本上以不同的方式展现（或不展现）。

本文的后续部分如下安排：在第 2 节，我们解释确定性多线程执行为什么无法解决多线程服务的复制问题。第 3 节介绍系统模型。第 4 节给出了协议的概述。第 5 节我们更细致地讨论执行阶段。第 6 节我们给出被验证阶段用在两个有趣的案例的一致性协议，同时讨论 Eve 屏蔽并发错误的能力。第 7 节给出了对 Eve 的实验测评。第 8 节介绍了相关的工作。第 9 节总结全文。

2. 为什么不采用确定性的执行方式？

多线程程序的确定性执行[2, 3, 5, 30]是保证在相同输入下，所有正确的多线程应用副本将产生相同的内部应用状态和相同的输出。尽管初看起来这一方法似乎是一种对于多核服务器的多线程 SMR 挑战的完美解决策略，但是有两个问题值得我们更深入地分析它。第一个问题[4]是很直接的：目前的确定性多线程技术要么需要硬件支持[14, 15, 20]，要么对于生产环境来说太慢（1.2 倍至 10 倍的系统开销）[2, 3, 5]。第二个问题是，在现代 SMR 协议和用于实现确定性多线程的技术之间存在语义鸿沟。

为了寻找获得更高吞吐量的机会，近几年来 SMR 协议一直在寻找对副本所处理请求的语义的利用方式，以便在不强制所有副本处理相同的输入序列的前提下，实现副本的协调。例如，许多现代 SMR 系统不再坚持要求读入请求在所有的副本上都要以同样的顺序处理，这是因为读入请求不会改变应用副本的状态。这种*只读优化*[7, 9, 24]常常与另一种优化结合：读入请求仅仅在特殊选定的一组*仲裁*副本上执行，而不是在*所有*副本上执行[21]。后者这一优化被用于若干 SMR 系统[11, 47]，并且在这些系统的无故障执行期间被扩展至同样适用于会改变应用状态的请求，只有当某个仲裁副本出现故障时，才让其它的副本执行这些请求。

不幸的是，确定性多线程技术对它们所执行的操作的语义一无所知，它们能够保证多线程服务器的副本协调的能力纯粹基于一些语法机制。这一机制极强地依赖于“所有副本接收到相同的输入序列”的假设，因为只有这样，确定性多线程才可以保证副本的状态和输出都是相同的。只读优化和仲裁操作违反了这一假设，导致了正确的副本出现分歧。例如，只读请求会让一个副本的指令计数器增加，也可能导致副本获得额外的读入锁：我们容易构造出一个执行过程，使得如上所述的低级的差异或许会最终导致正确副本的应用状态出现分歧[22]。矛盾的是，确定性复制的困难源于坚持状态机方法[26, 38]，与此同时现代的 SMR 协议放松了它的要求，保留它的精神。

3. 系统模型

我们提出的状态机复制的新结构是非常通用的：无论是同步系统还是异步系统，Eve 都可以被用来协调多线程副本的执行，并且从容纳崩溃故障到容纳拜占庭故障，Eve 可以被配置为容忍各种严重程度的故障。

在本文中，我们主要针对异步环境，网络可以有任意长的延迟，重排或不危及安全性（safety）的信息丢失。对于正常运转（liveness），我们要求存在同步间隔：在这一间隔中网络状况良好，在两个正确结点之间的信息收发以及处理是在有界的延迟内。因为有可靠连接的同步的主-备是一个有趣的布局[13]，我们也在一个服务器对（server-pair）的布局上评估了 Eve，与主-备类似[6]，这个服务器对依赖对于安全性（safety）和正常运转（liveness）的时间安排的假设。

通过相应的设定，利用 Eve 结构可以做出保证正常运转的（live）系统，即，在总故障不超过 u 次的前提下，提供一个反馈给客户请求，这里的故障既可以是缺失故障（omission failure），也可以是委托故障（commission failure），同时可以确保在出现至多 r 次委托故障和任意多次的缺失故障的前提下，所有被正确客户接受的反馈都是正确的 [9]。委托故障包含所有的不属于缺失故障的故障。委托故障与缺失故障的并集就是拜占庭故障。不过，我们假定这些故障不会打破密码原语（cryptographic primitive）；即，一个出错的结点永远不会产生一个正确结点的 MAC。我们将一个包含鉴别码（一个 MAC 的向量，每个接收副本对应一个）的由 Y 发出的消息 X 记为$\langle X \rangle_{\overline{\mu}Y}$。

4. 协议概述

图 1 展示了 Eve 的整体的结构，它的“执行再验证”的设计与传统 SMR[7, 27, 50]的“一致再执行”的方法有所不同。

4.1 执行阶段

Eve 将请求划分为若干批次，然后让副本并行地执行同在一个批次内的请求，并不要求在同一批次内的请求在每个副本上具有相同的执行顺序。然而，对于每个批次，Eve 采

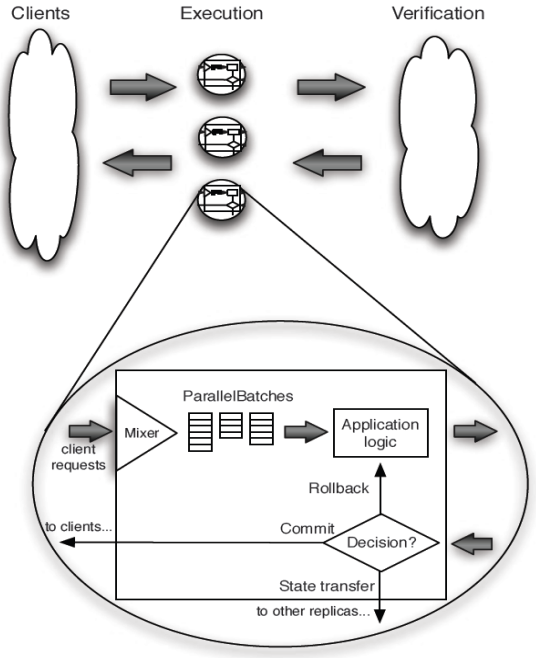

图 1：Eve 整体结构.

取了一系列步骤，设法让每个副本产生相同的最终状态和输出成为可能。

分批（Batching） 客户将他们的请求发送给当前的主执行副本（primary execution replica）。主执行副本将请求划分为若干批次，为每个批次分配一个序列编号，然后将它们送给所有的执行副本。不同批次的请求可能在同一时刻被发送，但是它们是按顺序执行的。伴随着请求的发送，主执行副本将一致化处理非确定性请求所需的所有数据（如，random()所需要的种子，gettimeofday()所需的时间戳[7,9]）一并发送出去。然而，主执行副本不做与“减少多线程副本独立执行这些批次可能引发的不确定性”相关的工作。

混合（Mixing） 每个副本通过运行相同的确定性的混合器，将从主执行副本获取到的每个批次划分为相同顺序的*并行批次（paralledBatch）*序列，混合器相信它所划分的处于同一并行批次中的请求具有这样的特性：即使我们在不同的副本上以不同的方式交替处理请求指令，副本也几乎不可能产生具有分歧的结果。例如，如果请求ρ_1和ρ_2都会修改对象A，那么混合器将把它们放入不同的并行批次中。第 5.1 节将更详细地给出混合器的描述。

并行执行（Executing in parallel） 每个副本按确定性混合器给出的顺序执行每个并行批次。在执行完第i个批次的所有的并行批次后，副本将它的应用状态和对应于每个请求的输出计算一个哈希值。这个哈希值，连同批次序号为$i-1$[1]的哈希值以及批次序号i，组成一个*标记（token）*被送到验证阶段，以便判断副本之间是否出现了分歧。第 5.2 节将描述我们如何有效地、确定性地计算最终状态和输出的哈希值。

