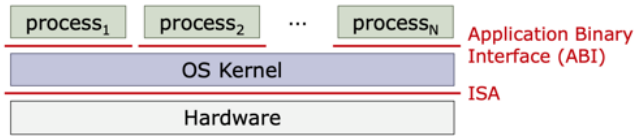
目录

System virtualization

Operating system (OS) and its Goals

- **Resource Management**
  - OS controls how processes share hardware (CPU, memory, storage, network, etc.)
- **Abstraction**
  - Hide underline details
  - Provide usable interfaces
- **Protection and Privacy**
  - Process cannot access other process data



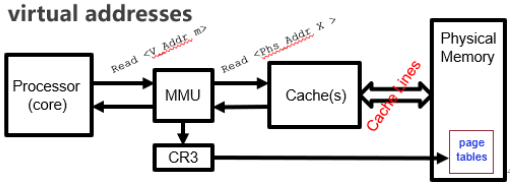
资源管理，向下管理各种硬件资源
向上抽象，让上层以为自己独占资源，可以正常使用资源。
保护、隐私。不能接触到其他进程的数据。

操作系统与硬件之间的接口是 ISA。
操作系统与进程之间的接口是 ABI（系统调用）。

进程抽象
1. 每个进程有自己的私有地址空间
2. 操作系统调度进程占用 CPU（分时）
3. 进程通过系统调用来使用 OS 的服务

Memory Management Unit (MMU)

- **OS configures page tables for processes**
- **Install a page table by modifying CR3 register**
- **MMU checks a page table when translating virtual addresses**



模式转换的方式：

Three Types of Mode Switches

- **System Calls**
  - Process requests a system service, e.g., exit
  - Like a function call, but outside the process
  - Do not have the address of the system function to call
- **Exceptions**
  - Internal synchronous event in process triggers context switch
  - Protection violation (segmentation fault), Divide by zero, ...
- **Interrupts**
  - External asynchronous event triggers context switch
  - Timer, I/O device
  - Independent of user process

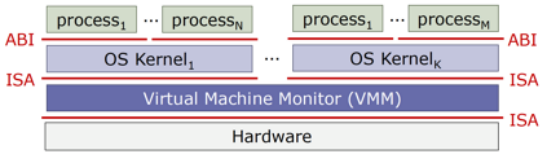
OS 提供的系统隔离
进程之间的隔离

技术：page tables，context switch，file abstraction
进程和内核之间的隔离
CPU 特权级

VMM

Virtual Machine Monitor

- **A VMM (aka Hypervisor) provides a system virtual machine to each OS**
- **VMM can run directly on hardware (type-1) or on another OS (type 2)**
- **Hardware virtualization**
  - VT-x: root and non-root mode
  - EPT
  - IOMMU and SR-IOV

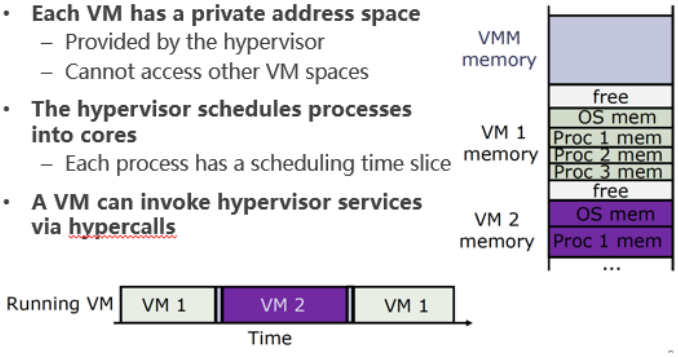


两种 VMM
Type1：直接运行在硬件上
Type2：运行在已有的操作系统上
Type2 可以只关注虚拟化的部分，其他例如设备驱动的功能可以复用 OS。

和 OS 对进程的抽象类似，VM 抽象：

Virtual Machine (VM) Abstraction

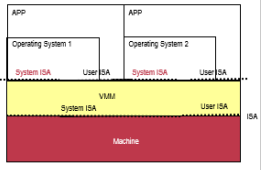
- **Each VM has a private address space**
  - Provided by the hypervisor
  - Cannot access other VM spaces
- **The hypervisor schedules processes into cores**
  - Each process has a scheduling time slice
- **A VM can invoke hypervisor services via hypercalls**



System ISA：可以访问敏感资源的 ISA

System ISA

- **Access Sensitive Registers**
  - CR0, CR3, CR4...
- **Control CPU**
  - Example: HLT
- **Control virt/phy memory**
  - Configure & Install PT
- **Control Devices**
  - DMA, Interrupts

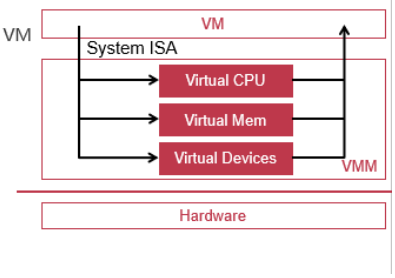


CR3：进程对应的页表地址。

系统虚拟化的步骤：

Procedure of System Virtualization

- **Step 1**
  - Catch all system ISA of a VM
- **Step 2**
  - Provide three functions
    - Virtualize CPU
    - Virtualize Memory
    - Virtualize I/O
- **Step 3**
  - Resume VM



首先要捕获 VM 的**所有的**系统 ISA
识别并提供相关的服务。
回到虚拟机中继续执行。

CPU 虚拟化

如果简单把 OS 当作普通的 Application，在 OS 上去跑，问题是客户机 OS 要执行特权指令，但是其所在特权级不能执行特权指令。

Solution: Trap & Emulate

- **Trap: running privilege instructions will trap to the VMM**
- **Emulate: those instructions are implemented as functions in the VMM**

问题是，架构并非是“严格可虚拟化”的。

Problems of Trap & Emulate

- **Not all architectures are "strictly virtualizable"**
- **An ISA is strictly virtualizable if, when executed in a lesser privileged mode:**
  - All instructions that access privileged state trap
  - All instructions either trap or execute identically

X86 结构中，一些特权级指令，在用户特权级下运行时，不会正常生效，只是静默地跳过去，什么都不发生，不会 trap。

解决方法：

How to Deal with the 17 Instructions?

1. **Instruction Interpretation: emulate them by software**
2. **Binary translation: translate them to other instructions**
3. **Para-virtualization: replace them in the source code**
4. **New hardware: change the CPU to fix the behavior**

Sol-1：指令翻译

Sol-1: Instruction Interpretation

- **Emulate Fetch/Decode/Execute pipeline in software**
  - Emulate all the system status using memory
    - E.g., using an array `GPR[8]` for general purpose registers
  - None guest instruction executes directly on hardware
- **E.g., Bochs**

把每一条指令都用软件模拟执行。
最大的问题就是慢。

Sol-2 二进制翻译

Sol-2: Binary Translator

- **Translate before execution**
  - Translation unit is basic block (why?)
  - Each basic block -> code cache
  - Translate the 17 instructions to function calls
    - Implemented by the VMM
- **E.g., VMware, Qemu**

在执行之前提前翻译好。
翻译的单元是基础块。
把 17 个指令翻译成 call

Sol-3 半虚拟化

Sol-3: Para-virtualization

- **Modify OS and let it cooperate with the VMM**
  - Change sensitive instructions to calls to the VMM
    - Also known as **hypercall**
  - Hypercall can be seen as trap
- **E.g., Xen**
  - **Was** widely used by industry like Amazon's EC2

改操作系统，把敏感指令换成 hypercall。
（具有可行性是因为要改的敏感指令少，常用的操作系统少）

Sol-4 硬件虚拟化

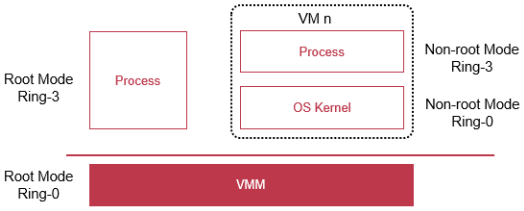
把 CPU 特权级进一步细化。
在 4 个特权级之外的维度，加了 root 和 non-root 的模式。

Sol-4: Hardware Supported CPU Virtualization

- **VMX root operation:**
  - Full privileged, intended for Virtual Machine Monitor
- **VMX non-root operation:**
  - Not fully privileged, intended for guest software

Both forms of operation support all four privilege levels from 0 to 3

Sol-4: Hardware Supported CPU Virtualization



VMCS：VM control structure

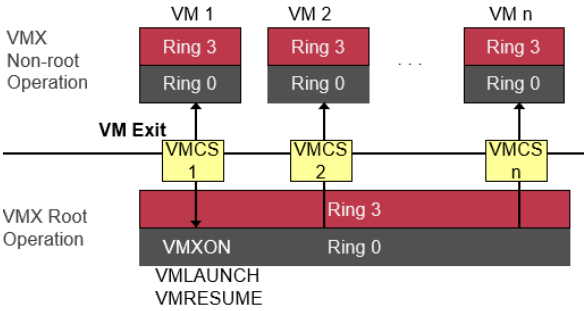
VT-x VMCS

The VMCS consists of six logical groups:

- **Guest-state area:** Processor state saved into the guest-state area on VM exits and loaded on VM entries.
- **Host-state area:** Processor state loaded from the host-state area on VM exits.
- **VM-execution control fields:** Fields controlling processor operation in VMX non-root operation.
- **VM-exit control fields:** Fields that control VM exits.
- **VM-entry control fields:** Fields that control VM entries.
- **VM-exit information fields:** Read-only fields to receive information on VM exits describing the cause and the nature of the VM exit.

VT-x 工作流程

VT-x Workflow



首先执行 VMXON 指令，告诉 CPU 要使用虚拟化了。
执行 VMLAUNCH，就会进入虚拟机中执行。
遇到特权指令后会 VM Exit，回到 VMM，执行完之后执行 VMRESUME，再回到 VM。
每个虚拟机有一个 VMCS，告诉有哪些指令需要下陷。

内存虚拟化

扩展了内存地址种类：

Terminology: 3 types of address now

- **GVA->GPA->HPA** (Guest virtual. Guest physical. Host physical)
- Guest VM's page table contains GPA

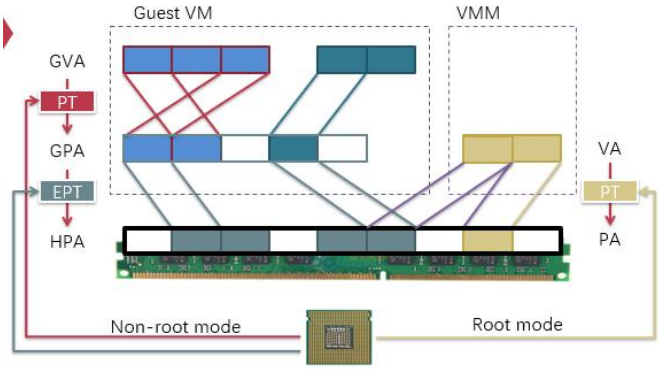
解决方法：

- Traditional solutions: **shadow paging & direct pa**
- Today's solution: **new hardware**

扩展页表

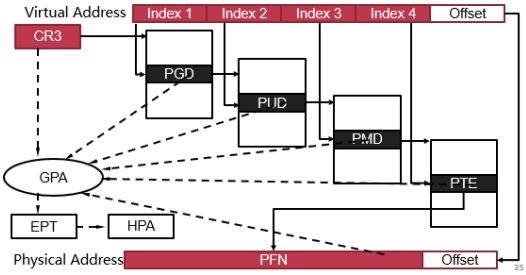
- **Hardware implementation**
  - Intel's EPT (Extended Page Table)
  - AMD's NPT (Nested Page Table)
- **Another table**
  - EPT for translation from **GPA to HPA**
  - EPT is controlled by the hypervisor
  - EPT is per-VM

EPT 用来把 GPA 翻译成 HPA



原本的地址翻译过程：

Any GPA is translated to HPA



5 次内存访问。
EPT 翻译与此类似。
意味着需要 20 次（4*5）内存访问。

IO 虚拟化

I/O Virtualization

- **Goal**
  - Multiplexing device to guest VMs
- **Challenges**
  - Each guest OS has its own driver
  - How can one device be controlled by multiple drivers?
  - What if one guest OS tries to format its disk?

四个方案：

Solutions for I/O Virtualization

1. **Direct access: VM owns a device exclusively**
2. **Device emulation: VMM emulates device in software**
3. **Para-virtualized: split the drivers to guest and host**
4. **Hardware assisted: self-virtualization device**

Sol-1 直通独占

问题：虚拟机可以通过 DMA 让设备访问到其他虚拟机的内存

解决方法：设备页表，IOMMU

IOMMU: Page Tables for Devices

- **Allow guest OS direct access to underlying device**
  - Guest just reuses its own device driver
- **Q: What if a VM asks device to access memory of other VMs?**
  - It is possible because device accesses HPA in DMA
  - Thus a device can access any memory
- **Solution: Page tables for devices**
  - Another MMU: IOMMU for devices
  - Q: what addresses will IOMMU translate?

优点：快，简化 VMM

缺点：guest 直接操作硬件接口（难以做 VM 迁移）（不走 VMM）。需要更多设备。

Direct Access Device Virtualization

- **Positives**
  - Fast, since the VM uses device just as native machine
  - Simplify monitor: limited device drivers needed
- **Negatives**
  - Hardware interface visible to guest (bad for migration)
  - Interposition is hard by definition (no way to trap & emulate)
  - Now you need much more devices! (image 100 VMs)

Sol-2 模拟设备

用软件模拟设备逻辑

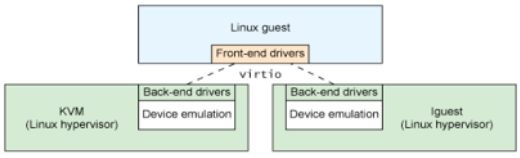
Emulated Devices

- **Positives**
  - Platform stability (good for migration)
  - Allows interposition
  - No special hardware support is needed
- **Negatives**
  - Can be slow (it’ s software emulated)

Sol-3 半虚拟化

Sol-3: Para-Virtualized Devices

- **VMM offers new types of device**
  - The guest OS will run a new driver (front-end driver)
  - The VMM will run a back-end driver for each front-end
  - The VMM will finally run device driver to drive the device



VirtIO Architecture by KVM

两边互相知道，可以更好的配合，比如数据可以在前端 batch 之后再给后端。

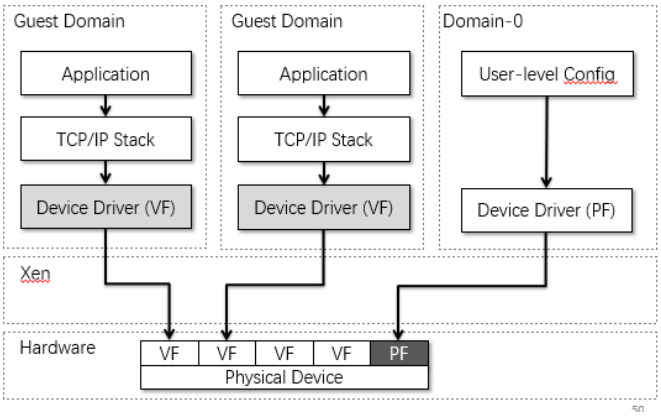
因此快。

Sol-4 硬件虚拟化

Sol-4: Hardware Support for I/O Virtualization

- **VMM**
  - An SR-IOV-capable device can be configured to appear in the PCI configuration space as multiple functions
- **VM**
  - The VMM assigns one or more VFs to a VM by mapping the actual configuration space of the VFs to the configuration space presented to the virtual machine by the VMM

SR-IOV：硬件的一种虚拟化标准



PF 具有管理功能，被 VMM 控制。

总结表格：

Virtualization Technologies

Virtualization	Software Solution	Hardware Solution

CPU	• Trap & Emulate
	• Instruction interpretation
	• Binary translation
		• VT-x
		  • Root / non-root mode
		  • VMCS

	• Para-virtualization: Replace 17 instructions	
Memory	• Shadow page table • Separating page tables for U/K	• EPT
	• Para-virtualization: Direct paging	
Device	• Direct I/O • Device emulation	• IOMMU • SR-IOV
	• Para-virtualization: Front-end & back-end driver (e.g., virtio)	

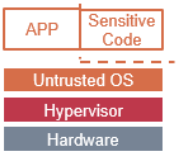
52

Topic：Isolation Via VMM

SeCage [CCS '15]: Isolate Sensitive Code

Problem

Private data leakage caused by memory vulnerabilities. Example: HeartBleed

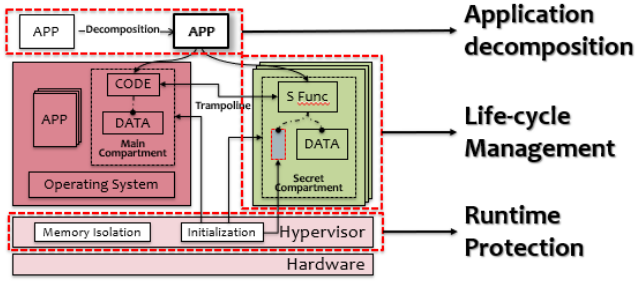


Solution

- Isolation Domain: Put sensitive code via an isolated VM
- Support different applications, including OpenSSL
- Utilize hardware features to switch between the application and its isolation domain, causing only 8% overhead

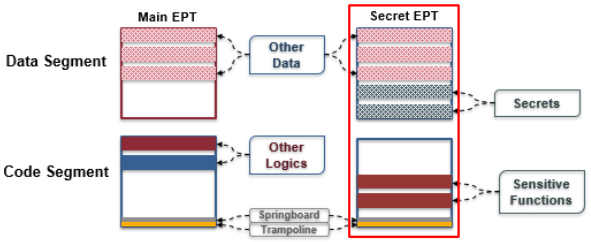
62

SeCage [CCS '15]: Isolate Sensitive Code



秘密数据部分摘出来，为其单独分配一个 EPT，使得 OS 不能直接访问。

SeCage: Different EPTs for two Parts



EPT Isolation: **Data Segment** is removed from main EPT; **Code Segment** is only mapped in Secret EPT

64

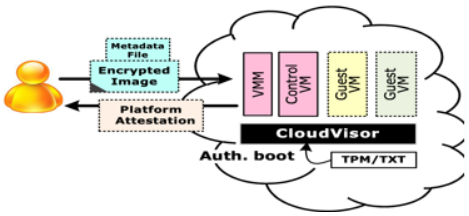
缺点：如果秘密数据很多，被频繁用，那么每次访问需要到 VMM 里换页表，很慢。

解决方案：intel 加了一个指令 VMFUNC，可以在 VM 里换页表，不需要 VM Exit

CloudVisor：在恶意的 hypervisor 情况下保护 VM

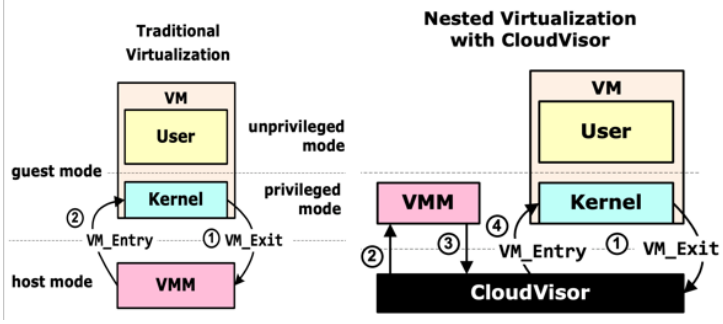
Key Idea

- **Separating security protection from VM hosting**
- **Add another layer of indirection**
  - Nested virtualization



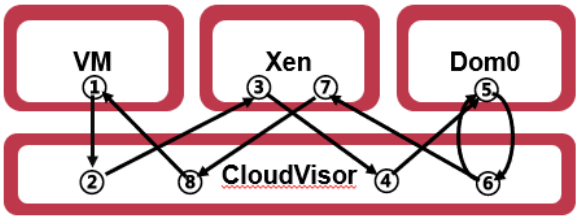
管理和安全分离。VMM 被降权，CloudVisor 在最底层，起保护作用。

General Workflow



开销过大，慢

The Cost of Protection: Excessive VM Exits

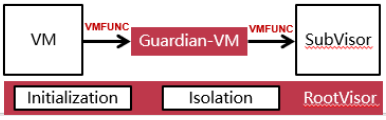


Operation	Times
Hypercall	>= 2X
EPT Violation	2 – 6 X
DMA Operation	>= 2X

改进 CloudVisor-D

Architecture of CloudVisor-D

- **A tiny nested hypervisor in root mode**
- **A Guardian-VM for each VM in non-root mode**
- **Most VM ops offloaded to Guardian-VM**
  - Hypercalls
  - Memory virtualization
  - I/O operations



Guardian-VM 在 non-root mode，承担保护安全的作用。用 VMFUNC 切换页表。

数据流控制流分离

Language Virtual Machines

Native 语言，比如 C++，编译、部署。

便利性问题：在多个平台下，多个 ISA，需要每个平台都有对应的编译器，编译出对应的 binary。部署环境越来越复杂，不同的操作系统，不同的硬件平台。
安全性问题：

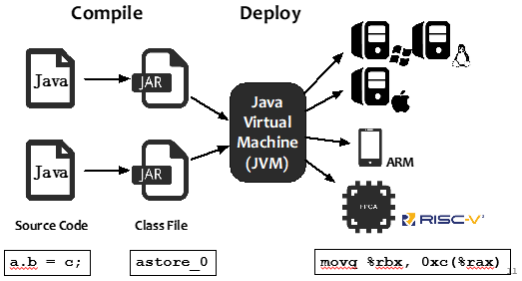
- **Out-of-bound access**

```
char * chs = malloc(10);
Char ch = chs[10]; // oops!
```

- **Use-after-free**

```
free(chs);
Char ch = chs[5]; // oops!
```

解决思路：加一层抽象



一次编译，处处执行。
安全问题的解决：

Security Enhancement

- **Out-of-bound access ← runtime check**

```
char * chs = malloc(10);
Char ch = chs[10]; // ArrayOutOfBoundException
```

- **Use-after-free ← no free at all!**

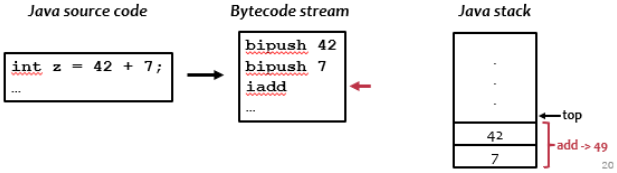
```
free(chs);
Char ch = chs[5]; // No problem!
```

托管给 JVM 运行，JVM 负责运行时检查、内存管理的工作。

执行模式：解释执行
在 Java 栈上执行。

Stack-based Interpretation

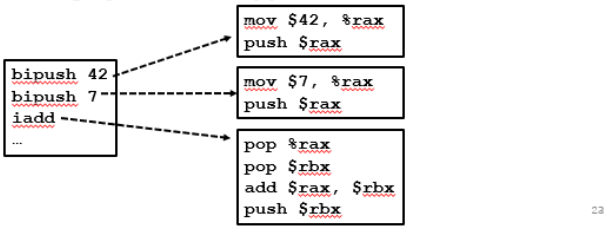
- **All instructions (bytecode) are operated on the stack**



