# 带有RDMA和缓存的高效分布式内存管理

## ABSTRACT

为此，我们提出GAM，这是一种高效的分布式内存平台，可通过远程直接内存访问（RDMA）提供基于目录的缓存一致性协议。 GAM管理分布在多个节点之间的空闲内存，以提供统一的内存模型，并支持一组用于内存操作的用户友好的API。

为了从关键执行路径中删除写操作，**GAM允许使用紧跟的读操作和写操作对当前的写操作进行重新排序，从而强制执行部分存储顺序（PSO）的内存一致性**。

设计了一个**轻量级日志记录方案**旨在提供GAM中的容错功能。我们进一步在GAM之上构建了一个事务引擎和一个分布式哈希表（DHT），以显示所提供API的易用性和适用性。

PS：什么是部分存储顺序的内存一致性？

## INTRODUCTION

## 系统设计

在本节中，我们介绍GAM的系统架构。 GAM基于**分区的全局寻址模型**，提供了一组用于内存操作的API，并维护了分布式节点之间的**缓存一致性**，以利用应用程序中的数据局部性。

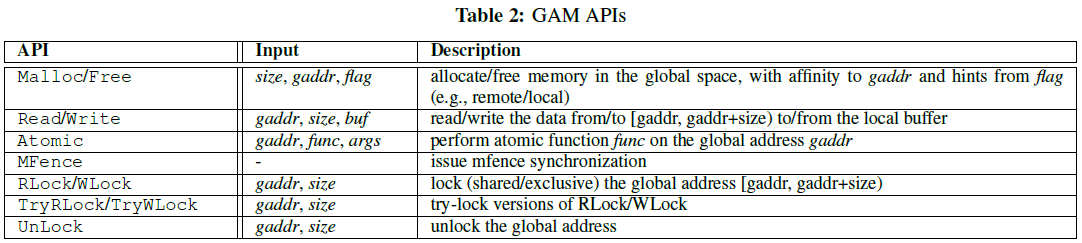
### 2.1 寻址模型和API

GAM采用**分区的全局地址空间（PGAS）寻址模型**[10，48]，该模型提供了逻辑上统一的地址空间，因通过将分布式应用程序实现为**多线程程序**，从而简化了分布式应用程序的开发。在硬件级别上，PGAS由通过RDMA网络互连的许多机器的存储器实现，从而每个节点负责全局地址空间的一个**分区**。

GAM提供了表2中列出的一组API，用于处理全局内存空间。这些API可以分为两类：

1. 全局内存访问，它包含前四个API：Malloc / Free和Read / Write；
2. 同步，其余的API都属于这类。

我们将仅在本节中讨论第一类的API，并将对同步API的讨论推迟到第4节。



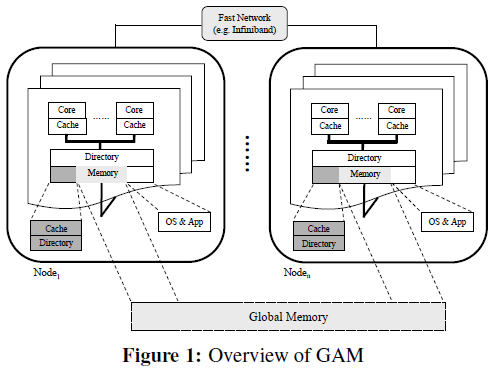
Malloc / Free与标准库中的malloc / free相似，不同的是它操纵全局内存而不是本地内存。**如果用户将基本全局地址gaddr提供给Malloc作为亲缘关系（affinity）的提示，则分配的内存在gaddr所在的节点上**。或者，用户也可以提供一些有关分配的提示（例如local/remote，作为flag的参数），这些提示将指导GAM进行相应的分配。**默认情况下，我们首先尝试在本地分配**；但是，如果本地节点面临内存压力，则我们会根据定期同步的全局内存分配统计信息，将分配请求转发到拥有最大空闲内存的节点。正如我们所看到的，GAM API是不言自明的，可直接用于在GAM之上构建高级应用程序。

### 2.2 缓存一致性协议

尽管RDMA网络的吞吐量和延迟有了显着提高，几乎接近QPI互连的吞吐量和延迟，**但是本地内存访问和远程内存访问之间仍然存在约10的差距**。考虑到大多数应用程序的局部性[13]，减轻延迟差距的最佳实践是依靠分层存储体系结构，即**多层高速缓存/存储器**，以减少到较低存储层的行程。在这方面，**我们在全局内存之上添加了一个额外级别的DRAM驻留缓存，以吸收远程内存访问。**

我们决定不使用基于探听（snoop-based）的缓存一致性协议，因为RDMA网络中的广播不可靠，并且建立可靠的广播非常昂贵，并且很容易使网络不堪重负。因此，我们依靠**基于目录的协议**，通过维护元数据来跟踪集群中的数据分布。这样，我们可以完全了解每个节点中缓存的数据，并通过点对点通信保持缓存一致。

如图1所示，全局内存之上的额外缓存级别导致GAM中的缓存一致性协议分为三个级别，即**NUMA节点内的基于探听的协议**，**NUMA节点之间的基于目录的协议**以及我们设计的分布式基于目录的协议。由于缓存一致性的最高两个级别已由硬件强制执行，因此我们仅关注最低级别的高速缓存一致性协议，该协议可实现对全局内存的一致视图。



对于每个数据片，有五种类型的节点：本地home/远程remote节点、请求节点，共享sharing/所有者owner节点，具体取决于数据位置和访问权限。本。请求节点是请求对数据的共享/独占（即对应读/写）访问的节点。共享/所有者节点是具有数据共享/独占**权限**的节点。为了便于表达，我们还说拥有者节点拥有数据。

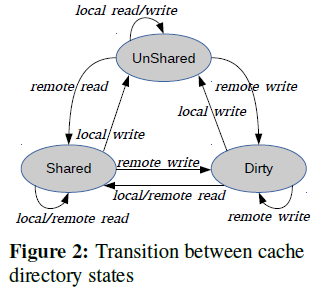
最初，本地节点拥有数据，因此既是共享节点又是所有者节点。在接收到对数据的访问请求后，本地节点将相应的权限授予请求节点，然后**将其提升为共享节点或所有者节点。**每个数据可以同时由多个共享节点共享，但一次最多具有**一个所有者节点**。