4.2 验证阶段

Eve 的执行阶段努力试图将出现分歧的可能降低到很小，但是并不保证一定不会出现分歧：例如，尽管竭尽全力，混合器可能无意地将存在冲突的请求包括在同一个并行批次中，进而导致不同的正确批次产生了不同的最终状态和输出。这时就需要验证阶段来确保这类的分歧不会影响到安全性（safety），至多只是影响到性能。于是，在验证阶段的末尾，所有正确的执行了第i个批次的副本都会到达同一最终状态并产生了相同的输出。

一致（Agreement） 验证阶段执行一个一致性协议，来决定处理完每一批次的请求后所有正确副本的最终的状态和输出。一致性协议的输入是接收自执行副本的标记。协议的最终决策有两种：如果有足够多的标记是相同的，那么最终决策是*提交（commit）*；反之，如果有过多的标记不同，那么最终的决策是*回滚（rollback）*。特别地，协议首先检验副本中是否出现了不一致：如果所有的标记都是相同的，那么副本所共用的最终状态和输出被提交；如果存在不一致，那么协议收到的标记进行统计，试图找出一个被足够多副本所获得的最终状态和输出，以此来保证最终状态和输出是由正确的副本产生的，如果这样的最终状态和输出被找到了，那么 Eve 会确保所有正确的副本都最终提交到这个最终状态和输出，否则，协议会做出回滚的决定。

提交（commit） 如果验证阶段的结果是*提交*，那么执行副本会将相对应的序列编号标记为已提交（commited），并将对于并行批次的反馈发送给客户。

回滚（rollback） 如果验证阶段的结果是*回滚*，那么执行副本会将它们的状态回滚至最近一次提交的序列编号所对应的状态，并顺序地重新执行这一批次，以保证执行进度的进展。另外，为了应对主执行副本出错的情形，回滚可能会导致主执行副本的更换。为了保证执行进度的进展，由新产生的主执行副本创建的第一个批次（典型情况下它会包括一些回滚请求的子集），会在所有的执行副本上顺序地执行。

“执行-验证”结构的一个额外收获是，Eve 时常可以屏蔽由并发错误导致的副本分歧，即由某些线程交错引发的应用的实际行为与预期行为的背离[18]。一些并发错误可能显现为委托故障；然而，因为这种错误通常是以一定概率发生的，并且它们不是关键错误，它们常可以被 Eve 原本只被设计用来容纳委托故障的配置所屏蔽。当然了，同每个使用冗余来容错的系统一样，在面对相互关联的错误时，Eve 是脆弱的，当太多的副本同时以完全相同的方式形成并发错误时，Eve 无法防止它们（并发错误）的发生。也就是说，Eve 的结构在屏蔽并发错误导致副本分歧方面有两方面的因素：一方面，混合器通过避免将存在冲突的请求并行处理，使得并发错误产生的可能性降低；另一方面，即使并发错误存在，并发错误在不同的副本上的表现方式也可能不同。

---

[1] 之所以我们将前一个批次的哈希值包括进来，是因为我们希望确保系统只接受有效的状态转移。验证副本（verification replica）只会在当它们已经对于“存在一个已经提交过的$i-1$批次的哈希值与当前提交的批次i标记中记录的$i-1$批次的哈希值匹配”达成一致时，才会认为这个批次i的标记是有效的，并接收它。

5. 执行阶段

本节我们更细致地介绍执行阶段。特别地，我们讨论混合器的设计，和状态管理框架的设计和实现。后者使得 Eve 可以高效地进行状态比较，状态转移和回滚。

Transaction	Read and write keys
getBestSellers	**read**: item, author, order_line
getRelated	**read**: item
getMostRecentOrder	**read**: customer, cc_xacts, address, country, order_line
doCart	**read**: item **write**: shopping_cart_line, shopping_cart
doBuyConfirm	**read**: customer, address **write**: order_line, item, cc_xacts, shopping_cart_line

图 2：被用于 TPC-W 负载的 5 个最常用处理任务的键值

5.1 混合器设计

只有保证分歧只在极少数情形下才会出现，并行执行才能获得更好的表现。混合器的任务是分辨出哪些请求可以高效地并行执行，并且保证这一分辨出现漏报（false negative，即将不可并行执行的请求判断为可以并行执行）与误报（false positive，即将可以并行执行的请求判断为不可并行执行）的比例是很小的。漏报将会导致存在冲突的请求被并行执行，进而导致分歧与回滚的可能。误报将会导致本可有效并行执行的请求在实际中被顺序地执行，进而导致执行并行度的降低。需要注意的是，无论混合器的漏报与误报率如何，Eve 的安全性和正常运转总是有保证的，一个好的混合器只是会提高系统的性能（当然这一点也是很重要的）。

我们在实验中，对每个请求，我们采用的混合器会进行解析，尝试预测它（请求）将访问哪个状态：根据应用的不同，状态的定义可以从一个单个文件、一个应用级别的对象到如数据库的行，数据库的表等高级别的对象。当两个请求访问了同一个对象，且至少有一个请求是写这个对象的（即只有“读/写”与“写/写”两种情况）时，这两个请求是冲突的。为了避免将两个有冲突的请求放到一起并行执行，混合器开始时有一个初始为空的并行批次和两个初始为空的哈希表，其中一个哈希表用于对象的读取，一个哈希表用于对象的写入。接下来混合器依次扫描每个请求，将请求中的对象操作视具体情况映射为读或写的键值。然后混合器判断请求的键值是否与两个哈希表中的键值存在“读/写”或“写/写”冲突。如果没有，混合器将请求加入并行批次，并将请求的键值视操作类型加入两个哈希表；如果出现了冲突，混合器尝试将请求放入一个不同的并行批次，如果请求与所有的现存的并行批次冲突，混合器会建立一个新的并行批次。

在 H2 数据库引擎和 TPC-W 负载上的测试中，我们简单地采用了在读或写模式访问的表的名字作为每一个处理任务[2]的读与写的键值（见表 2）。注意到即便混合器将请求错误地分类了，这也是安全的，所以我们不必要显式地捕获那些由数据库触发或浏览访问等我们可能看不到的行为而潜在生成的其它冲突。Eve 的验证阶段允许我们在不完美的情况下保证安全性（safe）。更进一步看，通过系统反馈，久而久之混合器是可以被提升的（例如，我们可以记录引起回滚的并行批次）。

尽管在一些情况下实现一个完美的混合器是一件繁琐的事情，我们期望对于许多有趣的应用和负载，我们可以用适度的努力写出一个好的混合器。数据库和键值存储是一类应用的例子，在这类应用中典型的请求均对应可能受影响的应用级别的对象，对于数据库来说这种对象是表，对于键值来说这种对象是值。到目前为止，我们的实验是激动人心的。我们的 TPC-W 混合器的搭建花费 10 小时的学生时间（student-hour），并且在此之前我们对 TPC-W 的代码并不熟悉。从第 7 节的展示可以看出，这一简单的混合器具有很好的并行度（只有可以接受的极少数误报），并且我们没有观察到任何回滚（极少或没有漏报）。

5.2 状态管理

一致-执行结构变为执行-验证结构为状态的检查点、比较、回滚、转移的实现增加了压力。例如，Eve 中副本在执行完每个批次的请求后，都必须计算出到达的应用状态的哈希值；相反，传统的 SMR 协议在检查点和应用状态比较方面都不是那么的频繁（例如，在请求日志被垃圾回收时）。

为了实现有效的状态比较以及细粒度的检查点和回滚，Eve 采用一个写入时复制的 Merkle 树来存储状态。这棵树的根是整个状态的简明代表。这一实现借鉴了 BASE[36]的两点思想。其一，它只存储了决定状态机操作的状态，忽略了可能在不同副本上不同但却对于应用的状态和输出没有语义影响的其它状态（例如 IP 地址或一个 TCP 连接）。其二，它对于部分对象提供了一层抽象包裹，来掩盖不同副本上的差异。