问题：efficiency

Problem: Efficiency

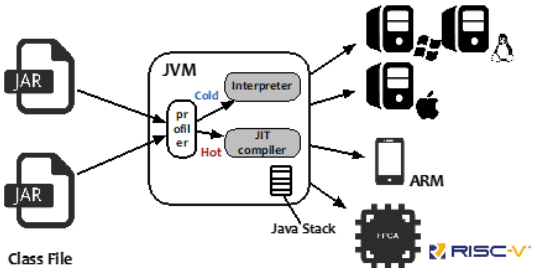
- **Too many memory operations (>=1 per bytecode)**
  - Low utilization of registers
- **Missing optimization opportunities**



内存操作太多，对寄存器的利用率低，难以优化。
解决方法：JIT compiler

Solution: JIT (Just-In-Time) Compiler

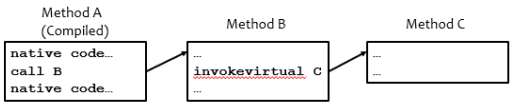
- **Hot code will be compiled and executed**
  - "Hot" means that it has been executed for times



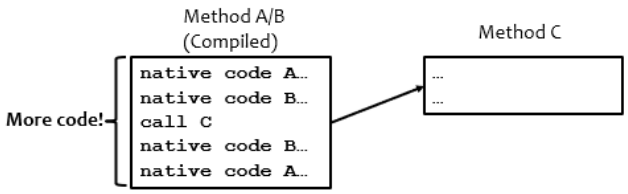
并不是所有的代码都编译，有一个 profiler 进行统计，偶尔执行的就直接解释执行，经常执行的就进行编译。

JIT：激进 or 保守

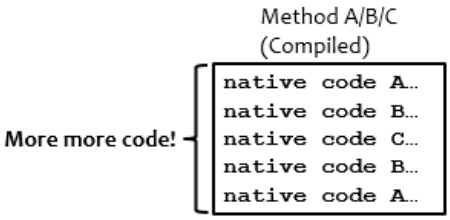
- **Choice 1: only compile A**



- **Choice 2: inline B**

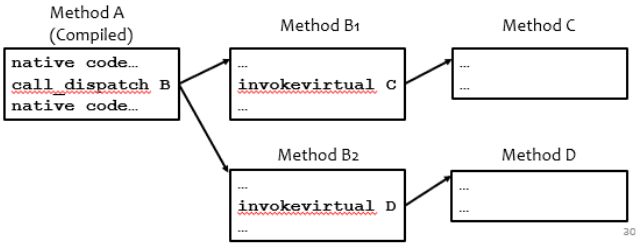


- **Choice 3: inline all**

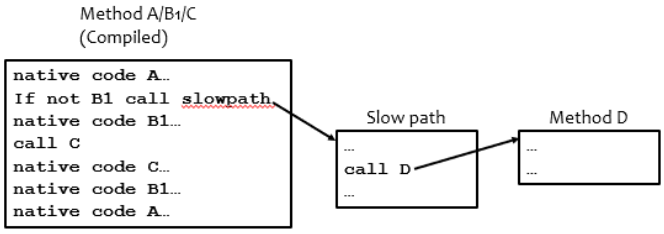


在多态的情况下

- **Choice 1: compile A only**
  - Suitable for all cases where all methods are hot



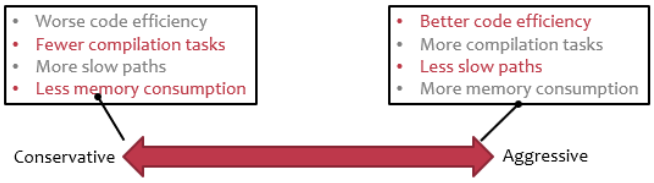
- **Choice 2: compile only for one path**
  - Suitable for cases where only one path is hot



激进和保守的权衡：
代码执行效率和运行时编译开销的权衡

JIT: Aggressive Or Conservative?

- **Summary: A tradeoff between code efficiency and runtime overhead**

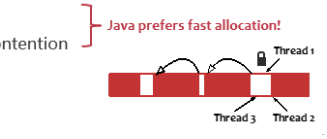


内存管理
数据放在 Java Heap 里

Free Memory Management
GC：识别死对象；管理堆空间。
堆的组织-1：free list
Free 的块组织成链表

在 Java 中不受欢迎

- **Not welcomed in Java**
  - Fragmentations
  - Multi-threaded contention



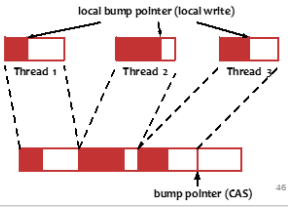
堆的组织-2：连续空间

Heap Layout 2: Contiguous Space

- **Free space is always contiguous**
  - A bump pointer to mark how much has been used
  - Allocation: Lock-free atomic instructions (CAS)

为了维持连续空间，需要在垃圾回收之后把 live object 压在一起。
小优化：local heap
每个线程有一个本地堆，省掉 CAS 开销。

- **Even faster allocation: local heap**
  - Allocate a large portion with CAS
  - Then allocate locally



GC 算法-1：引用计数
记录有多少指向该对象的引用。引用计数变为 0 时就进行回收。
在 JAVA 中不受欢迎的原因：性能差。和 free list 有较强耦合。不能处理**循环引用**。

GC 算法-2：tracing 追踪
通过图遍历的方法把所有的活对象都找到。
从“root”开始遍历。
Root 包含：线程栈上的对象。全局对象，包含每个类的静态变量等。
可以解决循环引用问题。

Tracing GC 应用比较广泛。

Tracing GC is Popular

- **Can be integrated with different layout**
  - Free-list: Mark-Sweep (Boehm GC)
  - Contiguous: Mark-Copy (PS, G1, Shenandoah...)
- **Can also be used for different purposes**
  - Throughput-oriented: Stop-the-world tracing
  - Latency-oriented: concurrent tracing

吞吐量优先 or 延迟优先 ？

Throughput-oriented GC

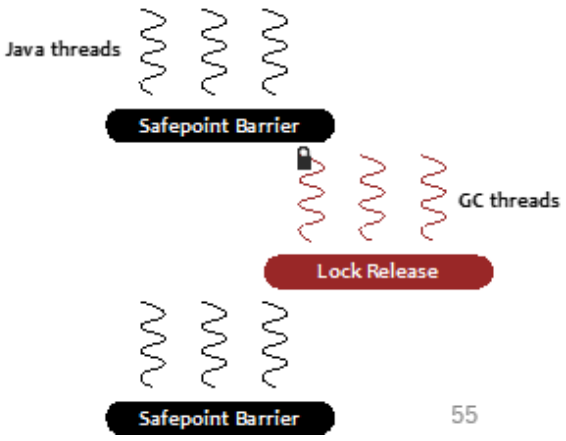
- **Design goal: high GC throughput**
  - Stop-the-world: Java threads must be paused during GC
  - Task-based parallelism: dividing collections into tasks
- **Consider latency together with throughput**
  - Generational

- 在 GC 期间暂停线程的运行，以提高吞吐量。
- 把垃圾回收拆分成小的任务，并行处理。
- 分代。

Stop the world：

JVM leverages *safepoint* to pause all Java threads

- Java threads queue up for a lock held by GC threads



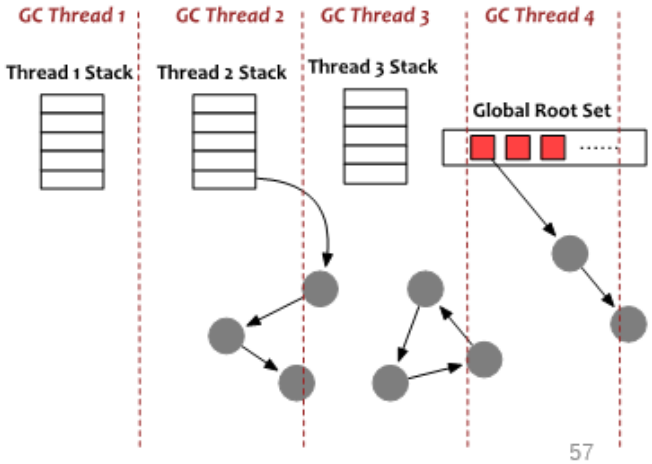
Aiming at high GC throughput

- CPUs are monopolized by GC threads
- No coordination between GC and Java threads

GC 线程可以充分利用 CPU;
无需考虑 GC 对 Java 线程的影响。

基于任务的并行

遍历过程：可以通过划分 roots 来分成多个任务，多个 GC 线程并行执行。



问题：根据起点分配，可能会造成负载不均衡。
可以通过 work-stealing 实现动态平衡。

考虑时延：

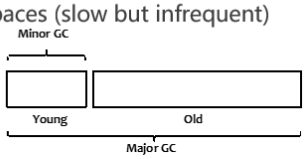
GC 暂停的时间和 live object 的大小正相关。当 live object 特别大时暂停的时间会很长。
因此不能等到完全没有空间了才回收。

Solution：分代

分为新生代、老生代。
刚分配的对象在新生代，存活时间长的对象在老生代。

Solution: Generational GC

- **Dividing Java heap into multiple spaces**
  - In PS it has two spaces: *young* and *old*
  - The size of young gen can be small and fixed
- **GC is also two-fold**
  - Minor GC: only collecting young-space (fast and frequent)
  - Major GC: collecting both spaces (slow but infrequent)



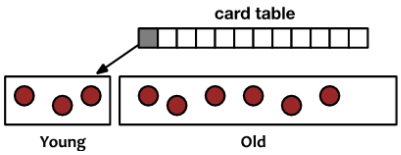
GC 也分为两个过程，对新生代区域的 GC 和对全局的 GC。
基于的假设：大多数对象死的很快。因此对新生代区域的 GC 效率很高。
但是，有些场景下“分代假设”不成立，比如大数据场景 spark Hadoop。对象可能都移到老生代里去了。
分代的问题：跨区的引用
我们只想对新生代区进行遍历、回收。但是有些引用是从老生代指向新生代的。
方法是把新生代中被老生代引用的对象标为 root。
那么如何识别有哪些是跨区的引用呢?

- 扫描整个老生代？（那还分个锤子代）
- 记住所有的引用（开销太大）

方法：card table

Solution: Card Table

- **Dividing old-space into many *regions***
  - Using 1 bit (card) in a table for each region
- **When a cross-space write happens, dirty the card**
  - Scanning dirty cards only during minor GC



是一个折中的办法。

面向时延的 GC

目标是更短的停顿，更小的延迟

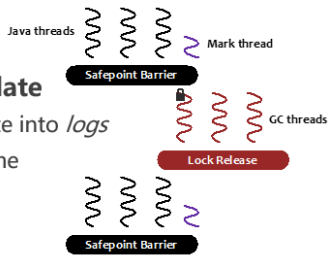
Design goal: shorter pauses/app latency

- Concurrent marking
  - Choose which parts are valuable for collection
- Concurrent collection
  - Collecting when application threads are active

并行标记

Java 线程和标记过程并行，处理并发更新的方法是 Java 线程会将更新内容写入 logs，标记线程会定期消费日志。

- **Handling concurrent update**
  - Java threads will write update into *logs*
  - Marking threads will consume the logs periodically

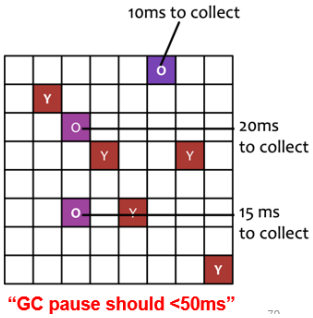


并发标记的好处：

- 标记不需要暂停 Java 线程
- 标记的结果可以指导 GC 线程进行更高效的垃圾回收。

An Example: Region-based Profiling

- **The whole heap is split into many *regions***
- **GC threads estimate per-region collection cost according to concurrent marking**
  - Meeting the “soft limits” from users



可以根据标记的结果预估每个区域垃圾回收的时间。以便满足用户对时延的要求。

并行回收

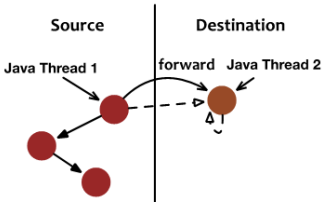
会使得暂停时间非常短。

要解决的主要问题：双拷贝，引用更新。

双拷贝问题：GC 时要进行拷贝，存在一个时间窗口使得两个拷贝同时存在，可能会使得不同的线程指向的是不同的拷贝，此时如果发生更新，会不一致。

解决：间接指针

- **Solution: indirect pointer**
  - Always point to the newest version
  - All read/writes will be forwarded to the newest one

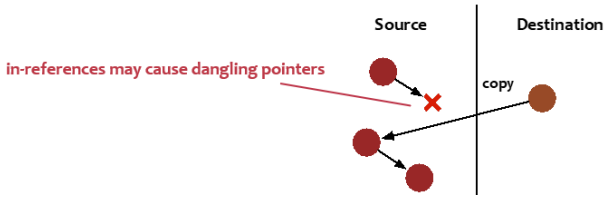


解决：双写，要解决双写的原子性问题。

引用更新问题：

In-Reference Updates

- **Only part of heap space is collected each time**
- **Only out-references are updated**



In-Reference Updates

- **Design 1: Stop-The-World update**
  - Large pause time (violating the goal of Shenandoah)
- **Design 2: memorizing all in-references**
  - Large memory overhead (all references are duplicated)
- **Design 3: lazily updated in next phases**
  - A next marking phase will scan the whole heap

较好的是 lazily update，在下一轮扫全栈的时候才真正回收，在此之前用**间接指针**占位。

总结：

Summary: Throughput vs. Latency

- **PSGC: Stop-The-World**
  - Throughput-oriented
  - Large GC pauses
- **G1GC: adjustable & partially-concurrent**
  - Concurrent marking
  - Controllable GC pauses
- **Shenandoah: mostly-concurrent**
  - Concurrent marking & collection
  - Ultra-low GC pauses
  - Hurting application throughput

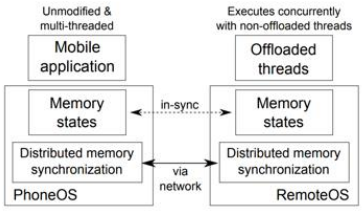
引入 JVM 这一层抽象的**好处**是可以捕捉更多的应用相关的语义。**坏处**是运行时的行为更加复杂（比如 GC）。基于这些好处和坏处，有以下一些工作。

12 年的工作：COMET

The Goal of COMET

- **Transparent offloading**
  - No programming effort, arbitrary apps
- **Fine-grained offloading**
  - Restricting transferred bytes
- **Robust offloading**
  - Resisting network failures

COMET builds a distributed shared memory (DSM) between phones and clouds

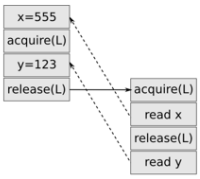


传统 DSM 是基于页的内存管理。

在 JVM 下的好处是可以做**细粒度**的 DSM。

Core Concept: Java Memory Model (JMM)

- **Dictates which writes a read can observe**
- **Specifies 'happens-before' partial order**
  - Accesses in single thread are totally ordered
  - Accesses in different threads are ordered by locks



94

单线程内部的读写操作是按顺序的。

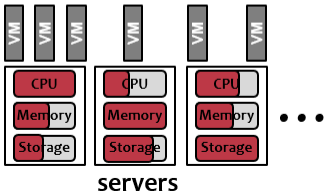
多个线程之间的顺序是锁来保证

- **Used to establish 'happens-before' relation**
- **Synchronizes**
  - Bytecode sources
  - Java thread context (all frames of all threads, including pc/registers/method)
  - Java heap (only dirty fields in tracked set)

关于 VM 带来的问题的工作：Semeru

Resource Disaggregation

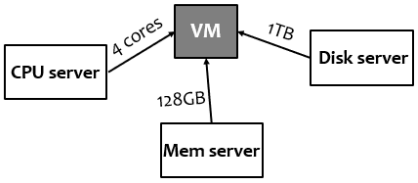
- **Traditional data center: monolithic server model**
  - Each server hosts all types of hardware resources
  - Inducing resource under-utilization



可能因为某一种资源用满了造成其他资源的浪费。

Resource Disaggregation

- **Disaggregated data center: the next generation?**
  - Each server(s) contains different types of resources
  - Better resource utilization

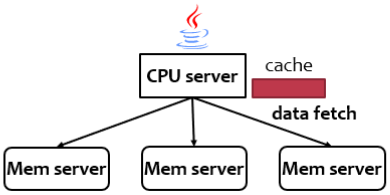


1

一种简单的模型，CPU server + memory server

A simplified model: CPU server + memory server

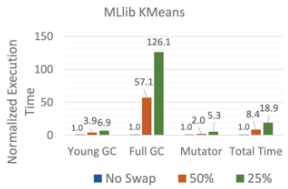
- Applications are running on CPU server
- Data are stored on memory servers
- CPU servers keep a local memory cache (so locality matters)



105

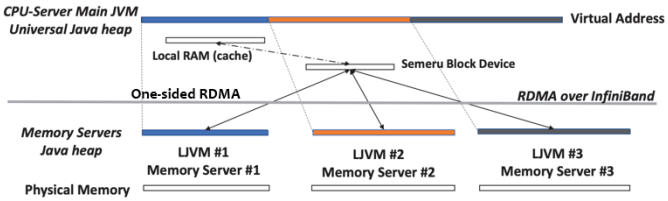
发现资源分离对 JVM 不友好。

- **Workload: *KMeans* in Spark**
  - One CPU server, two memory servers, G1GC
- **Result: 18.9X for 25% cached memory**
  - Full GC: 126.1X
  - Reason: too many data fetching



Semeru Overview

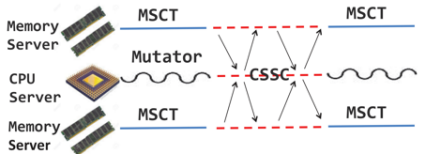
- **A unified Java heap among all servers**
  - Each memory server runs a LJVM



Key Insight: Close-Data GC

- **Let memory servers handle most GC tasks**
- **Now GC is divided into two sub-phases:**
  - Memory server concurrent tracing (MSCT)
  - CPU server STW collection (CSSC)

G1 mixed



108

思路：memory server 处理大多数 GC 工作

CPU server 把 root 发到 memory server

Conclusion

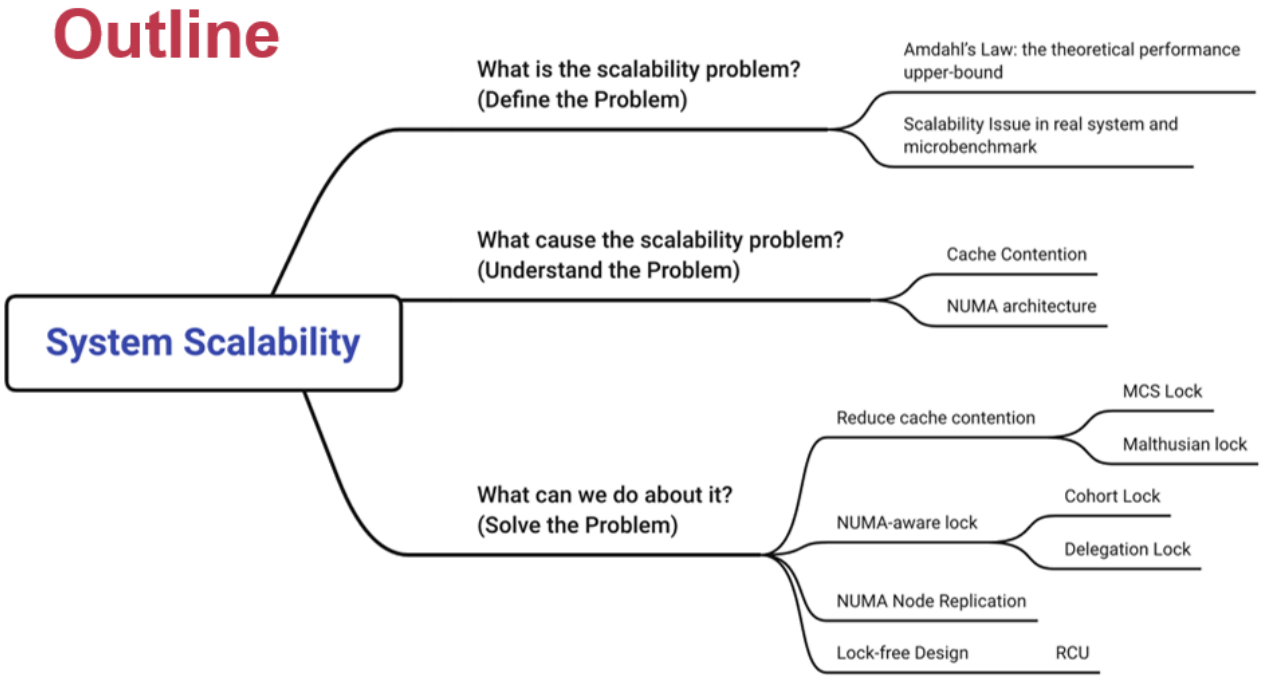
- **Language virtual machines: an "indirection" for portability and safety**
- **A typical example: JVM**
  - Code execution: interpreter & JIT compiler
  - Memory management (GC): basics & modern GC designs
  - JVM in systems: COMET & Semeru

Scalability & Serverless

- **粗体**是重要内容，非粗体大多是ppt废话

03 Scalability

- 大纲，越末尾越具体（越可以出题）



定义问题（什么是可拓展问题）

1. 单核和多核
   - 由于功耗墙的存在，单核性能有上限。突破上限需要引入多核
   - 如何完全利用多核是一个问题
2. **Amdahl's Law**
   - 计算多核情况下的加速比（与核心数和并行度有关）

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

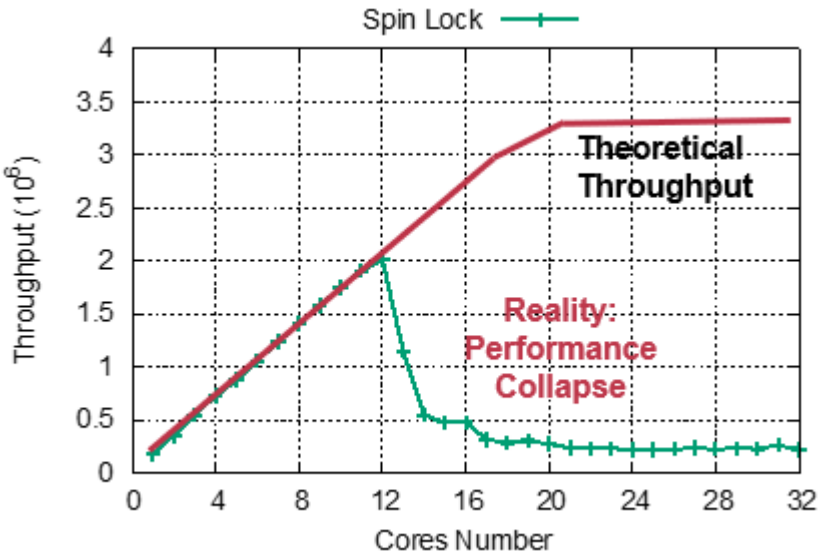
Speedup — Percentage that can parallel — Avaliable cores

The upper bound $\lim_{s \to \infty} S = \frac{1}{(1-p)}$

It depends on the program itself

- 我们需要逼近阿姆达定律的理论上限

3. 示例实验

- 在一个benchmark中体现可拓展性的瓶颈
- 为了保护共享数据，使用互斥锁（mutex lock）
  - 只有一个线程可以拿到互斥锁，执行critical section
- benchmark：一堆线程抢锁，读写共享数据
- 实验结果显示核心数超过一个阈值，throughput骤降
- 