另外，**除非所有者节点是唯一的共享节点，否则一个节点不能既是所有者节点又是共享节点**。当请求节点已被授予相应的访问权限时，可以利用数据局部性。

与硬件中的缓存机制类似，我们也采用缓存行的粒度来利用数据访问的局部性，只是分布式缓存中使用的缓存行是可配置的，并且比硬件缓存大得多，以减轻小数据包的高传输成本。

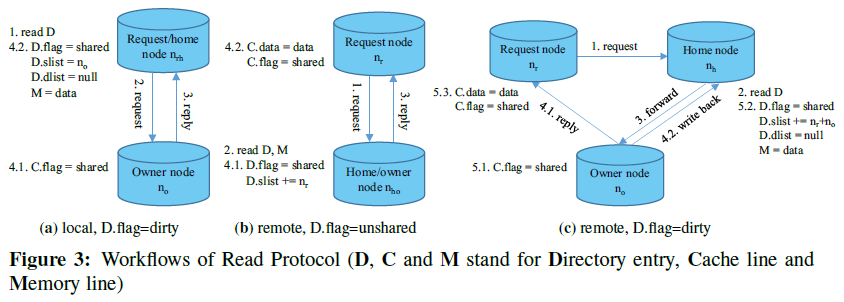
**本地home节点**上高速缓存行的目录状态可以是“共享Shared”（具有读取权限的某些远程节点可以共享），“脏Dirty”（由具有写入权限的远程节点所拥有）和“无共享UnShared”（由本地节点拥有）。这三种状态之间的转换如图2所示。

类似地，**远程节点**上的缓存行的缓存状态可以是“共享Shared”（只读），“脏Dirty”（可写）和“无效Invalid”（无效），取决于该远程节点持有的数据访问权限。另外，由于网络延迟，状态之间的过渡不是原子的，因此我们为每个可能的状态过渡（例如“SharedToDirty”）引入**过渡状态**。正如我们将在2.5节中看到的那样，过渡状态对于确保并发请求处理的正确性是必需的。



### 2.3 Read

GAM中有两种**读操作类型**，具体取决于请求节点的位置。我们将请求节点也是home节点的读取称为**本地读取**，将其他读取称为**远程读取**。在本节中，我们分别详细介绍两种类型的读操作的工作流程。图3给出了读取工作流程的图形说明。工作流的每个动作都与一个主号码和一个可选的次号码相关联，这样一个动作必须在具有更高主号码的那些动作之前执行并完成（**主号码小的动作需要在主号码大的动作之前执行并完成**），具有相同主号码的两个动作可以由不同的节点同时执行。



（**注意：只有数据的本地home节点，才具有Directory line和Memory line，其他类型的节点具有Cache line**）

#### 2.3.1 Local Read

发出本地读取时，如果没有远程节点拥有所有权，则数据仅驻留在本地内存中（“非共享Un-Shared）或处于只读模式（“共享”）。在这两种情况下，数据都是直接从本地存储器提供的，而不会引起任何网络通信。但是，如果有一个**远程节点拥有数据（“脏”）**，则工作流程将变得更加复杂，因为它涉及数据传输和缓存状态转换，这两者都需要节点间通信。

详细的通信工作流程如图3a所示。本地节点nh也是请求节点nr，首先读取相应的目录条目（D）以获取有关所有者节点no的信息（1），然后将READ请求发送给no（2），后者依次响应，并返回其自己的副本（3），并将相应缓存行（C）的状态更改为“共享”（4.1）。一旦接收到数据，本地节点nh将通过将标志改变为“共享”（4.2）来相应地更新其存储器（M）和目录条目。

#### Remote Read

对于请求节点与本地节点不同的远程READ，如果请求的数据已经被请求节点缓存，则缓存的副本可直接用于服务此读取请求，而不会引起任何通信。否则，根据缓存目录状态，有两个不同的工作流程，分别对应于“未共享/共享”状态（即“非脏”）和“脏”情况。

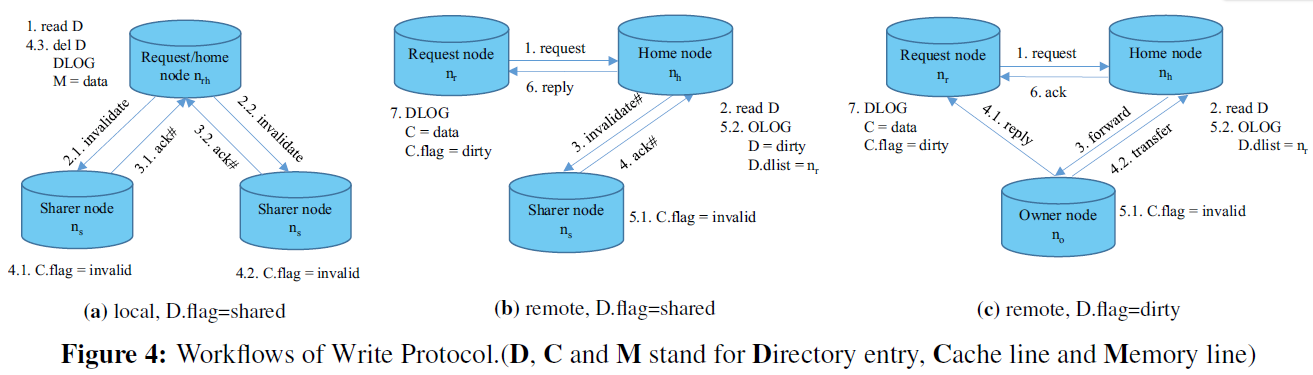
图3b显示了请求的数据为“未共享”的情况，因此home节点也是所有者节点。

首先，请求节点nr将所读取的请求通知给home节点nh（1）。读取缓存目录（2）并发现自己拥有所请求数据的最新版本后，主节点将用该数据返回给nr（3），然后相应地更新缓存目录和共享列表（4.1）。最后，请求节点nr用接收到的数据更新其缓存（4.2）。对于“共享”情况，尽管存在其他共享节点，但是工作流程与图3b完全相同，因为这些共享节点不需要参与。

在所请求的数据由远程节点拥有的第二种情况下，相应的缓存行的状态需要从“脏”更改为“共享”。如图3c所示，在意识到所请求的数据是脏的之后，home节点nh需要将读取请求转发到所有者节点no（3）。然后，no将向请求节点nr（4.1）和主节点nh（4.2）发送缓存行的副本，并最终将该缓存行标记为“共享”（5.1）。收到缓存行副本后，nh将更新缓存目录并将该副本写入其内存中（5.2），而nr会将副本放入其缓存中，以希望吸收更多的读取请求（5.3）。

### 2.4 Write

同样，我们还将写入请求分为两类：**本地写入**和**远程写入**，并在图4中说明了各自的工作流程。



#### 2.4.1 Local Write

对于由home节点发出的写请求，如果没有共享或所有者节点（数据没有共享或者在远程节点上有脏数据），则归属节点可以安全地写入数据而不会引起任何通信。否则，数据将被共享（“Shared”）或由远程节点拥有（“ Dirty”）。在第一种情况下，如图4a所示，请求节点nr（也是本地节点nh）将为每个共享节点（2）发出INVALIDATE请求。在接收到INVALIDATE请求后，每个共享节点将确认该请求（3），然后使其本地副本无效（4.1）。

