海量存储结课论文

**Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication**

高效可用的内存K-V存储与

混合删除编码和复制

院（系）名 称：计算机学院

专 业 名 称 ：计算机技术

学 生 姓 名 ：莫磊

学 生 学 号 ：2017282110300

摘 要

内存中的键/值存储（KV-store）是许多系统（如数据库和大型网站）的关键构建块。这种系统的两个关键要求是效率和可用性，这要求K-V存储每秒连续处理数百万个请求。一种常见的可用性方法是使用主备份（PBR）等复制方式，但是，这需要M + 1倍的内存来容忍M失败。这使得稀少的内存无法处理有用的用户作业。一种常见的可用性方法是使用主备份（PBR）等复制方式，但是这需要M+1倍的内存来容忍M次失败。 这使得稀少的内存无法处理有用的用户作业。

本文首先通过集成纠删编码来实现内存效率，同时不会显著降低性能，从而构建高度可用的内存式KV-store。一个主要的挑战是内存KV-store中有很多分散的元数据。由于对元数据进行了大量小的更新，单纯的KV放置可能会导致过度的编码操作和奇偶校验更新。我们的方法即Cocytus通过使用针对小尺寸和分散数据（例如，元数据和密钥）的PBR的混合方案来处理这个挑战，而仅对相对较大的数据（例如，值）应用擦除编码。为了缓解诸如删除编码的冗长恢复等众所周知的问题，Cocytus使用在线恢复方案，利用复制的元数据信息来连续提供KV请求。我们已经将Cocytus应用于Memcached。使用不同的KV配置对YCSB进行评估表明，Cocytus的延迟和吞吐量开销较低，可以通过快速联机恢复来承受节点故障，与PBR相比，在处理两次故障时可节省33％到46％的内存。

**ABSTRACT**

In-memory key/value store (KV-store) is a key building block for many systems like databases and large web- sites. Two key requirements for such systems are effi- ciency and availability, which demand a KV-store to con- tinuously handle millions of requests per second. A com- mon approach to availability is using replication such as primary-backup (PBR), which, however, requires M + 1 times memory to tolerate M failures. This renders scarce memory unable to handle useful user jobs.

This paper makes the first case of building highly available in-memory KV-store by integrating erasure coding to achieve memory efficiency, while not notably degrading performance. A main challenge is that an in- memory KV-store has much scattered metadata. A sin- gle KV put may cause excessive coding operations and parity updates due to numerous small updates to meta- data. Our approach, namely Cocytus, addresses this chal- lenge by using a hybrid scheme that leverages PBR for small-sized and scattered data (e.g., metadata and key), while only applying erasure coding to relatively large data (e.g., value). To mitigate well-known issues like lengthy recovery of erasure coding, Cocytus uses an on- line recovery scheme by leveraging the replicated meta- data information to continuously serving KV requests. We have applied Cocytus to Memcached. Evaluation us- ing YCSB with different KV configurations shows that Cocytus incurs low overhead for latency and throughput, can tolerate node failures with fast online recovery, yet saves 33% to 46% memory compared to PBR when tol- erating two failure.

目 录

[1 引入 1](#_Toc500427550)

[2背景及挑战 2](#_Toc500427551)

[2.1 背景 2](#_Toc500427552)

[2.2 机会和挑战 4](#_Toc500427553)

[3 设计 5](#_Toc500427554)

[3.1 接口和假设 5](#_Toc500427555)

[3.2 架构 5](#_Toc500427556)

[3.3 从数据中分离元数据 7](#_Toc500427557)

[3.4 与捎带一致的奇偶校验更新 8](#_Toc500427558)

[4 恢复 8](#_Toc500427559)

[4.1 数据恢复 8](#_Toc500427560)

[4.1.1 准备 9](#_Toc500427561)

[4.1.2 在线恢复 9](#_Toc500427562)

[4.1.3 请求处理恢复过程 11](#_Toc500427563)

[4.2 数据迁移 11](#_Toc500427564)

[5 实现 12](#_Toc500427565)

[6 评价 14](#_Toc500427566)

[6.1 实验装置 15](#_Toc500427567)

[6.2 内存消耗 16](#_Toc500427568)

[6.3 性能 16](#_Toc500427569)

[6.4 恢复效率 17](#_Toc500427570)

[6.5 差异编码 18](#_Toc500427571)

[7 相关工作 19](#_Toc500427572)

[8 结论和发展 20](#_Toc500427573)

[参考文献 21](#_Toc500427574)

1. **引入**

大规模Web应用程序日益增长的需求刺激了将大型数据集放入内存的模式，以每秒几百万次的操作满足亚毫秒级的延迟。 这种新的计算模式，即内存计算，最近才出现。 例如，像Memcached [13]和Redis [47]这样的大规模内存键/值系统已被广泛应用于Facebook [24]，Twitter [38] 和LinkedIn。将内存数据库（IMDB）应用于需要性能的场景（例如SAP HANA [12]，Oracle TimesTen [18]和Microsoft Hekaton [9]）也有相当大的兴趣。

即使许多系统有一个持久的后备存储来保存崩溃后的数据持久性，仍然需要将数据保留在内存中，以便瞬间接管失败节点的工作，因为将数TB的数据重新存储到内存中非常耗时。 例如，据报道，在Facebook上从内存数据库恢复大约120GB的数据到内存中的数据库花费了2.5-3小时[14]。传统的提供高可用性的方法是通过标准主备份（PBR）[5]和链复制[39]等备份方法，通过复制M+1次数据集来容忍M次失败。但是这也意味着不需要生成用户工作就可以献出M份CPU /内存，生成用户工作需要更多的备用机器，从而增加能源消耗。

本文描述了Cocytus，这是一种高度一致的有效且可用的内存中复制方案。 Cocytus的目标是减少复制副本的内存消耗，同时保持类似于PBR的解决方案的性能和可用性，尽管更新密集型工作负载需要额外的CPU成本。 Cocytus的关键是有效地将空间有效的时间编码方案与PBR相结合。

擦除编码是数据复制的节省空间的解决方案，并广泛应用于分布式存储系统，包括Windows Azure Store [15]和Facebook存储[23]。 然而，虽然节省空间，删除编码以其冗长的恢复和暂态数据不可用性而闻名[15,34]。

在本文中，我们研究了在内存密钥/值存储（KV-store）中应用删除编码的可行性。 我们的主要观察结果是，丰富而快速的CPU内核可以在线执行编码。 例如，单个Intel Xeon E3-1230v3 CPU内核可以对5.28GB / s的Reed-Solomon（3,5）编码数据进行编码，这比即使是40Gb / s带宽的当前高端NIC也要快。 然而，擦除编码的面向块的性质和KV-store的独特功能给Cocytus带来了几个挑战，以达到效率和可用性的目标。

第一个挑战是分散的元数据如散列表和KV存储器的内存分配信息将导致大量的编码操作和更新，即使对于单个KV库也是如此。这不仅会导致很多的CPU开销，而且还会导致高额的网络流量。Cocytus通过利用从数据中分离元数据的思想来解决这个问题[42]，并使用混合复制方案。具体来说，Cocytus使用应用程序数据的擦除编码，而对于小尺寸的元数据使用PBR。

第二个挑战是如何使用分布式数据块和奇偶校验块1一致地恢复丢失的数据块。Cocytus引入了分布式在线恢复协议，该协议始终收集所有数据块和奇偶校验块以恢复丢失的数据，而不会阻塞实时数据块和可预测内存的服务。

我们在Memcached 1.4.21中使用同步模型实现了Cocytus，服务器在收到备份节点的确认后，向客户端发送响应，以避免数据丢失。 我们还在Memcache 1.4.21中实现了一个纯粹的主备份复制来进行比较。通过使用YCSB [8]发布具有不同的密钥/值分布的请求，我们显示Cocytus在正常处理期间吞吐量和等待时间上几乎没有退化，并且可以很快地正常恢复数据。总体而言，Cocytus具有较高的内存效率，同时与PBR相比，开销较小，但读取主要工作负载的CPU成本很低，更新密集型工作负载的CPU成本较低。

总之，本文的主要贡献包括：

•第一个利用删除编码的案例，记忆KV-store。

•两个关键设计，包括混合复制方案和分布式在线恢复实现效率，可用性和一致性。

•在Memcached上实现Cocytus [13]以及确认Cocytus的彻底评估效率和可用性。

本文的其余部分安排如下。下一节介绍在现代计算环境中有关主备份复制和擦除编码的必要背景信息。第3节描述了Cocytus的设计，随后是第4节的恢复过程。第5节描述了实施细节。第6节介绍了Cocytus的实验数据。最后，第7节讨论相关工作，第8节总结本文。