缓存一致性导致低可拓展性

缓存一致性

1. CPU有三级缓存结构

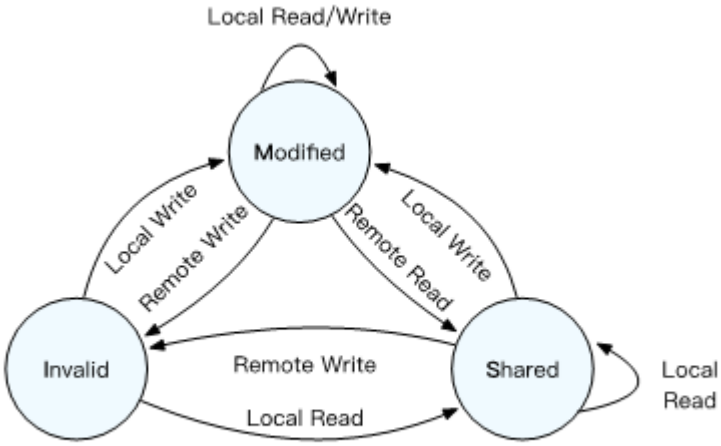
- L1缓存由一个核心独享，L2缓存由一组CPU共享，L3缓存由所有CPU共享。
- 一个数据在两个缓存中被读写，可能会导致不一致

2. Cache Coherence（缓存一致性）

- 目标是为了保证所有核心读到的数据是一致的
- 一些缓存一致的协议：Snoop(在总线上嗅探)/Directory-based(用目录记录变量的“主人”)

3. **MSI satate machine（一种cache coherence协议）**

- 定义了三种状态：Modified， Invalid， Shared
- 定义了多种操作：远程读写，本地读写
- 用状态机定义操作如何改变状态，以及各状态下合法的操作（如不能远程读写Invalid的cacheline）



- Cacheline State Machine

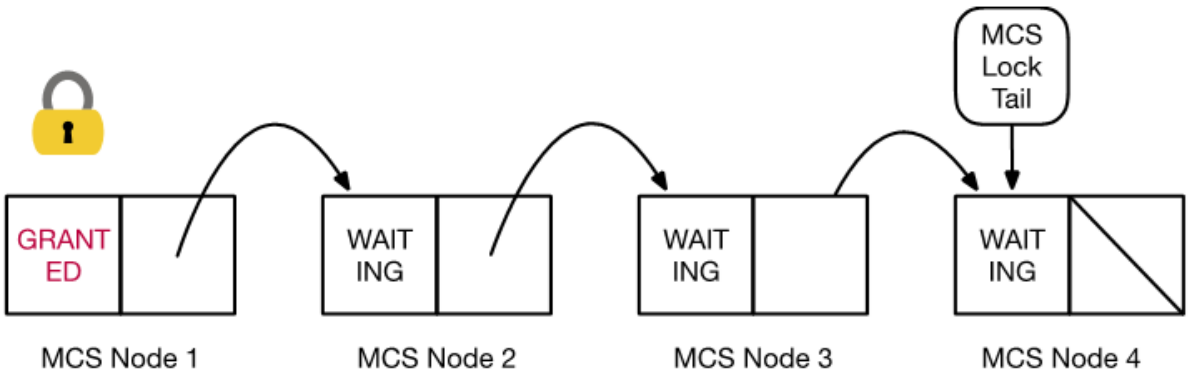
4. **Global Directory（一种MSI的实现）（Directory-based）**
    - 用一个目录记录每个变量属于哪个缓存（目录在内存或者别的硬件里），缓存中每个cacheline都有一个state
    - 当要写缓存时，变量在其他旧主人缓存中状态改成Invalid，自己设置成唯一的新主人，并把变量状态更新为Modified
    - PPT只举了一个例子，总的思路是M表示一个缓存独享该变量，S表示多个缓存共享变量，I表示这个缓存不能用要找别人读。拥有该变量最新值的缓存都是它的主人（M和S）。每次操作根据以上原则更新状态和目录

5. Cache Coherence和可拓展性的关系
    - 多个核争夺一把锁，导致锁的变量在各个缓存中跳来跳去
    - 在阿姆达定律中体现为p大幅降低（为了保持缓存一致导致大量开销）

解决方案：高可拓展性的锁

1. 简易实现：back_off
    - 轮询拿锁时每个循环sleep一段时间，避免多个CPU同时抢锁的情况
    - 没有断崖了，但是在低负载下性能变差（overhead高）。治标不治本

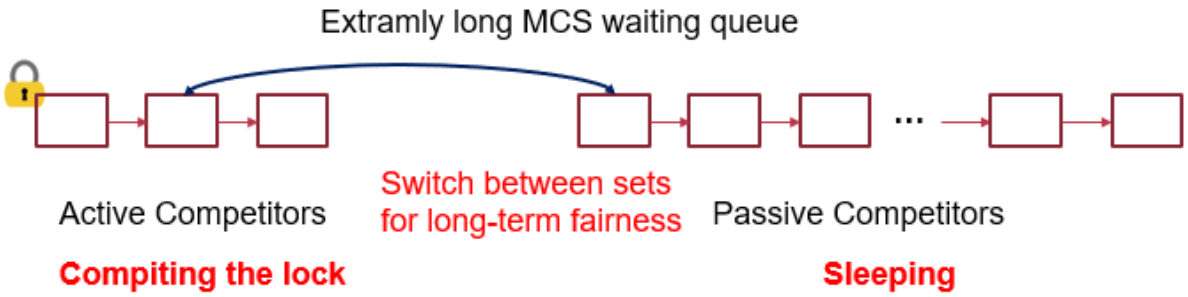
2. **MCS锁**
    - 把对单点的竞争改进为多点



    - 每一个拿锁的线程维护一个节点，排队的节点穿成一个链表。节点由两部分组成，一个表示锁当前状态，一个是指向排队的下一个锁的指针。

- 如何加入新节点：有一个公共尾指针指向排在最后的节点，把新节点状态设置成waiting，并把原尾节点的指针核公共尾指针指向新节点。
- 如何放锁：释放原节点，并根据指针找到排队的下一个节点，把该节点的状态更新为granted。
- 如何轮询：每个线程轮询自己节点的状态即可，锁不再跳来跳去
- 实验证明效果不错，但核心数量变非常多时性能依然有瓶颈，因为LLC(Last-level-cache，即L3缓存)大小有限

3. **Malthusian锁**

- 为了进一步提升MCS锁性能，减少缓存开销
- 

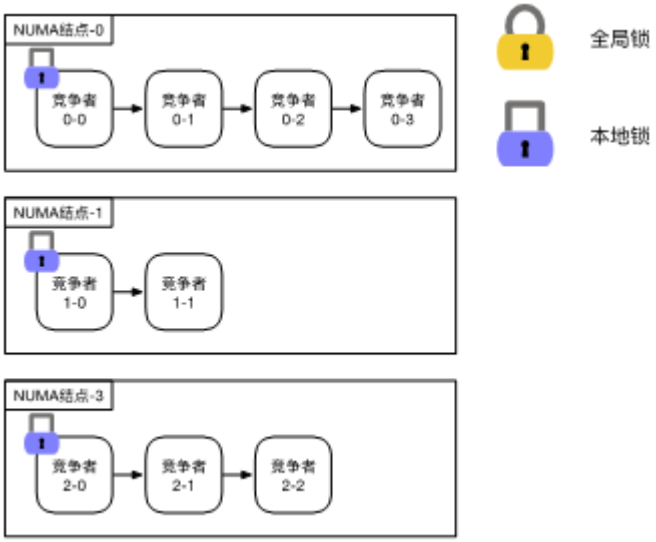
- 把队列头部几个节点设置成需要轮询的活跃节点，后面靠后的节点直接sleep等待唤醒（减少参与竞争的核心数）
- 实验证明可拓展性进一步增强（延迟也变高了）

面向NUMA架构的可拓展性

1. 新的问题

- MCS很快，但是NUMA上还是不行。NUMA架构的CPU是分开的，每组CPU有Local Memory。跨节点读取内存开销很大
- 跨节点共享锁和数据开销很大

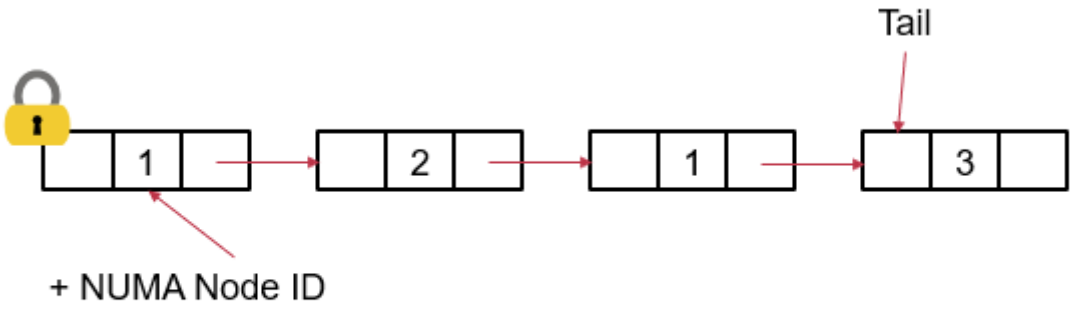
2. **面向NUMA架构的锁(Lock cohorting)**

- 设置一个全局锁和多个本地锁。当一个NUMA节点拿到全局锁，把资源拿到本地由本地的线程竞争（本地锁），用完后再放全局锁。
- 

- 减少了共享数据跨节点传输的次数，性能有所好转

3. **一个更聪明的Cohort锁**

- 即将被并入Linux 内核



MCS waiting list

- 节点记录所属的NUMA节点，放锁时跳过属于不同NUMA的节点，遍历完list找不到同节点的等候者则把锁交给其他节点。
- 进一步改进：放锁时遍历整个list太蠢了。等候者在等待时，更新等候队列的顺序，让同一NUMA下的节点串在一起，这样放锁时可以跳过一串属于其他NUMA的节点，更快地遍历整个list

代理执行

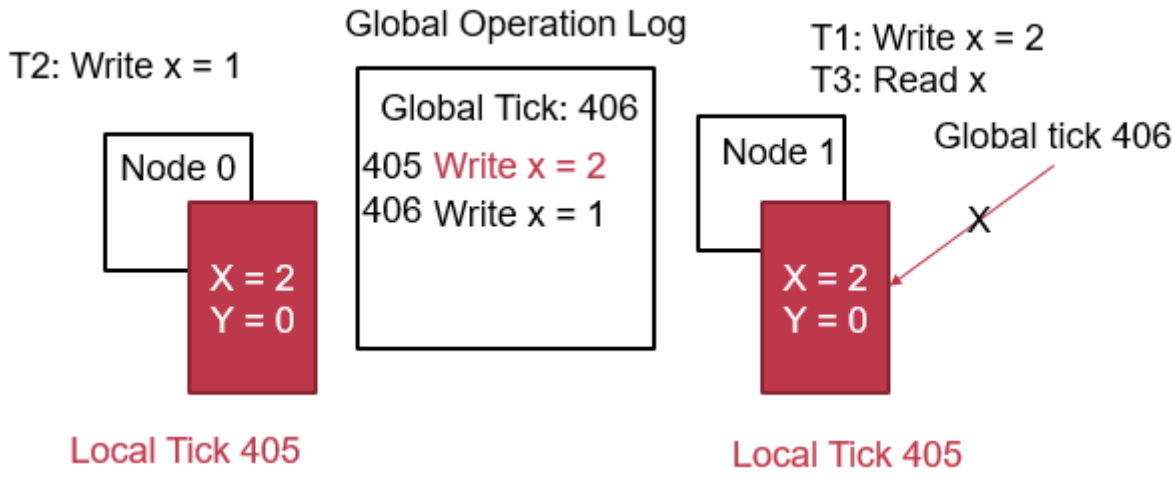
1. cohort锁的问题
   - cohort锁依然有性能断崖
2. **Delegation Lock**
   - 核心思想是把锁固定在一个核中（代理核），其他核通过访问请求的方式来轮询锁。其他锁把参数交给代理核，代理核执行本地函数并返回是否拿到锁。
   - 代理核可以是固定的也可以是动态的
   - 代理锁的性能远远超出cohort锁，在低竞争时overhead很大
   - 以上三种锁的对比

No one-size-fit all solution

	MCS/spinlock	Cohort Lock	Delegation Lock
Pros	Simple, efficient when contention is low	Better scalability when scaling to NUMA nodes	Best performance in NUMA systems
Cons	Bad scalability in NUMA system	Still has some performance penalty	Non-trivial code changes to application
Notes	Qspinlock is already used in Linux Kernel	A variant (CNA) is going to be used in Linux Kernel	Barely used after almost 10 year

Node Replication

1. **Node Replication**
   - 思路：把数据在每个NUMA节点中都存一个备份
   - 所有读操作都是本地读，写操作用一个全局log，用来更新每一个备份
   - 所有操作用tick记录，根据tick决定变量的最新值，本地有一个表记录所有变量的最新tick，以决定读变量时是否更新



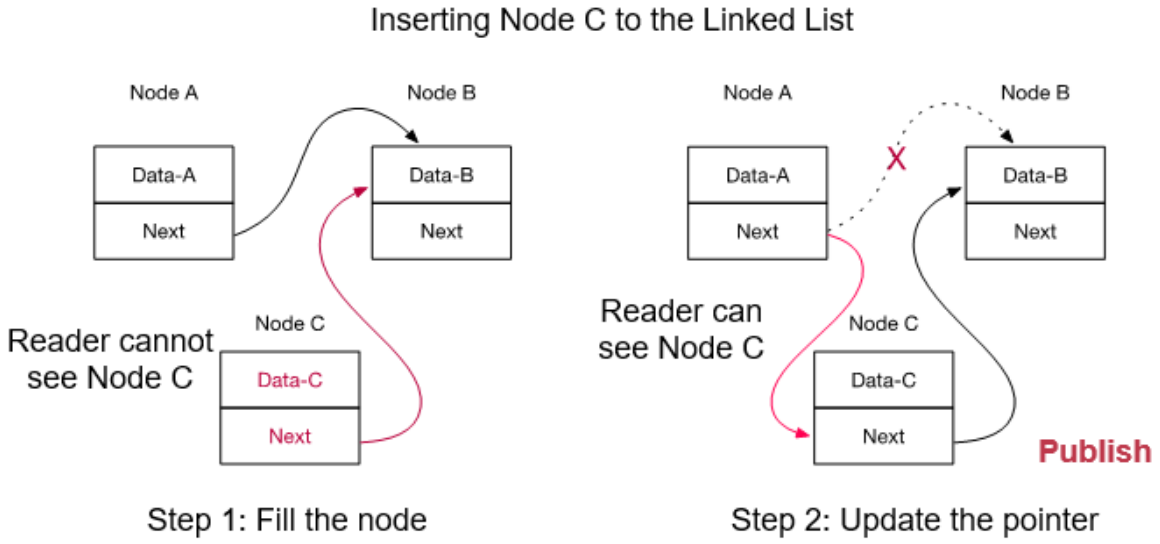
libnuma

1. libnuma
   - 显式地在别的numa节点或本地节点申请内存，由应用程序来维护内存位置

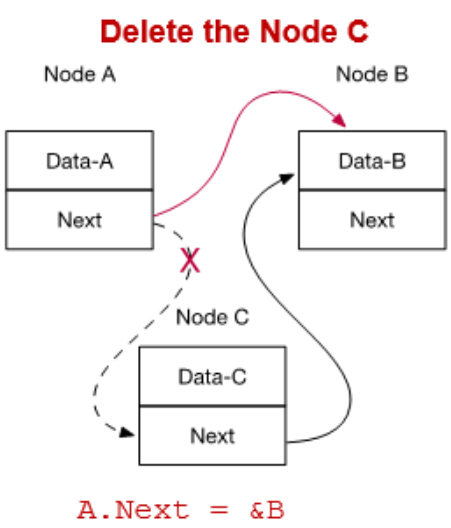
Lock-free design（RCU）

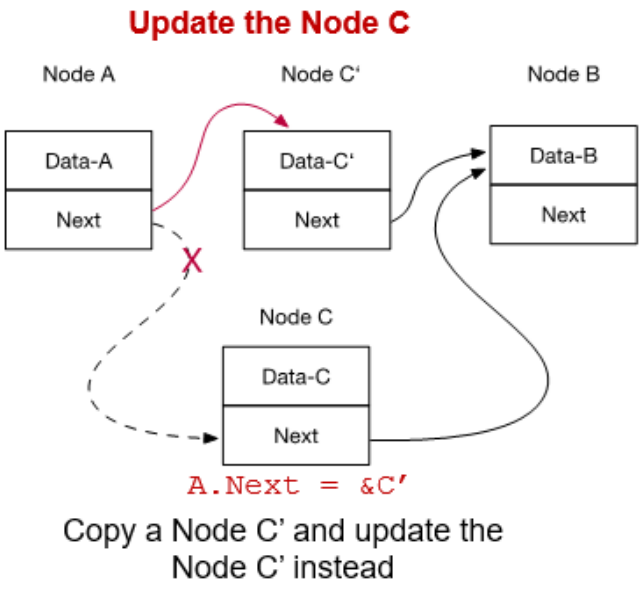
1. Read Copy Update（RCU）
   - 传统的读写锁会有拿锁开销，同时对数据读写会不可避免地阻塞住后来者
   - RCU抛弃锁，让读和写可以同时进行

2. 机制
   - 借助指针原子添加新的内容



   - 借助指针原子删除和更新内容





3. 监视读者

   - RCU中，当删除一个节点时，即使事先改变指针让该节点脱离原来的list，此时可能有读者在指针改变前进入了待删除节点。因此RCU需要监视读者的状态来决定是否释放掉被删除的节点。
   - 根据读者的读写时间判断。假设删除操作在时间T发生，那么只要还有在时刻T前开始读的读者没有结束，就不释放被删除节点。当所有T前开始读的读者结束，就可以释放掉被删除的节点了。

4. 性能

   - 读者读的行为本质是通过指针找到下一个节点。写者写的行为本质是更改节点，两者不冲突。
   - 观测读者和释放节点的工作可以异步地来做。
   - 缺点是RCU不好用，RCU无法在双向链表中实现。

Serverless

Serverless Overview

1. 传统云应用

   - 部署虚拟机，建立环境，编写代码并执行
   - 缺点：虚拟机难以拓展；部署环境很麻烦；运行单个虚拟机浪费钱（不是所有时间都跑满CPU）
   - 问题关键：VM这种抽象太底层，开发者需要花精力解决环境等细节问题。

2. **Serverless**

   - Serverless = FaaS (Function as a service)
   - 使用过程
     - 写一个函数并上传到服务器
     - 定义函数被调用的condition
     - 起一个VM或者container
     - 执行函数
     - 返回并kill掉VM或container

3. **Serverlss优点**

- 自动Scaling：服务端会自动分配计算资源（起VM）
- 部署很方便：只要写函数时import对应的包，剩下交给服务器
- 按需收费：只在需要的时候消耗服务器资源（省钱）

4. **Serverless缺点**

- lifetime有限：一个函数执行时间有上限
- 函数执行没有状态：没有一直保留的全局变量，没有共享内存或者IPC，只能用数据库来保存状态和跨进程沟通（慢）
- 启动慢：每次跑函数起一次VM
- IO限制：多个函数会抢IO带宽
- 阻碍分布式计算：分布式计算需要自定义的进程间通信，而serverless没有直接的网络地址
- GPU等其他异质计算资源难以serverless

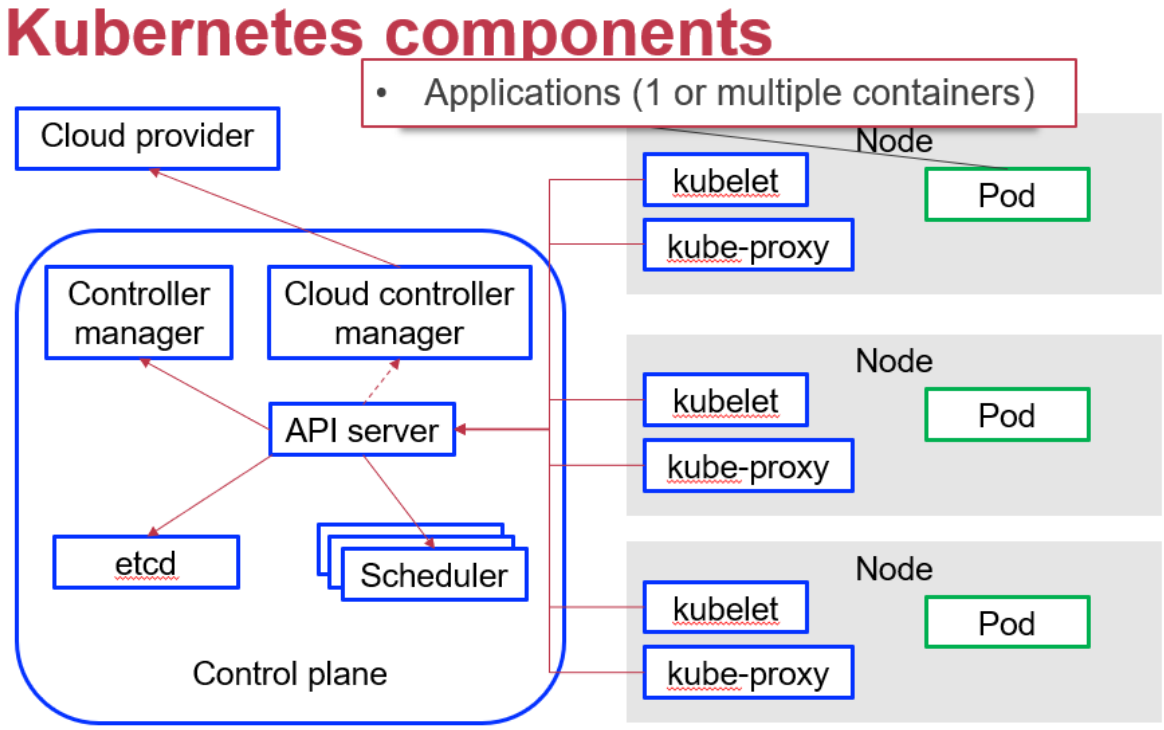
Kubernetes(K8s)