确认（3）是必需的，因为它允许nr决定何时通过内存隔离区（临界区），这将在第4节中讨论。在从共享节点收集所有确认之后，nh将删除该数据的目录条目，并写入新的数据存入内存（4.2）。**为了支持故障恢复，nh还将在实际写入新数据之前执行DLOG记录新数据。**第二种情况与第一种情况相似，其中请求节点（即home节点）也将指示所有者节点的缓存行变成无效并等待确认。**唯一的区别是，所有者节点需要将最新的缓存行副本与确认一起附带，以使请求节点更新其本地内存。**

#### 2.4.2 Remote Write

如果请求节点也是所有者节点（dirty），则可以立即执行远程写入操作。否则，根据缓存行的目录状态，存在三种工作流情况，可以是“共享”，“未共享”或“脏”。

第一种情况的工作流程如图4b所示。当从请求节点nr接收到写请求（1）时，home节点nh检查其目录以获得关于共享节点（2）的信息，然后使共享节点上的缓存副本无效（3）。从所有共享节点收到确认后（4），nh执行OLOG记录相应缓存行的所有权转移并相应地更新缓存目录（5.2），此后返回数据的最新版本到请求节点nr（6）。高速缓存行的目录状态为“未共享”的第二种情况与第一种情况类似。区别在于，由于没有共享节点，因此归属节点nh现在可以跳过图4b中的步骤3和4，并立即向请求节点授予写权限。

图4c说明了缓存目录为“脏”的最后一种情况。接收到来自请求节点nr的请求（1）后，本地节点nh首先检查缓存目录以找到所有者节点no（2），然后将请求转发给no（3）。然后，no将缓存行发送到nr（4.1），确认所有权转移到nh（4.2），并使该缓存行的本地副本（5.1）无效。在从所有者节点号（4.2）接收到所有权转移消息之后，归属节点nh现在能够将所有权授予请求节点nr（6）。但在此之前，它首先需要通过执行OLOG记录该所有权，并更新脏列表（5.2）。来自no（4.2）和nh（6）的确认都是必要的，因为我们必须通过所有权转移消息（4.2）来更新主节点的目录，即使在某些情况下（例如，TryLock）转发请求（3）被拒绝了。在接收到来自所有者节点（4.1）的答复和来自home节点（6）的确认之后，请求节点nr现在可以继续进行操作，方法是记录新数据，更新缓存行并将其状态设置为“脏”（7）。 **应当注意，只有在接收到两个消息即（4.1）和（6）之后，才能执行（7）的操作**。否则，如果请求节点nr在接收（4.1）之后立即执行（7），则它可能会在home节点nh从所有者节点no接收所有权转移消息（4.2）之前放弃所有权（比如缓存行淘汰）。 在这种情况下，home节点将认为请求节点实际上已经拥有所有权，而实际上却不再拥有该所有权。（？）

### 2.5 竞争情况

在前两个小节中讨论的工作流程中可能会发生竞赛。例如，当提交远程读/写操作并等待其答复时，同一节点中的另一个线程可能也同时执行一个针对同一地址的远程读/写操作。这会导致**重复的请求**并浪费网络带宽。我们避免了这样的情况，即在处理第一个请求期间，此时缓存行被设置为过渡状态，此后针对同一缓存行的后续请求的处理将被阻塞，直到完成第一个请求为止。因此，**过渡状态有助于确保高速缓存行中每个基本操作（例如，读取和写入）的原子性。**

但是，过渡状态会导致**死锁**。假设具有读取权限的请求节点想被提升为具有写入权限，于是发送WRITE\_PERMISSION\_ONLY消息给home节点，而与此同时，home节点也希望写入相同的数据，于是将INVALIDATE消息发送至此请求节点。在这种情况下，两个节点都需要彼此等待以确认相应的消息，如上所述，由于所请求的缓存行的转换中状态，该消息的处理将被阻塞。为了处理这种情况，我们要求请求节点退回，就像它处于先前状态一样，并让home节点来解决不一致问题。具体而言，在发出INVALIDATE请求时，主节点将首先将相应的目录设置为“UnShared”。在接收到“WRITE PERMISSION ONLY”消息后，它将知道由后退策略引起的不一致，因此就好像是写请求一样处理此请求。(???)

### 2.6 LRU-based Caching

GAM采用最近最少使用（LRU）缓存替换策略来管理软件缓存。

每个节点维护一个**本地哈希表**，该哈希表将**全局内存地址映射到相应的缓存行**。随着越来越多的全局内存访问，新的缓存行不断添加到哈希表中。一旦哈希表大小超过预定义的阈值，高速缓存替换模块就会选择最近使用最少的高速缓存行来驱逐。

由于当多个线程同时更新列表时，仅使用单个LRU列表会产生巨大的开销，因此我们通过引入**多个LRU列表**以获得更好的性能来对此情况进行优化（虽然这样会降低LRU精度，需要适当平衡）。对于每个全局内存访问，并行线程**随机**选择一个LRU列表进行更新；对于每次高速缓存行逐出，将随机选择一个LRU列表来指导高速缓存行的替换。

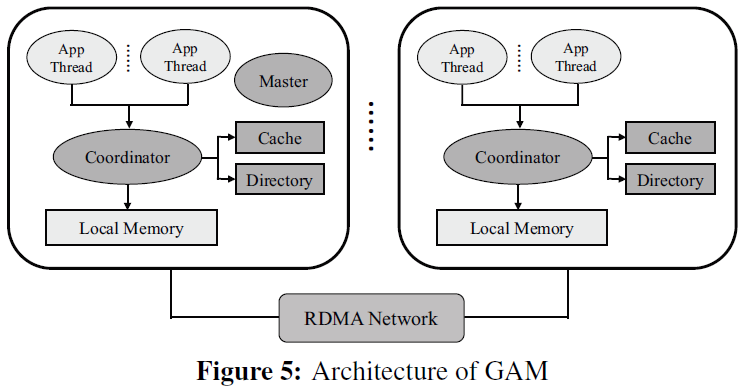
替换的工作流程取决于缓存行状态，该状态可以是“共享”或“脏”。为了收回“共享”缓存行，直接将替换（eviction）请求发送到home节点，此后可以立即为其他数据重新分配缓存行（不需要等待确认？）。为了替换“脏”高速缓存行，所有者节点将高速缓存行副本附加到替换请求中，并在删除高速缓存行之前等待主节点的确认。因此，在接收确认之前，该高速缓存行仍**可用于服务**发出的请求。

### 2.7 讨论

尽管GAM大大增加了应用程序可以访问的内存区域，但是总内存消耗可能仍然超过GAM中提供的全局内存。我们将这种内存不足的情况留给GAM的后续工作来解决，并探讨如何利用RDMA的最新进展（例如ODP（按需分页））来处理它。

