F2FS

普通FLASH好处：非易失性；独立的cell可以比较；对机械震动不敏感。组成：cell,page,block,chip。不足：1. Erase-b4-write：数据由page为单元，erase block由page组成->erase b4 write; 2. 随机写会产生内部碎片,因此序列化写入到erased block; 3. 每个erase block的写周期是被限制的。Segment(基本管理单元)<section(擦除的单位)<zone；4.当要用到多块NAND时，需要一个控制器（FTL：Flash translation layer）连接多块存储片，带来了开销。 随机写的负面影响可以通过LFS或者copy on right来降低，但是会减少寿命降低性能。提出F2FS。

设计特点

1. Flash友好型磁盘layout：把抽象出的元数据放在一起(locality)便于擦除；多头logging便于区分冷/热数据。磁盘被分为6部分：superblock、checkpoint、segment info table、node addr table、segment summary area、main area。如何读文件”/dir/file”：
2. 先从NAT里获取根目录inode；
3. 在根目录的block里，找dir目录以及对应的inode；
4. 通过NAT将dir的inode转化为物理地址；
5. 获得dir的inode，读取相应block；
6. 在dir中，找到file的dir entry，重复3,4，最终从MA虫得到数据。
7. 文件结构：原LFS(log-structured fs)中，如果leaf数据被更新，它被指向的direct和indirect指针都被递归更新；而F2FS中，为防止这种传播问题(wandering tree：我们要修改一个file data，就必须修改direct pointer block。因为这里需要擦除data block，就得给它一个新的block id，从而必须修改direct pointer block，以此类推，一层层修改，直到最上层)，只更新一个direct指针和NAT entry。
8. Multi-head logging：F2FS一共有6个日志area来区分hot/cold数据。数据温度：Node > Data；Direct node > Indirect node；Directory > User file。LFS拼命地顺序往后写，如果我们只有一个head的话会导致Log里夹杂着不同的数据，node和data混合在一起，缺点是，一个block每次都会一整块擦除。我们希望改block的时候，它的数据都是要修改的，不应该是不修改的。因此把频繁改变的数据和不怎么修改的数据分开存放是很好的。根据数据hot 和cold的区别，设计不同的head，（元数据和数据）\*（hot warm cold）6种。GC垃圾回收：释放invalid的数据。
9. Cleaning：在section内进行。当存储容量用完时就会发生cleaning，所以降低此开销很重要。1)找section：foreground：没有足够section时被触发，使用greedy，先去从稀疏的section找，优先清除这里的空间，这里需要valid block check，判断block数据是不是有效；background定期由某线程触发并在后台运行,综合考虑使用率和使用年龄。2)迁移valid的block：fg中，先通过SIT的validity bitmap找所有的valid segment，再从SSA找到parent index structure，再讲valid block移到main area中其他地方；而bg clean,F2FS将block加载到page缓存并标记为dirty，然后等待内核来把它们写入存储(lazy igration)。3)当validblock被迁移后，该section(称为pre-free section)就将在下一个checkpoint产生时被擦除。
10. Adaptive logging：为了降低cleaning cost,F2FS综合使用normal logging和threaded logging，clean section多于5%用normal。前者是序列化append，当容量不够时cleaning开销大；后者是log到dirty segment中，复用了dirty segment中的invalid block，并不用clean，但会导致随机写。
11. Checkpoint和Recovery：为了能在断点和系统崩溃时恢复。Checkpoint的被触发后：1)所有在page cache中的dirty node和目录项都会被flush. 2)延迟写操作，包括syscalls. 3)fs元数据、NAT、SIT、SSA都会被写到它们的区域.4)将checkpoint pack写到CP area.

Roll-back recovery：F2FS会对checkpoint、NAT、SIT有一个shadow copy来降低checkpoint的latency和I/O；从最近的chkp恢复。

Roll-forward recovery：只写data blocks和direct node blocks。将最后一个checkpoint的日志记录N之后n个与之前n个比较，不同的话证明有改动->refresh。

Non-volatile memory

PM 字节寻址

随着NAND(非易失性存储技术)的出现，降低了存储设备和内存之间的巨大开销。

一致性：通过保证一系列写操作（A）是在写操作B之前具有持久性。以前如何保证一致性：write-through（内存和缓存数据同时更新，会带来开销），这样就需要PM写操作使用non-temporal存储指令来绕过CPU缓存（增加编程难度）。