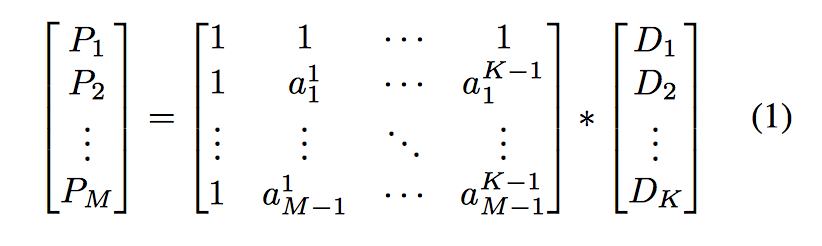
1. **背景及挑战**

本节首先简要回顾了主备份复制（PBR）和纠删编码，然后确定了将纠删编码应用于内存KV存储的机会和挑战。

* 1. 背景

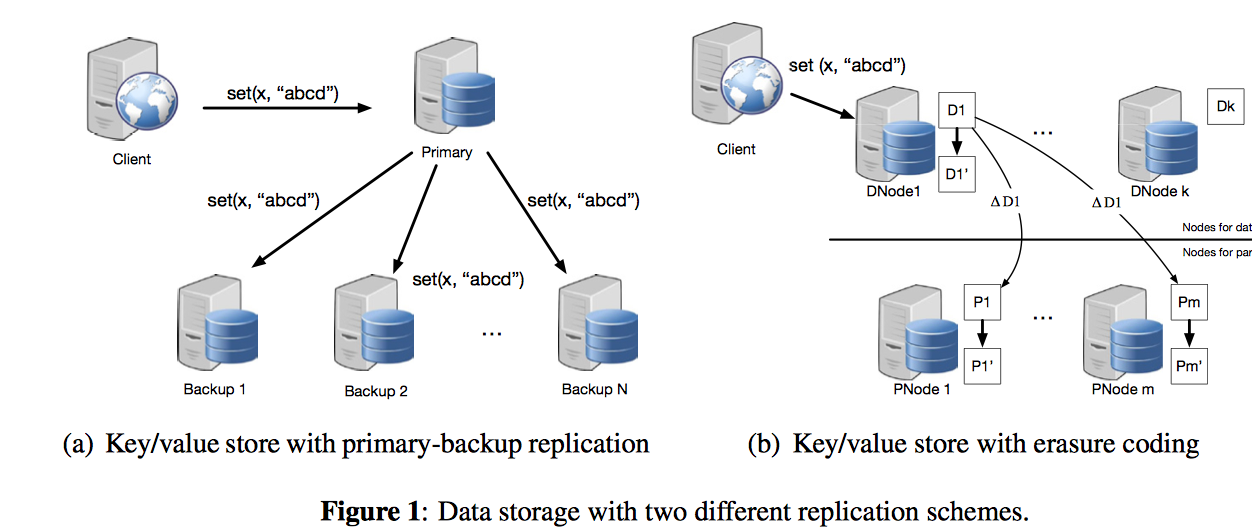
**主备份复制：**主备份复制（PBR）[3]是一种广泛使用的提供高可用性的方法。 如图1（a）所示，每个主节点都有M个备份节点来存储其数据复制以容忍M个故障。 如果主节点失败，其中一个备份节点将作为新的主节点，导致视图更改（例如，使用Paxos [19]）。因此，系统仍然可以在节点故障时提供连续的服务。然而，这是以高数据冗余为代价的，例如M个额外的存储节点和相应的CPU来容忍M个故障。例如，为了容忍两个节点的故障，KV-store的存储效率只能达到33％。

**擦除编码：**擦除编码是提供数据耐久性的一种有效方法。 如图1（b）所示，使用纠删编码，一个N节点集群可以使用K个节点作为数据，M个节点作为奇偶校验（K + M = N）。常用的编码方案是Reed-Solomon码（RS码）[30]，该码根据其在有限域中的数据通过下面的公式计算出奇偶性（矩阵称为Vandermonde矩阵）：



在上面的例子中，我们将相应的RS码方案表示为RS（K，N）。 在节点故障时，集群中的任何K个节点都可以恢复故障节点中丢失的数据或奇偶校验，因此RS（K，N）可以最多处理M个节点的失败。在恢复期间，系统通过求解由上述等式产生的等式来重新计算丢失的数据或奇偶校验。

由于只有M个节点用于存储奇偶校验，所以内存效率可以达到K / N。例如，RS（3,5）编码方案具有60％的存储效率，同时容许多达两个节点故障。

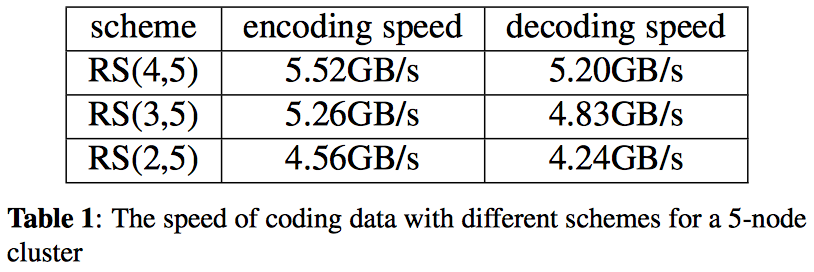


* 1. 机会和挑战

内存计算的出现显着提升了许多系统的性能。但是，这也意味着大量的数据需要放在内存中。 由于内存当前不稳定，节点故障会导致大量内存数据丢失。 即使数据在持久存储中有其备份，也需要非平凡的时间来恢复单个节点的数据[14]。

但是，简单地使用PBR可能会导致显著的内存低效。尽管数量有所增加，但内存仍然是稀缺资源，特别是在处理“大数据”应用程序时。经常有报道说，内存膨胀会显着地使性能下降或者造成服务器崩溃[4]。对于工作负载共享集群来说尤其如此，因为存储特定应用程序数据的预算并不大。

**机遇：**对可用性和内存效率的需求使得擦除编码成为一个新的有吸引力的设计点。CPU速度和CPU核心数量的增加使得擦除编码甚至适用于数据处理的关键路径。表1给出了在具有平均CPU核心（2.3 GHz Xeon E5，详见第6.1节）的5节点集群上，不同Reed-Solomon编码方案的编码和解码速度。与20年前相比，编码和解码都可以在4.24-5.52GB / s的速度下完成数百次（例如，10MB / s [31]）。 这意味着平均速度的内核足以处理通过40Gb / s的网络链路传输的数据。 这揭示了交换CPU资源以获得更好的内存效率以提供高可用性的新机会。



**挑战：**然而，将删除编码细微地应用于内存中的KV-store可能会导致显着的性能下降和一致性问题。

第一个挑战是编码只有在批量导向的情况下才能有效地完成。然而，KV-store中的更新操作可能会导致一些小的更新，这将引入显着的编码操作和网络流量。例如，在Memcached中，哈希表和分配元数据都需要修改以进行设置操作。对于第一种情况，插入桶中的KV对将改变双链表的四个点。一些类似LRU替换的统计信息也需要改变。在散列表扩展或收缩的情况下，可能需要重新定位所有的键/值对，导致大量的更新。对于分配元数据，由于Memcached使用slab分配器，分配操作通常会更改四个变量，而自由操作会更改六个到七个变量。

第二个挑战是数据更新涉及跨机器的多个奇偶校验块的更新。在数据恢复过程中，还涉及多个数据块和奇偶校验块。如果正在进行并发更新，则可能会导致数据恢复不一致。

1. **设计**
   1. 接口和假设

Cocytus是针对键/值存储（KV-store）的内存复制方案，可提供高内存效率和高可用性，且开销较低。 它假设KV-store有两个基本的操作：Value←get（Key）和set（Key，Value），其中Key和Value是任意字符串。 根据之前关于商业工作中重要/价值store的大规模分析[1,24]，Cocytus认为价值的大小通常比密钥的大小要大得多。

Cocytus只处理节点故障停止并不会污染其他节点的节点故障。佣金或拜占庭故障不予考虑。它也没有考虑到一个彻底的停电事故，使整个集群崩溃。 在这种情况下，它假定还有一个不断存储数据的存储层以保持持久性[24]。 或者，也可以使用电池备份的RAM，如NVDIMM [37,35]以保持耐用性。