在 BASE 以及其它如 PBFT，Zyzzyva，UpRight 等传统 SMR 系统上，程序员被要求手动标注出哪些状态应该被包含进状态机的检查点[7, 9, 24, 36]，类似的，我们在目前对 Eve 的实现中也手动标注了应用的代码，以指示出那些应被加入 Merkle 树并在它们被修改时标记为“脏”（dirty）的对象。

然而与 BASE 相比，Eve 面临两个新的挑战：在并行执行和并行哈希生成的前提下维护一个确定性的 Merkle 树结构，以及因为我们选用 Java 实现 Eve 所引起的相关问题。

5.2.1 确定性 Merkle 树

为了实现同样的校验和（checksum），不同的副本必须将同样的对象放在 Merkle 树的同一个位置。对于单线程执行来说，我们可以容易地获得确定性，因为我们只需要在一个对象创建时把它加入树中就行了。对于多线程执行来说，确定性是更具有挑战性的，因为对象可能被并发地创建。

直觉上有两个方法可以解决这个问题。第一个方法是同步地且确定地进行内存申请。这一方法不仅将并发内存申请所做的努力都忽视掉了[17, 40]，而且是不必要的，因为内存申请顺序通常不会根本性地影响副本的一致。第二个方法是根据对象的内容产生一个 ID，利用这个 ID 决定对象在树中的位置。然而这种方法也有问题，因为许多对象（尤其是在刚创建时）具有同样的内容。

我们的解决策略是，把新建对象加入 Merkle 树的时间推迟到当前批次的末尾，这时它们便可以确定性地加入 Merkle 树。Eve 扫描现存的被修改过的对象，如果某个对象包含一个对不在树中的对象的引用，那么 Eve 会将这个对象加入树

[2] 因为 H2 不支持行级别的锁定，我们没有实现比表（table）级别的冲突检测更细粒度的冲突检测。

的下一个空位，对于所有新加的对象，Eve 迭代地重复上述过程。

因为两个原因，对象扫描是确定性的。首先，现存对象已经被放入树中确定性的位置。其次，对于单个对象，Eve 可以按照确定性的顺序迭代它所有的引用。通常我们可以使用定义引用的类中所定义的序，然而一些类，如 Hashtable，并没有确定性地存储它们的引用，我们在 5.2.2 节讨论如何处理这部分类。

我们没有并行化扫描新对象这一过程，因为这一过程的系统开销很低。然而，我们并行化了哈希生成：我们将 Merkle 树分割成若干子树，并行地计算子树的哈希值，然后将它们合并，并计算出 Merkle 树树根的哈希值。

5.2.2 Java 语言及运行

使用 Java 实现我们的原型为我们提供了若干有利的特性，这些有利特性之一是它为我们提供了一个将引用与其它数据分离的简单的方法，这简化了确定性扫描的实现；同时，使用 Java 也带来了一些挑战。

其一，Merkle 树保存的有引用的对象不能被 Java 的自动垃圾回收处理。我们的解决方法是定期地进行一次 Merkle 树级别的扫描，使用一种类似于 Java 自带的垃圾回收的标记-清扫（mark-and-sweep）算法，来寻找未使用的对象，并将它们从树中移除，这样就保证了这样对象可以被 Java 的垃圾回收功能正确地回收。对于我们已经分析过的应用，这一扫描的频率可以低于 Java 的垃圾回收，因为树中的对象有较高的可能性是“重要的”并有较长的生存周期。在我们的实验中，这一扫描不是系统开销的主要来源。

其二，在 Java 中许多标准的集合型数据结构——包括广泛采用的 Hashtable 和 HashSet 类——与构造顺序是有关联的。例如，一个 Java 的 Hashtable 对象的序列状态是敏感于值的插入和删除顺序的。因此，两个集合型数据结构在不同的副本上即使包含有相同的元素，它们也可能产生不同的校验和，当这种校验和被加入 Merkle 树时，尽管它们在语义上是等价的，但是这两个副本的状态却会被看成已经出现分歧了，从而导致了没有必要的回滚。

对此我们的解决方法是建立一层包裹[36]，通过这层抽象，将处于不同副本上语义层面上相同的不同集合型类的实例视作相同的实例。对于每个集合型数据结构，这层包裹可以生成一个确定性的包含该数据结构中所有元素的列表，并且如果必须，可以生成一个与之对应的迭代器。如果元素的类型 Java 已提供了比较器（如，Integer，Long，String 等），这是容易做到的。否则，这些元素在被加入集合型数据结构之前，Eve 会赋给这些元素一个有序对$(requestId, count)$，然后在包裹层生成列表之前，元素会依照这一有序对来排序。这里，$requestID$是请求的唯一标识，用于元素的增加，$count$表示的是由标识为$requestID$的请求已向数据结构中增加的元素的数量。实际中，我们发现只需要生成两个包裹，分别对应 Java 中常用的集合型数据结构接口 Set 和 Map。

6. 验证阶段

验证阶段的目标是判断在执行完一个请求批次后，是否有足够多的执行副本在它们的状态和反馈（输出）上是一致的。假定由执行副本产生的标记能够反映它们的当前状态以及它们经历的状态变换，验证阶段所需要做的仅仅是判断是否有足够多的标记是一致的。

为了做出这一判断，验证部分采用了一个一致性协议[7, 27]，这一协议的细节设定极大依赖于系统模型。作为优化，只读请求首先尝试在众多副本上执行时不参与验证阶段。如果足够多的反馈是一致的，那么客户便接受返回的值；否则，只读请求会被重新下发，并且按照处理一般性请求的方式处理。我们展示对于两种极端情形的协议：一个异步拜占庭容错系统和一个同步主-备系统。然后我们讨论验证阶段怎样提供对于并发错误的抵御，以及它如何被配置到可以最大化容忍并发错误的数量。

6.1 容许异步拜占庭故障

这一节我们描述一个用于异步拜占庭容错系统的验证协议，其中有$n_E = u + max(u, r) + 1$个执行副本和$n_v = 2u + r + 1$个验证副本（verification replica）[8, 9]，它可以保证在u个错误（无论是缺失故障还是委托故障）内系统正常运转（live），并保证在r个委托故障和任意多缺失故障内系统是安全的（safe）。熟悉 PBFT[7]的读者将发现这两个协议之间是有许多相似的；这并不令人惊讶，因为这两个协议都是尝试在$2u + r + 1$个（若用 PBFT 的定义，都是$3f + 1$个）副本上处理一致性问题。这两个协议的主要区别在于两个因素。第一，在 PBFT 中，副本试图对于单个结点（主执行副本）上的输出达成一致。在 Eve 中，一致性的对象是若干副本（执行副本）的表现。因此，Eve 中验证副本采用来自执行副本的标记的仲裁作为它们“提议”的值。第二，在 PBFT 中，副本会尽力在状态机的输入上（输入的请求以及它们的顺序）达成一致。与 PBFT 不同的不是，Eve 中，副本会尽力在状态机的输出上（应用的状态以及对于客户的反馈）达成一致。就其本身而论，在视图转换协议（view change protocol）（由篇幅原因我们只能在[22]做更详细地讨论）中，如果知道了存在给定序列编号的证明，那么就足以将这一序列编号提交到下一个视图（view）——不再需要已提交序列编号的前缀。