PowerGraph

图运算的挑战：poor locality of mm access; I/O密集型; 分区之后难以并行计算。

社交网络等自然图具有幂等分布特性（power degree）：大量点只有很少邻居；只有少量点与大部分点相连。幂等指数越大，就有越多的店只有少量邻居。

幂等性带来的限制：1. GS都与邻居数成正比，因此带来不均衡的并行处理；2. 具有幂等性的图很难划分，划分要求最小化传输；3. 带来通信不对称和瓶颈；4. 存储：high-degree的点要存储所有邻居的信息，占用巨大内存；5. 计算量大。

图并行计算的概念：G={V, E}，有一个程序Q可以运行在V中任意一个点v上，并且可以与其他相邻实例u交互，且(u,v)属于边集合E。

GAS。Gather：收集与点u相邻的所有v以及边的值；Apply：然后更新点u的值；Scatter：最后将更新的值传给之前的所有边。

Pregel（同步,面向点）：如pagerank，使用message的方式来做Gather，累加message，再把更新的value传给邻居，AS都是以vertex为单位。

GraphLab（异步分布式共享内存，面向边）：Pagerank为例，将所有邻居节点的值加起来，再把最新的值发给所有out neighbors。点的范围是全体邻居，GA可以由用户定义，所有的点和边都对邻居可见，不区分有向无向边。

PowerGraph就是为了解决图的幂等性带来的限制，综合了GraphLab和Pregel的优点：采用了前者的共享内存、避免让用户来定义消息的传播；采用后者的大量同步Gather特性。PowerGraph维护一个data-graph with用户定义的点集和边集。既可以同步执行也可以异步执行。

什么是delta caching？

当用户想要重新计算某些点时，会调用Activate(v)来激活它们。当完成scatter后会变回非活跃状态。点也可以激活自己和它的邻居。

PG同步执行时, GAS是顺序执行，通过barrier来限制顺序执行；PG异步执行时，当AS操作执行后会立即更新图，使得最新的数据对接下来的计算和邻居可见。异步可以被很好运用在很多算法上，但是它的效率取决于机器数量和网络资源。所以PG支持强序列化的运算，并且加入了并行锁协议，能保证对高度点的公平。

Vertex-cut GAS:当顶点按点切分方式被分到3台机器之后，在多个节点上指派一个为Master,其余的为Mirror。

Mirror上可以运行Gather程序来收集所有邻居的信息，并进行聚合计算(sum)后发送给Master。

Master上的Gather程序收集这些结果，最终将这个结果应用到Apply程序上，得到新的节点状态。然后通过Scatter程序将新的节点状态广播给各个Mirror，Mirror进而广播给各个邻居。

这种点切割的方式解决了幂等性问题。

边切割为什么不好: 每切割一条边都会带来存储和网络开销，因为每个machine都要维护一个边和点的副本。

总结：Powergraph：GAS分解(使得高度的点可以分开并行计算)+点分(解决幂等性问题和边分带来的网络和内存overhead)。

Transactional Memory

<http://kb.cnblogs.com/page/504824/>

传统同步技术，互斥锁，易于使用，并发存在问题: priority inversion(优先级反转：持有高优先级的锁的低优先级进程被终止)、convoying(一个持有锁的进程被deschedule，使得其他进程无法运行)、deadlock。

基本思想：利用高速缓存一致性协议来检测有冲突的access。

事务的两个属性：序列化执行+原子性。

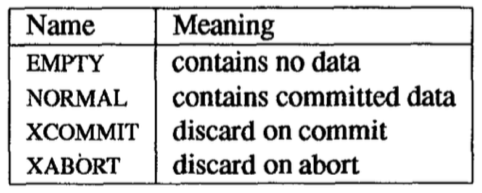
事务的三条指令：

Load-transactional(LT)：读取共享内存中的数据到私有寄存器；

Load-transactional-excluesive(LTX): LT+内存有可能被更新；

Store-transactional(ST): 将寄存器的值写到共享内存。

实现：为了最小化对非事务性操作的影响，每个处理器维护两个缓存：regular cache and transactional cache. 这两个都是主缓存。Regular缓存是直接相连映射，只能映射到特定缓存块；事务缓存是小容量的全项链映射，可以map到任意缓存块。



如果用窥探(snoopy)的缓存一致性协议，那么每个事务操作缓存都有两个entry：XCOMMIT(old value)和XABORT(new value)。如果这个事务成功commit了，XABORT->NORMAL, XCOMMIT->EMPTY；如果事务被abort掉，XABORT->EMPTY, XCOMMIT->NORMAL。