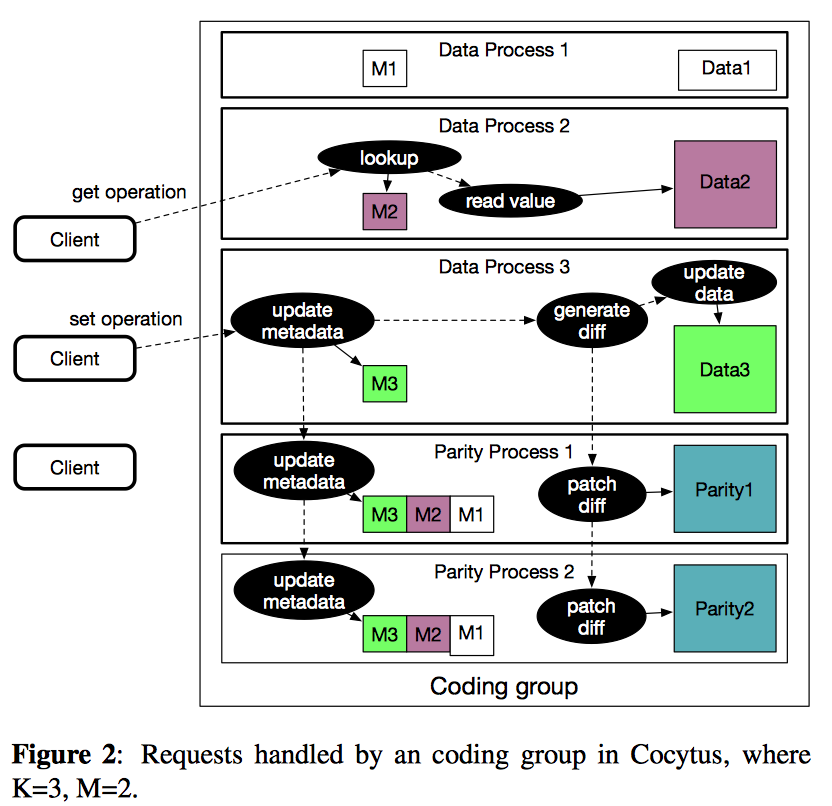
Cocytus被设计为同步的，即返回给客户机的设置请求的响应保证了数据已经被复制/编码，并且能够在节点失效的情况下幸免于难。

Cocytus可以有效地读取大部分工作负载，这对于许多商业KV-store来说是很典型的[1]。对于更新密集型工作负载，Cocytus将使用更多的CPU资源，这是由于擦除编码引起的额外计算，并且与简单的主备份复制相比，实现了类似的延迟和吞吐量。

* 1. 架构

Cocytus从元数据中分离数据并利用混合方案：元数据和密钥在使用主备份进行复制时进行擦除编码。

Cocytus的一个基本组成部分是编码组，如图2所示。每个组包括处理对数据块的请求的K个数据处理和接收来自数据处理的更新请求的M个奇偶校验处理。 get操作只涉及一个数据节点，而set操作则更新主节点和备份节点中的元数据，并生成差异补丁到部分代码。



Cocytus的一个基本组成部分是编码组，如图2所示。每个组包括处理对数据块的请求的K个数据处理和接收来自数据处理的更新请求的M个奇偶校验处理。 get操作只涉及一个数据节点，而set操作则更新主节点和备份节点中的元数据，并生成差异补丁到部分代码。

Cocytus使用分片将键/值元组分割成不同的组。 一个编码组处理一个密钥分片，在该分组中进一步划分为P个分区。每个分区由一个特定的数据进程处理，在虚拟地址空间级别执行编码。只要数据处理的地址空间不变，这就使编码操作适合于KV对的值大小的变化。数据处理之间没有数据通信，保证了数据处理之间的故障隔离。当数据进程崩溃时，一个奇偶校验进程立即处理对属于崩溃节点的分区的请求，并恢复丢失的数据，而其他数据进程不间断地提供服务。

Cocytus被设计为强大的一致性，从不丢失数据或恢复不一致的数据。 然而，Cocytus并不需要严格的平价过程。例如，两个数据进程同时更新其内存，这涉及到奇偶校验过程的两个更新。然而，奇偶校验过程可以按照任何顺序执行更新，只要他们知道更新已经被所有奇偶校验过程接收到了。因此，尽管更新顺序不变，稍后恢复的数据仍然保持一致。第4.1.2节将显示Cocytus在发生故障时如何实现一致的恢复。

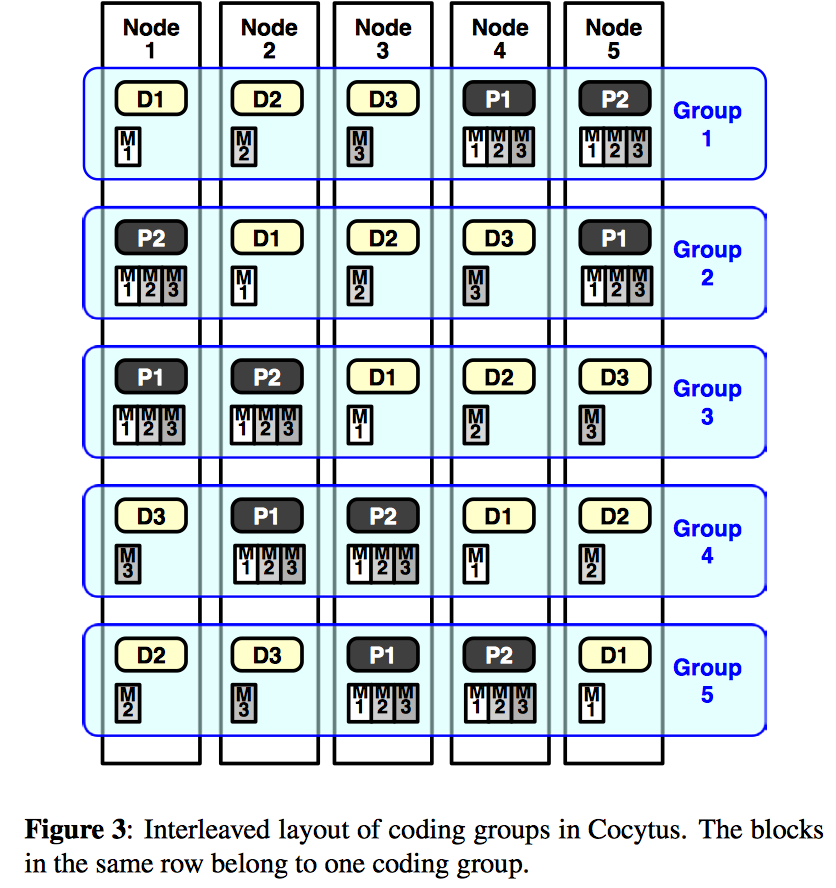
* 1. 从数据中分离元数据

对于典型的KV-store，有两种重要的元数据来处理请求。第一个是映射信息，例如将密钥映射到其值地址的（分布式）哈希表。第二个是分配信息。如前所述，如果元数据被擦除编码，则会有更多的小更新和长时间的不可用的持续时间。

Cocytus使用主备备份复制来处理映射信息。具体而言，奇偶校验进程将所有数据进程的元数据保存在相同的编码组中。对于分配信息，Cocytus应用一个基于板块的分配来进行元数据分配。它进一步依赖额外的确定性分配器来处理数据，以便每个数据处理在每次操作之后都会产生相同的存储器布局。

**交错布局：**由此设计引起的一个问题是奇偶校验进程比数据进程中的元数据保存更多的元数据，这可能会导致内存不平衡。此外，由于奇偶校验过程只需要参与设定的操作，所以它们可能变成闲置以用于读取，主要是工作负载。相反，对于读写工作负载，校验过程可能变得繁忙，并可能成为KV-store的瓶颈。

为了解决这些问题，Cocytus在集群中交错编码组，以平衡每个节点上的工作负载和内存，如图3所示.Cocytus中的每个节点都运行校验进程和数据进程; 相应地，节点将在奇偶校验过程或数据过程中忙于更新密集型或读取主要工作量。



交错布局还可以通过利用集群资源而不是一个节点来实现恢复过程。由于一个节点上的碎片属于不同的组，所以单个节点故障导致每个组上的进程故障。 但是，这些组的第一个奇偶校验节点分布在整个集群中，所有节点将一起工作来实现恢复。

为了在大规模集群中扩展Cocytus，需要考虑三个维度，包括编码组中数据处理的数量（K）和奇偶处理的数量（M），以及编码组的数量。 较大的K会增加内存效率，但是对于读写工作负载而言，奇偶校验过程会因较高的CPU压力而受到影响。较大的M导致更多的失败被容忍，但降低了存储效率并降低了设置操作的性能。扩展Cocytus的中立方式是部署更多的编码组。