1. 简介

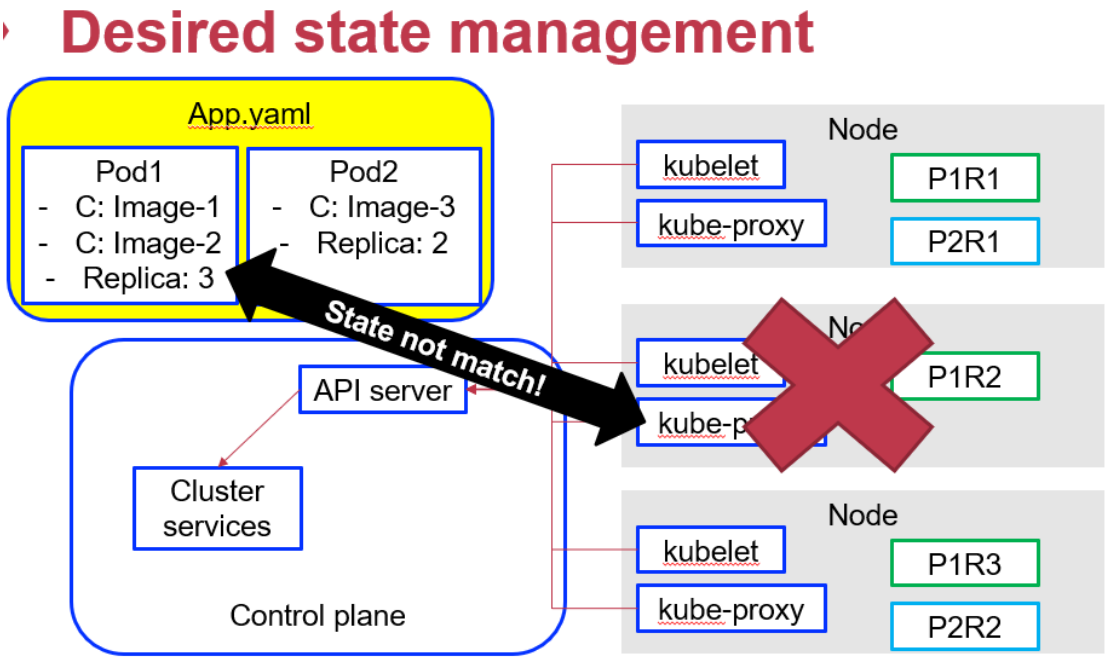
- K8s是面向cluster/数据中心的新OS
- 负责管理容器
  - 启动部署，scaling，容器可用性(地址访问)，调度infrastructure，自动更新，健康检查等

2. **组件**

- 基本组成：control plane + workers(nodes)
- control plane
  - Controller Manager
    - 跑控制进程
    - 监视并控制cluster状态
  - Cloud controller manager
    - 运行和底层云服务商交互的controller
    - 把cluster和云服务商api连接起来
  - API server
    - 暴露K8s API
    - cluster中所有单位靠这个API交流
  - etcd
    - 高可用的键值存储
    - 保存cluster中所有数据
  - 调度器
- Nodes(workers)
  - kubelet
    - 每个node的manager
    - 暴露node状态给全局manager
    - 管理内部实例
  - kube-proxy
    - Node间通信
    - Node和control plane通信
  - Pod
    - 部署的应用



- Desired state management
  - 由配置文件给出需求的状态
  - 由controller manager监视cluster是否满足需求的状态



3. Service mesh

- 一个面向微服务的框架，功能和k8s类似
- 微服务要解决的问题
  - 各种通信协议
  - 权限管理
  - 监视记录(logging)
  - 服务发现
  - 中间件load balance
  - 其他各种unexpected情况的处理(timeout， retry等)
- 区别

        - 由rust编写，安全性更高，更快

Serverless进阶(较新研究成果)

1. SOCK
    - 论文标题 SOCK: Rapid Task Provisioning with Serverless-Optimized Containers
    - 解决slow start问题
2. SOCK实现：提高容器性能
    - 传统docker的隔离机制是AUFS + namespace
    - SOCK换成了mount bind + chroot
    - 少了点功能(Serverless够用)，性能提高很多
3. SOCK实现：Zygotes
    - 原本是安卓的一个实现，在系统初始化时创建一个Zygotes进程，把所有要用的库都加载，然后工作线程从Zygotes fork出来，共享初始化的库。
    - SOCK的不同点：可以创建多个Zygotes，并对加载的库有安全性检查
4. cloudburst
    - 云服务商的极端实现
    - 规定所有函数只能用python跑，然后维护一个长期的python解释器
    - 快是快了，削足适履

07 SQL

- DBMS **和 KV 数据库**比较：In a key-value store, all the relationships and operations are managed **by the programmer** (**p29**左右)

关系模型（p32起）

关系relation, 元组tuple

怎么定位一个tuple？Table name + primary key

Short summary of relational model

Goal: hide the implementation detail of data behind a clean interface
- Yet, the interface is expressive enough
- The interface is also pr

Relational model
- Data model is based on set
- Data query language is based on set algebra

The query language (e.g., SQL) is **declarative**
- The developers only specify what they want

DML （Data Manipulation Languages）

- **procedural 过程式、declarative 声明式**

关系代数（p40-50）

复杂数据关系的表示（p59起）

- Duplication and normalizing 重复和规范化（p65）
- representing one-to-many 表示一对多关系（p66）

1. 解决方案1：**添加新表和外键**
2. 解决方案2：**document model 文档模型** （一对多关系）

Document model 文档模型（e.g. JSON）

- schema flexibility：任意修改schema
    - 关系数据库中的schema更改**很慢**

关系模型和文档模型对比

Summary: relational model vs. document model

Document model is a representative model in NoSQL databases

Benefits of **document model**
- Better locality
- Schema flexibility

Benefits of **relational model**
- Join supports
- Better modeling many-to-one & many-to-many relationships

OldSQL → NoSQL → NewSQL → HTAP

OldSQL（p81）

- **OldSQL = Relational Model + SQL + ACID**
- OLTP transaction
  - 生命周期短
  - 需要的数据量小

OldSQL = Relational Model + SQL + ACID

The architectural or historical baggage of SQL

Abstraction
- Relational + Transaction

Semantic
- ACID

Architecture
- Mostly focus on Single-node & On-disk

NoSQL（p91）

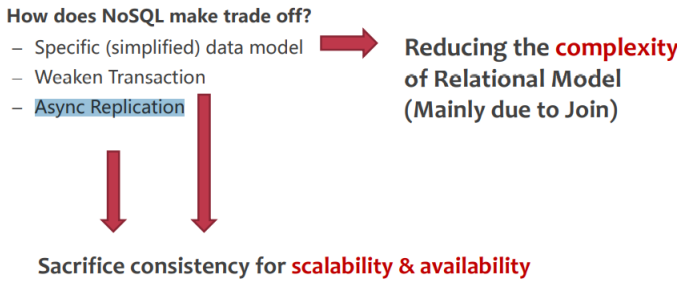
Scale Horizontally with Middleware 使用中间件横向扩展

- **[√]** Read/Write single Record
- **[×]** No Distributed Transaction & Join
  - Unavailable when: Changing schema - Server failover
  - Data Scaling & Re-sharding

tradeoff（p95-99）

- 简化数据模型 => 降低复杂度
- 弱化transaction，异步复制 => 牺牲一致性，换取scalability和availability

NoSQL - Build from scratch



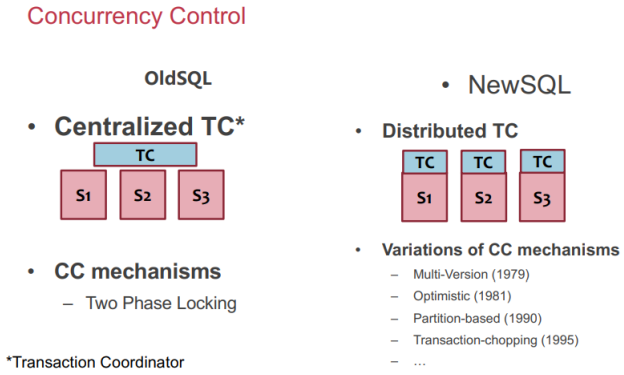
NewSQL（p105）

定义：A class of modern RDBMS, which provides:

- NoSQL's Scalability
- SQL's ACID Transaction & Relational Model

怎么扩展scale?

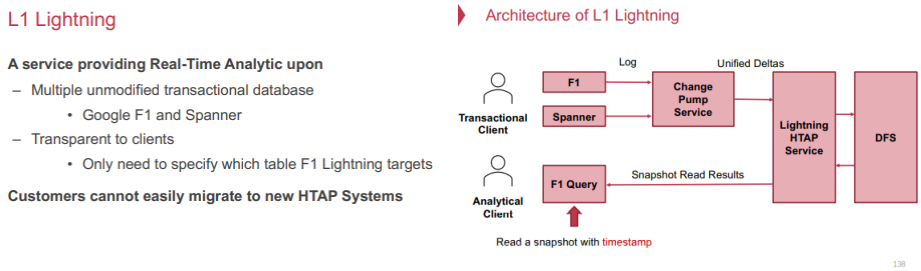
- **Shared-nothing Partitioning**



HTAP(Hybrid Transactional/Analytical Processing)（p131）

- **real-time analytics on fresh data**
- OLAP (ML)

L1 Lightning（p137-140）



08 Transactions

- **Atomicity**: TX is either performed entirety or not performed at all. **(rollback回滚)**
- **Cosistency**: Transaction must change the data from a consistent state to another
- **Isolation**: Two concurrently executed transactions are isolated from each other
- **Durability**: Once a transaction is committed, its changes must durably stored to a persistent storage

如何保证？ **(p28)**

- **I：一致性控制方法**

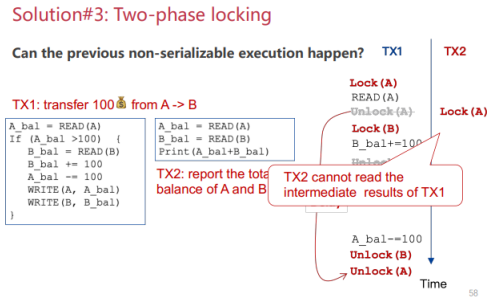
Serializability（p31）

- 是理想化的：**并发事务T1,T2,...,TN好像是按顺序执行的一样**
- 如何检查是不是serializability？replay the concurrent execution to a serial of reads/writes（**例子p35-42**）

Serializability的实现（p45）

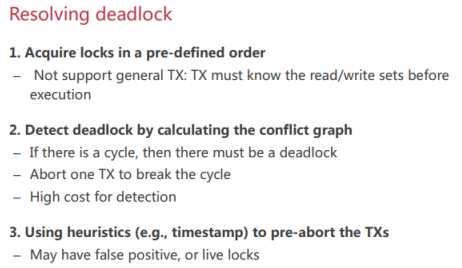
使用锁lock

- 全局锁global lock：一次只能一个TX， 不并发（性能差）
- 简单细粒度锁simple fine-grained lock：每个record有一个锁（锁容易放太早，不正确）
- **Two-phase locking** 2PL：TX commit之后再放所有的锁 -> **保证Serializability**



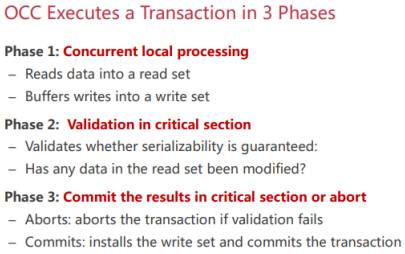
2PL的问题——deadlock（p61）

解决方法：



Optimistic concurrency control -- OCC（p67）

- Executing TXs **optimistically** w/o acquiring the lock
- Checks the **results of TX** before it commits
    - If violate serializability, then **aborts & retries**



具体例子（p69-80）

好处：

- phase1: Operates in **private** workspace; **rare inter-thread** synchronization (optimistic)
- phase2, 3: Needs synchronization, but usually very **short at low contention**

问题：

- False Aborts 错误的abort
- livelock：high contention情况下，一直abort，没有progress

Summary of realizing serializability

Pessimistic methods
- Presume that interference is likely
- Prevent any possibility of conflict actively
- E.g., global lock, 2-phase locking

Optimistic methods
- Allow write in any order and at any time
- If detect conflict, then "sorry, conflict write, please abort, clear the history and then retry"
- E.g., OCC

Modern Transaction Systems（p89）

HTM（p92）

intel RTM（p96）

Fun facts about RTM

How does Intel implement RTM?
- Basically, OCC!
- Use CPU cache to track TX's read/write sets

Why efficient?
- Cache is a perfect place for tracking TX's temporal updates
- Leverage existing cache coherence protocol to detect conflicts: reuse existing multi-core hardware to implement transactional memory!

Hardware support for transactional memory总结

- Easy programming model for the programmer
- Good performance if using properly
- However, the programmer should handle its pitfalls

DBX（p111）

a TX system to use **RTM for acceleration**, but **avoids its pitfalls** for TXs

Conclusion of DBX

RTM is a promising feature provided by Intel CPU

RTM alone is not sufficient for TXs

DBX provides a study of how to use RTM to accelerate in-memory databases

RTM can simplify the system building and get comparable performance
- Limitations of RTM force us to craft the transaction region and memory access pattern carefully

ROCOCO（p135）

回顾：优化TX的方向？Improve the TX algorithms properties

- E.g., better deadlock detection algorithms in 2PL
- E.g., reduce aborts in OCC

Overview of ROCOCO

1. Two-phase protocol
- Most pieces are executed at the second phase

2. Decentralized dependency tracking
- Servers track pieces' arrival order
- Identify non-serializable orders
- Deterministically reorder pieces

3. Offline workload checking
- Identifies safe workloads (common)
- Identifies small parts that need traditional approaches (rare)

Conclusion of ROCOCO

Traditional protocols perform poorly w/ contention
- OCC aborts & 2PL blocks

Rococo defers execution to enable reordering
- Strict serializability w/o aborting or blocking for common workloads

Rococo outperforms 2PL & OCC
- With growing contention
- Scales out

RDMA

Basic knowledge

Kernel bypassing network: DPDK

- Userspace, driver lib 来使用网卡
- 优势
  - bypass kernel
  - 减少memcpy overhead
  - 用户可以定制化网络，不需要使用TCP/IP协议
- 不足：
  - 扔就需要软件栈来实现general-purpose networking, 比如用户态协议栈

CPU bypassing:

- CPU很难变快了 ——power wall
- NIC网卡持续变快， 200Gbps
- 数据中心需要节能，polling会导致能耗高，中断会导致性能差(超过100X)
- 方案：单边RDMA，可以bypass CPU

RDMA目前的特定

- 价格更低
  - $19/Gbps (RDMA 40Gbps) vs. $60/60Gbps (Ethernet 10Gbps)
- 向下兼容性(downward compatibility)
  - RDMA atop RoCE supports fast ethernet (compatible with existing datacenter apps)
- 在现代数据中心中应用广泛
  - Even avaliable in the public cloud

Case study: dsitributed kv store

Farm-KV @NSDI'14

Goal: 使用单边RDMA来建立分布式KVS

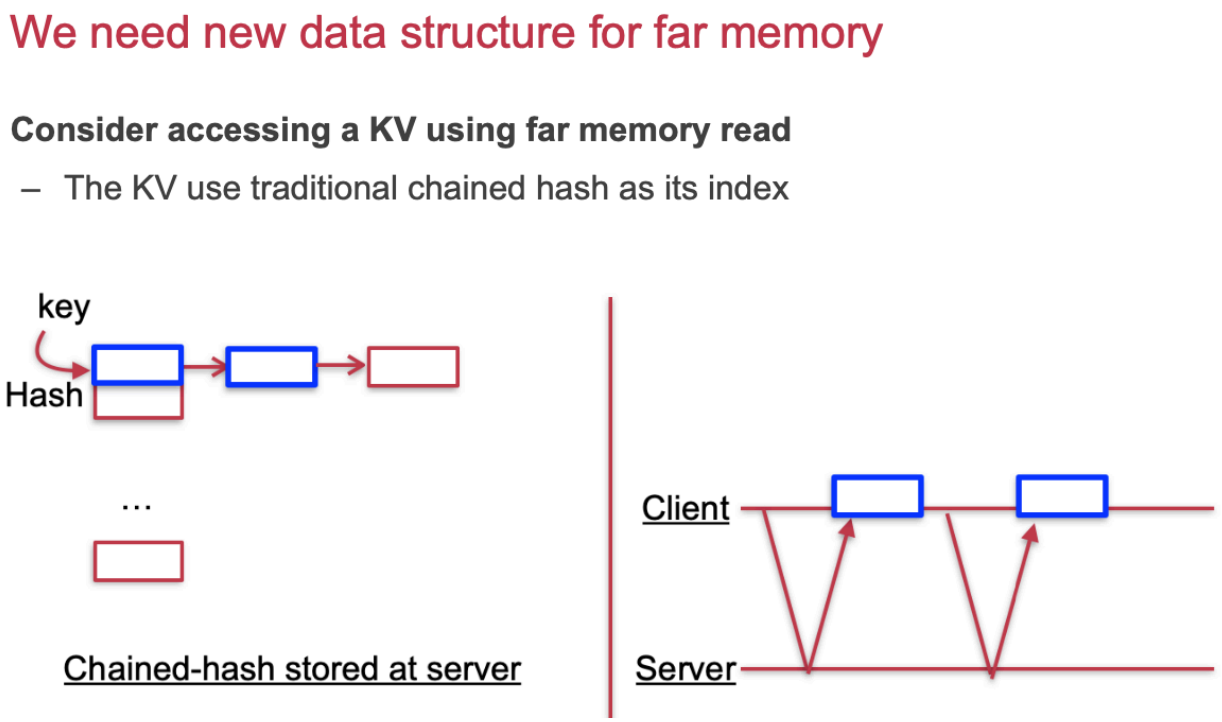
Question:

- 我们使用的是single node KVS
- 我们是否可以将内存访问替换为远端内存访问?

Problem:

- 单边RDMA速度还是比本地内存访问慢很多 （2 us vs 100ns）

传统HashIndex会增加更多的network roundtrips



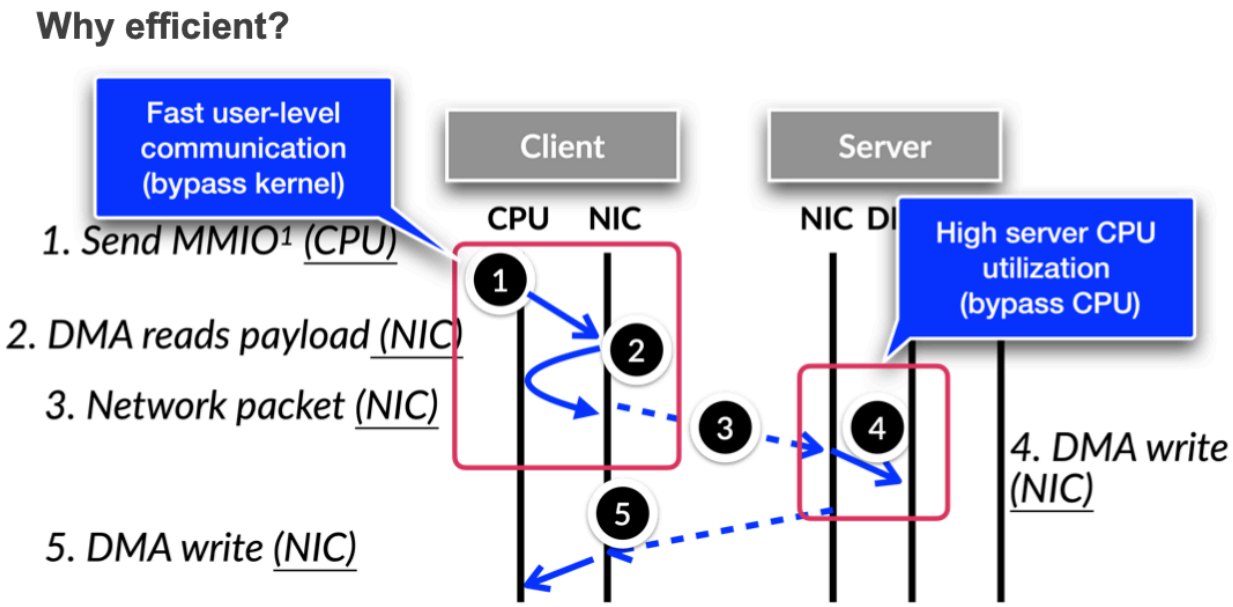
Network amplification: 远端内存在网卡端只有非常简单的抽象；实现high level的KV抽象需要更多NIC网卡操作

因此，FaRM对数据结构(hopscotch hashing)进行了重新设计。Hash之后的key对应的bucket，能够保证目标entry一定在有限个bucket offset中（相当于进行了一个限制）。这样就可以通过一次RDMA Read来把offset个bucket一次性读回。

但是Update会变得更加复杂（需要更多rehash操作）

RDMA实现细节

Life cycle of an RDMA one-sided WRITE



但是RDMA的cost还是很高：PCIe latency, network latency.

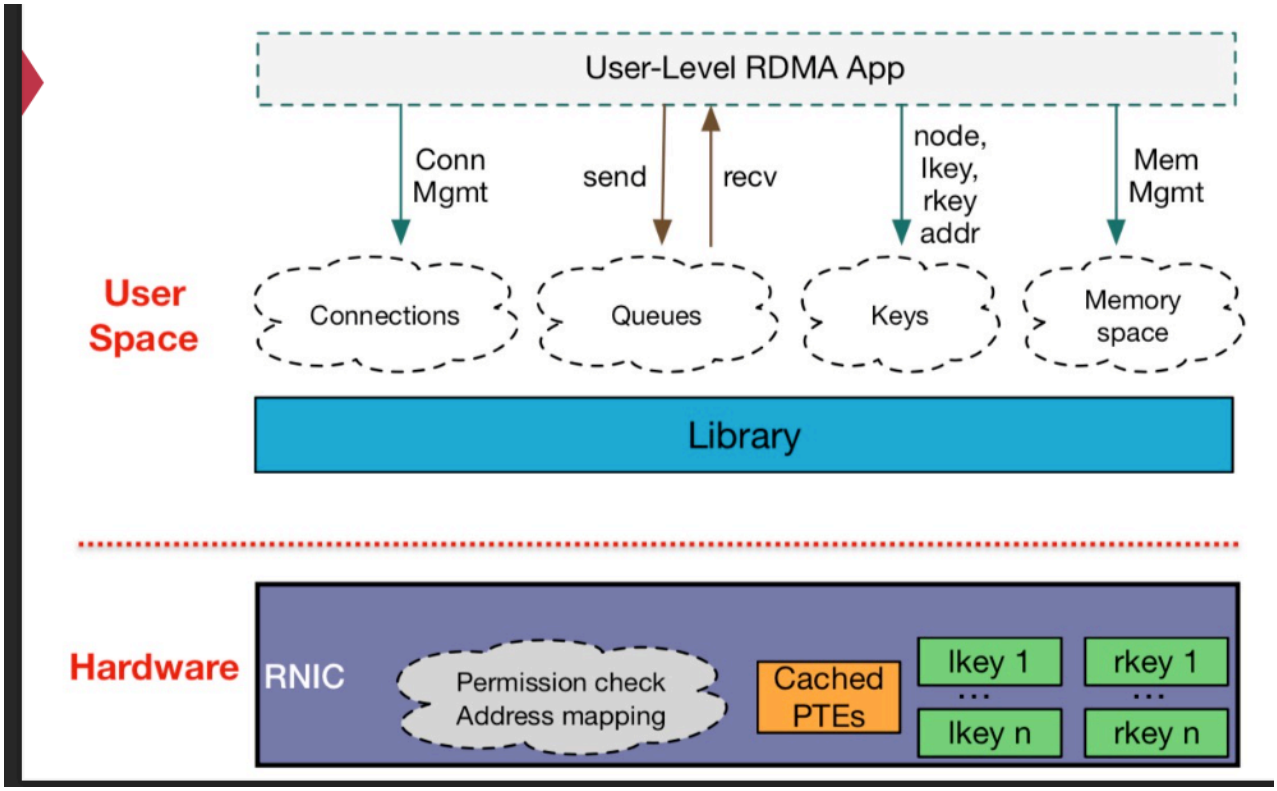
Outstanding request & coroutine

- Outstanding requests means keep multiple RDMA requests on-the-fly
- Coroutine is the same as traditional asynchronous I/O programming
- Goal: hide polling latency

doorbell batching

- One MMIO to post a batch of requests, NIC uses DMA to read them

RDMA: out memory design



Connection, translation, permissions, etc. information are stored on the host memory, NIC only caches them in its SRAM, but with a limited space(~2MB). When cache miss, use PCIe read to fecth them.