当事务cache需要一个new entry时，会先找EMPTY entry，再找NORMAL entry，最后找XCOMMIT entry。如果XCOMMIT是DIRTY状态，里面的数据必须先被回写到内存。回写模式有它的优势：它能过滤掉对同一地址的反复写操作，并且，如果大多数缓存段都在回写模式下工作，那么系统经常可以一下子写一大片内存，而不是分成小块来写，效率更高，降低网络开销。

在总线上，本文添加了三种bus，主要是针对事务性缓存操作的。如果是Busy状态，那么其他的事务会被abort。

事务内存优点：易于使用，只需要将需要独占运行的部分放入一个事务block里；没有锁和死锁；并行度、性能提高了。缺点：应用程序的执行被限制，要考虑怎么分解为事务；很难debug，因为在事务里面不能打断点。

TM的不足：TM的实现是建立在事务持续时间少和数据集少的前提下，有可能会被中断或者缓存不够用；缓存一致性可能比一些其他用来实现同步的要低效，因为它们在临界区用了barrier，不过barrier太多也会影响性能。

Bigtable

目的：为了使PB级的数据能在上千个machine有很好的可扩展性。Wide applicability, scalability, high performance, high availability.

Bigtable的building block：GFS(稳定地存储数据，用SSTble文件格式); Scheduler(在Bittable服务中调度job); lock service(选取master、location boostrapping)。

SSTable：SSTable的全称是Sorted Strings Table，是一种不可修改的有序的键值映射，提供了查询、遍历等功能。每个SSTable由一系列的块（block）组成，Bigtable将块默认设为64KB。在SSTable的尾部存储着块索引，在访问SSTable时，整个索引会被读入内存。

Tablet：包含一个table中的一些range of rows。是distribution和负载均衡的基本单位。一个Tablet可能包含多个SSTables。

Table：多个tablet组成一个table。SStable可以被多个Tablet共用。Tablet不可以overlap，但是SStable可以。

如何找到一个tablet：客户端会缓存片的位置信息，如果在缓存里找不到一个片的位置信息，就需要查找这个三层结构了，包括访问一次Chubby服务，访问两次片服务器。

Servers：一个tablet server管理多个tablet，每个tablet占10-200MB不等，每个tablet只存在于一个server上，当tablet过大时会被server分割。Master负责负载均衡和容错，使用Chubby来监控tablet server的情况、重启宕掉的服务器；GFS用来replicate data，当一个server挂掉，会在已经有数据的服务器上重启tablet server(data locality)。

如何编辑/读table：更改会被提交到GFS的commit log，然后被提交到memtable，memtable的每一个row都是Copy on right的(for concurrency)；读的话是从SSTable和memtable的合并视图中读。因为SSTable和memtable本身都是已排序的，所以合并相当快。在tablet分裂和合并的过程中，R/W不受影响。每一次minor compaction都会产生一个新的SSTable文件，SSTable文件太多读操作的效率就降低了，所以Bigtable定期执行merging compaction操作，将几个SSTable和memtable合并为一个新的SSTable。BigTable还有个更厉害的叫major compaction，它将所有SSTable合并为一个新的SSTable。

片的存储和访问：当片服务器收到一个写请求，片服务器首先检查请求是否合法。如果合法，先将写请求提交到日志去，然后将数据写入内存中的memtable。memtable相当于SSTable的缓存，当memtable成长到一定规模会被冻结，Bigtable随之创建一个新的memtable，并且将冻结的memtable转换为SSTable格式写入GFS，这个操作称为minor compaction。

BT支持非关系型数据库。BT是一个稀疏的、分布式的、持久化存储的多维度排序map。

Bigtable的键有三维：分别是行键（row key）、列键（column key）和时间戳（timestamp），行键和列键都是字节串，时间戳是64位整型；而值是一个字节串。可以用 (row:string, column:string, time:int64)→string 来表示一条键值对记录。行键可以是任意字节串，通常有10-100字节。行的读写都是原子性的。Bigtable按照行键的字典序存储数据。Bigtable的表会根据行键自动划分为片（tablet），片是负载均衡的单元。最初表都只有一个片，但随着表不断增大，片会自动分裂，片的大小控制在100-200MB。