* 1. 与捎带一致的奇偶校验更新

由于擦除编码组具有多个奇偶校验过程，因此将更新消息发送到这些过程需要原子广播。 否则，KV-store可能会导致不一致。例如，当一个数据进程接收到一个设置的请求并且正在向两个奇偶校验进程发送更新时，就会发生故障，并且只有一个奇偶校验进程接收到该更新消息。以下恢复可能会由于奇偶校验之间的不一致而恢复不正确的数据。

这个问题的自然解决方案是使用两阶段提交（2PC）来实现原子广播。然而，这需要两轮消息，并且将两次设置请求的I / O操作。Cocytus用捎带的方法解决了这个问题。每个请求都分配有一个xid，它在每个数据处理过程中像逻辑时钟一样单调增加。接收到奇偶校验更新后，奇偶校验进程首先将操作记录在与xid相对应的缓冲区中，然后立即向其数据进程发送确认。在数据处理接收到来自所有奇偶校验处理的确认之后，该操作在KV-store中被认为是稳定的。

1. **恢复**

当一个节点崩溃时，Cocytus需要在服务客户端请求的同时在线重建丢失的数据。Cocytus假设KV-store最终将通过分配新节点来托管恢复的数据来保持其容错级别。或者，Cocytus可以退化其容错级别，以容忍更少的故障。在本节中我们首先描述Cocytus如何将数据原位恢复到奇偶校验节点，然后说明Cocytus在崩溃的节点重新引导或添加新的备用节点时如何迁移数据以恢复奇偶校验和数据进程。

* 1. 数据恢复

因为数据块只是在处理用xid顺序执行的set请求的最后一步中更新的。 我们可以将最近完成的请求的xid视为数据块的逻辑时间戳（T）。类似地，对于奇偶校验块，存在K个逻辑时间戳（V T [1..K]），其中K是相同编码组中的数据处理的数量。 每个K逻辑时间戳是来自相应数据处理的最近完成的请求的xid。

假设数据进程1到F同时崩溃。Cocytus选择所有活动的数据块和F奇偶校验块来重建丢失的数据块。假设数据块的逻辑时间戳为TF + 1，TF + 2，...，TK，奇偶块的逻辑时间戳为VT1，VT2 , ..., VTF . If V T 1 = V T 2 = ... = V T F and V T 1 [F + 1..K] = <T F+1 , T F+2 , ..., T K>, 那么这些数据块和奇偶校验块就与公式（1）一致。因此，它们是一致的。

恢复包括两个阶段：准备和在线恢复。在准备阶段，奇偶校验进程同步它们对应于失败进程的请求缓冲区。准备阶段完成后，所有奇偶校验块在失败的进程上保持一致。在线恢复期间，活动数据进程以其逻辑时间戳发送它们的数据块，所以奇偶校验进程可以容易地提供一致的奇偶校验块。

* + 1. 准备

一旦检测到数据处理失败，则选择相应的奇偶校验处理作为恢复处理来执行恢复并代表崩溃的数据处理提供服务。恢复过程首先收集所有奇偶校验过程中对应于失败数据过程的最新xid。因此，奇偶校验过程对于每个数据过程都有最新的xid，因为它为每个数据过程维护一个单独的请求缓冲区。然后选择最小的最新的xid作为稳定的xid。由失败的数据进程收到的更大的xid请求没有被所有奇偶校验进程成功接收，因此应该被丢弃。然后，稳定的xid被发送到所有的奇偶校验进程。奇偶校验进程应用更新请求来代替xid等于或小于相应缓冲区中的稳定xid。之后，所有奇偶校验过程在失败的数据过程中都是一致的，因为它们相应的逻辑时间戳都与稳定的xid相同。

准备阶段在很短的​​时间内阻止关键/值请求。根据我们的评估，即使在高工作负载下，阻塞时间也只有7ms到13ms。

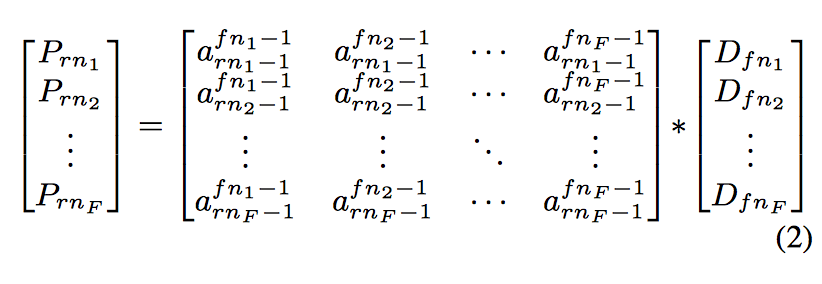
* + 1. 在线恢复

元数据和数据的分离使得能够联机恢复键/值对。在恢复过程中，恢复过程可以利用复制的元数据在线重建丢失的数据以服务客户端请求，同时使用空闲的CPU周期主动重新构建其他数据。

在线恢复期间，数据块以4KB的粒度被恢复，这被称为恢复单元。 根据地址，每个恢复单元都被分配了一个ID，以方便处理之间的通信。

如图4所示，我们的在线恢复协议有五个步骤：

1. 为了重建恢复单元，恢复过程成为恢复发起者并且将包括恢复单元ID和涉及的恢复过程的列表的消息发送到活动数据过程。
2. 当第i个数据进程接收到消息时，它将相应的数据单元连同其逻辑时间戳Ti发送到所有的恢复进程。
3. 3（a）当恢复进程接收到数据单元和逻辑时间戳T i时，首先将xid等于或小于T i的请求应用于相应的缓冲区。此时，此恢复过程的第i个逻辑时间戳等于Ti。
4. 3（b）恢复过程用接收的数据单元用预定的系数减去相应的奇偶校验单元。 减法完成后，奇偶校验单元不再与第i个数据进程相关联。 它停止被第i个数据进程更新。 因此，这个恢复过程中的其余奇偶单元仍然与第i个数据进程相关联。
5. 当恢复进程接收并处理来自活动数据进程的所有数据单元时，它将最终相应的奇偶校验单元发送到仅与失败的数据进程相关联的恢复启动器。
6. 当恢复启动器已经从恢复过程接收到所有的奇偶校验单元时，通过求解以下等式来解码它们，其中fn 1，fn 2，...，fn F表示F故障数据过程的数量，并且rn 1， 2，...，Fn表示被选择为恢复过程的F个奇偶校验过程的数量。



**正确性论证：**这里我们简单地论证协议的正确性。因为当数据块更新时，所有的奇偶校验过程都应该收到相应的更新请求。因此，在步骤3（a）中，奇偶校验过程必须已经接收到所有需要的更新请求，并且可以将其相应的逻辑时间戳与接收到的逻辑时间戳同步。由于接收的数据块和奇偶校验块具有相同的逻辑时间戳，所以接收到的数据块应当与用于构建奇偶校验块的数据块相同。由于奇偶校验块是Vander-monde矩阵中各个预定义系数的数据块之和，所以在步骤3（b）中的减法之后，奇偶校验块仅由其余数据块构成。在步骤4的开始，奇偶校验块仅由失败的数据处理的数据块构成，因为对于每个活动数据处理，奇偶校验处理已经完成步骤3。最后，在准备阶段，在稳定的xid同步的帮助下，步骤5中接收到的奇偶校验块都是一致的，应该与方程2一致。

* + 1. 请求处理恢复过程

Cocytus允许恢复过程在恢复过程中处理请求。 对于获取请求，它会尝试通过备份散列表查找键/值对。 如果发现该对，恢复过程将检查该值所需的数据块是否已被恢复。 如果数据块尚未恢复，则恢复过程将启动每个数据块的数据块恢复。 数据块恢复后，恢复过程将响应以请求的值发送给客户端。

对于设置的请求，恢复过程借助备份中的分配元数据为新值分配新空间。 如果分配的数据块没有恢复，恢复过程将为它们调用恢复功能。 恢复后，恢复过程像正常的数据处理一样处理操作。

* 1. 数据迁移

**数据处理恢复：**在数据处理恢复期间，Cocytus可以将数据从恢复过程迁移到新的数据处理。恢复过程首先将密钥以及哈希表中的值（即，大小和地址）的元数据发送到新的数据处理。在接收键/值对的同时，新的数据进程重建散列表和分配元数据。所有键/值对发送到新的数据进程后，恢复过程停止向客户端提供服务。