当一个执行副本执行一个批次请求（也就是说，一个并行批次的序列）时，它将消息$< \text{VERIFY}, \upsilon, n, T, e >_{\vec{\mu}_e}$发送给所有的验证副本，这里$\upsilon$是当前的视图编号（view number），$n$是批次序列编号，$T$是由这一批次计算所得的标记，$e$是发送请求的副本。前文曾经提到过，$T$包含了批次$n$和批次$n-1$的哈希值：一个验证副本接受批次$n$的VERIFY消息，当且仅当它曾经提交过批次$n-1$的哈希值，并且这一哈希值与$T$中存储的批次$n-1$的哈希值是匹配的。

当一个验证副本收到$max(u, r) + 1$条标记匹配的 VERIFY 消息时，它将这一序列编号标记为*预先准备好的（preprepared）*，并将消息$< \text{PREPARE}, \upsilon, n, T, v >_{\vec{\mu}_v}$发送给所有其它的验证副本。与之类似地，当一个验证副本收到$n_V - u$条匹配的 PREPARE 消息时，它将这一序列编号标记为*准备好的（prepared）*并将消息$< \text{COMMIT}, \upsilon, n, T, v >_{\vec{\mu}_v}$发送给所有其它的验证副本。最终，当一个验证副本收到$n_V - u$条匹配的 COMMIT 消息时，它将这一序列编号标记为*已提交的（committed）*，并将消息$< \text{VERIFY} - \text{RESPONSE}, \upsilon, n, T, v >_{\vec{\mu}_v}$发送给所有执行副本。注意到这里视图编号$\upsilon$与 VERIFY 消息的视图编号是一样的；这表示一致被达成，并且不必要做视图转换。

如果一致无法被达成，那么这要么是因为副本出现了不一致，异步，要么是因为主执行副本出现了拜占庭故障，验证副本中出现了视图转换[3]。在视图转换期间，验证副本从已经被至少$n_v - u$个副本准备好的序列编号中找出序列编号最高者（以及对应的标记），并以这一标记展开一个新的视图。它们将消息$< \text{VERIFY} - \text{RESPONSE}, \upsilon + 1, n, T, v, f >_{\bar{\mu}_v}$发送给所有的执行副本，这里$f$是一个标识，它指示了下一个批次应当被顺序的执行，以保证执行进度的进展。注意到在这一情形里视图编号增大了，这表示一致没有达成，对于序列编号n的回滚是必要的。

提交，状态转移，和回滚 当接收到$r + 1$个匹配的 VERIFY-RESPONSE 消息时，一个执行副本e辨别以下三种情形：

提交 如果视图编号没有增大，并且得到一致认可的标记与e发送的标记匹配，那么e将序列编号标记为稳定，将过时的状态进行垃圾回收，并将这一批次的请求的计算结果反馈给相应的客户。

状态转移 如果视图编号没有增大，但是得到一致认可的标记与e发送的标记不匹配，这意味着这一副本的状态与一致认可的状态出现了分歧。为了修复这一分歧，这一副本向其它副本发布一条状态转移请求。这一状态转移是增量的：相比改变整个状态，Eve 只改变在最后一个稳定序列编号之后的那些部分。增量式转移采用了 Merkle 树来辨别需要转移的状态，这使得即使出现分歧的副本很慢，Eve 也可以迅速地让它们更新到正确的状态。

回滚 如果视图编号增大，那么这意味着一致没有被达成。副本e丢弃掉所有未执行的请求，并将它的状态回滚至标记T指示的序列编号，同时验证这一新的状态与标记一致（如果不一致，它将进行一次状态转移）。视图编号的增大也隐含地轮换了主执行副本。副本开始从新的主执行副本接收批次，同时，因标识f的设置，副本会顺序地执行由新主执行副本创建的第一个批次，以保证执行进度的进展。

6.2 同步主-备系统

为同步主-备配置的系统只有两个副本，它们既负责执行，也负责验证。主服务器接收客户请求，并将请求划分为若干批次。当产生出一个批次$\mathcal{B}$后，它（主服务器）将消息$< \text{EXECUTE} - \text{BATCH}, n, \mathcal{B}, ND >$发送给从服务器。这里$n$是批次的序列编号，$ND$是一致性执行非确定性调用（如 random()、gettimeofday()）所需要的数据。主、从服务器均将混合器应用于批次，执行由混合器给出的并行批次，并计算状态的标记（如第 4 节所介绍的那样）。从服务器将它的标记发送给主服务器，主服务器将自己计算的标记与从服务器的标记比较，如果标记匹配，那么主服务器将这一序列编号标记为稳定，并将反馈发送给客户；如果标记不匹配，那么主服务器将它的状态回滚至最近一次的稳定的序列编号，并告知从服务器也进行同样的回滚操作。为了保证执行进度的进展，主、从服务器将顺序地执行下一个批次。

如果主服务器崩溃了，那么最终从服务器会注意到这一点，并承担起主服务器的任务（即成为新的主服务器）。在旧的主服务器的问题解决前，新的主服务器将独自执行请求。在旧的主服务器的问题解决后，在处理新的请求前，我们采用增量式状态转移将它的状态更新至最新。

6.3 容许并发错误

执行-验证结构的一个令人高兴的结果是，当 Eve 被配置为容忍u个缺失故障时，即使并且我们给予它容忍这些错误所需要的最少数量的副本，Eve 仍然可以提供对并发错误的屏蔽。

并发错误既可能导致缺失故障（如，一个副本可能卡主），也可能导致委托故障（如，一个副本可能产生一个错误的输出，或转移到一个错误的状态）。然而，由于并发错误导致的故障有一个重要的性质，而一般来说这一重要性质无法作为拜占庭容错中的一个假设：由并发错误导致的故障可以容易地修复。如果 Eve 发现了并发故障，它可以通过回滚并顺序地重新执行来修复它。

异步情形 当 Eve 被配置为$r = 0$时，它提供了以下保证：

定理 1. 当 Eve 被配置为$n_{exec} = 2u + 1$并且$r = 0$时，异步的 Eve 在不超过u个并发故障或缺失故障时，是安全的（safe），可正常运转的（live），和正确的（correct）。

注意到安全性（safety）和正常运转（liveness）对应了状态机复制的要求——前者对应了“正确副本提交的状态和输出是一致的”，后者对应了“请求最终会被提交”。正确性对应了状态机自身；一个提交的状态是正确的，那么它是一个可以由状态机在无错运行时达到的状态。

证明梗概： 系统总是安全和正确的，因为在提交一个批次时，验证器要求$u + 1$个匹配的执行标记。如果有至多u个并发故障并且没有其它委托故障，那么每个提交批次都有至少一个执行标记是由正确副本产生的。

这个系统是可正常运转的，这是因为如果一个批次无法收集$u + 1$个匹配的标记，验证器会强制执行副本回滚并顺序地重新执行。在顺序执行的过程中，确定性的正确副本不会出现分歧；因此重新执行至多会受到u个缺失故障的影响，并产生至少$u + 1$个匹配的执行标记，这保证了这个批次可以提交。 □

当超过u个相互关联的并发错误产生完全相同的状态和输出时，Eve 仍然可以保证状态机复制的安全性和正常运转要求，但是 Eve 不再能保证正确性。

同步情形 当 Eve 被配置为只有$u + 1$个执行副本时，如果u个副本遭受了缺失故障， Eve 可以继续在 1 个副本上运转。在这一配置下，Eve 没有额外的冗余，来在剩余的 1 个副本上屏蔽并发错误。

在好区间中的额外保护 所谓*好区间*是指，除了由并发错误导致副本错误或超时外，没有其它的副本错误与超时。在好区间中，无论是同步还是异步情形，Eve 都可以利用额外的冗余竭尽所能地保护$n_E - 1$个执行副本，使它们免受并发错误的影响。