## 基于RDMA的实现

在本节中，我们描述如何利用RDMA网络来实现第2.2节中描述的缓存一致性协议。 GAM的体系结构如图5所示，其中主要模块是GAM协调器，该协调器实现了缓存一致性协议，并用于协调本地和远程内存访问。整个系统中只有一个GAM主机（master，主节点），仅用于初始化和状态跟踪。统一的内存模型公开给上层应用程序，这些应用程序可以访问群集中的整个内存，而无需知道数据的实际位置。



### 3.1 协议的实现

RDMA提供了一种有效的数据传输机制，可以绕过CPU和OS直接访问远程内存。但是，完全不同的传输模型（例如，不同的API、单面传输）和显式的通信缓冲区管理，使得有效地和高效地使用它是不平凡的。基本上，有两组RDMA动词可用于启动数据传输，即READ和WRITE，以及SEND和RECEIVE。 READ / WRITE动词是单边操作，它们仅涉及请求者。但是，SEND动词要求接收者方预先将RECEIVE谓词发布到其接收队列中。（也就是send动词需要预先知道接受方的接收区）通常，单侧的READ和WRITE动词的性能要比双侧的SEND / RECEIVE更好。因为前者在接收方不涉及CPU或OS，而后者则要求接收方**预先发布RECEIVE动词**，并在收到消息后得到**通知**。然而，对于单边动词，由于接收者没有得到任何通知，因此很难弄清数据传输的完成。

为了利用RDMA的功能满足各种需求，我们分别设计了单独的**控制信息**和**数据传输**的通信管道。其背后的理由是控制消息（例如，READ/WRITE 请求）相对较小，需要立即通知接收方，而数据则以大单位传输（即缓存行）。对于控制通道，我们**避免使用RDMA Write动词**和繁忙的内存轮询（如在[14，21]的描述），而只使用双面RDMA SEND/RECEIVE动词。这是因为，与基于事件的机制（例如 epoll 和select）相比，繁忙的轮询将消耗大量的 CPU 资源，而每对（发送者/接收者）通信缓冲器将占用大量内存。此外，我们的通信并不总是遵循键值存储这样的请求/回复模式，因此，接收方的通信缓冲环上的**更新**很难附带（piggybacked）给发送方。但是，**对于传输量较大的数据传输**，我们采用单边的 RDMA WRITE 来构建数据通道。它允许直接写入最终目的地址，这与专门的通信缓冲区 [14，21] 的使用不同，**后者始终需要本地内存和注册通信缓冲区之间的额外数据副本**。

此外，还通过使用带/不带负载的 RDMA WRITE\_WITH\_IMM实现特殊的通知通道。对于纯通知的通信，只有请求标识符嵌入到头部中，作为直接（immediate）值（32 位），没有任何有效载荷，这在请求方和接收方 [23] 中都更有效。**如果还需要有效负载，数据通道和通知通道将组合在一起**，**以便接收方在收到数据后得到通知**。在大多数情况下，数据通道与通知通道相结合，实现大数据传输和高效通知。我们不使用 READ 动词，因为它的性能比WRITE [21] 差，而且很难以有效的方式保证数据的一致性 [14]（对于实现了数据cache的系统，read原语确实不好保证cache一致性，还是得额外的通信）。对于所有通信方式，我们使用**RC（可靠连接）**传输类型，因为我们需要可靠的传输和严格的消息顺序。

第2.3节和第2.4节中描述的工作流程是通过上述**三个通信渠道**实现的。

短控制消息，即“request”，“forward”和“invalidate”消息，通过控制信道（RDMA SEND / RECEIVE）传输，而“reply”和“writeback”消息通过组合数据传输和通知通道（带有效负载的RDMA WRITE\_WITH \_IMM）。另外，由于确认（例如，“ack”和“transfer”）仅传递成功信息，因此仅通知信道（没有有效载荷的RDMA WRITE\_WITH\_IMM）就足够了。**错误回复仍然需要控制通道，因为请求者需要的反馈信息多于32位立即数。**

### 3.2 Optimizations

我们将介绍以下优化，这些优化利用RDMA的特殊功能来进一步提高GAM的性能。

由于NIC高速缓存的大小有限，重要的是要使RDMA NIC所需的数据保持尽可能小，这些数据通常包括**页表、队列对和接收队列**。为此，我们首先将暴露给远程访问的内存组织成**大页**，以减少页表条目，进而减少TLB（转换后备缓冲区）未命中。另外，**一个节点内的所有线程共享同一组RDMA连接（即队列对）**，与n2 ×t2相比（如果每个线程都直接相互连接），这将队列对的总数减少到n2（其中n是群集中的节点数。t是每个节点中的线程数）。队列对数量的这种减少不会损害吞吐量。如我们所测量的，每个节点一个队列对足以使RDMA NIC饱和。此外，对于每个节点，我们仅使用一个**共享的接收队列**，该队列在所有关联的队列对之间共享。**这样，我们不仅减少了NIC必须维护的状态信息，而且减少了RDMA RECEIVE动词（对于需要预先发布的动词，如send）。**

为了减少CPU开销，选择性信号（selective signaling）技术用于完成通知。特别是，每r个请求都会向RDMA动词发送信号，从而减少了r倍的完成通知和清除例程（例如，释放send的缓冲区）的数量，经测试，这可以显着提高性能。为了进一步减少网络通信开销，我们尽可能使用RDMA内联技术直接通过PIO（编程输入/输出）（??）发送较小的有效负载，从而消除通过PCIe进行的额外DMA往返行程。**用于内联请求的缓冲区不需要预先注册，并且可以在发布后立即重新使用。**这特别有利于*控制信道*，因为其有效载荷大小通常足够**小**以内联。

为了利用请求之间的共享机会，我们将多个小数据包合并为一个大数据包，同时保持严格的数据包顺序。此外，如第2.5节所述，为了不发送重复的请求，我们将对同一缓存行的所有请求添加到待处理列表中，并按照发出的顺序逐一处理。

## y

内存一致性模型的范围很广，从强一致性模型（例如严格一致性和顺序一致性）到一些宽松模型（例如总存储顺序（TSO），部分存储顺序（PSO）和发布release一致性）。

基本上，**一致性模型取决于放宽全局内存访问顺序的程度**。因为只有两种类型的内存访问：读和写，所以总共有四种内存访问顺序3：读后读，写后读，读后写，写后写和放宽不同的顺序导致不同的内存一致性。例如，放宽“写后读”以使读取将返回较旧的数据，这是TSO，而进一步放宽“写后写”将在PSO中产生结果。允许对所有四个内存访问顺序进行重新排序，可以实现发布一致性。

较强的一致性使用户更容易进行内存访问推理，并降低了编程复杂度和调试难度。理想情况下，我们可以通过强制执行强一致性（例如顺序一致性甚至严格一致性）来最大程度地减少使用统一内存模型进行编程的用户负担。但是，尽管RDMA已经使网络延迟变得不再像以前那么重要了，但是强制强一致性仍然会导致无法承受的远程内存访问延迟，因为它需要同步执行读写操作（读操作会被写操作阻塞）。因此，我们首先放宽了写后读的顺序，以允许异步写操作，并将写操作从程序执行的关键路径中删除。