列是第二级索引，每行拥有的列是不受限制的，可以随时增加减少。为了方便管理，列被分为多个列族（column family，是访问控制的单元），一个列族里的列一般存储相同类型的数据。一行的列族很少变化，但是列族里的列可以随意添加删除。列键按照family:qualifier格式命名的。

时间戳是第三级索引。Bigtable允许保存数据的多个版本，版本区分的依据就是时间戳。时间戳可以由Bigtable赋值，代表数据进入Bigtable的准确时间，也可以由客户端赋值。数据的不同版本按照时间戳降序存储，因此先读到的是最新版本的数据。

每一个row range被称为tablet， 如同一域名的各个网页就属于一个tablet。Column keys被分组，成为column family，是access ctrl的基本单元。在同一个column family的数据基本是同一类型，格式为family:qualifier，如一个网站有很多个anchor，每一个column key都是一个引用该网站的网站，content存的是网页链接。

每个cell都可以有同一个数据的多个版本，indexed by时间戳。但是Bigtable要对一些版本数据垃圾回收。Client可以指定只要多少时间内的数据，或者只保存最新数据。

Bigtable api提供了API可以对table进行各种操作。还可以支持single-row的事务，用来原子性操作存储在single row key下的序列。暂不支持夸row key的原子性操作。

Bigtable集群包括三个主要部分：一个供客户端使用的库，一个主服务器（master server），许多片服务器（tablet server）。

Bigtable和GFS的关系：集群包括主服务器和片服务器，主服务器负责将片分配给片服务器，而具体的数据服务则全权由片服务器负责。但是不要误以为片服务器真的存储了数据（除了内存中memtable的数据），数据的真实位置只有GFS才知道，主服务器将片分配给片服务器的意思应该是，片服务器获取了片的所有SSTable文件名，片服务器通过一些索引机制可以知道所需要的数据在哪个SSTable文件，然后从GFS中读取SSTable文件的数据，这个SSTable文件可能分布在好几台chunkserver上。

Spanner

<http://dblab.xmu.edu.cn/post/google-spanner>

是一个分布式的多版本的数据库：支持事务(ACCID)、提供类SQL查询语句、模式化的半关系型table。

为什么需要Spanner：bigtable无法应用到一些特定类型的应用上，比如具备复杂可变的模式，或者对于在大范围内分布的多个副本数据具有强一致性要求(Gmail、Calendar、Android market、AppEngine)；现有的存储方案不够高效，如Megastore的吞吐量不高、Bigtable的弱一致性。

一个Spanner部署称为一个universe。Spanner有两个单例：universe master和placement driver。Spanner被组织成许多个zone的集合。

zone是管理部署的基本单元。zone的集合也是数据可以被复制到的位置的集合。当新的数据中心加入服务，或者老的数据中心被关闭时，zone可以被加入到一个运行的系统中，或者从中移除。zone也是物理隔离的单元，在一个数据中心中，可能有一个或者多个zone，例如，属于不同应用的数据可能必须被分区存储到同一个数据中心的不同服务器集合中。一个zone包括一个zonemaster，和一百至几千个spanserver。Zonemaster把数据分配给spanserver，spanserver把数据提供给客户端。客户端使用每个zone上面的location proxy来定位可以为自己提供数据的spanserver。

Spanner的软件栈：

每个spanserver负载管理100-1000个称为tablet的数据结构的实例。一个tablet就类似于BigTable中的tablet，也实现了下面的映射：

(key:string, timestamp:int64)->string

与BigTable不同的是，Spanner会把时间戳分配给数据，这种非常重要的方式，使得Spanner更像一个多版本数据库，而不是一个键值存储。

为了支持replication，每个spanserver都在每个tablet上实现了一个单个Paxos状态机，存储tablet的元数据和日志。当前的Spanner实现中，会对每个Paxos写操作进行两次记录：一次是写入到tablet日志中，一次是写入到Paxos日志中。Paxos是为了实现一系列mapping副本一致性。Write必须在leader上初始化Paxos协议，Read可以直接从底层的任何副本的tablet中访问状态信息，只要这个副本足够新。这些replica集合组成一个Paxos group。