RDMA: out memory design makes cache miss costly

Driver stores these at the host memory

- NIC caches a portion of such information

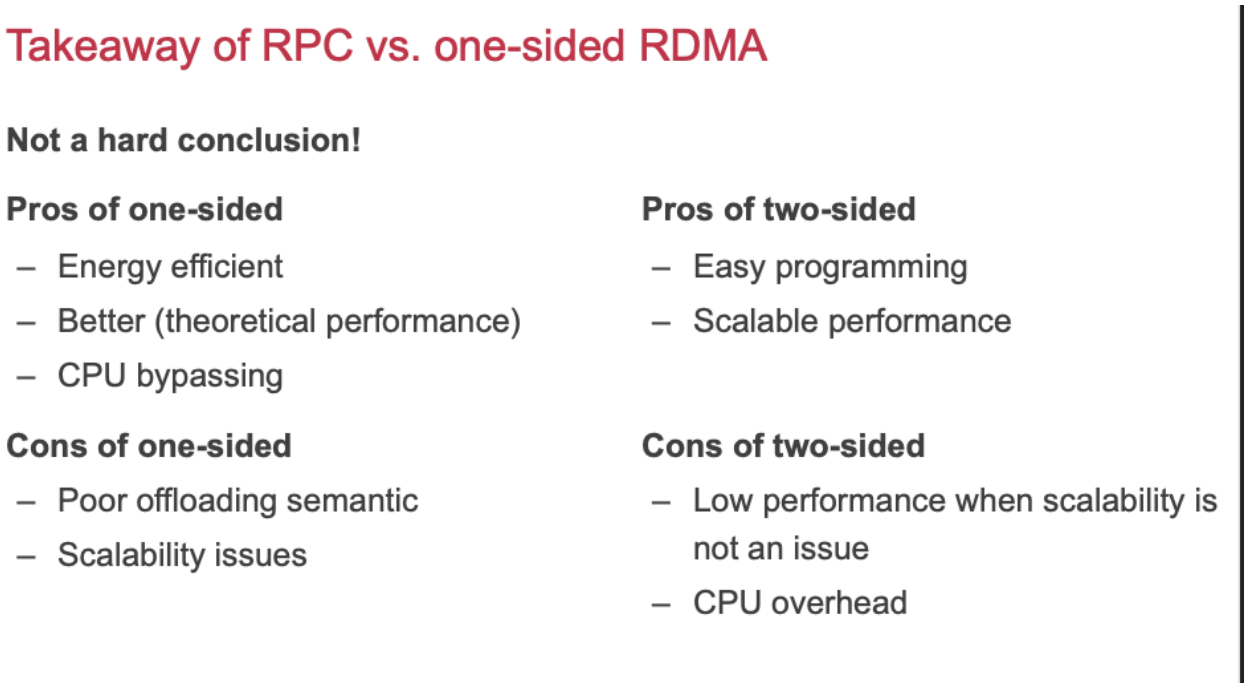
Fetch miss items using PCIe read

- ~ 1us additional latency, fetch from DRAM using PCIe read
- For comparison, RDMA one-sided read takes ~2us

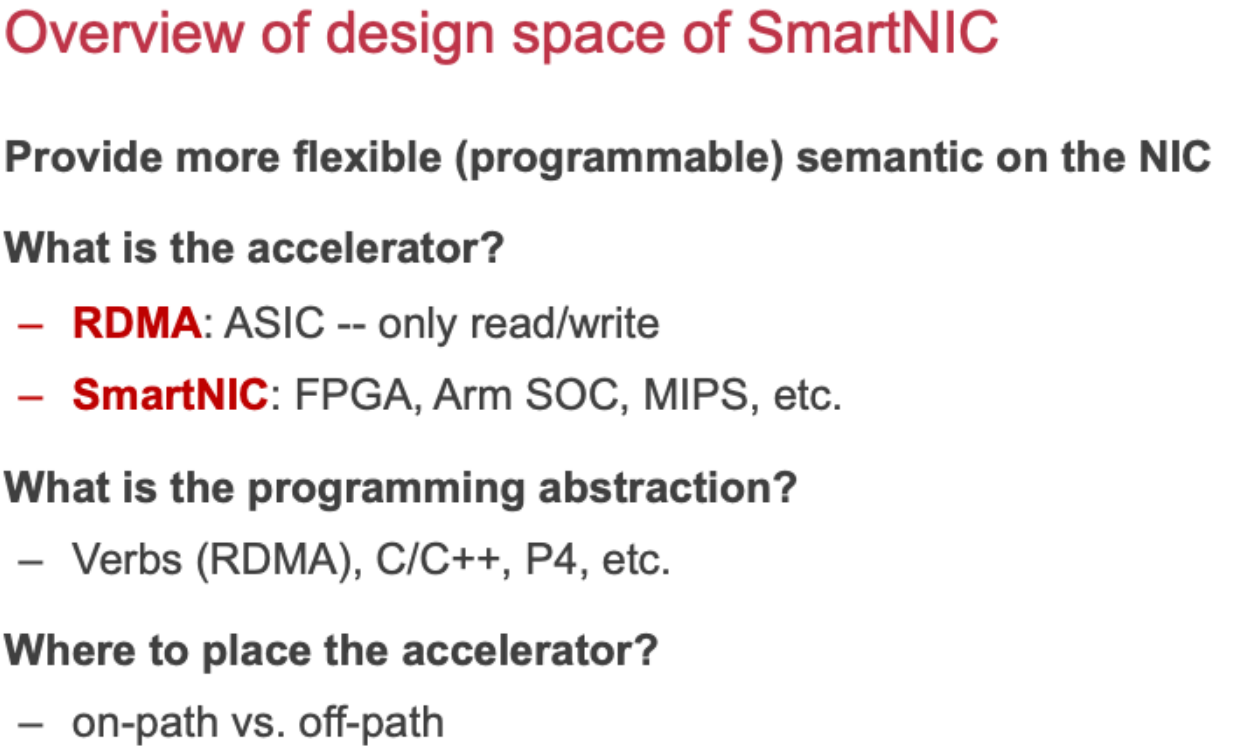
一些缓解方案：

- FaRM@NSDI'14: use huge page
  - Reduce #PTE entries
- LITE@SOSP'17 takes to another extreme; it directly physical memory, and uses kernel for security check
  - No PTE and keys need to be cached at the NIC!
- FaSST@OSDI16: 使用UD

- 减少QP，没有connection信息

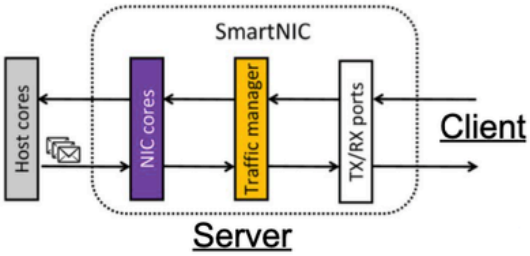


SmartNIC



On-path vs. off-path

On-path vs. off-path

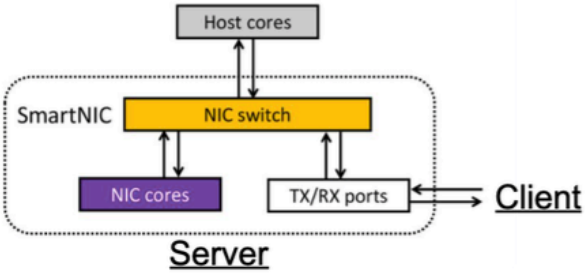


Pros

- Lower latency

Cons

- No resource isolations



Pros

- Clear resource isolation

Cons

- Latencies added due to NIC switch

92

FPGA vs. Processors

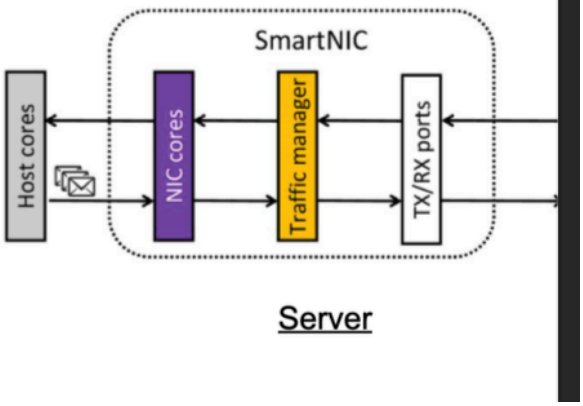
FPGA vs. Processors

The NIC cores can be

- FPGA
- ARM processors [1]

In the case of ARM, the NIC cores run a full OS

- E.g., Linux
- Represented RNICs: Bluefield, Stingray

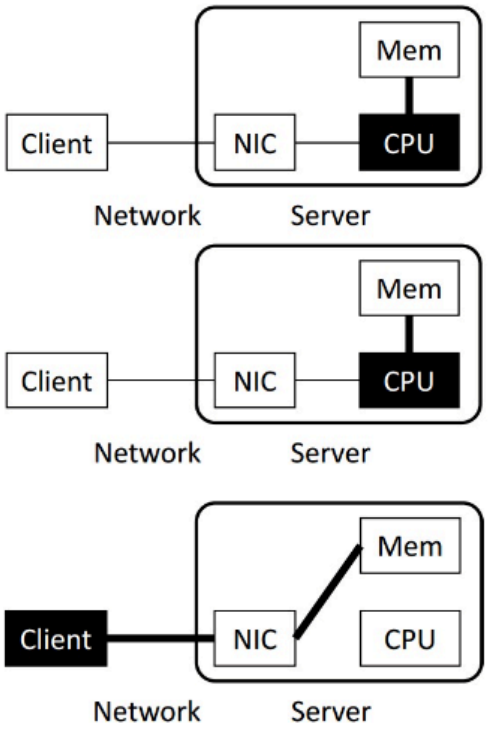


为什么智能网卡慢：

- 功耗限制
- 读取Host DRAM需要进过PCIe

KV-Direct:

Distributed key-value architecture



Software (Kernel TCP/IP)

Bottleneck: Network stack in OS
(~300 Kops per core)

Software (Kernel Bypass)

e.g. DPDK, mtcp, libvma, two-sided RDMA

Bottlenecks: CPU random memory access and KV operation computation

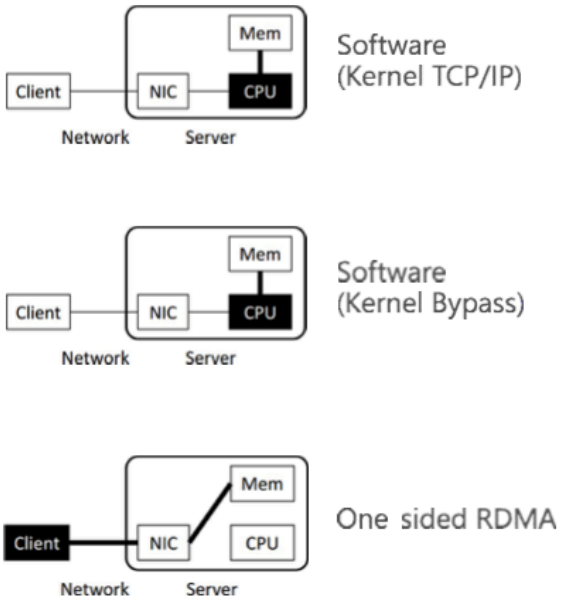
(~5 Mops per core)

One sided RDMA

Communication overhead: multiple round-trips per KV operation (fetch index, data)

Synchronization overhead: write operations

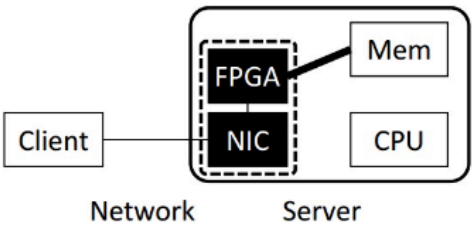
Distributed key-value architecture



Software (Kernel TCP/IP)

Software (Kernel Bypass)

One sided RDMA



KV Direct FPGA + NIC

Offload KV processing on CPU to Programmable NIC

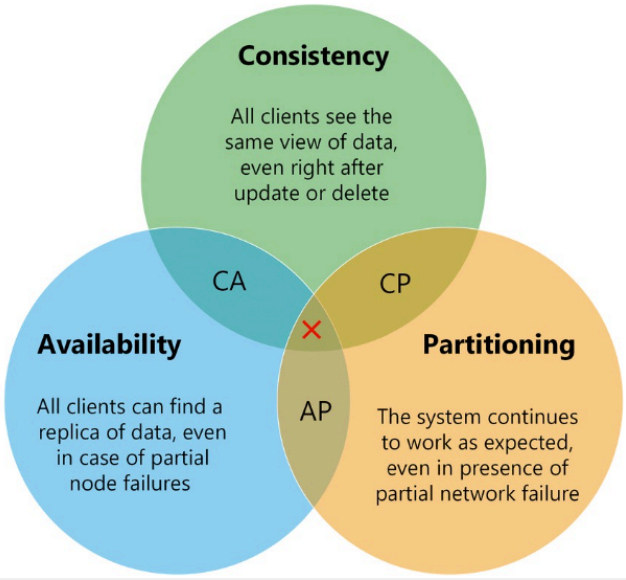
Fault Tolerance

Fault, Error, Failure

- **Fault can be latent or active**
  - If active, get wrong data or control signals
- **Error is the results of active fault**
  - E.g. violation of assertion or invariant of spec
  - Discovery of errors is ad hoc (formal specification?)
- **Failure happens if an error is not detected and masked**
  - Not producing the intended result at an interface

CAP theorem

- **CAP: pick two & trade-off**
- **CA**
  - Single machine
- **CP**
  - Hbase, Redis
- **AP**
  - Cassandra, CouchDB



Recovery恢复

Goal: 从crash 或者网络failure中恢复状态

正确性保证：未commit的continue or abort；已经commit的数据可以持久化

Solution: **Logging**

Write-ahead logging (WAL)：在更新数据状态之前，把log先写入磁盘；并且log是append-only的;

Structure of WAL Record

- **LSN**: unique log sequence number
- **Type**: operation (insert/delete/update)
- **After image**: new state
- **Before image**: old state
  - Insert location, delete location

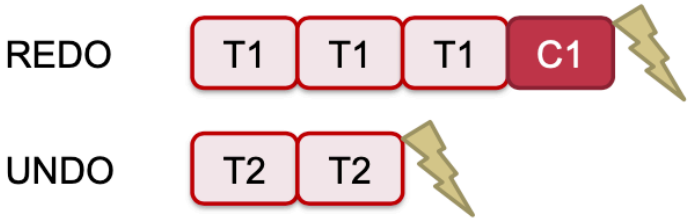
Checksum	LSN	Log Record Type	Transaction Commit Timestamp	Table Id	Insert Location	Delete Location	After Image

在Commit的时候，添加一个特殊的Log：CMT (Commit Log)

- Commit log (CMT)

Recovery Rules:

- **Recovery rules**
  - Travel from end to start
  - Mark all transaction's log record w/o CMT log and append ABORT log
  - REDO: committed
  - UNDO: uncommitted/aborted



日志与并发事务：

- 单个log queue
- 多个log queue

性能考虑：

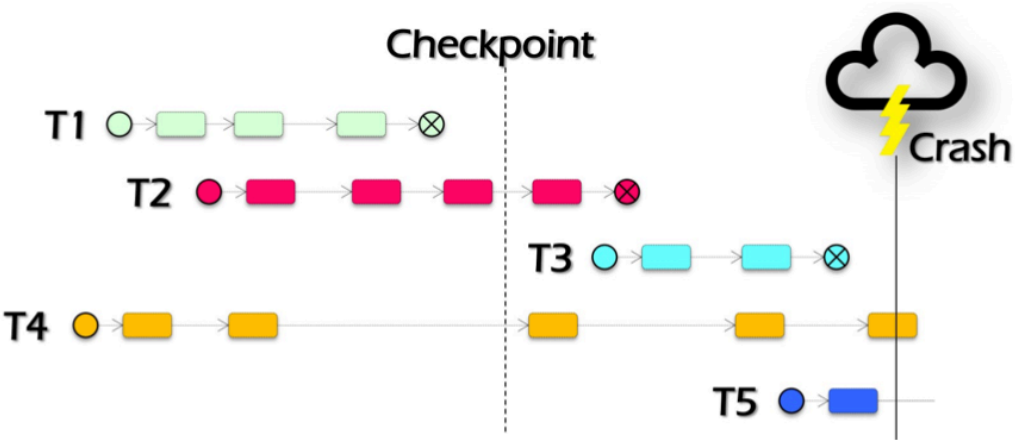
- 局部性（log序列化考虑Sequential IO）
- 并发控制（会增加tail latency）

我们为什么需要flush disk:

- No need to recovery from a blank state
- Avoid aborting long transaction

刷磁盘的频率：不是只在最后的时候把log刷入磁盘——checkpoint

Put it together



T1: do nothing; T2: redo; T3: redo; T4: undo; T5: undo

WBL (write-behind logging)

WBL: write-behind logging

- **Reduce data duplication**
  - flushing changes to the database in NVM during regular transaction processing
- **When insert a tuple**
  - Write data *before* meta-data in the log
- **Log is behind the contents of the database**

Primary-backup 主从备份

Fault tolerance机制：

- Replicated machins
  - Fully replication of data
- Log
  - Synchronize data updating

AI background

Machine learning内容：data, model, loss function, Optimization Algorithm

Problems: 泛化、过拟合、欠拟合 解决方案: 使用cross validation评价模型

ML算法分类：

- 监督学习：回归，分类，推荐系统
- 无监督学习：聚类，降维
- 强化学习

训练过程：降低loss function

- 线性回归

$$\hat{\boldsymbol{\theta}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

  线性回归由于计算逆矩阵过于复杂，因此较少使用

- 梯度下降
  - Batch Gradient Descent:一次训练所有样本
  - Stochastic Gradient Descent:一次训练一个样本 (快，不稳定)
  - Gradient Descent with mini-batch:一次训练几个样本 (快，稳定，便于并行化)

选取更新率：Momentum，Adagrad，Adam

Deep learning 从LeNet到AlexNet的挑战

- 数据需求：ImageNet
- 计算需求：GPU
- 避免过拟合：Dropout regularization（随机将神经元输出设为0）
- 梯度消除、爆炸：Residual Net

System for AI

System for AI的目的：

- 训练 Training:
  - 加快训练
  - 训练大模型 scalability
- 推断 Inference:
  - 推断速度
  - 在不同设备上部署（例如边缘设备）

关注点

准确率 & 资源利用率

- 训练 Training: 吞吐量
- 推断 Inference: latency

User API层面

Why not just numpy?

- 只支持部分简单算子
- 需要编程者自己计算梯度
- 需要自己实现更新规则

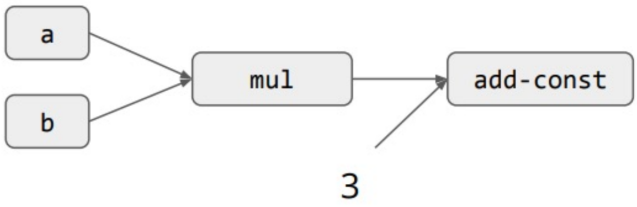
大家更喜欢声明式语言

声明式AI语言实现方法：计算图

Nodes represents the computation (operation)

– E.g., Matrix multiplications, softmax operator, activation functions

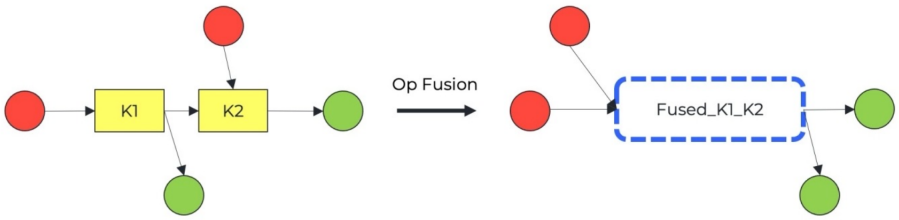
Edge represents the data dependency between operations



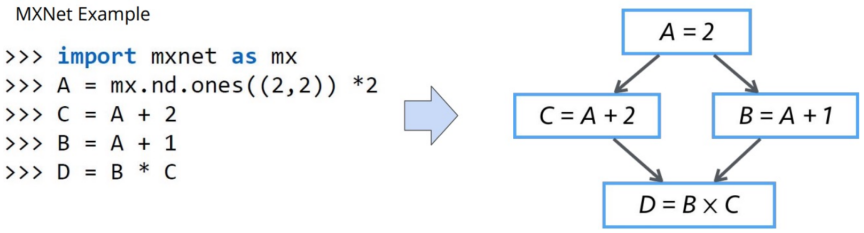
Example: computational Graph for a * b +3

计算图优点：

- 自动求导：利用求导的链式法则
- operation fusion: 避免算子之间的内存拷贝



- parallel scheduling：提高计算效率（例如，将B C过程放到两个计算单元中并行计算）



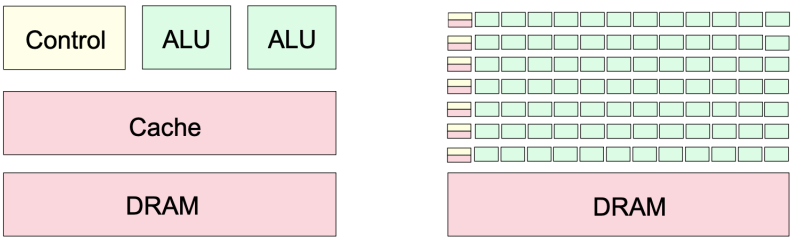
声明式语言更加兼容异构的计算硬件：使用为特定硬件优化的编译器，链接库 etc，无需为特定硬件重新编程