当元数据迁移完成时，数据（即值）迁移开始。那时，数据进程可以像恢复进程一样处理请求。 它们之间的唯一区别是数据处理本身不能恢复数据块。当数据进程需要恢复数据块时，它会向恢复进程发送一个请求。 如果恢复过程已经恢复了数据块，它将恢复的数据块直接发送到数据过程。否则，它开始恢复过程。所有数据块迁移到数据进程后，迁移完成。

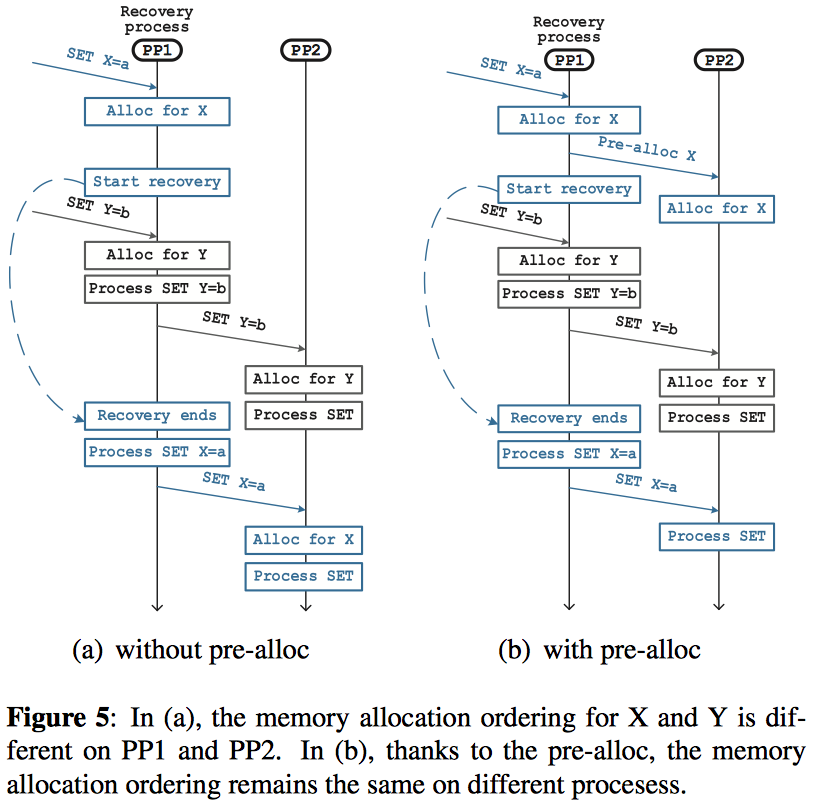
如果在数据迁移期间新数据进程或相应的恢复进程失败，则两者都应该被终止。 这是因为只有其中一个会导致提供连续服务的信息不足。 Cocytus可以将此失败视为数据处理失败。

**奇偶校验过程恢复：**奇偶校验过程恢复很简单。在奇偶校验过程崩溃之后，数据过程用奇偶校验过程标记所有数据块。数据处理首先将元数据发送到恢复奇偶校验处理。元数据传输完成后，新奇偶校验过程的逻辑时间戳与其接收到的元数据相同。在元数据传输之后，数据进程迁移可能与奇偶校验更新请求重叠的数据。在发送涉及标记有未命中位的数据块的奇偶校验更新请求之前，数据处理需要将涉及的数据块发送到新的奇偶校验处理。这样，发送到新的奇偶校验过程的数据块与之前发送的元数据具有相同的逻辑时间戳。新的奇偶校验过程接收到所有数据块后，恢复完成。如果在恢复奇偶校验过程中任何一个数据进程失败，则恢复失败，Cocytus开始恢复失败的数据进程。

1. **实现**

我们首先从零开始建立了一个Cocytus的KV-store。为了理解它在真正的KV存储上的性能意义，我们还在Mem-cached 1.4.21的顶端使用同步模型实现了Cocytus，向Memcached添加了约3700个SLoC。目前，Cocytus仅适用于单线程模式，并没有完全支持数据迁移。为了利用多核，Coc-tus可以部署分片和多进程而不是多线程。事实上，对于可能遭受不必要的资源争用和中断数据隔离的数据处理，使用多线程并没有显着的改进。校验过程可以以多线程的方式实现，以便在写入密集型工作负载下分配高CPU压力，这是我们未来的工作。在RS码中，我们使用Jerasure [27]和GF-complete [26]进行伽罗瓦域操作。请注意，Cocytus在很大程度上与编码方案正交;这将是我们未来的工作，应用其他网络或空间有效的编码方案[33,28]。本节介绍一些实现问题。

**确定性分配器：**在Cocytus中，分​​配元数据与数据分离。每个数据进程都使用mmap syscall维护数据的内存区域。每个奇偶校验过程还保持一个等价的存储器区域的奇偶性为了管理数据区域，Cocytus使用两个AVL树，其中一个记录空闲空间，另一个记录分配的空间。树节点由内存块的起始地址及其长度组成。长度保证是16的倍数，并被用作树的索引。每个内存位置都存储在任一树中。一个分配操作将在自由树中找到一个合适的内存块，并将其移动到分配树中，而自由操作将在oppo-site中执行。树以类似于伙伴内存分配的方式管理内存块：在分配操作期间，大块可能被分割成小块，并且在空闲操作期间连续的块被合并成更大的块。为了快速拆分和合并，所有的内存块根据广告连接一个列表。请注意，只有元数据存储在树中，与分配器管理的实际内存分开存储。



**预先分配：**Cocytus使用确定性分配器和散列表来确保每个节点中的所有元数据是一致的。 因此，Cocytus只需要保证每个进程将以相同的顺序处理相关的请求。捎带的两阶段提交（第3.4节）大多可以提供这样的保证。

图5（a）显示了一个例外。当恢复进程收到X = a的设置请求时，需要为该值分配内存。如果需要恢复该值的内存，恢复过程首先启动X的恢复，并将此设置请求置于等待队列中。在Cocytus中，恢复是异步的。因此，恢复过程能够在恢复完成之前处理其他请求。在这段时间内，Y = b的另一个设置请求进入恢复过程。恢复进程为它分配内存，幸运的是分配的内存已经被恢复。因此，恢复过程直接处理Y = b的设置请求而没有任何恢复，并将请求发送到其他奇偶校验进程以进行容错。只要他们收到请求，其他进程（例如图中的PP2）就会为Y分配内存并像往常一样完成其工作。最后，当X的恢复完成时，恢复过程继续处理X = a的设置请求。它也发送容错请求到其他的奇偶校验进程，在这个进程上为X分配内存。到目前为止，恢复进程已经为X和Y分配了内存。但是，在其他奇偶校验过程中，Y的内存分配发生在X之前。恢复过程和奇偶校验过程之间的这种不同的分配顺序将导致不一致。

Cocytus通过在每个设置操作由于恢复而排队之前发送预分配请求（如图5（b）所示）来解决这个问题。 通过这种方式，奇偶校验过程可以为排队请求预先分配空间，并保证内存分配的顺序。

**恢复领导者：**因为当多个恢复进程想要同时恢复两个等效块时，他们都希望启动在线恢复协议，这是不必要的。为了避免这种情况，Cocytus在每个组中分配了一个恢复组长。复苏领导者是一个平等的过程，负责启动和完成集团的复苏。如果需要恢复数据，组中的所有其他校验进程将向恢复负责人发送恢复请求，恢复负责人将在恢复完成后广播结果。恢复领导者并不是绝对必要的，但这种恢复的集中管理可以防止同一个数据被多次恢复，从而减少网络流量。考虑到系统的交错布局，恢复领导者均匀分布在不同的节点上，不会成为瓶颈。

**连续性故障的快速恢复：**当存在多个数据处理故障并且某些故障进程的数据已经被恢复进程恢复时，如果我们不考虑恢复进程，进一步恢复的数据可能是错误的。

在图4给出的例子中，假设DP1（数据处理1）首先失败，并且PP1（奇偶处理1）成为恢复处理。在PP1恢复了一部分数据块后，DP2失败，PP2成为DP2的恢复过程。那时候，PP1上的一些数据块已经被恢复了，其他的没有。为了恢复DP2上的数据块，如果DP1上对应的数据块已经被恢复，则应该以涉及3个数据块和1个奇偶校验块的方式进行恢复，否则应该以涉及2个数据块的方式恢复 2个奇偶校验块。这两种恢复的程序是完全不同的。

**主备份复制：**为了评估Cocytus，我们还实现了基于Memcached-1.4.21的主备份（PBR）复制版本，其设计与Cocytus几乎相同，如同步写入，捎带，但Cocytus将数据在编码空间中，并且需要在发生故障之后解码数据。我们没有直接使用Repcached [17]有两个原因。一个是Repcached只支持一个从属工作者。另一个是Repcached中的set操作是异步的，因此不能保证崩溃的一致性。