例如，在同步主-备情形中，当两个执行副本都在运转，主服务器接收到自身和从服务器两个执行结果，如果执行结果不匹配，那么它令自己和从服务器回滚，并顺序地重新执行。因此，在一个好区间中，这一配置可以屏蔽一个副本的并发错误。我们期望这是一种普遍的情形。

[3] 当提交吞吐量比预期低时，这一视图转换会被触发，与[10]类似。

在同步和异步两种情形中，当 Eve 被配置为$r = 0$时，若所有n_E个执行副本在连续k个批次都提供了匹配且及时的回馈，Eve 便进入了*额外保护模式（extra protection mode）*（EPM）。当 Eve 处于 EPM 时，在验证器收到的执行回馈的数量达到最小的必要的数量后，验证器继续等待并接收所有n_E个回馈直到略微超时。如果验证器收到了所有n_E个匹配的反馈，那么验证器提交这一反馈的结果，否则，验证器要求执行副本回滚，并顺序地重新执行。接下来，如果验证器在允许略微超时情况下收到n_E个匹配的反馈，那么验证器提交这一反馈结果并仍然保留在 EPM 中；相反，如果顺序重执行并没有产生n_E个匹配且及时的反馈，那么验证器会怀疑产生了非并发错误，此时验证器会终止 EPM 以通过允许系统在更少的匹配反馈下向前进展，来保证系统的正常运转（liveness）。

7. 评估

我们的评测试图回答以下问题：

- 与传统的顺序执行方法比，Eve 的吞吐量有多大的提升？
- 与不采用复制的多线程执行以及其它复制方法比，Eve 的表现如何？
- 混合器以及其它负载特性对 Eve 的表现的影响如何？
- Eve 屏蔽并发错误的效果有多好？

我们通过利用键值存储应用和 H2 数据库引擎来回答这些问题。我们实现了一个简单的键值存储应用来展现 Eve 对于不同参数的敏感度的微基准测试度量。特别地，对于每个请求所需的执行时间的多少、应用对象的大小、混合器的准确度（包括误报和漏报），我们测试了将它们设置为不同大小会对 Eve 产生怎样的影响。对于 H2 数据库引擎，我们采用了 TPC-W 基准测试的一个开源实现。为了表述的简洁，我们将给出它浏览工作负载的结果，因为浏览工作负载有更高的并发度。

我们当前的原型忽略了一些前文描述的特性。特别地，尽管我们在 6.3 节实现了对于同步主-备复制的额外保护模式这一优化，我们没有为我们的异步配置实现它。另外，在我们当前的实现中，如果一类对象， Java 的 *finalize* 方法会为它修改某些需要在副本间保持一致的状态，那么我们不能处理包含这类对象的应用。最后，我们当前的原型仅支持存放于内存中的应用状态。

我们在一个 Emulab 试验平台上运行我们的微基准测试，

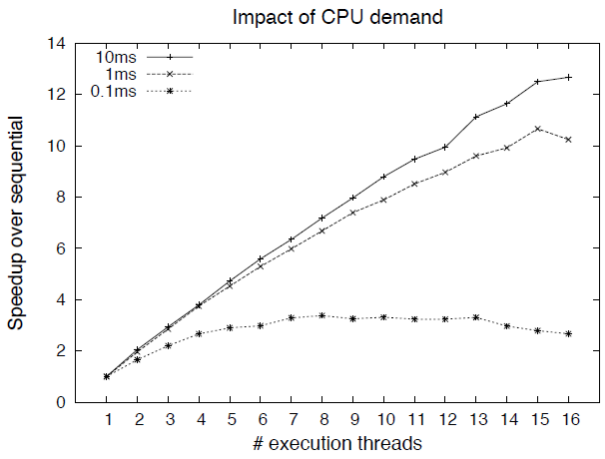

图 4：单个请求的 CPU 需求对 Eve 的吞吐量的性能提升的影响

我们采用了 14 个 4 核 Intel Xeon @2.4GHz，4 个 8 核 Intel Xeon @2.66GHz 和 2 个 8 核超线程 Intel Xeon @1.6GHz，它们以 1Gb 以太网连接。我们能够有限地访问 3 个 16 核 AMD

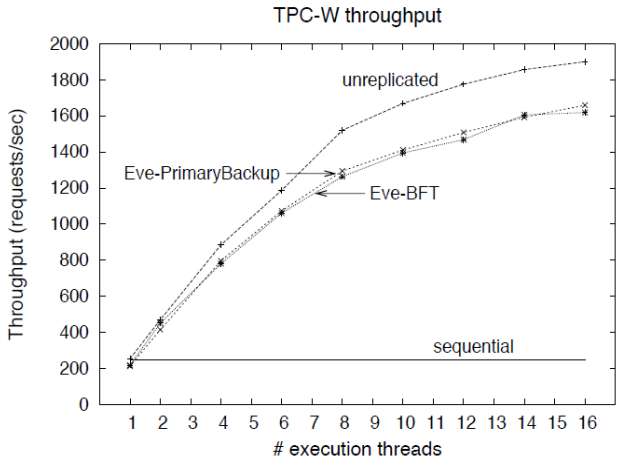

图 3：在 H2 数据库引擎上运行 TPC-W 浏览工作负载的 Eve 的吞吐量

Opteron @3.0GHz 和 2 个 8 核 Intel Xeon L5420 @2.5GHz。无论同步主-备，还是异步拜占庭故障容许的配置，我们均采用 AMD 的机器作为执行副本来在 H2 数据库引擎上运行 TPC-W 基准测试（图 3）。对于异步拜占庭故障容许的配置，我们采用 3 个执行结点和 4 个验证结点，它们足够容忍 1 个拜占庭故障（$u = 1, r = 1$）。L5420 机器上运行了 Xen，我们利用它们来展示我们与 Remus 的比较（图 10 和图 11）。

7.1 H2 数据库+TPC-W

图 3 展示了对于 H2 数据库引擎，Eve 在 TPC-W 浏览工作负载上的性能表现。我们报告以下两种配置下 Eve 的吞吐量：一个异步的拜占庭故障容许的配置（*Eve-BFT*）和一个同步的主动的主-备的配置（*Eve-PrimaryBackup*）。我们将它们与一个不采用复制的顺序执行的服务器的吞吐量进行比较（*sequential*）。注意到它代表了过去采用顺序执行[7, 9, 27, 31]的复制系统可达到的性能的上界。另外，我们也与一个不采用复制的并行执行的服务器的吞吐量进行了比较。

在 16 个执行线程下，与顺序执行相比，Eve 取得了 6.5 倍的性能提升。而不采用复制的 16 线程 H2 数据库服务器相比顺序执行也只能达到 7.5 倍的性能提升。

在两种配置下，在所有的数据点的所有运行过程中，Eve 都没有出现回滚。这表明了我们设计的简单的混合器从未将应顺序执行的若干请求放入同一并行批次。同时，良好的性能提升结果表明了，我们设计的简单的混合器在辨别给定请求是否可以并行执行上是足够有效的。

7.2 微基准测试

在这一节中，我们采用一个简单的键值存储应用来衡量不同参数的变化对 Eve 性能的影响。由于篇幅限制，我们只展示主-备配置的图表；异步复制的结果与此相似。后文除非特别指出，否则默认的工作负载是每一个请求消耗 1ms 执行时间，每一个请求更新一个应用对象，且应用对象的大小是 1KB。