架构层面（硬件）

GPU

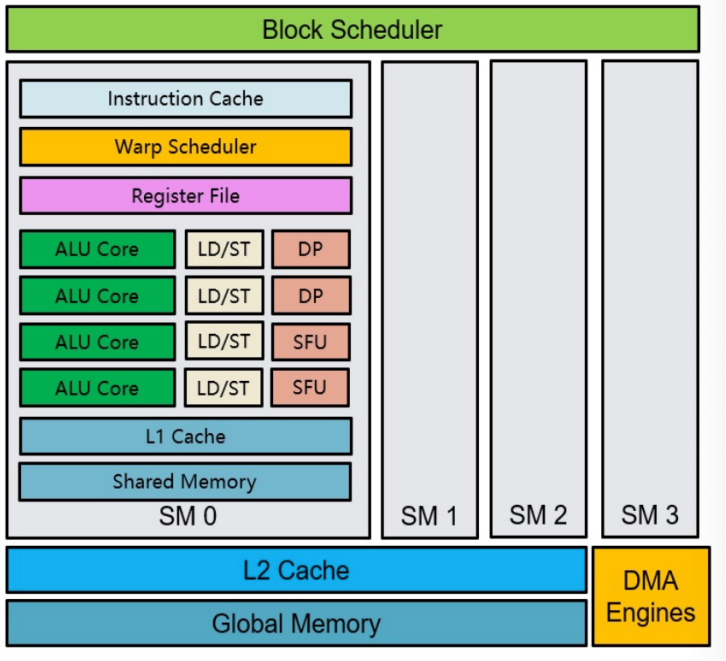
GPU硬件架构简介：

GPU较CPU有更多ALU单元



Simplified architecture of CPU　　Simplified architecture of GPU

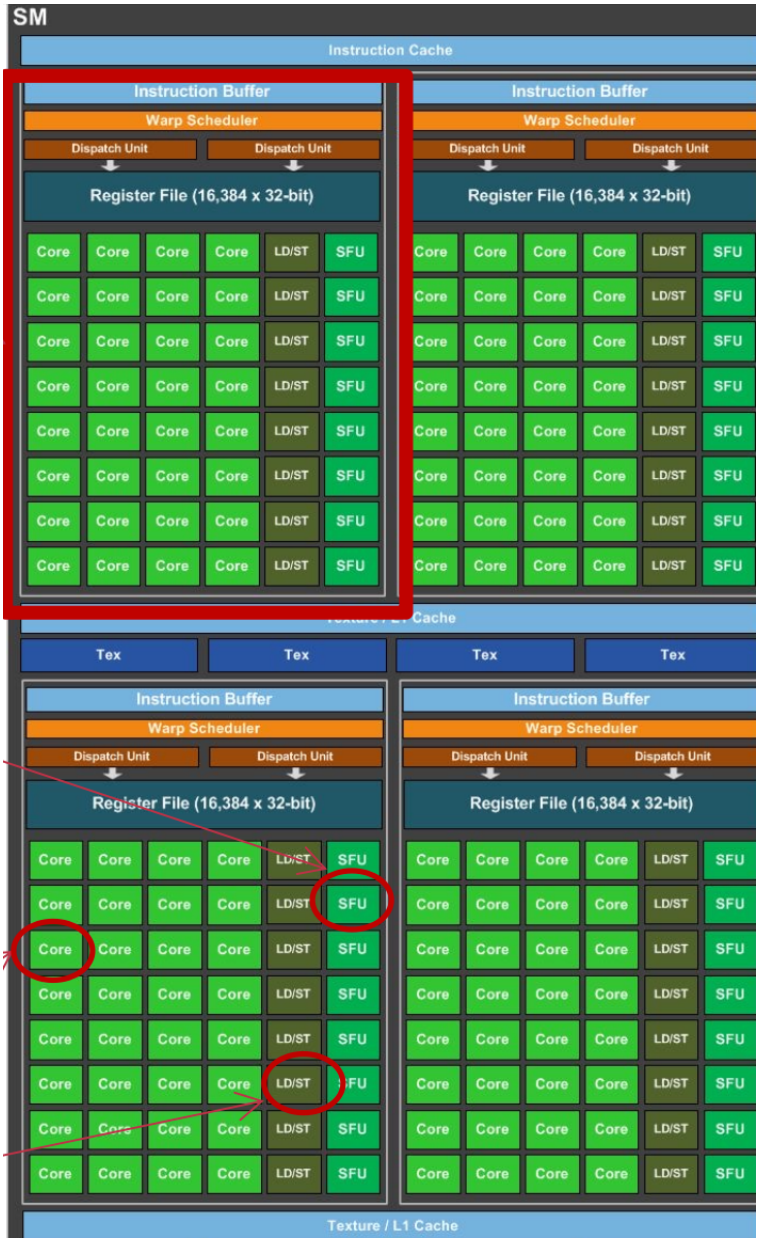
GPU支持SIMD（单指令多数据）并行执行能力更强 （CPU也有支持，但是支持并发程度仍低）



Block：一组GPU线程

Streaming Multiprocessors (SM): 一组ALU核，每个SM有一个L1缓存

SM中分为多个Warp，每个warp中的核执行同一个指令



CPU vs GPU summary：

- CPU
  - 对内存访问优化
  - 对乱序执行控制更优
- GPU
  - 算力高

GPU的内存层级结构仍然是复杂的，因此需要编程时注意，因此有CUDA模型：

- SIMT：单指令，多线程
- 抽象C code：编程者为线程编写C代码，每个线程执行相同代码（支持分支）
- 线程被组合成block
- kernel：一组block组成的grid

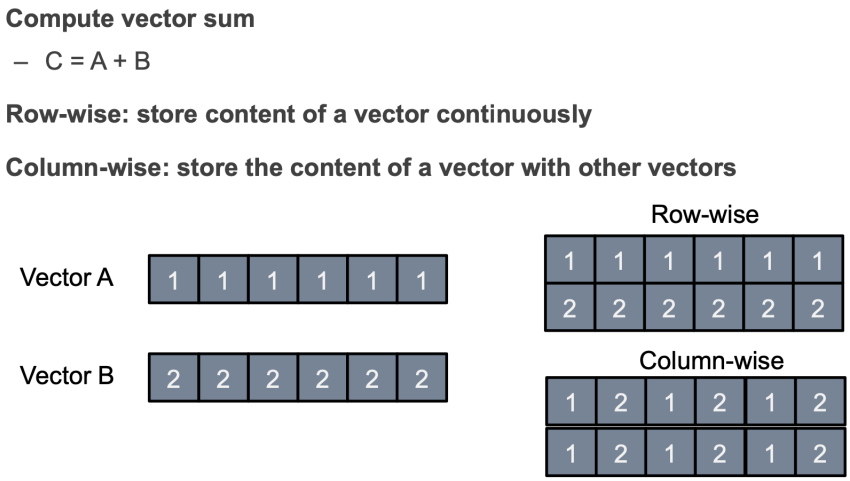
SM调度器会将block分配到SM上，由调度器进行block的换入换出，每个block则被拆分成多个warp，在硬件上执行

每个warp中的线程共享一个program counter，因此在CUDA编程时需要慎重考虑分支

为什么使用block概念(本质就是内存访问优化)：1.block层面共享内存效率高(类似L1 cache) 2.可以有轻量快速的同步barrier

需要考虑的问题：

- 内存分布（需要编程者仔细考虑）



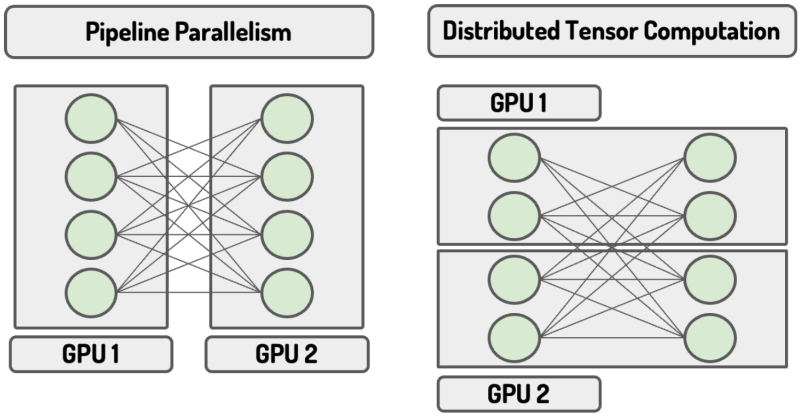
- dead lock（共享PC）

```
if (threadIdx.x == 0) {
    consume();
} else {
    produce();
}
```

GPU使用out-memory design，通过PCIe访问，在GPU上执行单次计算效率低

Scale

模型并行



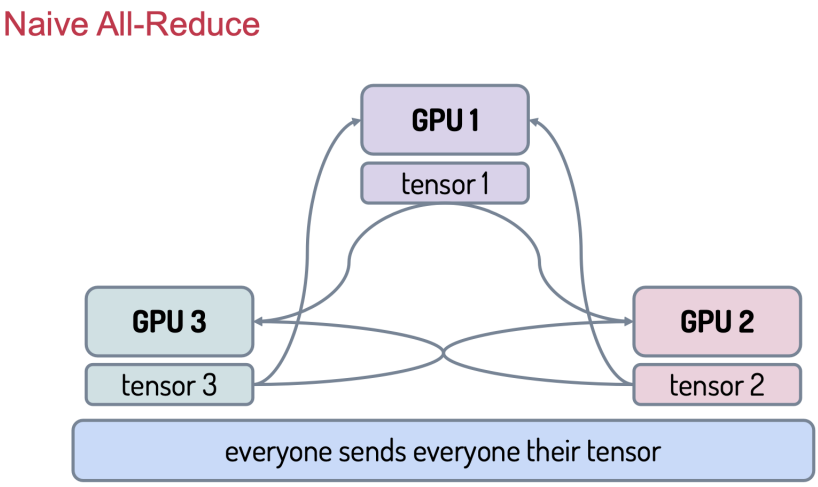
主要讲了pipeline式的模型并行

往往一个GPU中会有多层

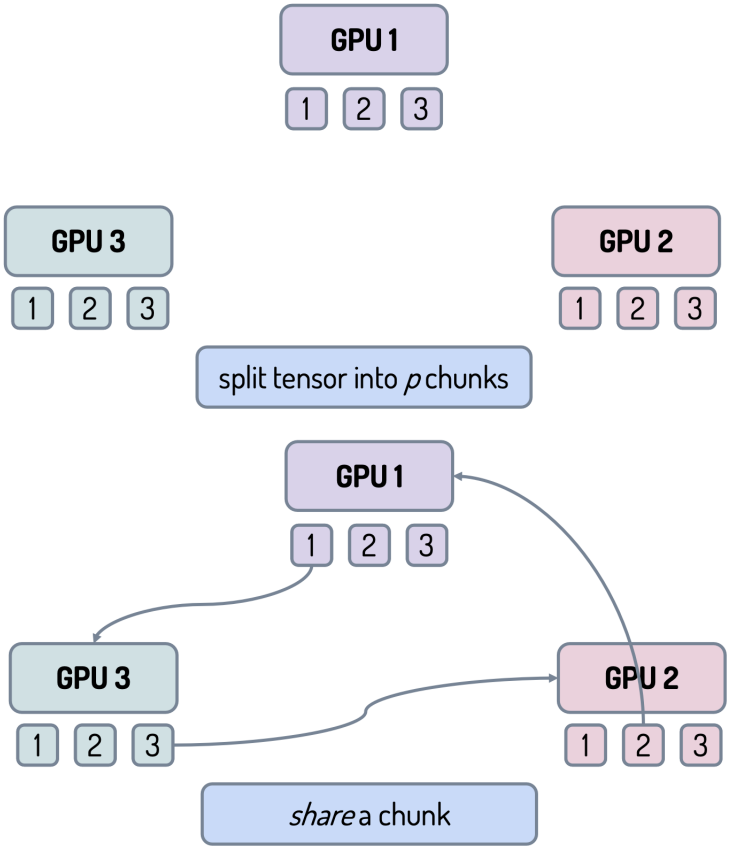
但是pipeline仍然无法解决需要存储activation的问题，activation memory usage睡着minibatch size 和 参数的数量增加。通信开销过大，无法换入换出。解决方案，扔掉GPU中间的层的activation，在方向传播时再重新传播出中间扔掉的activation

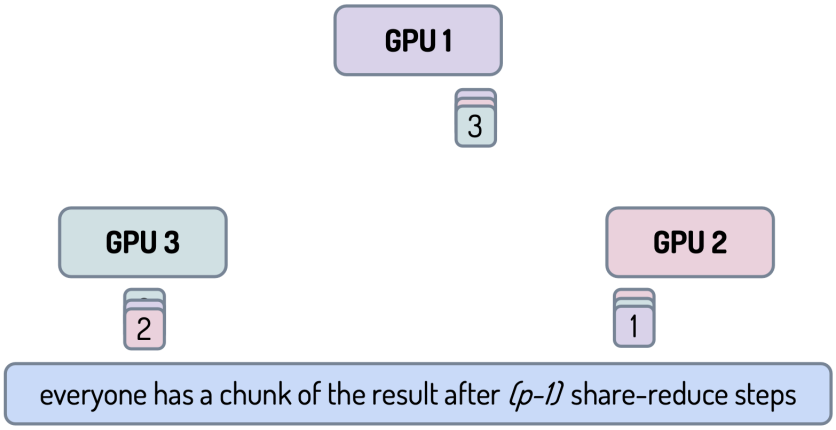
数据并行

需要进行一个all reduce过程来保证模型收敛到同一状态

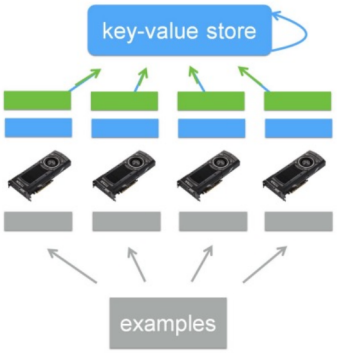


使用Ring ALL reduce方法，降低通信成本O(np^2) -> O(np) (p个GPU, tensor size是n)





或者使用Parameter server来做key-value store，在使用参数前pull，通过barrier保证同步



但是这样性能受最慢节点影响，因此尝试异步方法Async Training

异步会使精度降低，收敛变慢

此外Parameter server带宽可能不足

13. AI for system

AI能够给系统以下部分进行提升: indexes, joins, sorting, caching, hash tables, scheduling, planning

索引优化

举例: 为100到1M的整数建立索引 **Data=[100,101,102,...,1M]**

- 如果我们知道数据分布,则索引可以表示为**Data[key-100]**
- 如果数据为偶数分布 **Data=[100,102,104,...1M]**
- 索引可以表示为**Data[(key-100)/2]**

使用传统的数据结构例如B+树,最差情况下的复杂度为O(log(n)),如果使用AI模型可以缩减到**O(1)**

1. learned index(仅查找)

限制条件:

- 只读不能进行写操作,因为会改变数据分布
- 数据以顺序数组形式存储

动机:
将BTree当作一种模型, BTree的每一个叶节点都是一个page, 每个page中包含一段连续数据, 在查找key时, 首先映射对应的叶节点, 即pageID, PageID = BTree(key), 随后在[pos, pos+page size]这段范围查找value.

learned index:
建立模型将 key直接映射为有序数组中的pos, key->pos
在 [pos-err_min, pos+err_max] 中查找, key->pos的映射关系在有序数组中相当于累积概率模型CDF,#keys为key的数量

pos = P(X <= key) * #keys = F(key) * #keys

err_min和err_max在训练中的计算方法

```
for key,pos in sorted array:
    if model(key) >= pos:
        err_min = max(err_min, model(key) - pos)
    else:
        err_max = max(err_max, pos - model(key))
```

使用learned model的潜在优势:

- 更小的内存访问: 在预测后只有O(1)的内存查找
- 更小的索引: BTree使用很大的内存空间存储索引, 而如LR(线性回归)只需要占用8B, pos = key * w + b

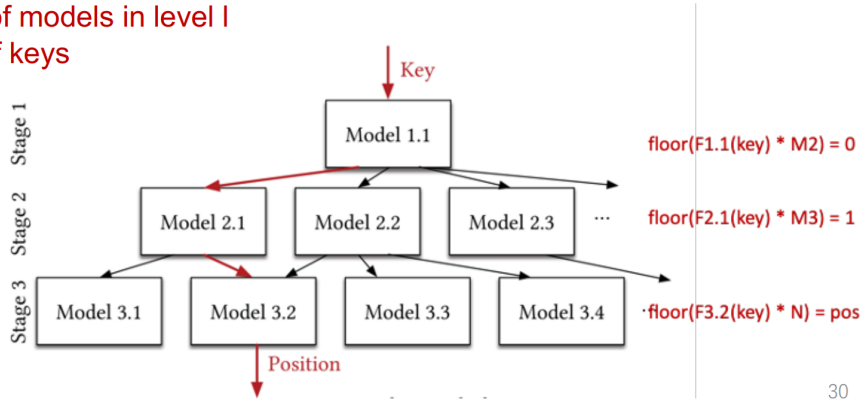
直接使用tensorflow实现模型出现的问题:

- tensorflow为大型模型设计,不适用于小模型训练,效率低
- 不能过拟合,意味着模型预测有很大的错误

Last Mile Problem:

- 数据中的离群值(不贴近CDF曲线的数据点)会对模型预测产生影响,偏离值越大,预测错误越多
- 使用RMI(Recursive Model Index),递归模型索引来解决,将key预测任务分散到各个小模型之中减少错误.

Fl.k - CDF of Ml.k
Ml – Number of models in level l
N – Number of keys



key通过顶部模型来决定分发到哪个子模型,依次传递到最终预测模型来预测key对应的数组中pos

如何在有序数组预测到的范围内查找key:

- 使用二分查找
- 优点: 如果搜索空间很大,复杂度从O(N)降为O(log(n))
- 缺点:如果搜索空间很小,CPU会执行很多分支预测操作开销

2. ALEX(查找与插入)

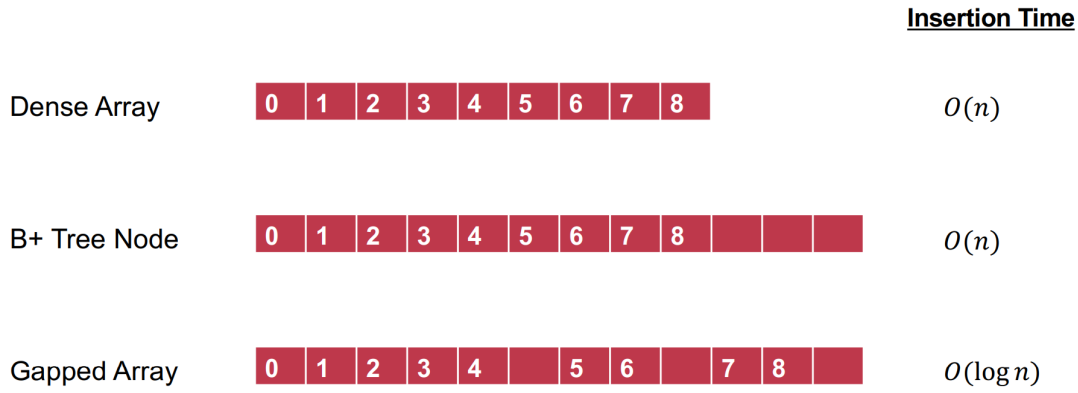
为什么learned index不能进行插入操作,因为它假设数据有序排列没有空间进行插入,如果插入数据需要移动其他key的位置

ALEX创新点

- Gapped Array
- Model-based Insert
- Adaptive Tree Structure

ALEX缺点:不支持并发,只支持单线程设置,当一个节点预测模型正在训练时不支持另一个线程并发查找

Gapped Array
gapped array中数据不是连续放置,而是存在空隙



gapped array使得插入数据只需要移动几个key,实现更快的插入,它的缺点为

- 可能将key移动到其他位置上,则模型预测就会偏离
- 不能使用二分查找, 使用指数搜索

Model-based Inserts
相对于普通的插入,比如先插入数据,然后训练模型能够预测插入位置,ALEX在新的数据开始插入时,首先使用模型预测插入位置并在模型预测位置处插入数据.之后一段时间后重新训练模型.

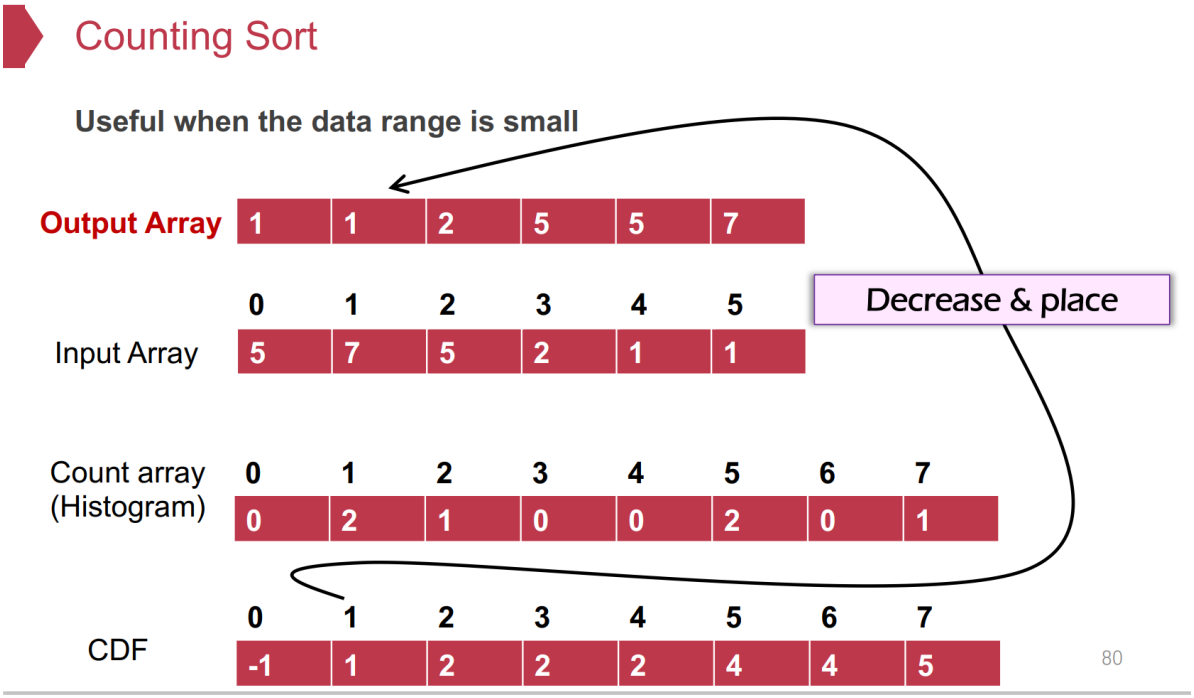
Adaptive Structure
灵活的树结构

- 将节点分裂为兄弟节点
- 将节点分裂为子结点
- 在gapped array没有多余的空隙时,拓展数据节点,插入更多空隙
- 合并节点

排序优化

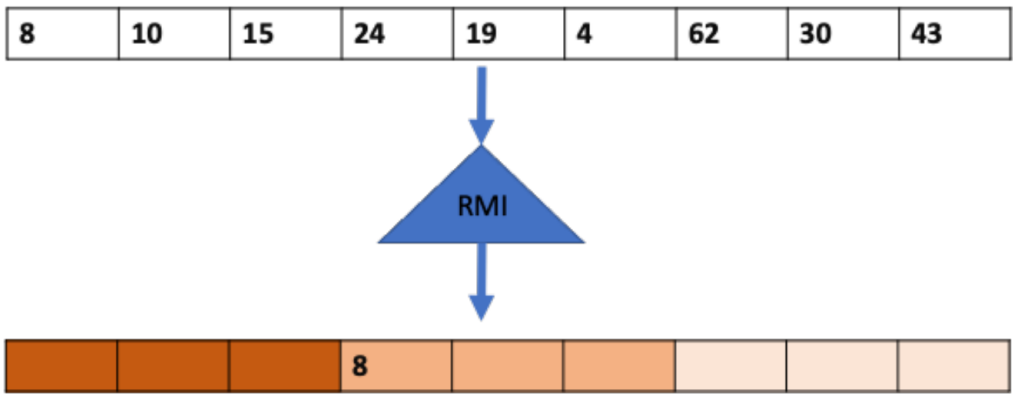
常见的基于比较的排序方法: 冒泡排序, 插入排序, 快速排序, 归并排序
上述的基于比较的排序方法至少需要 O(Nlog(N))的复杂度

Counting Sort 计数排序



- 对input array内的数据进行技术写入count array
- 使用count array统计出累积概率,写入CDF数组, 例如count array中0,1,2出现1,2,2次,则CDF中0,1,2位置记为1,1+2=3,1+2+2=5
- 根据CDF来填写output array并减去CDF中的值,例如CDF中2位置的值为5,表示在output array中pos=5的位置一定是2,则在pos=5放入2,并在CDF中2处减1.