1. **评价**

* 我们通过将它与主备份复制（PBR）和香草Memcached进行比较来评估Cocytus的性能。 我们评估结果的亮点如下：
* Cocytus实现高内存效率：当容忍两个节点故障时，从1KB到16KB的值大小可将内存消耗减少33％到46％。
* Cocytus的开销很低：与香草KV-store相比，它与PBR和vanilla KV-store（即Memcached）具有相似的吞吐量，并且延迟较小。
* Cocytus可以按设计容忍失败，并且可以快速，优雅地重新覆盖：即使在两次节点崩溃的情况下，Cocytus也可以优雅地恢复丢失的数据并处理客户端请求，并且与PBR密切相关。
  1. 实验装置

**硬件和配置：**由于我们的硬件限制，我们在一个6节点的机器集群上进行所有的实验。每台机器有两个10核2.3GHz的In-tel Xeon E5-2650,64GB的RAM，并连接10Gb网络。我们使用6个节点中的5个作为服务器运行，剩下的一个作为客户端进程。

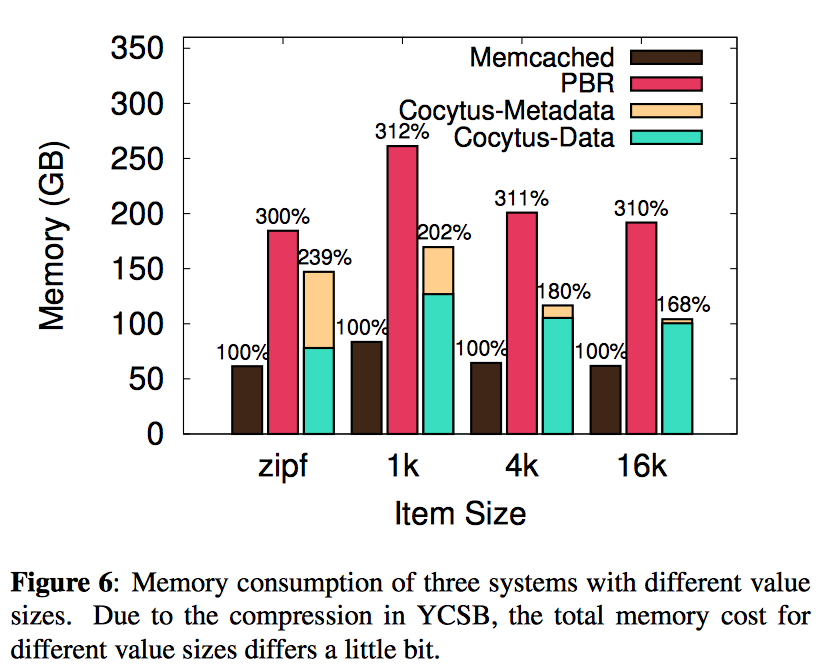
为了获得更好的记忆效率，Cocytus可以在编码组中使用更多的数据处理。 但是，在一个组中部署太多的数据处理会增加平价处理的负担，这可能是系统的一个瓶颈。 由于我们集群的局限性，我们将Cocytus配置为配置为RS（3,5）的5个交织EC组，以便系统能够容忍两个故障，同时最大化数据处理。 每个组由三个数据进程和两个奇偶校验进程组成。通过这种部署，每个节点包含三个数据进程和两个不同组的奇偶校验进程。

**比较目标：**我们比较Cocytus与PBR和香草Memcached。为了评估PBR，我们在五个节点中分配了15个数据过程。对于每个数据流程，我们启动2个备份过程，以便系统也可以容忍两个节点故障。与Cocytus（25个进程）相比，此部署启动更多的进程（45个进程），在某些情况下可能会占用更多的CPU资源。 我们通过在五个节点之间平均分配15个实例来部署Mem缓存。这样Mem缓存的进程数与Cocytus的进程数相同。

**工作量：**我们使用YCSB [8]基准来生成我们的工作量。我们通过连接一个表名和一个标识符来生成每个键，一个值是一个压缩的HashMap对象，它由多个字段组成。关键概率的分布是Zip fi [10]，其中一些密钥是热的，一些密钥是冷的。钥匙的长度通常小于16B。我们还评估了不同读写比例的系统，包括等份（50％：50％），大部分读取（95％：5％）和只读（100％：0％）。

由于来自Facebook [24]的值大小的中位数为4.34KB，而Clus-ter为10.7KB，我们测试这些具有相似值大小的高速缓存系统。和YCSB一样，一个值由多个字段组成，为了评估我们的系统具有不同的值大小，我们保持字段号为10，同时改变字段大小使得总值大小为1KB / 4KB / 16KB，即字段大小相应地是0.1KB / 0.4KB / 1.6KB。要将总数据大小限制为64GB，1/4/16 KB的项目编号分别为64/16/1百万。然而，由于对象压缩，我们无法预测KV-store接收到的实际值大小，并且这些值可能不一致; Cocytus将压缩的值对齐到16个字节以执行编码。

* 1. 内存消耗



如图6所示，由于使用了擦除编码，Cocytus与PBR相比节省了显着的内存。与PBR相比，Cocytus的值为16KB，可以节省46％的内存。使用RS（3,5），Cocytus的预期内存开销应为1.66X，而实际的内存开销范围为1.7X至2X。这是因为复制元数据和密钥引入了更多的内存成本，例如，对于1KB，4KB和16KB的值大小，所有消耗的内存的25％，9.5％和4％。我们认为，这样的成本是值得的快速和在线恢复的好处。

为了调查小和可变大小的值的影响，我们进行一个测试，其中值的大小遵循Zip范围从10B到1KB的分布。由于难以预测总的内存消耗，我们只需插入1亿个这样的项目。结果如图6中的zipf所示。正如预期的那样，更多的项目会带来更多的元数据（包括键），这会减少Cocytus的好处。即便如此，与PBR相比，Cocytus仍然可以节省20％的内存。

* 1. 性能

如图7所示，Coctus对于只读和只读工作负载所带来的性能开销很小，与Vanilla Memcached相比，写入密集型工作负载的开销很小。Cocytus与PBR具有相似的延迟和吞吐量。以下使用一些数据来解释数据。

**Cocytus和PBR的小额开销：**由于三种配置处理类似操作的请求，在这种情况下，性能也是类似的。但是，在处理设置请求时，Cocytus和PBR会引入更多的操作和网络流量，因此，吞吐量会有较小的延迟。我们发现，尽管PBR和Cocytus在CPU使用率（表2）和网络流量（Memcached：540Mb / s，PBR：2.35Gb / s，Cocytus：2.3Gb / 更多的CPU操作和网络流量，两者都不是瓶颈。因此，来自客户端的多个请求可以被重叠和流水线化。因此，与Memcached的香草类似。因此，Cocytus和PBR都可以交换一些CPU和网络资源以实现高可用性，同时产生小的用户感知的性能开销。

**PBR和Cocytus的写入延迟较高：**当读写率为95％时，延迟较高：5％，这是一个很奇怪的现象。 原因是set操作被get操作占先。 在Co-cytus和PBR中，set操作是FIFO，而set操作和get操作是交错的。特别是在主要读取工作量的情况下，集合操作往往被预先占用，因为集合操作在PBR和Cocytus中具有更长的路径。