对于每个是领导者的副本而言，1).每个spanserver会实现一个锁表来实现并发控制。这个锁表包含了2PL锁机制的状态：它把键的值域映射到锁状态上面; 2). 每个spanserver也会实现一个事务管理器来支持分布式事务。这个事务管理器被用来实现一个participant leader，该组内的其他副本则是作为participant slaves。如果一个事务只包含一个Paxos组（对于许多事务而言都是如此），它就可以绕过事务管理器，因为锁表和Paxos二者一起可以保证事务性。如果一个事务包含了多于一个Paxos组，那些组的领导者之间会彼此协调合作完成两阶段提交。其中一个参与者组，会被选为协调者，该组的participant leader被称为coordinator leader，该组的participant slaves被称为coordinator slaves。每个事务管理器的状态，会被保存到底层的Paxos组。

目录和放置：

在一系列键值映射的上层，Spanner实现支持一个被称为“目录”的桶抽象，也就是包含公共前缀的连续键的集合。对目录的支持，可以让应用通过选择合适的键来控制数据的局部性。

一个目录是数据放置/转移的基本单元。属于一个目录的所有数据，都具有相同的replicatoin configuration。当数据在不同的Paxos组之间进行移动时，会一个目录一个目录地转移。一个Paxos组可以包含多个目录，这意味着一个Spanner tablet是不同于一个BigTable tablet的。一个Spanner tablet没有必要是一个行空间内按照词典顺序连续的分区，相反，它可以是行空间内的多个分区。我们做出这个决定，是因为这样做可以让多个被频繁一起访问的目录被整合到一起。

一个目录也是一个应用可以指定的地理复制属性（即放置策略）的最小单元。当一个目录变得太大时，Spanner会把它分片存储。

并发控制：

进一步说，把Spanner客户端的写操作和Paxos看到的写操作这二者进行区分，是非常重要的，我们把Paxos看到的写操作称为Paxos写操作。例如，两阶段提交会为准备提交阶段生成一个Paxos写操作，这时不会有相应的客户端写操作。

True Time API:

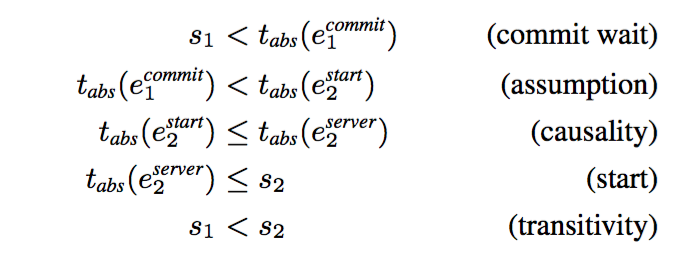
TrueTime明确地将时间表示为一段间隔TTinterval:[earliest, latest]。TTinterval的终点是TTstamp类型；TT.after(t)，true if 时刻t完全pass之后；TT.before(t), true if时刻t还没到。在每个Paxos组中，Spanner对Paxos写操作分配时间戳是单调递增的，即使跨leader。

Spanner的external consistency：如果事务T2的开始事件发生在T1的提交事件之后，那么T2的提交时间戳一定比T1的提交时间戳靠后。该协议遵循以下两个规则：

1. Start. 协调者leader给Ti分派的提交时间戳si不小于TT.now().latest。

2. Commit wait. 协调者leader会保证在TT.after(si)返回true之前，客户端看不到Ti提交的任何数据。也就是说从客户端提交到真正在服务器被提交是间隔了一段时间的。

像Bigtable一样，发生在同一个事务里的Write会缓存在客户端直到被commit，所以在同一个事务里的Read操作是看不到write更改的数据的。



Spanner做到的；

Lock-free的跨数据中心的读事务；外部一致性；时间戳分配；Truetime。

Spanner没做到的：

如何read at present time；原子性模型变化；非阻塞式的读操作。

Tail at scale

重点：会考得比论文中具体；tail latency为什么在scale大时会很严重？如何在large scale的情况下减少tail latency？

什么是latency tail-tolerant system: A predictably responsive whole out of less-predictable parts.

为什么存在响应时间的variability？响应时间的多样性导致了high tail latency，而响应时间的多样性主要是由于以下几个原因：

资源共享：多个app可能会共享同一机器，导致cpu,cache,内存和网络资源竞争；

Deamons：后台运行的守护进程虽然只用很少的资源，但是在它们被调度的时候可能会有多毫秒的间断。

Global resource sharing：在不同机器上运行的app也会竞争全局资源，如网络交换机和共享的文件系统。

Maintenance activity：后台的activity，类似分布式FS的数据重组、Bigtable的定期log compaction、定期的垃圾回收等。

如何减轻variability？资源的超供给、软件实时工程、提高可靠性。

如何reduce tail latency?