我们的下一个决定是进一步放宽**写后写**顺序。如果要保持此顺序并因此提供TSO一致性，则需要发出后台写入请求，以便仅在**所有较早的请求完成后**才可以发出每个写入请求，这意味着已向该请求节点授予了所有必要的写权限。这大大减少了写入请求合并的机会，进而导致更高的网络开销。而且，在这种情况下，**过载的节点将阻塞随后对其他节点的写入请求，并减慢整个系统的速度**。这样，我们放宽了写后写的顺序，以允许在较早的写请求之前使写请求可见。

尽管放宽写后读和写后写顺序的同时也放宽了GAM的内存一致性，但是编程的复杂性和程序的正确性并没有受到太大的影响，因为大多数程序员都熟悉异步写的编程模型（例如，文件IO），并且大多数程序的正确性不依赖于写入。但是，进一步放宽其他两个顺序将导致完全异步的编程模型，这大大增加了编程的复杂性和难度。

因此，GAM通过放宽写后读和写后写顺序来提供**PSO一致性**，因此采用了大多数程序员都熟悉的**同步读和异步写的编程模型**。为此，**一个工作线程专用于处理来自应用程序线程的所有请求。**发出读取请求后，应用程序线程将被阻塞，直到工作线程将请求的数据提取到给定的缓冲区中为止。另一方面，发出写请求后，应用程序线程立即返回，而无需等待该请求完成。通过使用将在下一节中讨论的显式同步原语（例如MFence，Lock / UnLock），可以轻松地实现更强的一致性。例如，通过在每个写操作之后插入MFence，可以轻松实现顺序一致性。锁原语还有助于实现应用程序级的可序列化性（例如，由第7.2节中说明的事务引擎实现的事务可序列化）。

### 4.1 同步操作

为了在更高级别的应用程序中实现更强的一致性，我们提供了两组显式同步操作–内存屏障（即MFence），类似于x86指令集中MFence指令的操作以及分布式锁操作（包括共享锁、排他锁及其try-lock变体）。锁定操作是隐式的内存屏障。对于每个Lock和MFence操作，它的后续操作将被挂起，直到工作线程完成所有之前的操作为止。此外，GAM还提供了Atomic操作，这些操作原子性地针对全局地址执行给定的功能。对于每个原子操作，工作线程将其视为对给定地址的隐式写操作，因此尝试获取相应的写权限。此后，工作线程针对该地址**调用**给定的函数。

Lock操作为程序员提供了一个自然同步机制，以协调共享内存环境中的数据访问。在我们当前的设计中，锁和数据是耦合在一起的，因此节点上的分布式锁具有与数据共享相同的粒度，即缓存行，**这意味着同一缓存行中的数据共享同一锁定（即锁的范围是按照缓存行的粒度）**。Lock 操作将请求的缓存行预装到本地缓存中，使请求节点成为共享节点（在 RLock 的情况下）或所有者节点（在 WLock 中），使后续在同一缓存行上的读/写或解锁操作免于额外的网络通信。同样，如果请求节点已获得请求缓存行的适当许可，则可以在本地服务Lock 操作。从这个意义上说，锁操作，如同他们的读/写对应，也可以缓存。

另外，阻塞锁定操作是以基于队列的方式实现的，而不是通过RDMA重复请求，这可能会使网络溢出。通过基于队列的锁定方案，在解锁操作之后，我们将选择锁定等待队列的头节点来授予锁定权限。但是，TryLock操作不会排队，如果其他人持有互斥锁，则锁定失败将立即返回。为了维持一致的状态，在锁定失败时需要一些撤消过程和特殊处理，因为它可能涉及多个节点，并且有可能只有一个节点拒绝授予锁，而其他节点则同意。

我们省略了共享或排他Lock操作的通信工作流程的描述，因为它们分别类似于对应的Read（图3）或Write（图4），但是在获取数据后需要一些锁定语义。

请注意，同一缓存行中的数据在节点之间共享相同的锁，因此阻塞的Lock操作可能在例如某些复杂的事务处理应用程序中导致意外的死锁。因此，除非在所有过程中都确保在锁获取阶段遵循一定顺序，否则在这些应用程序中建议使用非阻塞锁原语。我们打算通过将来解除对锁和数据的耦合来提供一种细粒度的锁定机制。

## Log和故障恢复

在本节中，我们为 GAM 设计了一个log方案，并展示了如何将其用于故障恢复。

有两种类型的log：DLOG 和 OLOG，它们分别记录数据写入和所有权转移。如图 4 所示，DLOG 在将数据写到内存/缓存之前通过请求节点调用，OLOG 在每次所有权转移之前由home节点调用。此外，在获取排他锁时，请求节点还调用 DLOG 记录与锁获取一起预取的缓存行。我们避免由于读取请求从所有者节点复制缓存行而导致的数据写入log。这有助于减少log开销，并且不会影响故障恢复。此外，每个日志条目还包括一个计数器，该计数器在每次所有权转让时增加。**为了最大限度地减少记录产生的性能退化，我们依靠 NVRAM 来容纳内存日志，并在日志即将满时异步将内容溢出到 SSD/硬盘中**。

我们现在将解释日志如何用于故障恢复。我们首先考虑单个失败节点案例，然后讨论多个失败节点的情况。为了讨论，我们假设每个数据书写都覆盖整个缓存行，因此每个 DLOG 条目也会记录整个缓存行，从而允许从单个 DLOG 条目中恢复缓存行的内容。无需付出太多工程努力，就可以在实施中放宽这一假设。

当检测nf失败时，所有未失败的节点都会从nf出现的所有共享列表中删除nf，这样nf就不会成为任何缓存行的共享节点。因此，数据恢复只需要处理 nf 是home节点或所有者节点的数据。我们逻辑上将数据恢复分为两个阶段，实际上可以并行执行。在第一阶段，nf 恢复把它作为home节点的数据，在第二阶段，每个未失败的节点都会恢复由 nf 拥有的脏缓存行。在恢复过程中，应保留当前由非故障节点持有的锁，以便继续运行在这些节点上的应用程序。数据恢复后，我们将展示如何恢复当前由其他节点锁定的 nf 缓存行的目录。

在数据恢复的第一阶段，nf 执行**反向日志扫描**，并利用与**未恢复的缓存行**对应的每个日志条目。对于每个此类条目 e，如果由 DLOG 记录，nf 将直接使用 e 的内容来恢复相应的缓存行 c。否则（说明所有权已经转交给其他节点），nf 将要求 c 的所有者节点验证其最近写的 c 的计数器是否与 e 中记录的值相同。如果是，则nf将提升该节点为c的所有者，并相应地更新缓存目录。否则，nf会记录一个UNDO条目，使e无效。例如，在图 4c 的情景中，home节点 nh 在 OLOG 后崩溃时，就会发生这种情况。在这种情况下，如果未收到确认，请求节点nr将永远不会执行DLOG，导致nr和nf之间的计数器不匹配。需要注意的是，每个缓存行**最多会经历两次恢复试验**，因为其第二个到最后一个未撤消的日志条目始终有效（应该是说后面的log信息被覆盖了，没必要也不应该恢复）。