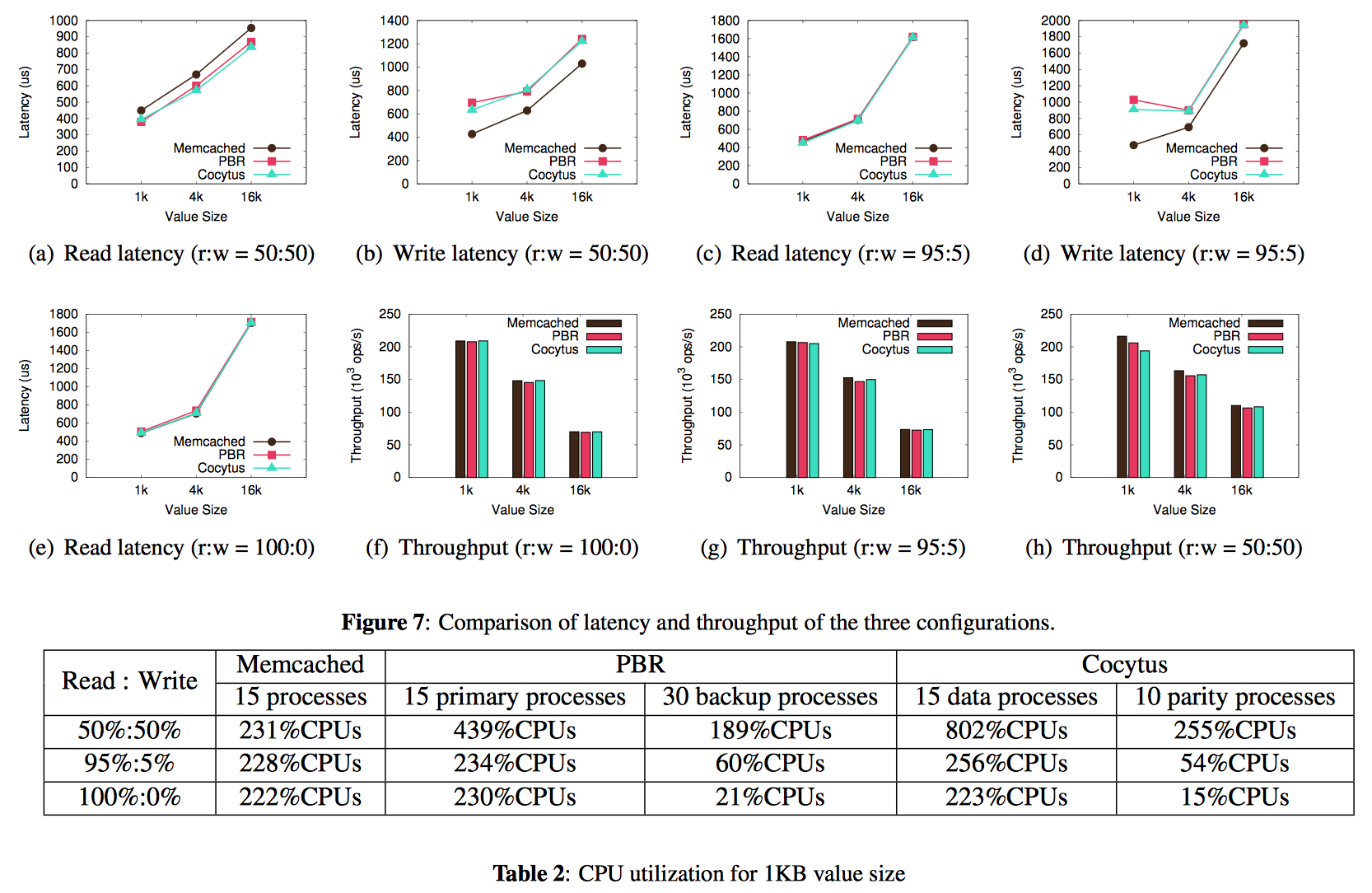
**PBR和Cocytus的读取延迟较低：**有一个有趣的现象是，在更新密集的情况下（即r：w = 50:50），较高的写入延迟导致较低的PBR和Cocytus读取延迟。 这可能是因为写入延迟较高时，更多的客户端线程一次等待设置的操作。但是，等待设置操作不会阻止来自其他客户端线程的获取操作。因此，等待get操作的客户端线程可以做得更快，因为会有更少的客户端线程阻塞这个操作。 结果get的延迟较低。

* 1. 恢复效率

我们使用1KB值的大小来评估恢复效率，以读取为主，读取为主和读写的工作负载。我们通过手动杀死节点上的所有进程来模拟两个节点故障。基准测试开始后的第一个节点发生故障。而另一个节点故障发生在100秒，在第一次故障恢复之前。图8的每个子结构中的两个吞吐量崩溃是由TCP连接机制引起的，并且可以巧合使用来指示节点失败的时间。垂直线表示所有数据已被恢复的时间。

我们的评估显示，在第一个节点发生故障后，Cocytus可以在没有客户请求的情况下以550MB / s的速度修复数据。如果我们使用更多的流程，速度可能会更快。但是，为了实现高可用性，Coc tus首先对请求的单元进行恢复，并在系统空闲时恢复冷数据。

如图8（a）所示，当工作负载是只读时，Cocytus与PBR的执行方式类似，这可以确保数据恢复可以与读取请求并行执行，而不会产生明显的开销。 50％，90％，99％的潜伏期在Cocytus恢复期间为408us，753us和1117us。如图8（b）所示，当读写率为95％时，可以达到类似的性能。在频繁设置请求的情况下，如图8（c）所示，恢复影响正常请求处理的吞吐量。原因是为了处理集合操作，Cocytus需要分配新块，这通常会触发这些块的数据恢复。等待这样的数据恢复完成会降低性能。事实上，在第一个节点崩溃之后，性能仍然是可以接受的，因为恢复比较简单，并不是所有进程都参与恢复。但是，当两个节点同时发生故障时，性能会受到更显着的影响。幸运的是，这是一个非常罕见的情况，即使发生，Cocytus仍然可以提供性能合理的服务，快速完成数据恢复。



为了证明我们的在线恢复协议的好处，我们还实施了Cocytus的阻止版本进行比较。在Cocytus的阻塞版本中，如果有任何恢复正在进行并且获取操作不受影响，则设置的操作会延迟。从图8可以看出，即使只有一个节点故障和5％的设置操作，阻塞版本的吞吐量也会崩溃。

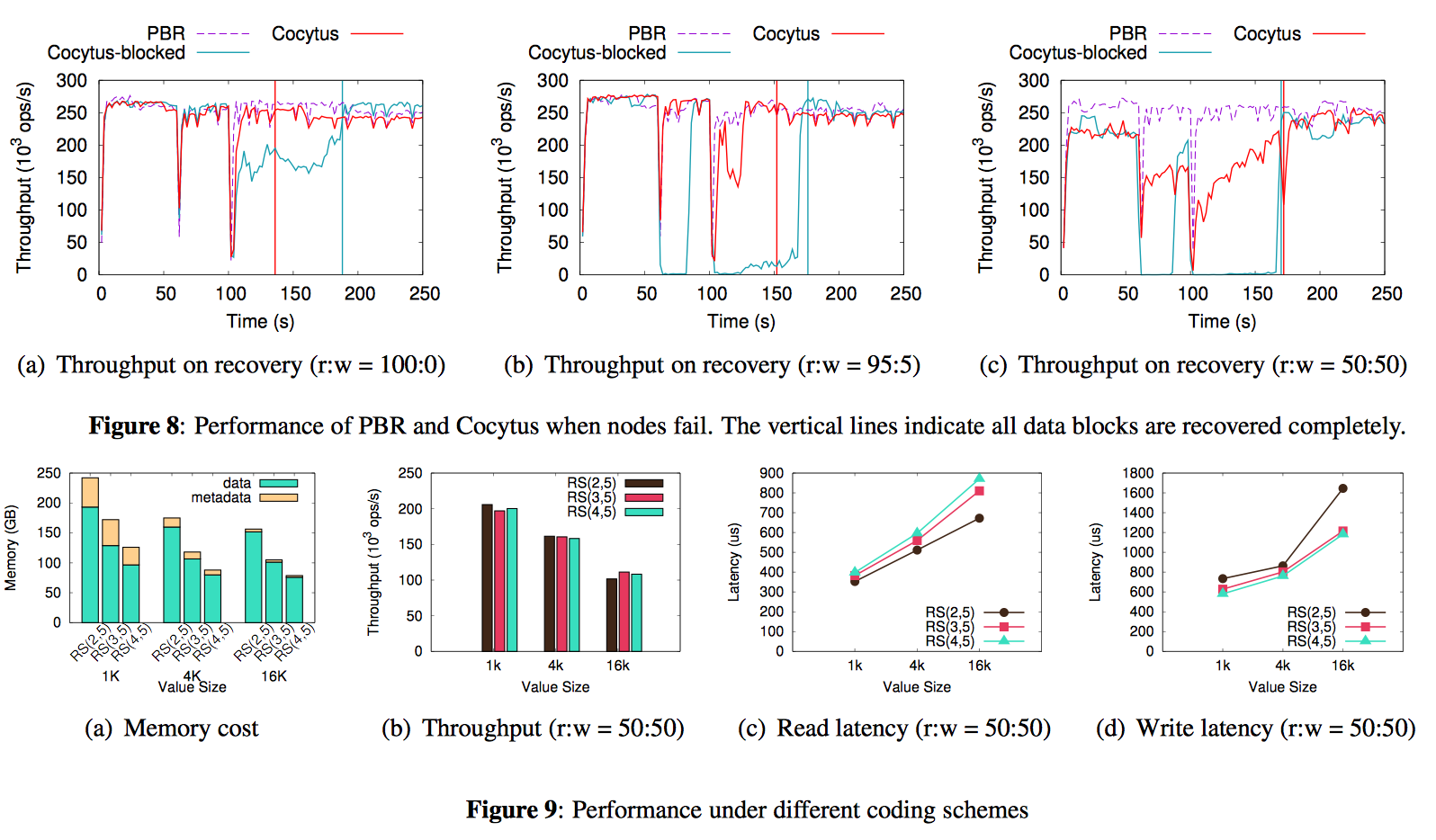
* 1. 差异编码

为了解不同编码方案下的效果，我们用RS（4,5），RS（3,5）和RS（2,5）评估Cocytus。如图9所示，RS（2,5）的内存消耗最大，RS（4,5）中的最小。所有这三种编码方案都能从更大的值中获益。他们的吞吐量相似，因为服务器上没有瓶颈。然而，RS（2,5）的写入延迟比较长，因为它将更多的消息发送到奇偶校验进程。RS（2,5）具有较低读取延迟的原因应该是较长的写入延迟导致较低的读取延迟（类似于前面所述的情况）。