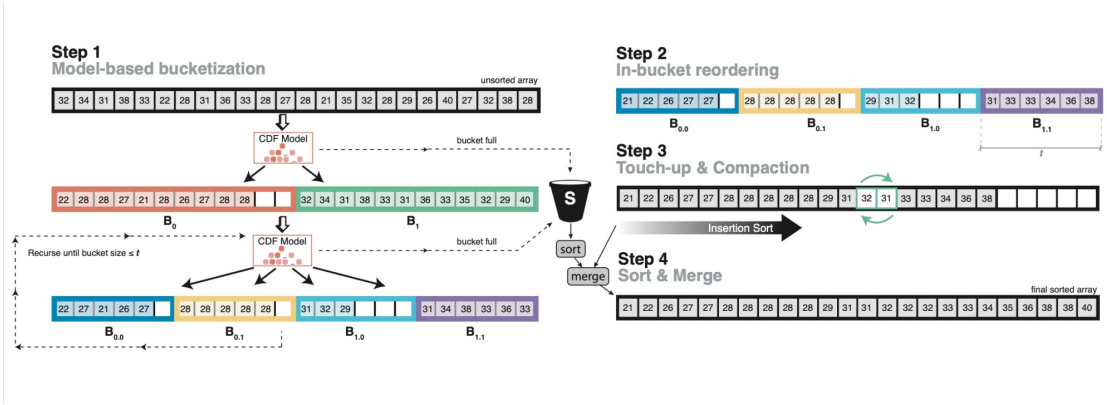
使用AI模型来优化排序
将排序任务作为预测任务,根据元素的值预测在数组中的位置
pos <- ECDF(key) * N



)

- 使用RMI来作为预测模型
- 为了防止预测位置出现冲突或前后颠倒,使用bucket存储区间内的元素,随后bucket内进行排序

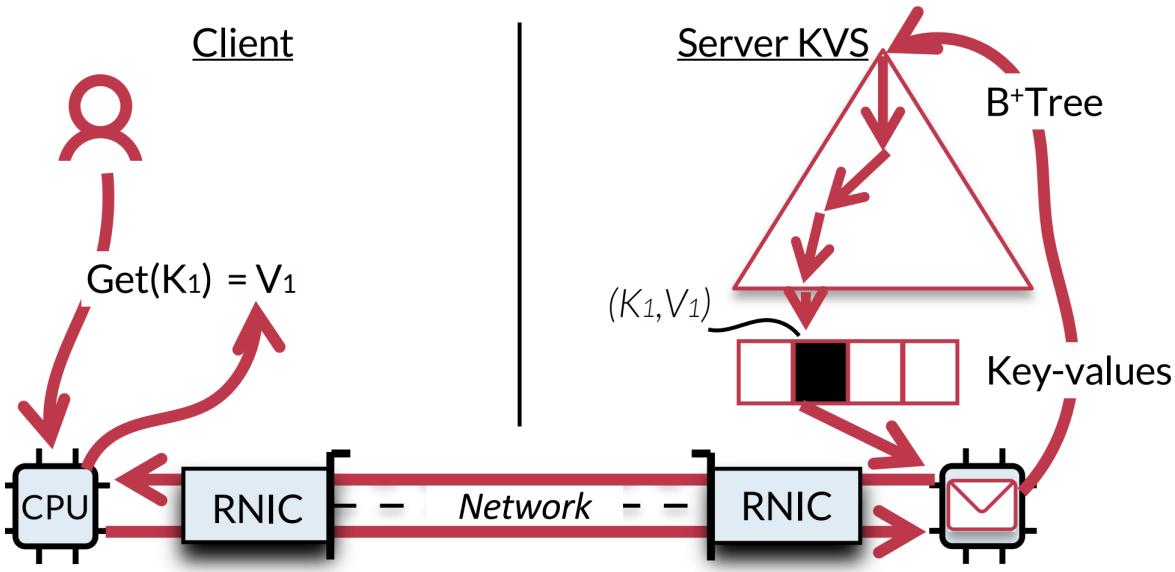
- 当bucket填满后,将额外的bucket扔到单独的数组中
- 在bucket内部,使用快速排序进行排序



)

使用模型加速基于RDMA的key-value存储查找

传统的分布式KVS

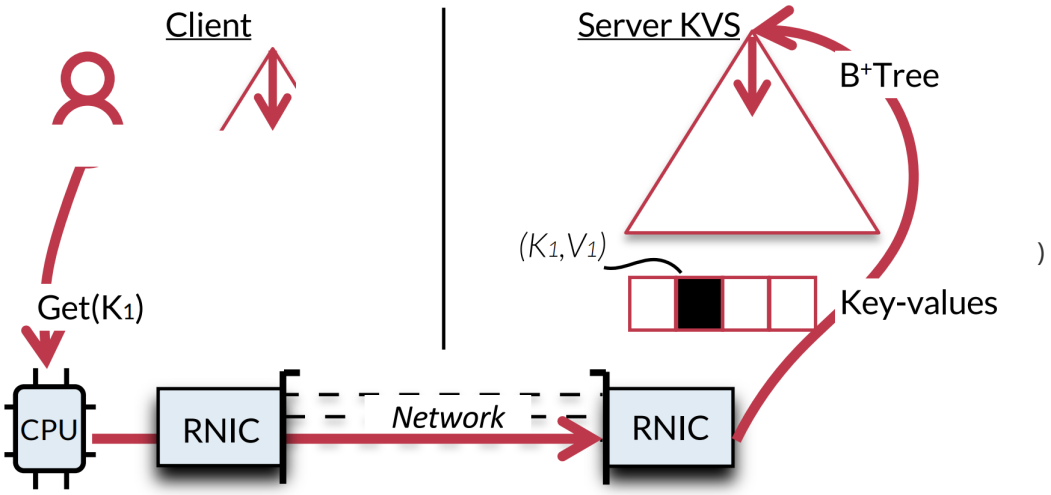


)

缺点:

- 服务器CPU是瓶颈
- CPU能耗是另一个问题,因为CPU必须使用polling来处理消息实现低延迟

使用NIC查找的KVS



)

122

使用NIC直接读写内存,将服务器CPU绕过
缺点:

- NIC只具有简单的抽象,例如内存读/写,对于简单索引例如hash可以使用,对于复杂索引例如B+Tree在一个RTT中一次只能读写一层
- 对于100M的KVS,至少需要7个RTT才能查找到key位置

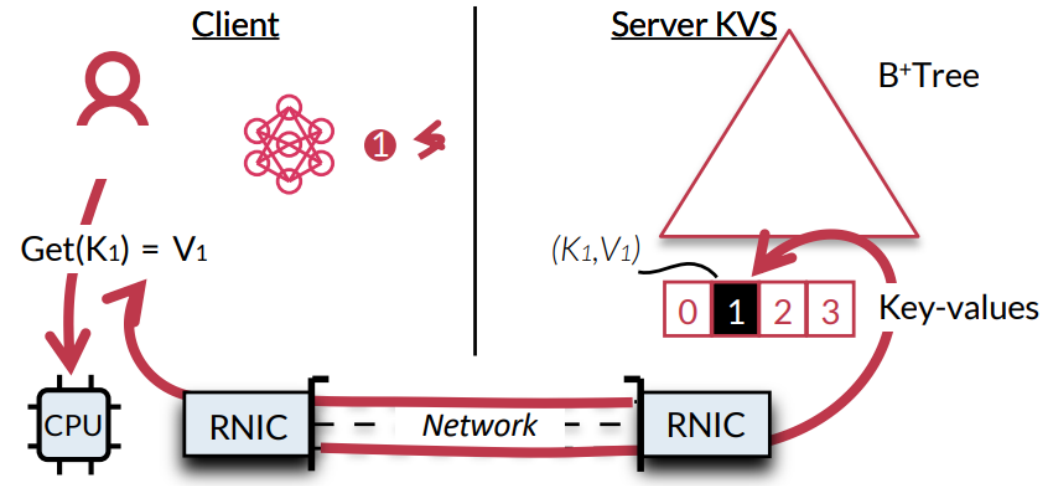
客户端缓存B+Tree来查找KVS
缺点:

- B+Tree索引会消耗很大的客户端存储
- Tree node大小可能比KV还要大
- 插入元素后客户端不能及时更新

小结:

- 服务端为中心的KVS: 高CPU利用率
- 客户端通过NIC: 低性能
- 客户端cache: 高存储

使用模型的KVS:XSTORE
使用模型直接计算key在服务端内存中的地址



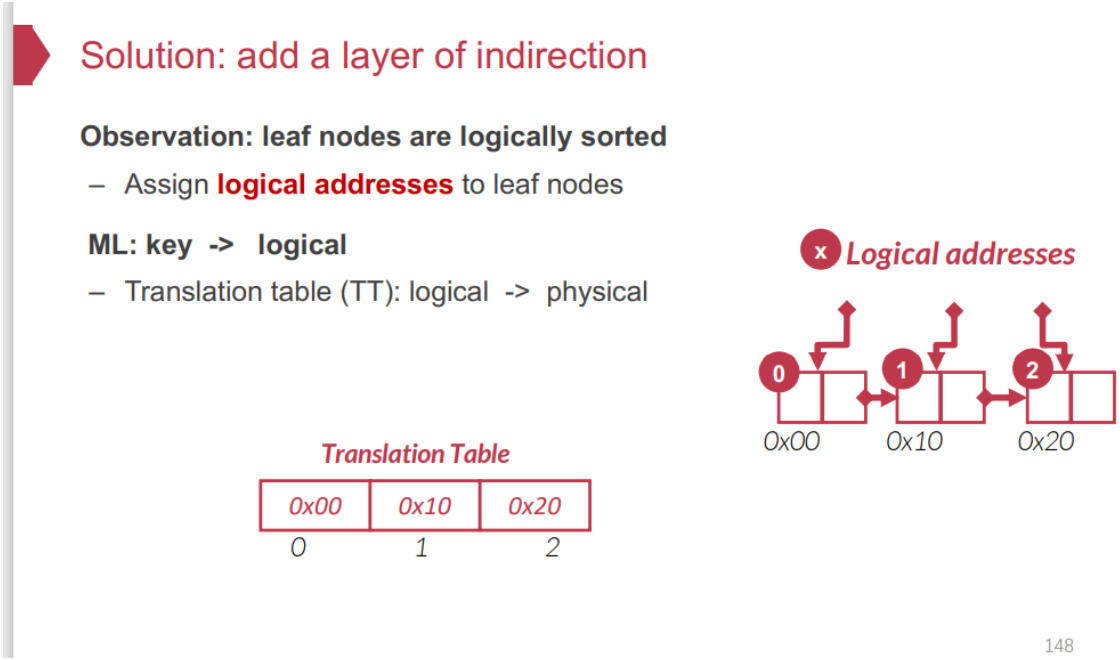
)

- 通过服务端B+Tree学习key->address的映射
- 客户端cache中的模型通过key获取在KV中key-value的地址

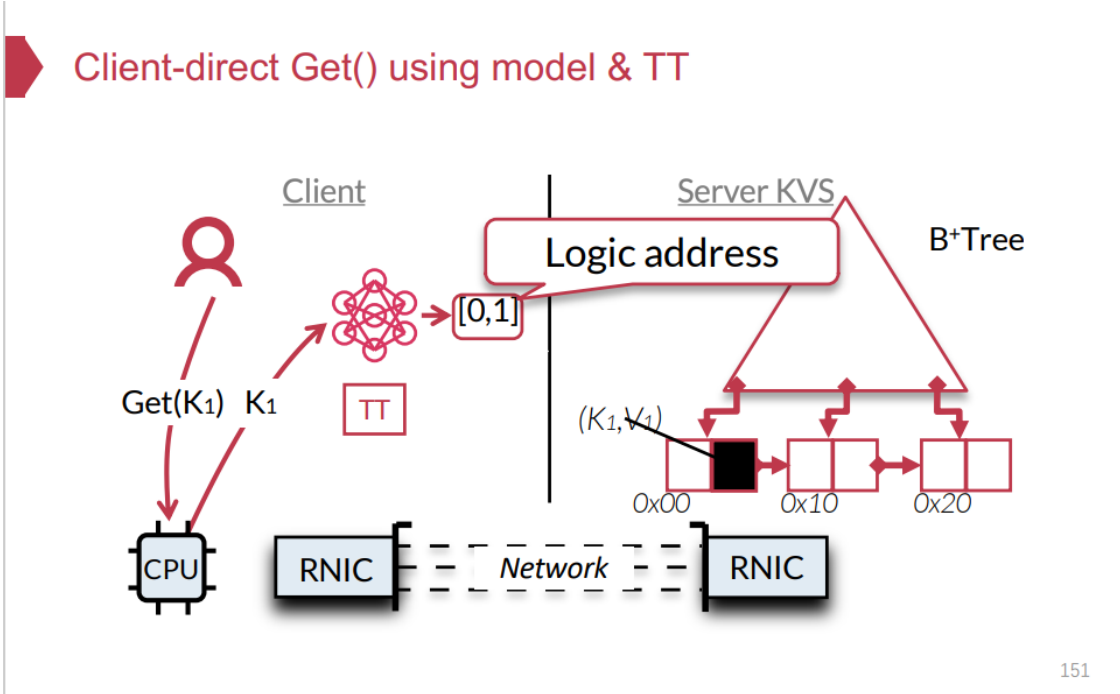
- 通过NIC读取对应value

XSTORE面临的问题

- 频繁的插入元素会使得KV地址改变
  在有序元素中插入新元素会改变原来KV的地址使得模型无效
  解决办法: **Translation Table(TT)**,使用逻辑地址来表示key的地址, 增加TT来存储逻辑地址和物理地址的映射



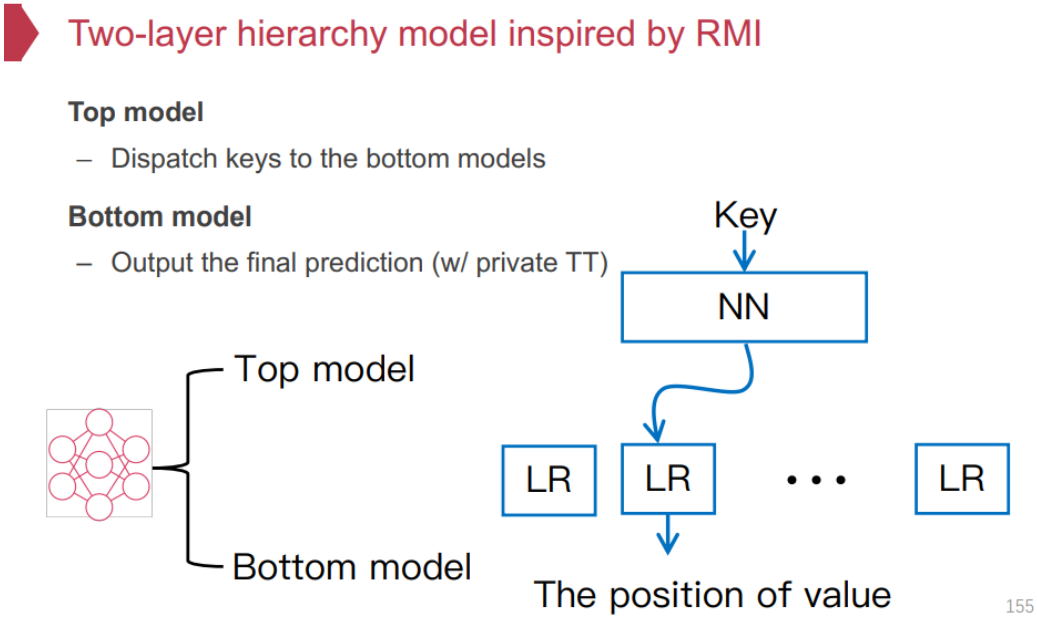
)



)

- 插入/删除会使得原先训练模型无效
  解决方法:1. 两层结构的模型 2. 特定领域选择特定模型 3. 过期检查和快速回退

- 两层模型架构,top model使用NN,bottom model使用LR



)

top model为key分发对应的bottom model,bottom model即LR选择具体的地址,由于LR单调和TT,只有B+Tree节点分裂的时候模型才会过期.

- 特定领域选择特定模型
  - 顶部使用NN:1.高准确性 2.在元素插入后不需要重新训练
  - 底部使用LR(线性回归):1. 重新训练速度快,无需迭代训练 2.可以使用很多LR提高预测准确性
- 过期检查和回退
  - 失效的情况:只有当Btree节点分裂时，模型才会失效
  - 过期检测:
    - 在TT中编码Btree序列号
    - 将TT的版本与fetch节点的序列匹配
    - 如果不匹配，回退到RPC,重新获取模型

XSTORE使用一个后台线程训练模型,当插入导致分裂时重新生成模型

14. Security: Attack and Defense

在任何安全系统中,最脆弱的环节是人

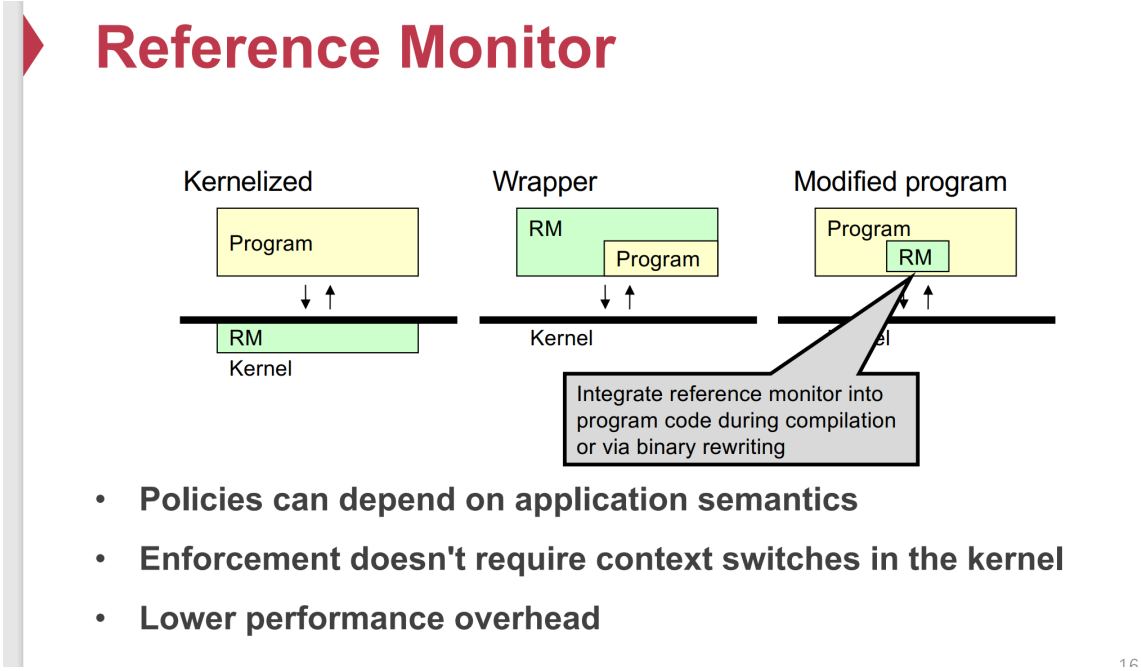
Trusted Computing Base (TCB):

- 需要要有信任的起点, 即TCB
- 其他的信任机制基于root of trust
- TCB可以是人, 软件, 硬件
  - 人: 制造商, 关键组件的开发者
  - 软件: BIOS, VMM, kernel loader
  - 硬件: TPM, secure processer

OS作为Reference Monitor(RM):

- OS是运行process(subjects)和文件files(objects)的集合
  - 进程与user关联
  - 文件有 access control lists(ACLs)控制
- OS执行一系列安全策略

    - 文件需要通过ACL检查
    - 进程不能在其他进程内存中写操作
    - 一些操作需要superuser权限
- 对于同一用户所有进程使用同一个策略



)

什么使得一个进程是安全的?

- 内存安全: 不越过数组边界，不要操作另一个进程的内存，不要像执行代码一样执行数据
- 控制流安全: 所有的控制传输都是由原程序所设想的
- 类型安全: 所有函数调用和操作都有正确类型的参数

隔离:

- 每个进程应该存在于自己的地址空间中, 但是进程间通信的性能成本(IPC)正在增加
- 上下文切换非常昂贵
- 捕获到OS内核需要刷新TLB和缓存，计算跳转目的地，复制内存

Software Fault Isolation (SFI):

- 进程位于相同的硬件地址空间中;软件RM将它们隔离开来
    - 每个进程被分配一个逻辑"fault domain"
    - 检查所有的内存引用和跳转，确保它们没有离开进程的域
- 权衡:检查成本 vs. 通信成本
    - 为每次内存写和控制转移付出执行检查的成本
    - 当陷入到kerne时，节省上下文切换的成本

Control flow attack 控制流攻击

举例:ROP攻击 Return-oriented Programming

- 在已有代码基础上找到code gadgets, 通常gadgets由1-3个指令组成,ret结尾
- 将这些gadgets的地址压到栈上
- 通过gadgets末尾的ret连接执行各个gadgets
- 实现无代码注入的攻击

如何防范这种攻击?

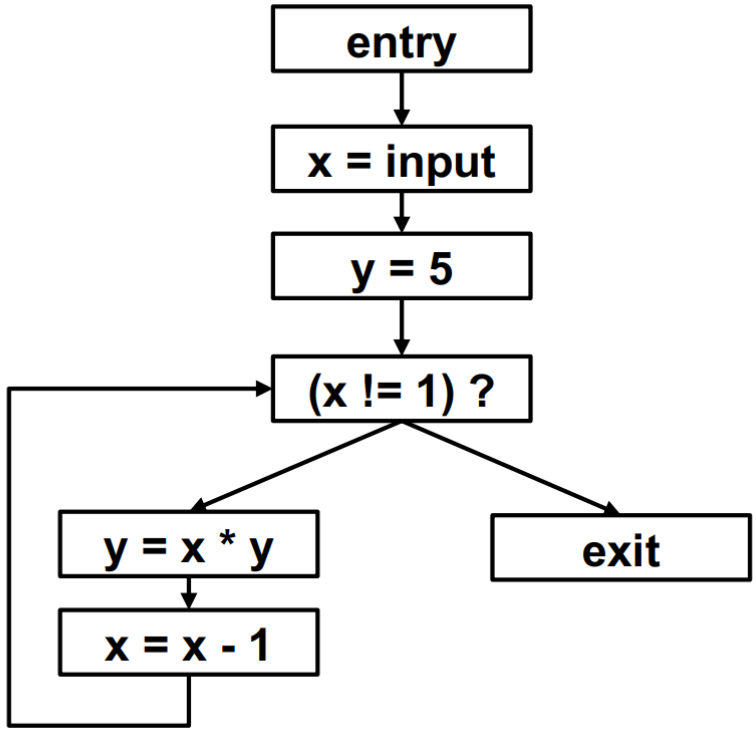
- 隐藏二进制文件

- ASLR(Address space layout randomization)来随机化代码地址
  - 改变内存空间布局
  - 每一次进程创建时随机
- Canary金丝雀机制保护栈, 预防栈溢出攻击
  - 将金丝雀,即一段随机数字,放入栈帧中(返回地址下),在每次函数返回时检查是否被改变
  - StackGuard作为GCC的补丁实现

CONTROL-FLOW INTEGRITY (CFI)

什么是control flow graph?