第二阶段与第一阶段相似。对于 nf 拥有的每条脏缓存行，一个非失败的节点要求 nf 验证其它的最近写的缓存行是否与当前计数器一致。根据验证结果，匹配则此节点将促进 nf 作为所有者节点，否则，进行以它为该缓存行记录的UNDO log，从而使该缓存行无效。

（如何恢复锁）数据恢复后，在 nf 故障之前为"Shared"状态的 nf 缓存行已更改为"Dirty"（它作为所有者节点）或"UnShared"（它作为home节点），这意味着其他节点获得的缓存行上的共享锁已隐性释放。因此，我们需要恢复这些缓存行的目录。**为此，在数据恢复后，每个非失败的节点都会通知 nf 以前与它共享的缓存行。对于每个此类缓存行，nf 将读取当前所有者节点的内容（如果可能），将其状态更改为"共享"，最后相应地更新其共享列表。**我们不需要恢复排他锁，因为如前所述，每次排他锁的设置都与 DLOG 条目相对应，因此锁持有人始终会被提升为各自缓存行的所有者。

上述恢复过程可应用于多个节点同时发生故障的情况。在这种情况下，恢复过程被推迟到所有失败的节点都联机，然后并行执行每个节点。该过程类似于单节点故障的情况。具体来说，每个失败的节点首先从内部恢复以它自身或其他非故障节点作为home节点的缓存行。这留下一些未修复的缓存行，这些缓存行的主节点和所有者节点是两个不同的失败节点。我们将其恢复推迟到第二阶段，在此期间，每个失败的节点都会恢复其拥有的缓存行，从而恢复缓存结构。

## 应用

在本节中，我们演示如何通过两个示例开发使用 GAM 的应用程序：事务引擎和分布式散列表 （DHT）。

### 6.1 事务引擎

共享内存模型通过隐藏复杂的网络通信，更容易实现分布式事务引擎，因此开发人员只用关注核心单节点事务处理逻辑。基本上，每个事务处理节点都有指向全局地址的**全局索引的根条目**，以便它可以通过使用读写 API 遍历全局索引以访问所有表。对于事务处理协议，我们只需d。我们不依赖于两阶段提交 （2PC） 之类的协议来实现分布式共识，因为当交易准备提交时，请求节点已经获取了数据。通过利用暗中发出MFence的锁同步原语，我们可以在GAM提供的PSO一致性模型下轻松实现序列化。由于 GAM 提供的统一内存模型，我们的事务引擎可以自然地避免 2PC 间接费用，并完全消除应用层中数据传输的复杂性。（加锁，实际上就是悲观并发控制）

### 6.2 分布式哈希表

基于 GAM 提供的共享内存模型，DHT 可以作为包裹在多个节点上的分布式存储桶阵列来实现。具体来说，每个 GAM 节点负责 64 位密钥空间的子集，密钥与其驻留节点之间的映射由最高位决定。在每个存储桶中，有多个 12 个子条目和一个溢出指针，以处理哈希冲突并增强哈希表的占用率。每个哈希表条目都包含从密钥的最低 12 位中提取的 12 位标记，以区分同一存储桶中的键，20 位整数记录索引键值对的大小，以及指向存储键值对的全局地址的 64 位指针。通过这种方式，与传统的 DHT 不同，哈希表及其索引键值对位于同一物理节点内，我们的 DHT 实现将键值对与其索引条目脱钩，从而不仅能够降低 DHT 更新的成本（无论原始映射如何，都可以在本地重新分配），还可以自动平衡 GAM 中的节点负载。