1. **相关工作**

**工作分离：**元数据/关键和价值的分离受到之前分离工作的努力的启发。例如，Wang等人 从元数据中分离数据以实现有效的Paxos式存储异步复制。尹等人当容忍拜占庭故障时，执行与协议分离以减少执行节点。Clement等人 [6]区分遗漏和拜占庭故障，并利用它们之间的冗余来减少所需的副本。相比之下，Cocytus将元数据/关键字从值中分离出来，以实现空间有效和高度可用的关键/价值存储。

**擦除编码：**学术界和工业界的擦除编码已被广泛应用于存储系统，以实现耐久性和空间利用率[15,34,29,32,23]。一般来说，它们提供了许多优化编码效率和恢复带宽的优化，如局部重构代码[15]，Xor-bas [32]，捎带代码[29]和懒惰恢复[34]。PanFS[44]是一个并行文件系统，它使用全局擦除编码来保护大于64KB的文件，但是复制元数据和小文件使元数据更新的成本最小化。



**复制：**复制是一种容错的标准方法，它可以分为同步[5,3,39]和异步[19,2]。 Mo-jim [48]结合了NVRAM和双层主备份复制方案来优化数据库复制。目前，Cocytus利用标准的主备份复制来提供元数据的可用性，以及在遗漏失败的情况下提供密钥。 我们未来的工作是应用其他复制方案或处理佣金失败。

RAMCloud [25]利用集群的规模来实现快速的数据恢复。Imitator [41]利用分区图中已有的顶点来提供容错图计算，它还利用多个副本来恢复一个节点中的失败数据。但是，它们不提供在线恢复，使得恢复的数据不能同时访问。相比之下，由于复制元数据和密钥，Co-cytus不需要快速恢复集群的规模，而是提供始终在线的数据访问。

**关键/价值存储：**在优化关键/价值存储方面有相当多的兴趣，利用像RDMA这样的高级硬件[22,36,16,43]或增加并发性[11,20,21]。Cocytus在很大程度上与这些改进是正交的，我们相信Cocytus可以类似地应用于这样的关键/价值存储以提供高可用性。

1. **结论和发展**

内存中的键/值存储的效率和可用性是两个关键的要求。我们已经展示了这样的设计，通过构建Cocytus并将其集成到Memcached中，可以同时实现效率和可用性。Cocytus使用混合复制方案，通过对元数据和密钥使用PBR，而对大尺寸的值使用擦除编码。Cocytus能够实现与PBR类似的正常性能，恢复期间性能影响不大，同时实现更高的内存效率。

我们计划以几种方式扩展我们的工作。首先，我们计划探索更大的集群设置，并研究其他优化编码方案对Cocytus性能的影响。其次，我们计划研究如何使用NVRAM将Cocytus应用于其他内存store[40,7,45]。最后，我们计划研究如何将Cocytus应用于内存数据库的复制。

参考文献

[1]  B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key- value store. In SIGMETRICS, pages 53–64. ACM, 2012.

[2]  W.J.Bolosky,D.Bradshaw,R.B.Haagens,N.P.Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In NSDI, 2011.

[3]  T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[4]  Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ACM SIGPLAN In- ternational Symposium on Memory Management*, pages 119–130. ACM, 2013.

[5]  N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[6]  A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Pro- ceedings of the ACM SIGOPS 22nd symposium on Oper- ating systems principles*, pages 277–290. ACM, 2009.

[7]  J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non- volatile memories. In *ACM Conference on Architectural Support for Programming Languages and Operating Sys- tems*, pages 105–118. ACM, 2011.

[8]  B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[9]  C.Diaconu,C.Freedman,E.Ismert,P.-A.Larson,P.Mit- tal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server’s memory-optimized oltp engine. In *Proceed- ings of the 2013 international conference on Management of data*, pages 1243–1254. ACM, 2013.

[10]  L. Egghe. Zipfian and lotkaian continuous concentration theory. *Journal of the American Society for Information Science and Technology*, 56(9):935–945, 2005.

[11]  B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, volume 13, pages 385– 398, 2013.

[12]  F. Fa ̈ rber, N. May, W. Lehner, P. Große, I. Mu ̈ ller, H. Rauhe, and J. Dees. The sap hana database–an ar- chitecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[13]  B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[14]  A. Goel, B. Chopra, C. Gerea, D. Ma ́ta ́ni, J. Metzler, F. Ul Haq, and J. Wiener. Fast database restarts at face- book. In *Proceedings of the 2014 ACM SIGMOD inter- national conference on Management of data*, pages 541– 549. ACM, 2014.

[15] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX Annual Technical Conference*, pages 15–26, 2012.

[16] A.Kalia,M.Kaminsky,andD.G.Andersen.Usingrdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 295–306. ACM, 2014.

[17] KLab Inc. http://repcached.lab.klab.org, 2011.

[18] T. Lahiri, M.-A. Neimat, and S. Folkman. timesten: An in-memory database for enterprise applica- tions. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[19] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[20] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freed- man. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[21] R. Liu, H. Zhang, and H. Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Pro- ceedings of the 2014 USENIX Annual Technical Confer- ence, USENIX ATC*, volume 14, pages 219–230, 2014.

[22] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[23] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Face- books warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Associa- tion, 2014.

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

[25] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Op- erating Systems Principles*, pages 29–41. ACM, 2011.

[26] J. S. Plank, E. L. Miller, and W. B. Houston. GF- Complete: A comprehensive open source library for Ga- lois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[27] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[28] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 81– 94. USENIX Association, 2015.

[29]  K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network chal- lenges of data recovery in erasure-coded distributed stor- age systems: A study on the facebook warehouse cluster. *Proc. USENIX HotStorage*, 2013.

[30]  I. S. Reed and G. Solomon. Polynomial codes over cer- tain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[31]  L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM computer communication review*, 27(2):24–36, 1997.

[32]  M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceed- ings of the VLDB Endowment*, pages 325–336. VLDB En- dowment, 2013.

[33]  N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchan- dran. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage- bandwidth tradeoff. *Information Theory, IEEE Transac- tions on*, 58(3):1837–1852, 2012.

[34]  M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair band- width costs in erasure-coded distributed storage. In *Pro- ceedings of International Conference on Systems and Storage*, pages 1–7. ACM, 2014.

[35]  SNIA. Nvdimm special interest group. http://www.snia.org/forums/sssi/NVDIMM, 2015.

[36]  P. Stuedi, A. Trivedi, and B. Metzler. Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached. In *USENIX Annual Technical Confer- ence*, pages 347–353, 2012.

[37]  V. Technology. Arxcis-nv (tm): Non-volatile dimm.  http://www.vikingtechnology.com/ arxcis-nv, 2014.

[38]  Twitter Inc. Twemcache is the twitter memcached. https://github.com/twitter/twemcache, 2012.

[39]  R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[40] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Camp- bell, et al. Consistent and durable data structures for non- volatile byte-addressable memory. In *FAST*, pages 61–75, 2011.

[41] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 562–573. IEEE, 2014.

[42] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage repli- cation. In *USENIX Annual Technical Conference*, pages 413–424, 2012.

[43] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Sys- tems Principles*, pages 87–104. ACM, 2015.

[44] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[45] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 167– 181. USENIX Association, 2015.

[46] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253– 267. ACM, 2003.

[47] J.Zawodny.Redis:Lightweightkey/valuestorethatgoes the extra mile. *Linux Magazine*, 79, 2009.

[48] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mo- jim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18. ACM, 2015.