1) Within-request immediate-response techs

很多种web服务都是有多个副本部署在服务器上的，这种方式很适合具有只读、松散一致性的数据集。

解决方案：利用redundancy 冗余

a) Hedged(两面下注的) request：先向一个机器发送请求，如果在一定时间内没有得到响应再向第二个、第三个机器发送。一旦受到了第一个响应，就取消其他的请求。这种naive的方法会带来额外负载，但是可以有效减少latency。怎么控制第二个请求发出去的时间？一种方法是等第一个请求在95percentile还没收到响应时发送，这样就能将额外的负载控制在5%。

b) Tied request：Hedged request tech使得多个服务器不必要的执行了同一个请求。variability的一个常见的原因就是队列等待。Tied request就是在多个服务器上都有copies的request，同时服务器之间可以通过交流来更新copies的状态。比如，一个客户端给两个服务器发送了同一个请求，每个都打了该服务器标签，当其中一个请求开始执行时，就会给另一个服务器发送个取消执行的消息，如果此时对方的请求仍在队列中排队，就可以即刻被取消。但如果在客户观发请求之前，两个服务器的队列都为空，就会导致两端同时向对方发cancel请求，解决方案就是给发送请求增加1ms的延迟。在文中对bigtable读取数据的实验可以得到，tied request在磁盘使用的开销小于1%，证明它的取消策略可以有效消除冗余的读操作。

c) Alternatives：(1) 先探测各服务器的请求队列，再将请求发到负载最轻的服务器上。这个方法比同时将请求发到两个服务器上低效，因为：负载情况可能在探测和请求实际到达服务器的时间间隔中有改变；负载轻重难以衡量，因为有类似硬件和底层系统的可变性；多个客户端可能会同时选择同一个负载轻的服务器，导致压力骤增; (2) 请求先被发到一个服务器。当且仅当此服务器没有该请求的缓存数据时，请求会被转发到另一个服务器，该方法对于限制在单个request的情况下也可以用。

以上的技术只在导致variability的因素不会同时影响到请求的多个副本时起作用，但事实却经常不是如此。

2) Cross-request long-term adaptions

这些techs是用来降低由粗粒度的因素(如service-time variation and load imbalance)带来的latency variability。这些因素主要有两个原因：1. 底层机器性能不统一，而且会有变化；2. 有可能有数据引起的负载不平衡，如某一个item突然迎来了访问高峰。

解决方案：利用分区

a) Micro-Partitions，微分区。分区数量大于提供服务的服务器数量。这样可以动态将分区分到机器上去。如bigtable是把数据存到tablet中的，一个machine通常管理20-1000个tablet。微分区也有利于错误恢复，因为当错误发生时只用恢复为数不多的工作。

b) Selective replication： Micro partitoin的一个加强就是检测或者预测可能导致负载不均衡的item，并且新增item的replica，这样负载均衡系统就可以使用新增的replica来将这些热度高的micro partitions分布到多个机器上。谷歌的web搜索系统就使用了这个方法，在多个微分区上制造另外的item replica.

c) Latency-induced probation：通过对延迟分布的观察，一些服务器可以检测到当把某个slow machine排除在外的时候，系统性能会很好。这些slowness的原因通常是由一些暂时的因素如网络拥塞和CPU被占用引起的。

实践-google海量信息检索系统(IR system)的两个策略：

Good enought：在大型IR系统中，只要超过某个比例的叶子服务器给出了响应，用户就拿到结果(good enough strategy)。因为一个服务器拿到最好的结果的几率小于千分之一；

Canary request：可能有某个请求走了条非同寻常的路线，导致整个系统crash或者大量延迟，在某一刻延迟的响应会突然分发出去，导致high fan-out。为了防止这个现象，根服务器会首先把一个request发送给一两个叶子服务器，只有在一定时间内得到成功响应后才把这个请求发给另外上千个叶子服务器。否则就把请求标记为潜在危险的请求，不发送给其他服务器。（提高鲁棒性）这个策略之给系统带来少数整体延迟(以为整个系统要等根服务器的响应)，但是避免了整个crash导致的overall latency。

为什么写操作对latency容忍度高：

首先，对延迟敏感的写操作的规模一般比较小；第二，在给用户响应后，数据更新可以不从关键路径走；第三，很多服务可以被构建为能够容忍非一致性更新的模型(本质上就能容忍latency)；第四，对于那些要求一致性的写操作，常用的技术是quorum-based的算法，但这个算法只用提交到三到五个replica，所以本质上也是tail-tolerant的。