## 性能评估

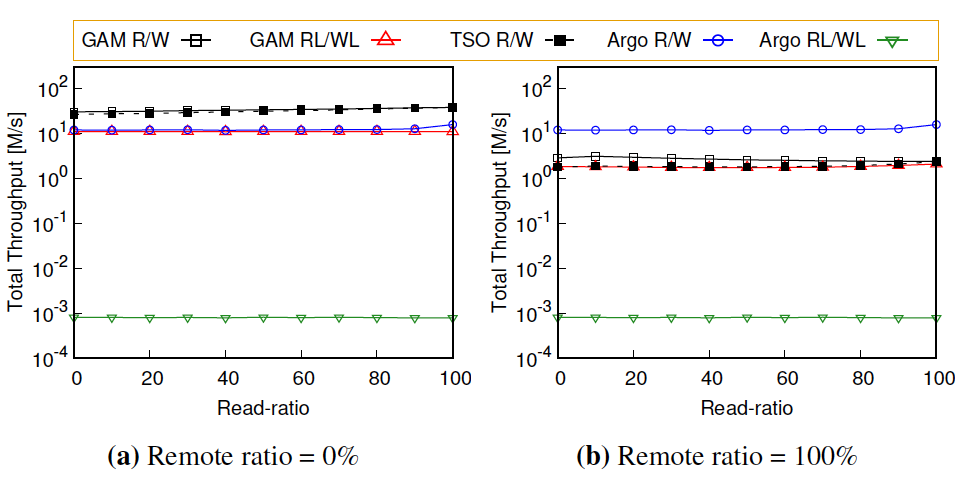
### 7.1 微基准测试

8个节点

每个节点：配置50%总数据量的Cache，运行一个进程（线程）

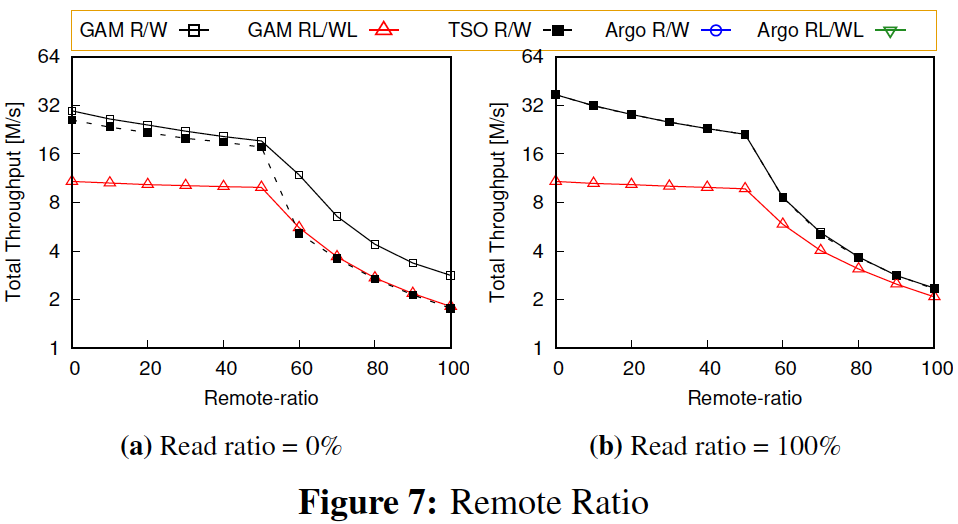
系统：GAM（PSO）、GAM-TSO、Argo’s（暂时忽略）

#### （1）读写比率



1. 在Remote ratio为0%的情况下，GAM的在PSO和TSO模型下，吞吐量是相同的。因为此时所有的操作都是本地的，系统的瓶颈RAM的吞吐量，与是否放宽写顺序无关。
2. 而在Remote ratio为100%的情况下，**对于纯写负载**，GAM在PSO模型下性能比在TSO模型下好60%。这是因为Cache的容量有限，一些请求需要远程通信。在PSO模型下，系统可以并行发送和处理多个请求，减少网络和CPU空闲的时间，所以吞吐率更高。对于纯读负载，TSO与PSO无差别。

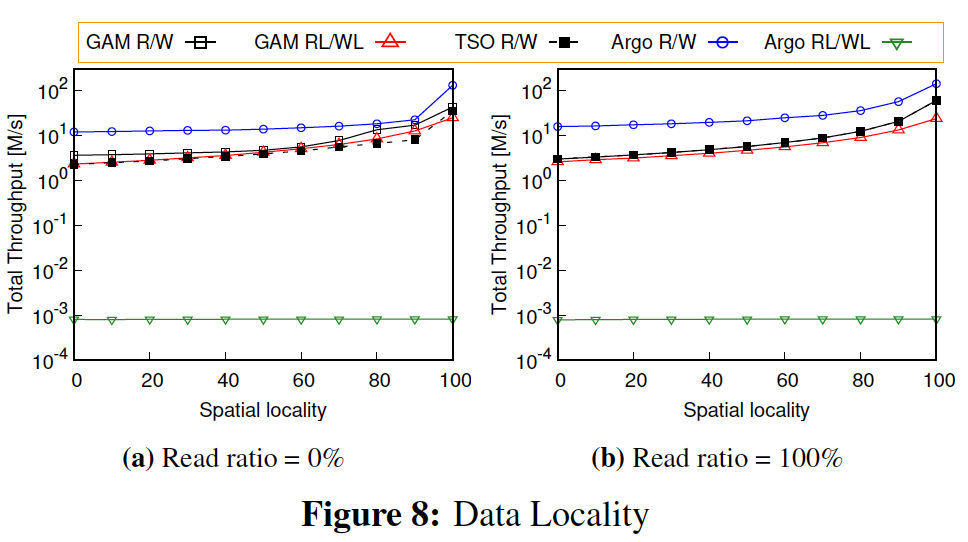
#### （2）远程比率



1. 由于Cache只配置50%，所以在**远程率低于50%**时，性能缓慢下降，此时主要受限于Cache数据结构的并发访问。远程率大于50%时，性能明显下降，此时的瓶颈是网络。
2. 如左图，远程率低于50%的情况下，由于50% Cache的存在，PSO和TSO的性能接近。但由于PSO的并行性，性能还是会比TSO稍微高一点。
3. 如左图，远程率大于50%的情况下，网络开销占主导，TSO模型下的性能逐渐接近PSO模型下的读锁和写锁负载的性能。因为这两种情况下，写操作都是按顺序的

#### （3）数据局部性

主要考究空间局部性。



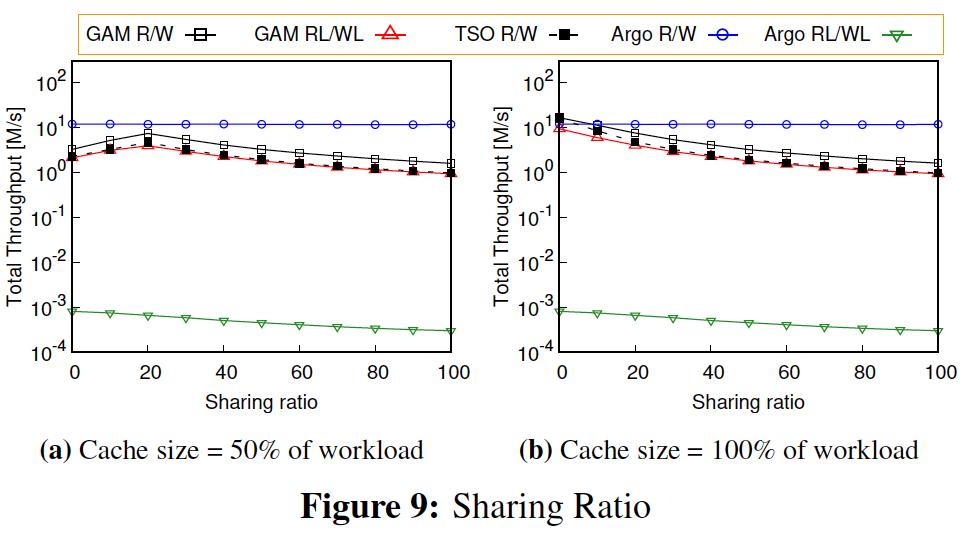
**系统可以从Cache中受益，随着数据局部性的提升，系统的性能也逐渐上升。**

如左图，此时负载为纯写，GAM的W（写）操作和WL（上锁写）操作的性能差距呈现有趣的趋势：逐渐减小，在60%的情况消失，然后又逐渐拉大。由于WL操作，在上锁时会附带缓存上锁的数据，所以能有更高的Cache命中率。所以局部率低于60%（Cache命中率较低）的情况下，WL操作能从Cache中获得更大的收益，所以差距逐渐减少。局部率高于60%的情况下，两类负载的命中率相差不大，但WL操作需要额外的锁开销，并占主导。