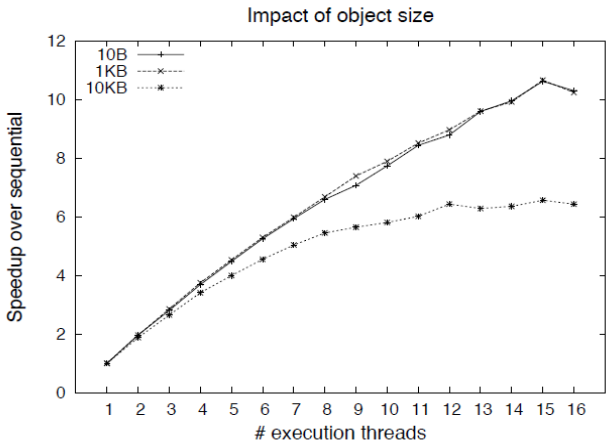

图 5：应用对象的大小对 Eve 的吞吐量性能提升的影响

图 4 展示了改变每个请求的 CPU 需求对 Eve 性能的影响。我们观察到，工作负载越重，即每个请求对 CPU 的需求越高（每个请求消耗 10ms 执行时间），性能提升幅度越大，在 16 线程上相比顺序执行至多有 12.5 倍的提升。当工作负载减轻，Eve 的系统开销变得越来越明显，当每个请求消耗 1ms 执行时间时，性能提升降至 10 倍，当每个请求消耗 0.1ms 执行时间时，性能提升降至 3.3 倍。3.3 倍的结果部分源于我们难以用工作负载轻的请求填满服务器负载。在我们的工作负载生成器中，客户每次有一个显著的请求，从而若要使服务器满载，需要大量的客户；这导致了我们的服务器在满载前已经用尽了套接字。通过测量，在这一试验中我们服务器的 CPU 利用率约为 30%。

图 4 中我们绘制了吞吐量的提升，可见提升的趋势是明显的。作为参考，对于图中对于 0.1ms 线，1ms 线，10ms 线，吞吐量的峰值（以请求/秒为单位）分别为 25.2K，10.0K 和 1242。

我们下一个实验探究了应用对象大小对系统吞吐量的影响。我们对于对象大小为 10B，1KB 和 10KB 分别进行了实验。图 5 给出了结果。当对象大小为 10B 和 1KB 时性能提升幅度较大，然而当对象大小变得更大时（10KB），性能提升的幅度变低。这是由我们采用的哈希库所导致的，因为我们采用的哈希库在计算一个对象的哈希前首先先需要拷贝这个对象：对于大的对象，这一内存拷贝过程限制了我们可能获得的吞吐量。注意到在这个图中，我们绘制的是性能提升而不是吞吐量的绝对值，以更好的表明各工作负载间的趋势。作为参考，对于图中 10B 线，1KB 线，10KB 线，吞吐量的峰值（以请求/秒为单位）分别为 10.0K，10.0K 和 5.6K。

接下来，我们评价了 Eve 对于不准确的混合器的敏感度。特别地，我们探究了 Eve 对于漏报（将冲突的请求报告为不冲突）和误报（将不冲突的请求报为冲突）的容忍的限度。这些参数的影响由一个关于两两冲突的概率（即两个请求可能冲突的概率）的函数衡量。为了做到这一点，实际中，我们让每个请求修改一个对象，然后变动应用对象的数量。例如，为了让冲突概率为 1%，我们创建 100 个对象。类似地，1%的漏报率意味着每一对冲突的请求有 1%的可能被分类为不冲突的。

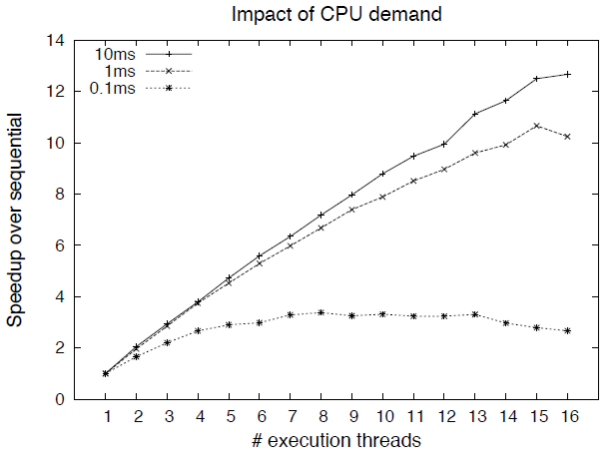

图 6：冲突概率和漏报率对 Eve 的吞吐量的影响

图 6 展示了漏报对于吞吐量的影响。首先可以观察到，即使漏报率为 0%，因可行的并行度的降低导致的两两冲突的概率的增加，也会导致吞吐量的降低。例如，如果批次中有 100 个请求，每个请求有 10%的概率与其它请求冲突，那么一个完美的混合器也只可能将这 100 个请求分为约 10 个并行批次，每个批次有约 10 个请求。

当我们增加漏报时，回滚就相应增加了。回滚的数量随着潜在的冲突概率和漏报率的增加而增加。注意到这一影响的增长速度比直观感觉要快，因为从本质上看这是一个生日悖论——如果我们有 1%的冲突概率和 1%的漏报概率，那么任意取一对冲突的请求，它们被漏报的概率为 1/10000。但是在有 100 个请求的批次中，每一个请求就有约 1%的概率被分入一个与之冲突的并行批次里，这意味着有约 39%的概率在这 100 个请求的批次中存在一个没有被检测到的冲突。进一步地看，当冲突概率有 1%时，给定批次将被分成数量不多的几个并行批次，因此有较大的可能我们会将冲突的请求分入同一个并行批次。事实上，在这一情形里，每 7 个并行批次的执行里就会出现 1 次回滚。不过虽然冲突的概率是比较高的，回滚次数是比较多的，但是相比顺序执行，Eve 仍然取得了 2.6 倍的性能提升。

图 7 展示了误报对吞吐量的影响。如预期的那样，增加

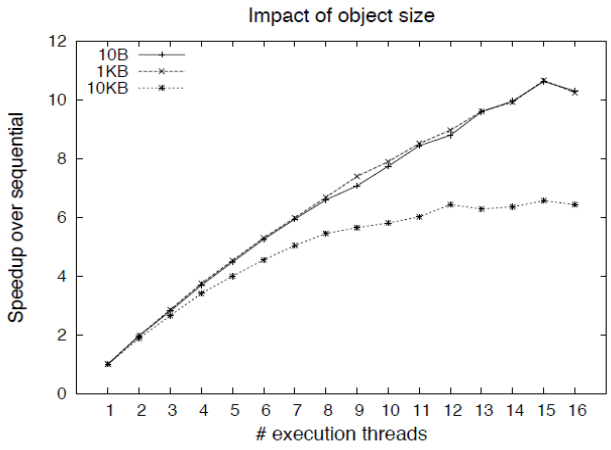

图 7：冲突概率和误报率对 Eve 的吞吐量的影响

误报的比例会降低吞吐量。但是这一影响并不像漏报那样强。原因是容易想到的：误报降低了并行执行的机会，但并不会招致额外的系统开销。

从这些实验中，我们可以总结出为了取得好的性能，Eve 确实需要一个好的混合器。不过这一要求并不特别令我们担忧。我们发现设计一个可以检测出所有冲突并且仍然保证较高数量的并行度的混合器是容易的（据我们所知）。其他研究者有着类似的经验[25]。尽管在一些情形里设计一个完美的混合器可能是困难的，我们认为通常情况下，为 Eve 设计一个具有低的漏报率和适度的误报率的混合器是可行的。

7.3 故障和恢复

在图 8 中，我们展示了 Eve 屏蔽故障以及从故障恢复的能力。在主-备配置中，我们运行了一个测试，测试中我们在$t = 30$秒时杀死主结点n_1，并在$t = 60$秒时恢复它（此时原从结点n_2已经成为了新的主结点）。接下来我们在$t = 90$秒时杀死新的从结点（n_1）并在$t = 120$秒时恢复它。我们观察到在第一次故障后，吞吐量降至 0，直到从结点在 4 秒[4]的超时后意识到主结点出了故障。接下来从结点开始担任主结点的角色，并开始处理请求。这一段时间的吞吐量较高，这是因为新的主结点知道另一个结点崩溃了，因此没有向它发送消息。在$t = 60$时，n_1恢复，在这接下来约 1 秒的时间内吞吐量降至 0，新恢复的结点利用这段时间更新状态。接下来吞吐量回到它的正常值。对于在$t = 90$时n_1再次崩溃以及$t = 120$时n_1再次恢复，与上述过程相同。