未来，有可能硬件级的variability会更高，因为越来越多的强劲的电源被设计出来了，导致了设备之间的不均匀性。这使得软件层面对variability的容忍要做得更好。幸运的是，几种新兴的硬件趋向会提高latency tolerating的效率。比如，更高的网络带宽可以降低tied-request的开销。

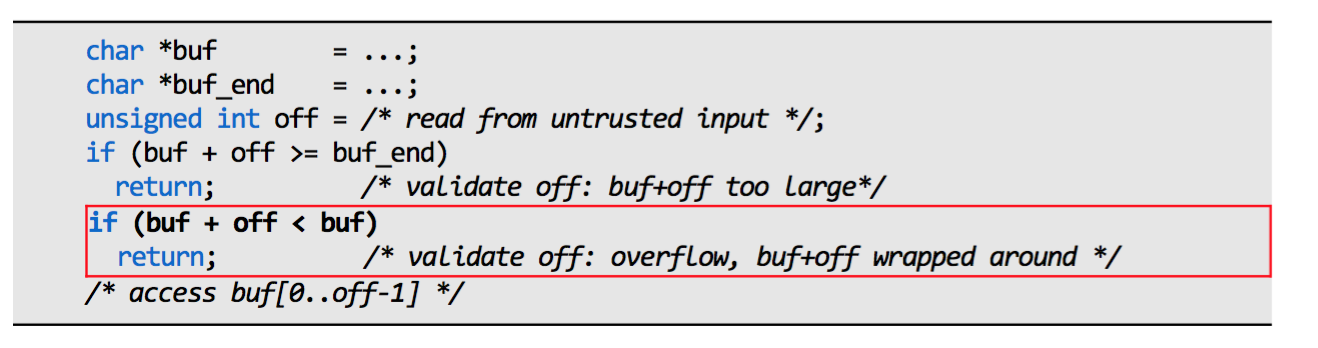
STACK

重点：什么是undefined behaviour? and unstable code? 如何检测？

Undefined behaviour: 是一些程序错误，如空指针解引用，缓冲区溢出、use after free等。

类似于这种undefined behavior在C编译器中还有很多：

* Pointer overflow: if ( p + 100 < p)
* Signed integer overflow: if (x + 100 < x)
* Oversized shift: if (!(1 << x))
* Null pointer dereference: \*p; if (p)
* Absolute value overflow: if (abs(x) < 0)



上面的代码中，buf + off < buf 就是一个undefined behavior：当off非常大的时候，会导致溢出，但gcc会认为buf + off总是小于buf\_end的。所以当溢出的时候，gcc检测不到，会绕过这一段溢出检测代码，然后去读取buf段以外的内存。

**Unstable code**: 由于被定义为是undefined behavior，从而被编译器略过的代码。

本文对12种C/C++编译器做了测试，发现：

1. 有一些编译器会悄悄地把unstable code给移除掉；
2. 不同的编译器会有不同的编译规则，不同版本的编译器对同一段代码会有不同的处理方式。

因此，需要一个成体系的approach来解决以上问题。

本文的方法是：把unstable code标记出来。

C/C++ source 🡪 LLVM IR 🡪 STACK 🡪 warnings

Assumption Δ: 程序员不会制造出undefined behavior

用公式表示Assumption Δ:

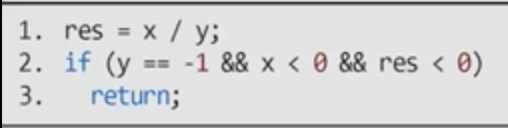
Reach(e): when to reach and execute code fragment e

Undef(e): when to trigger undefined behavior at e

../../../Library/Containers/com.tencent.qq/Data/Library/Application%20Support/QQ/Users/2207273547/QQ/Temp.db/95367528-B3D6-4021-B2BB-BF1B561AEFB7.png

STACK会在假设Δ被允许和不允许的情况下分别模拟出编译器。第一步先模拟假设不成立的情况；第二步模拟假设成立的情况；第三步就是查看前两步的执行结果有没有区别，有区别的地方就是unstable code。