#### （4）共享比率Sharing Ratio

Sharing Ratio是指访问的数据中，所有多少比例是在多个节点上有副本的。

测试该指标主要是观察Cache的存在，对写操作的性能的影响（下降）程度，此时需要很多额外的通信。



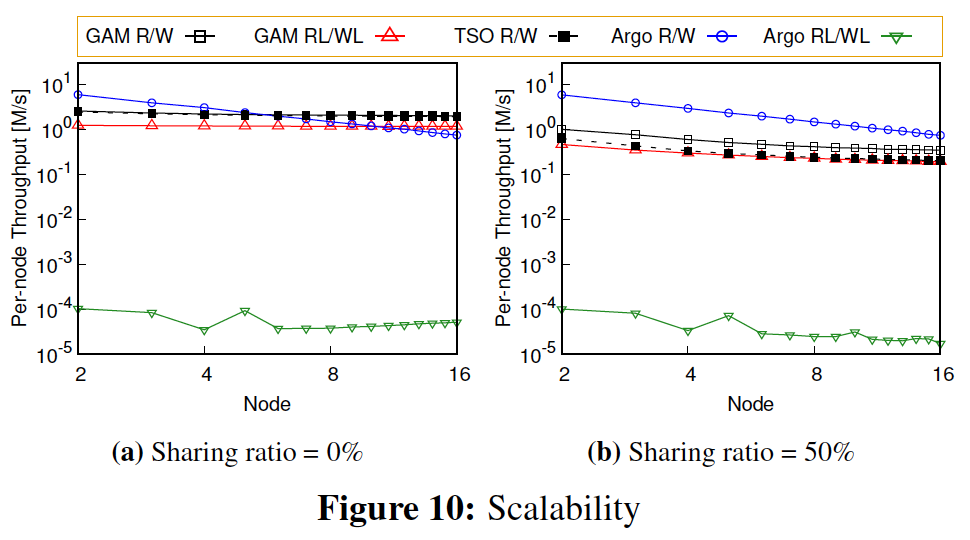
（读写比例为1：1）

**这里有个有趣的现象**：系统的性能并不像想象中的那样会降低。如左图，在共享率低于20%的情况下，GAM的性能甚至提高。这是因为随着共享率的提高，共享的Cache line会因为其他节点的写操作而变为无效，从而会被淘汰出Cache，使得非共享数据的访问可以使用更多的Cache，进而提高性能。**也就是说，在Cache有限（50%）、共享率较低的情况下，Cache反而是系统的瓶颈**。

这个结论可以从右图中得到验证。此时系统的性能逐渐降低。

**这个测试给我们的启示是**：实际应用中，需要根据负载的情况，合理设置Cache的大小。共享率较低的情况下，可以设置更大的Cache。而在共享率较高时，设置再大的Cache也对系统性能提高不大。

#### （5）扩展性



读写比例：1:1

共享率为0的情况下，系统**可以线性扩展**（不过这是通过在每个节点上运行一个**不相关**的测试程序来验证的，没有格外的系统同步开销）。

在共享率为50%的情况下，单个节点的性能会有所下降。

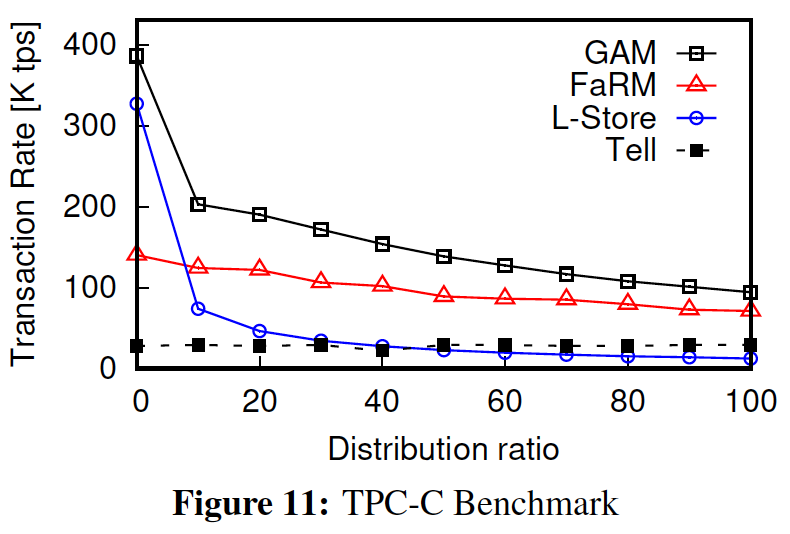
### 7.2 分布式事务处理

基准测试：TPC-C

测试系统：GAM、L-Store 、FaRM、Tell

公平起见：事务不能感知应用层面的数据布局

分布比率distribution ratio：控制一个事务中，访问其他（远程）节点数据的概率

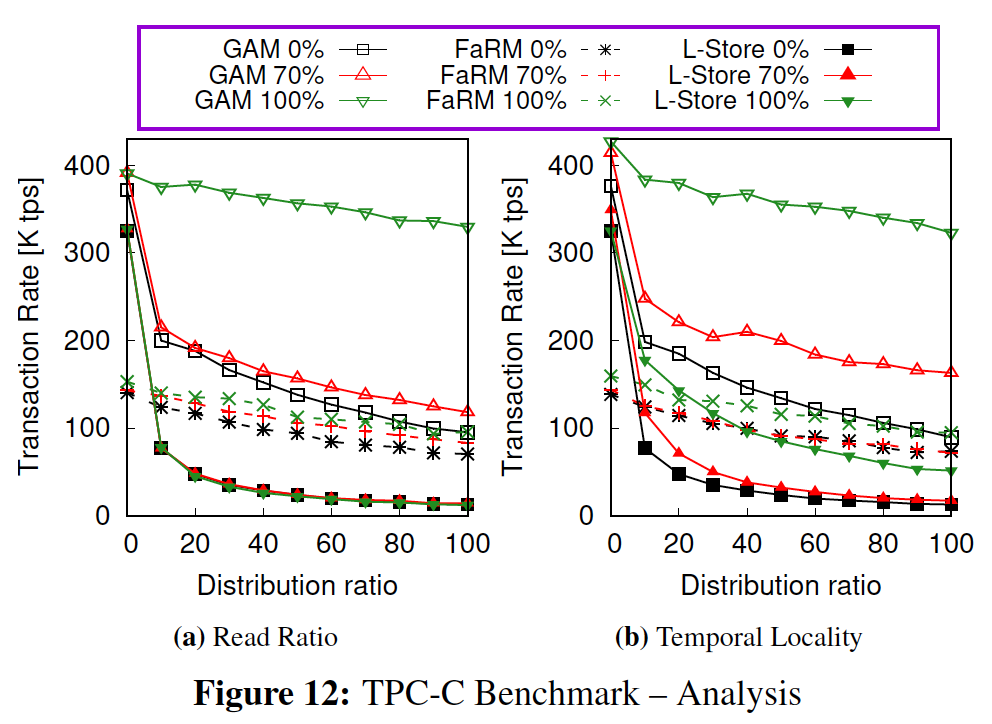


GAM的性能比FaRM要好，当分布比率升高时（比如70%），两者的性能逐渐接近。这是因为TPC-C基准测试是写明感的，这对于GAM系统来说是不友好的，当分布率升高时，为了维护缓存一致性，GAM需要更多的网络通信。

为什么GAM性能会更好，我想可以归结为两个原因：

1. 更好的通信方式，避免基于事件的编程模型，从而减少由于轮训而降低CPU的利用率
2. Cache的存在，避免两阶段提交。可以直接本地提交。

**论文中提到，即使在分布率为0%的情况，FaRM的性能仍比较低的原因是**：FaRM仍有一些数据（如，ITEM table)）不能完全放在本地，这个具体情况如何还需考究。



GAM对于读比率越高、局部性越高的事务，性能就越高，提升幅度较大。