7.4 并发故障

为了评测 Eve 屏蔽并发故障的能力，我们采用了一个具有 16 个执行线程的主-备配置，运用多种混合器，在 H2 数据库引擎上运行 TPC-W 浏览工作负载。H2 数据库引擎有一个过去没有发现的并发错误，在这个错误中，当多个请求在 *read_uncommitted* 模式中访问同一个表时，一个行计数器的值没有正确地增加。我们标准的混合器完全地屏蔽了这个错误，因为它不允许多个请求并行地修改同一个表。通过引入准确性相对低的混合器，我们探究了 Eve 第二条防守线——并行执行——在屏蔽这一错误中的表现。

图 9 展示了这一错误在一个或两个副本中出现的次数。当这一错误只在一个副本中出现时，Eve 会发现副本出现了分歧，并通过回滚与重新顺序执行来修复这一错误。如果错误恰好在两个副本中同时以相同的方式发生，那么 Eve 无法检测到它。

第一列展示了当我们采用一个简单且激进的混合器的结果，这一混合器将批次i的所有请求放入同一个并行批次。在这一情形中，我们允许并行地执行一个批次中的所有的请求。自然地，这种情形下出现的错误的数量很多。我们观察到即使混合器对请求完全不做筛选，Eve 也能屏蔽 82%的错误。

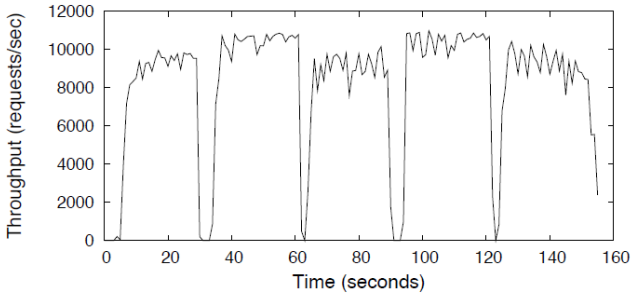

图 8：Eve 主-备配置在结点崩溃与恢复过程中的吞吐量

在剩余 18%的错误里，错误在两个副本中以相同的方式展现，因此无法被 Eve 修正。在第二列到第四列，我们引入了一个具有高漏报率的混合器。这导致了这一并发错误更少地显现，并且对于显现出的错误，Eve 会屏蔽掉大部分。在第五列，我们展示了我们的原始混合器的结果。我们的原始混合器没有引入漏报（就我们所知），因此在这一情形中，这一并发错误没有显现出来。

尽管我们不能肯定这些结果具有一般性，但是我发现它们最起码是有良好前景的。

7.5 Remus

Remus[13]是一个主-备系统，它采用虚拟机（VMs）将修改的状态从主副本发送给从副本。这一方法的优势是，它比较简单，不需要对应用做出修改。这一方法的一个缺点是，为了保证从副本与主副本的一致性，它过分地利用了网络资源。这一问题因为 Remus 的两点性质变得更加突出。第一，Remus 不做状态间的微粒区分，而这对于状态机和临时状态而言是必要的。第二，Remus 在虚拟机级别上操作，这迫使它需要发送整个页，而不是仅发送修改的对象。同时，因为 Remus 采用被动复制，它所能容许的故障的广度比 Eve 窄。我们的实验表明，尽管 Eve 给出的保障比 Remus 更高，Eve 在性能仍然是 Remus 的 4.7 倍，同时 Eve 利用的网络带宽比 Remus 小两个数量级。

图 10 展示了 Remus 和 Eve 在 TPC-W 基准测试的浏览工作负载上的吞吐量。同时我们也展示了对于相同负载，不采用复制的系统的潜伏期和吞吐量。两个系统均在 Xen 上运行 H2 数据库引擎，并采用 2 结点（主-备）配置。Remus 达到了最高每秒 235 个请求的吞吐量，Eve 最高达到了每秒 1225 个请求的吞吐量。当每秒请求数高于 235 时，Remus 崩溃了，因为它需要的带宽已经到了网络容许的上限（如图 11）。与 Remus 相比，Eve 在每个副本上独立地执行请求，并不需要在网络中传播状态修改的信息。这一实践结论是，相比如 Remus 的被动复制，Eve 利用了相对很少的带宽，实现了更高的吞吐量，同时提供了更强的保障。

	Group all	1% FN	0.5% FN	0.1% FN	Original Mixer
Times bug manifested	73	51	29	4	0
Fixed with rollback	60	38	18	3	0
All identical (not masked)	13	13	11	1	0
Throughput	1104	1233	1240	1299	1322

图 9：在使用不同混合器时，Eve 屏蔽并发错误的效果

[4] 可以用快速故障检测器[29]实现次秒级检测。

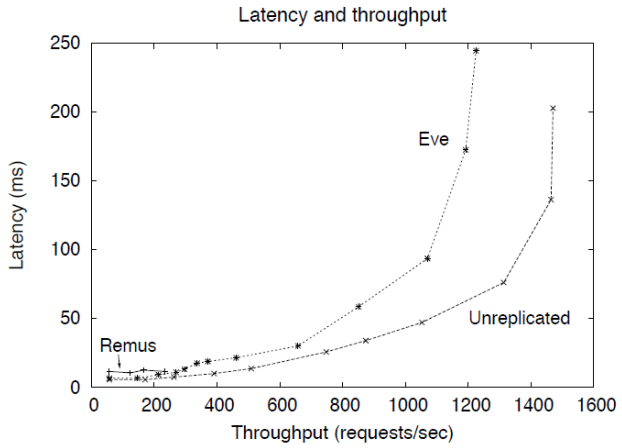

图 10：在 Xen 上，Remus 和 Eve 运行 H2 数据库引擎的潜伏期和吞吐量。两个系统均采用 2 结点配置。工作负载是 TPC-W 基准测试的浏览工作负载

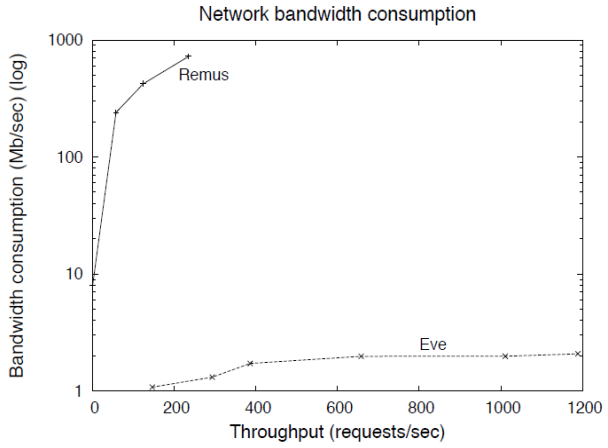

图 11：图 10 中的实验里 Remus 和 Eve 的带宽消耗

7.6 潜伏期与批次划分

从图 10 可以明了地看出 Eve 在潜伏期与吞吐量之间的权衡。当 Eve 的负载没有饱和时，它的潜伏期只是略微高于不采用复制的服务器。随着负载的增加，Eve 的潜伏期逐渐略微增加，直到最终在 Eve 负载达到饱和，即在每秒 1225 个请求时，它的潜伏期急剧增加；而没有采用复制的服务器的潜伏期在约每秒 1470 个请求时急剧增加。为了在保持最大吞吐量较高的同时使潜伏期尽量小，Eve 采用了一个动态批次划分的方案：当负载低时，批次的大小减小（这里假定潜伏期不长），当系统开始处于高负载时，批次的大小增大，从而尽可能地增大并行度。