如何检测unstable code？举个栗子：



Assumption Δ：y != 0, 且y != -1 or x != INT\_MIN

第一步：两个假设都不成立，意味着编译器会认为y可能会有等于0的情况，也可能会有y等于-1或x等于INT\_MIN的情况。这样使得有可能会有res < 0的情况；

第二步：两个假设都成立，编译器认为y不会有等于0的情况，也不会有y等于-1或x等于INT\_MIN的情况。这样的话永远都是res > 0；

第三步：比较前两步的结果，发现res有可能大于0也有可能小于0。因此res<0会被认为是unstable code。

STACK的这种检测方式的两个limitation：

如果执行第二步时得不到准确的结果，那么会漏报一些unstable code；

如果执行第一步时得不到准确结果，就会产生误报(false warning / false positive)。

如果避免unstable code：

对于程序员来说，通过fix bug和disable掉一些会被编译器discard的代码；

对于编译器来说，可以集成一些现有的bug-finding的工具，或者利用STACK的方式来判定unstable code；

对于编程语言设计者来说，可以通过修改语言的spec来定义更多的unstable code。

NFV-click

重点：为什么要提出这个技术，与传统的区别， 什么是NVF跟click router区别push和pull的技术-考试出画图-调度dropping policy

NFV:Network Function Virtualization Network Functions(Middleboxes): Firewall, IDS, DNS。网络功能虚拟化就是虚拟化这些网络功能，也就是通过软件模拟实现这些网络功能。 或者说是：运行在云设施的网络服务。 NFV目标：（1）省钱：使用更便宜的商业服务器来实现网络功能；减少专有硬件的使用，从而降低能耗，并且能够方便地维护。（2）赚钱：加速网络服务部署； 网络基础设施作为服务；

Click与传统的区别：（1）灵活:很容易添加新的功能 （2）模块化：通过组合元素来实现功能； （3）Open：允许开发者添加元素（Elements）;（4）效率：与硬件性能相差不大。

COW(copy on wirte)：写入时复制（Copy-on-write，COW）是一种计算机程序设计领域的优化策略。其核心思想是，如果有多个调用者（callers）同时要求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。这过程对其他的调用者都是透明的（transparently）。

本文的背景：

1) 路由器越来越不只是被用于转发包，对路由器有新的需求：NIDS(网络入侵侦测系统)、Random early detection dropping policy、Deep packet inspection、GFW等

。2) 现有的大部分路由器的设计都很静态、封闭、不灵活、贵。

Click的路由配置是一个有向图，每个节点被称为element，一个element代表路由处理的一个单元。Edge/connection是连接两个elements的，代表包传输的一条路径。每个element都是一个C++对象，每个connection都是一个指向element的指针，这样的设计使得每次包传输的开销只有一个虚拟函数调用。

每个路由的软件功能都封装在了element里面，包含了设备读写的设定、queueing、counting packets的行为，这些行为都可以由用户来决定。

Click提供了两种connection的方式：pull/push。Push：Push是从源Element到下游的元素，通过事件触发，比如有包到达。在Push连接方式里，上游的元素会提交一个包到下游的元素，当某个元素收到了一个push过来的包，它必须在存储、放弃和转发中做一个选择。Pull：Pull是从目的地元素到上游元素。在Pull的连接方式里，下游element要求上游element返回一个包。当输出接口准备好发包时，会开始一个pull过程：该过程会一直向后（上游节点）遍历图直到遇到某个元素‘吐出’一个包。

Click的element本身的输入/输出端口不含队列，包的存储是由一个独立的element（Queue）来实现的。这个queue element可以对包进行排队，因此可以用来实现调度。

Router的配置规则：1) push output必须与push input相连，pull同理：2) 每个agnostic端口都只能被一种操作(push/pull)独占；3) 如果一个元素的input和output都是agnostic的，那么两端必须都是pull或push；4) 每个push和pull端口都只能被连接一次。

调度：调度器就是一个有着多个输入端口，一个输出端口的**pull** element，并通过一定的调度策略来决定从这多个input进来的包应该如何共享这个Output。

调度器会选择一个input进行pull，如果没有返回包的话再选择下一个input进行pull。两种调度策略：Round-robin和PrioSched。

Dropping policy：Queue element实现了一个简单的丢弃方针，如果队列长度超过某个值，队列中所有包都被丢弃。另一种丢弃方针叫RED，有一种独立存在的RED element（只包含drop decision code）来通过flow-based router context来检测它下游的队列们的长度加起来是不是超过了某个值。当RED element被放置在Queue后面时，那么它会去看它上游的队列的长度。

Click的limitation：？4.6 看不懂