8. 相关工作

Vandiver 等人[45]描述了一个对于交易处理系统的拜占庭容错的半主动复制方案。他们的系统支持对于询问的并发处理，但是可用的范围比较有限：它可用在交易处理系统的一个子集中，这里的交易处理系统必须采用严格 2 阶段加锁（strict two-phase locking）（2PL）。最近一篇论文表明在交易系统中强行采用确定性并发控制或许是可行的[41]，但是一般情形中仍然很困难。Kim 等人[23]最近提出将这一思想应用到一个交易操作系统中。这一方法假定所有的应用状态是可以由内核来管理的，并且这一方法不能处理在内存中的应用状态。

一个替代方案是，我们采用一种不是 SMR 的复制技术。*半主动复制（semi-active replication）*[34]在确定性和执行的独立性两方面上弱化了 SMR：一个副本是主副本，它非确定性地执行，并记录所有它执行的非确定性动作。接下来，所有其它副本确定性地执行，来重现主副本的选择。关于这一点，人们可能希望借助在确定性多处理器重新执行上的大量工作[1, 12, 16, 28, 32, 33, 37, 46, 48, 49]。不幸的是，放松对独立执行的要求使得这些系统在委托故障（commission failure）面前显得很脆弱。同时，与确定性多线程执行方法类似，记录和重新执行方法假定所有的副本被给予同样的输入。如第 2 节讨论的那样，这一假设在现代复制系统中是被违反的。

Remus 主-备系统[13]采用了另一种方法：从副本不执行请求，而是被动地接受来自主副本的状态更新：因为执行仅仅在主副本上发生，调度并行执行的代价和难度被避开了。然而，这一优势在也有巨大的代价：Remus 只能容许缺失故障，而无法容许任何的委托故障，包括常见委托故障的如并发错误。类似 Remus，Eve 既不跟踪，也不剔除非确定性，但是它在不减少错误容许类型的前提下实现了这两点；另外，尽管 Eve 对错误容许有更强的保障，Eve 的性能是 Remus 的 4.7 倍，并且带宽占用比 Remus 少两个数量级（见 7.5 节），因为 Eve 可以保证在不要求传送所有修改的状态的前提下，副本的状态达成统一。

之所以 Eve 可以结合执行独立性与非确定性请求交错，关键的一点是混合器的采用，混合器使得副本可以在低的交错几率的前提下，并发地执行请求。Kotla 等人[25]采用了一个类似的机制来提高拜占庭故障容许（BFT）复制系统的吞吐量，然而因为他们仍然采用了传统的一致-执行结构，他们系统的安全性（safety）依赖于一条假设：混合器采取的分类准则不会让冲突的请求并行执行，这使得一个未预料到的冲突便会导致系统的安全性被违背。

Eve 和 Zyzzyva[24]均采用先“投机性地”执行，然后再做一致性判断，但是 Eve 和 Zyzzyva 所采用的假设有根本的不同。Zyzzyva 依靠的是“正确的结点是确定性的”，从而输入的一致性就足以保证输出的一致：因此，一个副本只需要发送它已执行的请求的序列（的哈希值），以此来向客户发送它的状态。与此相反，即使正确的副本已经执行了相同的批次的请求，Eve 仍然不保证这些正确的副本处于相同的状态，因为混合器可能错误地把冲突的请求放于同一个并行批次。

我们确实考虑过在 Eve 的实现中，将验证阶段从备份服务的逻辑边界内移走，并类似 Zyzzyva 那样，将验证阶段移送给客户，从而来减少系统开销。例如，对于一个客户的请求，服务器的回复所包含的可以不仅仅只是反馈，还可以包括编码了服务器状态的 Merkle 树的根。然而，因为一致性并非我们所考虑的对象的瓶颈，我们最终选择听从 Aardvark[10]的教训，避开这一实现可能引入的极端情形。

9. 结论

Eve 是一个新的执行-验证结构，它使得状态机复制可以扩展到多核心服务器。通过重新分析在副本协调中“确定性”的角色，Eve 使我们有了崭新的 SMR 协议：我们首次让副本

能够同时满足非确定性地交替处理请求和独立执行。这一首创的组合是 Eve 扩展能力和通用性的关键，它使得 Eve 可以在同步、异步两种设置下都可以被配置为容忍缺失故障和容忍委托故障（commission failure）。另外，作为额外的收获，Eve 的非传统结构可以被容易地调节，使它可以提供低成本、高效能的并发错误防护。

致谢

感谢我们的领导人 Robert Morris，以及 OSDI 审稿人给予的具有深刻见解的评论。这项工作由 NSF grants NSF-CiC-FRCC-1048269 和 CNS-0720649 支持。

参考文献

[1] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In SOSP, 2009.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In OSDI, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. SIGARCH Comput. Archit. News, 2010.

[4] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? In 2nd Workshop on Determinism and Correctness in Parallel Programming, 2011.

[5] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dOS. In OSDI, 2010.

[6] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. In CDCCA, 1992.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst., 2002.

[8] A. Clement. UpRight Fault Tolerance. PhD thesis, The University of Texas at Austin, Dec. 2010.

[9] A. Clement, M. Kapritsos, S. Lee, Y.Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In SOSP, 2009.

[10] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In NSDI, 2009.

[11] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In OSDI, 2006.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In SOSP, 2011.

[13] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In NSDI, 2008.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In ASPLOS, 2009.

[15] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. In ASPLOS, 2011.

[16] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution replay for multiprocessor virtual machines. In VEE, 2008.

[17] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD, April 2006.

[18] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In DSN, 2010.

[19] H2. The H2 home page. http://www.h2database.com.

[20] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? Free will to choose. In HPCA, 2011.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: waitfree coordination for internet-scale systems. In USENIX, 2010.

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multicore servers (extended version). Technical Report TR-12-23, Department of Computer Science, The University of Texas at Austin, September 2012.

[23] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter. Improving server applications with system transactions. In EuroSys, 2012.

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In SOSP, 2007.

[25] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In DSN, 2004.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. CACM, 1978.

[27] L. Lamport. The part-time parliament. ACM Trans. Comput. Syst., 1998.

[28] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In ASPLOS, 2010.

[29] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the Falcon spy network. In SOSP, 2011.

[30] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In SOSP, 2011.

[31] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In OSDI, 2008.

[32] J. T. Pablo Montesinos, Luis Ceze. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In ISCA, 2008.

[33] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessor. In SOSP, 2009.

[34] D. Powell, M. Ch ́er ́eque, and D. Drackley. Fault-tolerance in Delta-4. ACM OSR, 1991.

[35] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In OSDI, 2006.

[36] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In SOSP, 2001.

[37] M. Ronsse and K. De Bosschere. RecPlay: a fully integrated practical record/replay system. ACM TCS, 1999.

[38] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys, 1990.

[39] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine-fault tolerance. In NSDI, 2009.

[40] Sun Microsystems, Inc. Memory management in the Java HotSpot virtual machine, April 2006.

[41] A. Thomson and D. J. Abadi. The case for determinism in database systems. VLDB, 2010.

[42] TPC-W. Open-source TPC-W implementation. http://pharm.ece.wisc.edu/tpcw.shtml.

[43] Transaction Processing Performance Council. The TPC-W home page. http://www.tpc.org/tpcw.

[44] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. CACM, 1996.

[45] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In SOSP, 2007.

[46] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In ASPLOS, 2011.

[47] T.Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT. In Eurosys, 2011.

[48] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In ISCA, 2003.

[49] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B.Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In MOBS, 2007.

[50] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In SOSP, 2003.