- 每个函数中的代码被分割为多个基本块(Basic Block,BB),每个BB只有一个入口
- CFG中的顶点: 基本块
- CFG中的边: 控制依赖



)

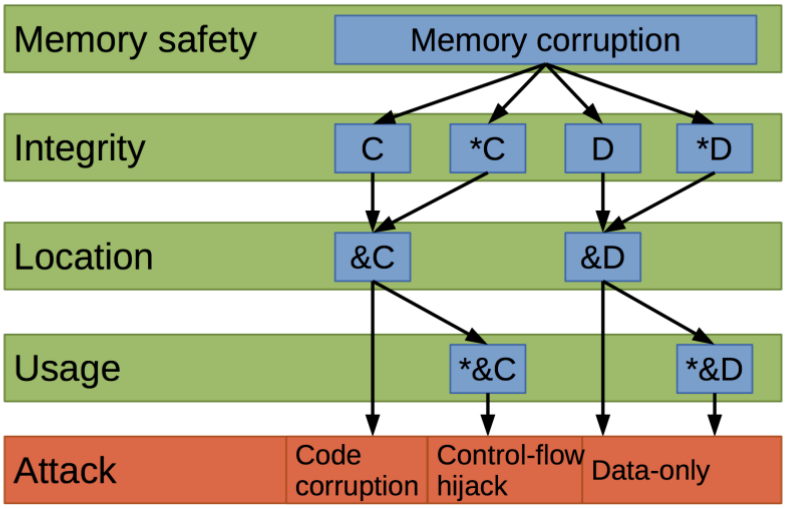
什么是CONTROL-FLOW INTEGRITY(CFI)?

- **main idea: 预先确定一个应用的CFG**
  - 对源码进行静态分析
  - 静态二进制分析 <-CFI
  - 执行采样
  - 明确安全策略
- **策略: 程序执行必须路线必须在预先确定的CFG内**
- **方法: 二进制插桩**
  - 使用二进制重写来进行插桩检查运行时代码
  - 插入检查代码确保执行总在确定的CFG内
  - 确保跳转目标是CFG内合法目标
  - 目标: 即使攻击者完全控制线程的地址空间也能确保安全
- **CFI控制流执行**
  - 对于每个控制转移，静态地确定其可能的目的地
  - 在每个目的地插入一个唯一的位模式(bit pattern)
  - 两个目的地是等效的，如果CFG包含从同一来源出发的边

    - 对于等价的目的地，使用相同的位模式
- **CFI安全保证**
    - 有效抵御基于非法控制流传输的攻击
    - 不能保护不违反程序的原始CFG的攻击

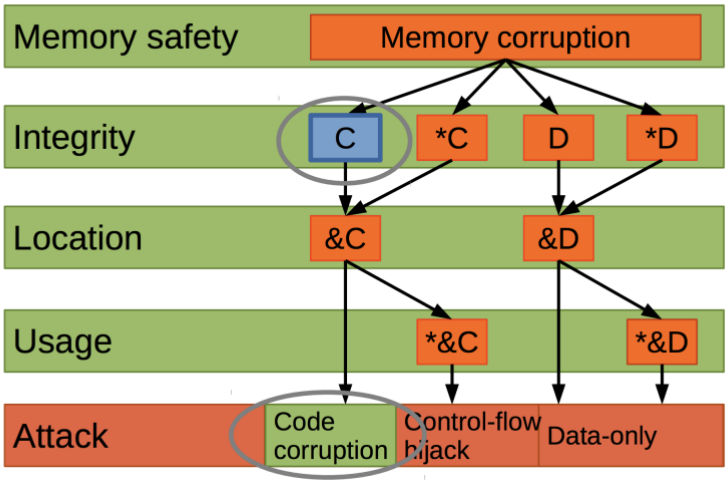
内存攻击模型

Memory Attack Model



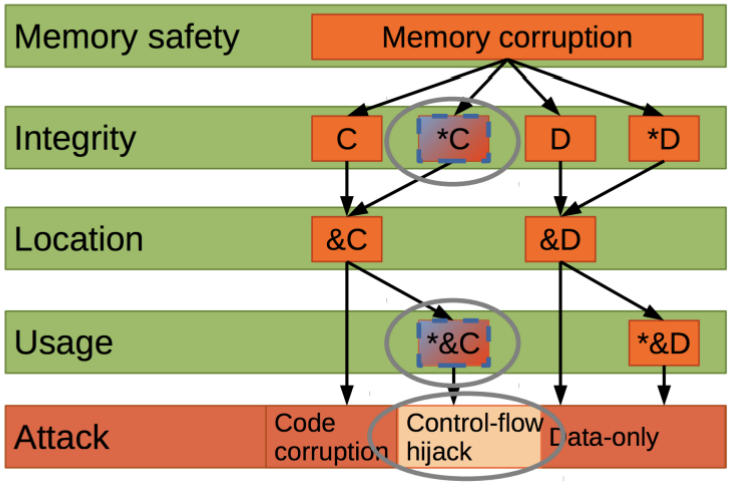
)

Data Execution Prevention, W ^ X



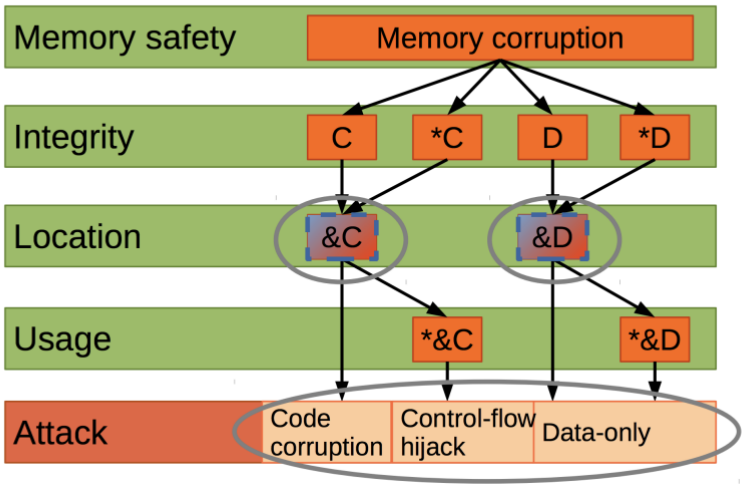
)

Stack Canaries and CFI



)

Address Space Layout Random



)

TAINT CHECK

TaintCheck的主要思想:

- 程序执行通常来自可信的来源，而不是攻击者的输入
- 所以外来输入可能是攻击源, 对所有输入计算机的数据进行 tainted(污染)
- 监视程序执行并跟踪受污染数据的传播方式(跟随字节、算术操作、跳转地址等)
- 检测被污染的数据何时以危险的方式使用

TaintCheck首先通过一个模拟环境(Valgrind)运行代码，并添加指令来监视受污染的内存, TaintCheck检测主要包含3个模块:

- **TaintSeed**: 将不受信任的数据标记为污染
- **TaintTracker**: 跟踪每个指令,确定结果是否被污染
- **TaintAssert**: 检查污染数据是否用于危险的地方,比如
  - 跳转地址: 函数指针
  - 格式化字符串: 是否使用污染数据作为格式化字符串参数
  - 用于系统调用

TaintCheck Detection Modules

- **TaintSeed:** Mark untrusted data as tainted
- **TaintTracker:** Track each instruction, determine if result is tainted
- **TaintAssert:** Check is tainted data is used dangerously

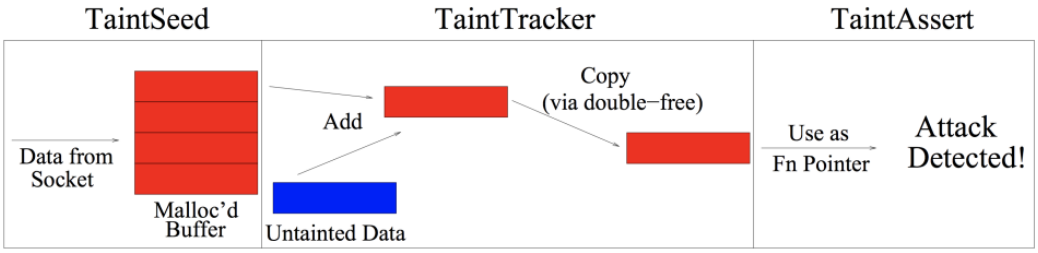


Figure 1. TaintCheck detection of an attack. (Exploit Analyzer not shown).

)

TaintCheck可能出现假阳性问题,即使没有攻击也会报使用非法,这个特性可以用来检查系统薄弱环节.

TAINT DROID

背景知识:Android:

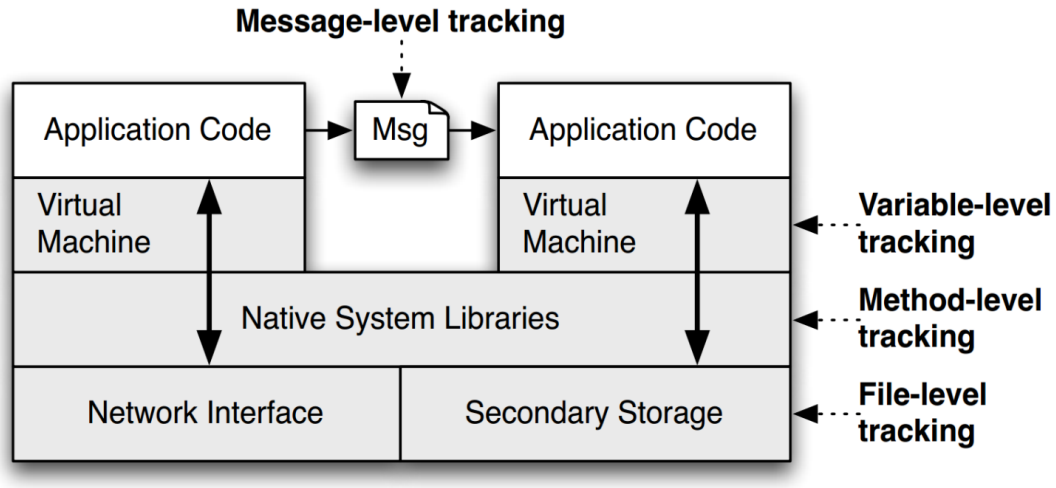
- Android中应用使用Java编写
- Java native interface(JNI)用于调用C++等底层库或系统API
- 源代码编译为Dalvik Executable(DEX)字节码形式
- 在Dalvik VM解释器中执行
- 组件间通过binder IPC机制通信

动机: 更好的保护当前Android系统隐私
和TaintCheck一样,使用某些方法对数据进行污染并跟踪

Taint Source 数据来源:

- 低带宽传感器: 地理信息, 加速计
  - 经常更改，并被许多应用程序使用
  - 大多数智能手机操作系统都有某种类型的管理器来多重处理这些信息
- 高带宽传感器: 麦克风, 相机
  - 返回大量的数据，并且一次只能被一个应用程序使用, hook放置在数据缓冲区中
- 信息数据库: 地址信息,SMS, 通话记录
- 设备标识符: IMEI
- 网络
- WiFi: 3G, SMS

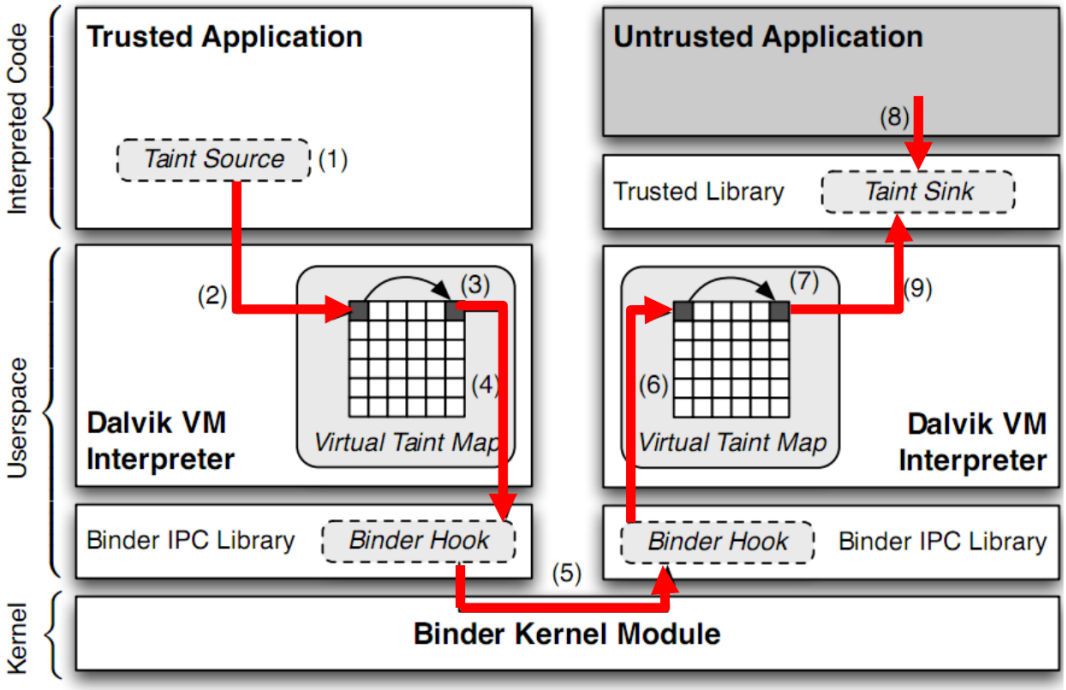
Taint Droid架构:



)
TaintDroid跟踪以下级别数据

- 消息级别
  - 使用Binder IPC 机制
  - 标记整个消息而不是标记变量
- 变量级别
  - 最主要的追踪方法
  - 仅对一个数组保存一个tag,可能导致假阳性
- 方法级别
  - VM内部方法: 使用较少
  - JNI方法
- 文件级别
  - 污染tag存储在扩展的文件属性中
  - 整个文件被标记
  - 变量访问时污染从文件传播到变量

Taint Droid流程:



)

1. 污点接口调用一个native方法
2. 此方法interface一个native解释器
3. 其中virtual taint map存储污点标记tag
4. 受信应用使用污染数据IPC通信时, 通信包中也包含污点tag
5. 使用binder kernel module传输, 被不可信app接收
6. 远程binder会从信息包接收污点tag,标记本地map
7. 传给动态库
8. 不可信app调用动态库,使用污染数据,被检测并报告

Taint Droid贡献
TaintDroid为每一个被测试的应用程序生成有用的结果, 并实现了一个有用的隐私分析工具

缺点:

- TaintDroid可以通过隐式信息流来规避
- TaintDroid不能告诉是否被污染的信息在离开后重新进入手机
- 可以使用侧信道(side-channel)攻击绕开
- 可以使用下面方法避免数据污染

```
if (x == 0) y = 0;
else if (x == 1) y = 1;
...
else if (x == 255) y = 255;
```

PUMP 基于hardware的安全

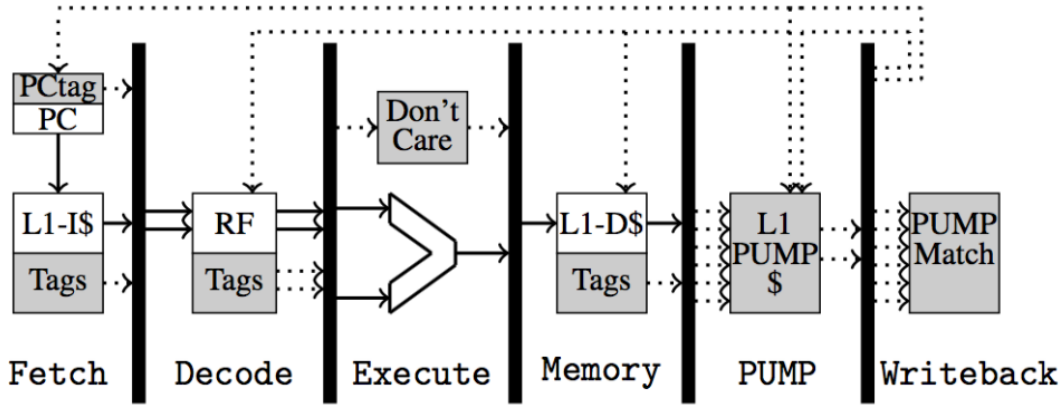
目标:

- 用户定义的元数据处理
  - 处理任意元数据大小
  - 任意策略
  - 低开销:低性能和硬件资源开销

PUMP: programmable unit for metadata processing

- 每个word有一个指针大小的tag
  - 小tag直接存储,大tag存储在内存中
- 标记内存, 缓存, 寄存器, PC等的使用
- tag不可被寻址

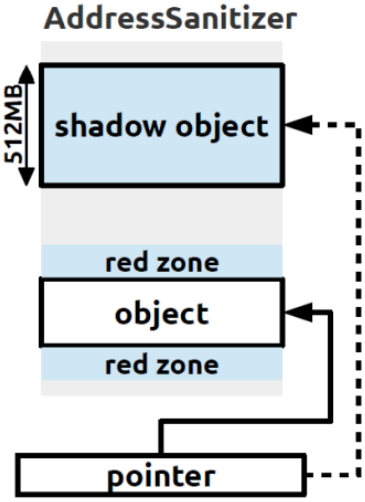
Pipeline Integration



)

ADDRESS SANITIZER 基于compiler的安全

ASAN Design



- 1/8 of the memory are use to save the shadow object.

)

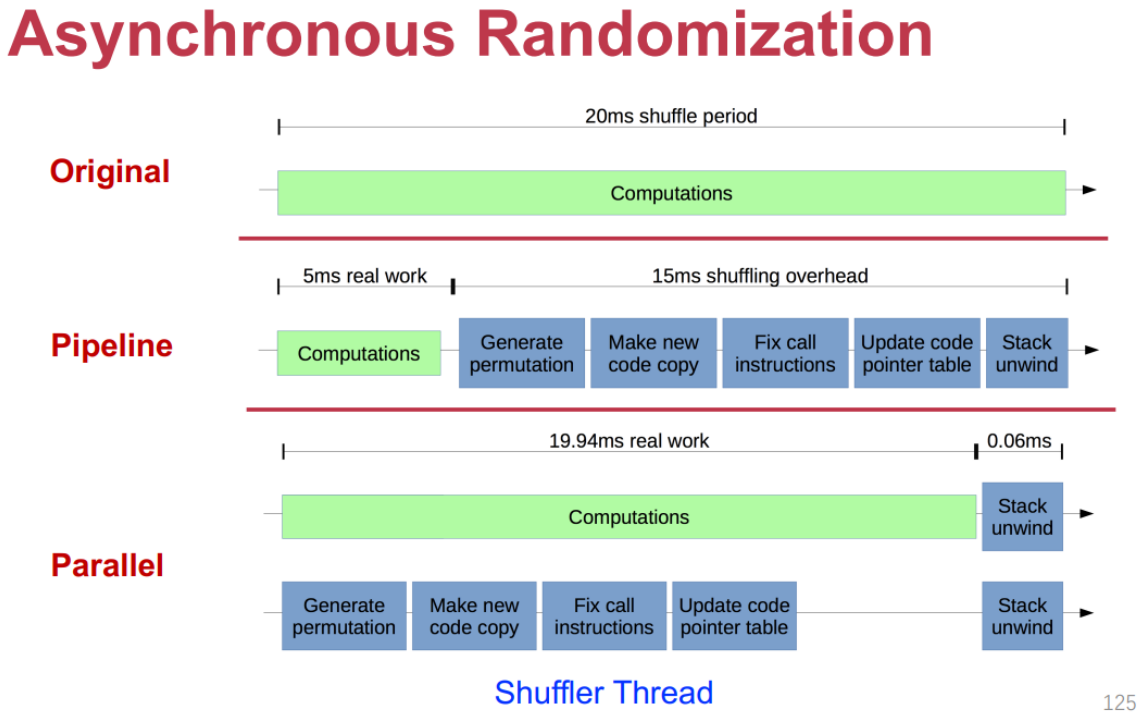
- 追踪内存访问的边界
- 主要包括两部分：插桩(Instrumentation)和动态运行库(Run-time library)。插桩主要是针对在llvm编译器级别对访问内存的操作(store，load，alloca等)，将它们进行处理。动态运行库主要提供一些运行时的复杂的功能(比如poison/unpoison shadow memory)以及将malloc,free等系统调用函数hook住。该算法的思路是：如果想防住Buffer Overflow漏洞，只需要在每块内存区域右端（或

两端，能防overflow和underflow）加一块区域（RedZone），使RedZone的区域的影子内存（Shadow Memory)设置为不可写即可。

SHUFFLER 基于系统运行时的安全

主要思想:

- 连续Re-Randomization,持续随机代码地址
- 在攻击者使用代码之前重新随机化代码
  - 比泄露漏洞执行时间更快;
  - 比小工具链计算时间快;
  - 或者，比网络通信时间更快



125

)

异步随机化

- 将新代码拷贝, 修改call指令, 更新code pointer table放入shuffler thread异步执行,最后同步修改代码栈
- 同步时间只占全部时间0.3%

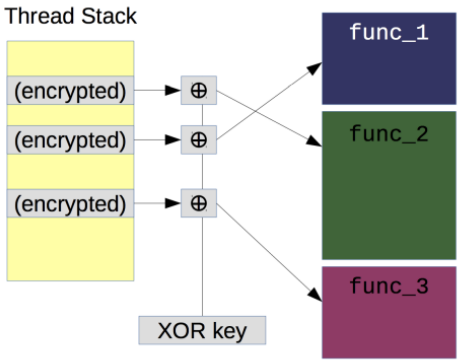
返回地址加密

- 使用正常呼叫指令
- 用XOR密钥加密返回地址
- 防止泄露返回地址

- 使用二进制重写

Return Address Encryption

- **Prevent return address disclosure**
- **Use binary rewriting (expand basic blocks)**



)

15. Hardware Security

Hardware Features Designed for Security

SMEP/SMAP: Supervisor Mode Execution Prevention/Supervisor Mode Access

Prevention

Return-to-user Attack: 内核代码中某些函数指针被设为NULL，dereference时跳转到用户代码。通过SMEP/SMAP禁止内核访问/执行用户代码。

- SMEP: Allows pages to be protected from supervisor-mode instruction fetches. If SMEP = 1, OS cannot fetch instructions from application
- SMAP: Allows pages to be protected from supervisor-mode data accesses. If SMAP = 1, OS cannot access data at linear addresses of application
- ARM的类似技术：PAN, Privileged Access Never; PXN, Privileged eXecute Never; UAO, User Access Only

应用：使用SMAP做进程内隔离，将用户代码放在内核态，需要保护的数据放在用户态

ret2dir Attack: 每一页物理内存都同时拥有kernel和user两个虚拟地址，虽然禁止内核执行用户代码，但通过跳转到恶意代码对应的kernel地址仍可攻击。

MPX: Memory Protection eXtension

Background: C/C++ bounds error (可通过gcc的-fcheck-pointer-bounds flag防止)

Intel introduces MPX since Skylake

- Specified by two 64-bit addresses specifying the beginning and the end of a range
- New instructions are introduced to efficiently compare a given value against the bounds, raising an exception when the value does not fall within the permitted range
    - bndmov: Fetch the bounds information (upper and lower) out of memory and put it in a bounds register.
    - bndcl: Check the lower bounds against an argument (%rax)
    - bndcu: Check the upper bounds against an argument (%rax)
    - bnd retq: Not a "true" Intel MPX instruction
- usage: make CFLAGS="-mmpx -fcheck-pointer-bounds -lmpx" LDFLAGS="-lmpxwrappers -lmpx"
- 将bounds存在bounds table(a two-level radix tree)中，最坏情况可多消耗400%内存，且使用较多bound时性能降低

MPK: Memory Protection Keys

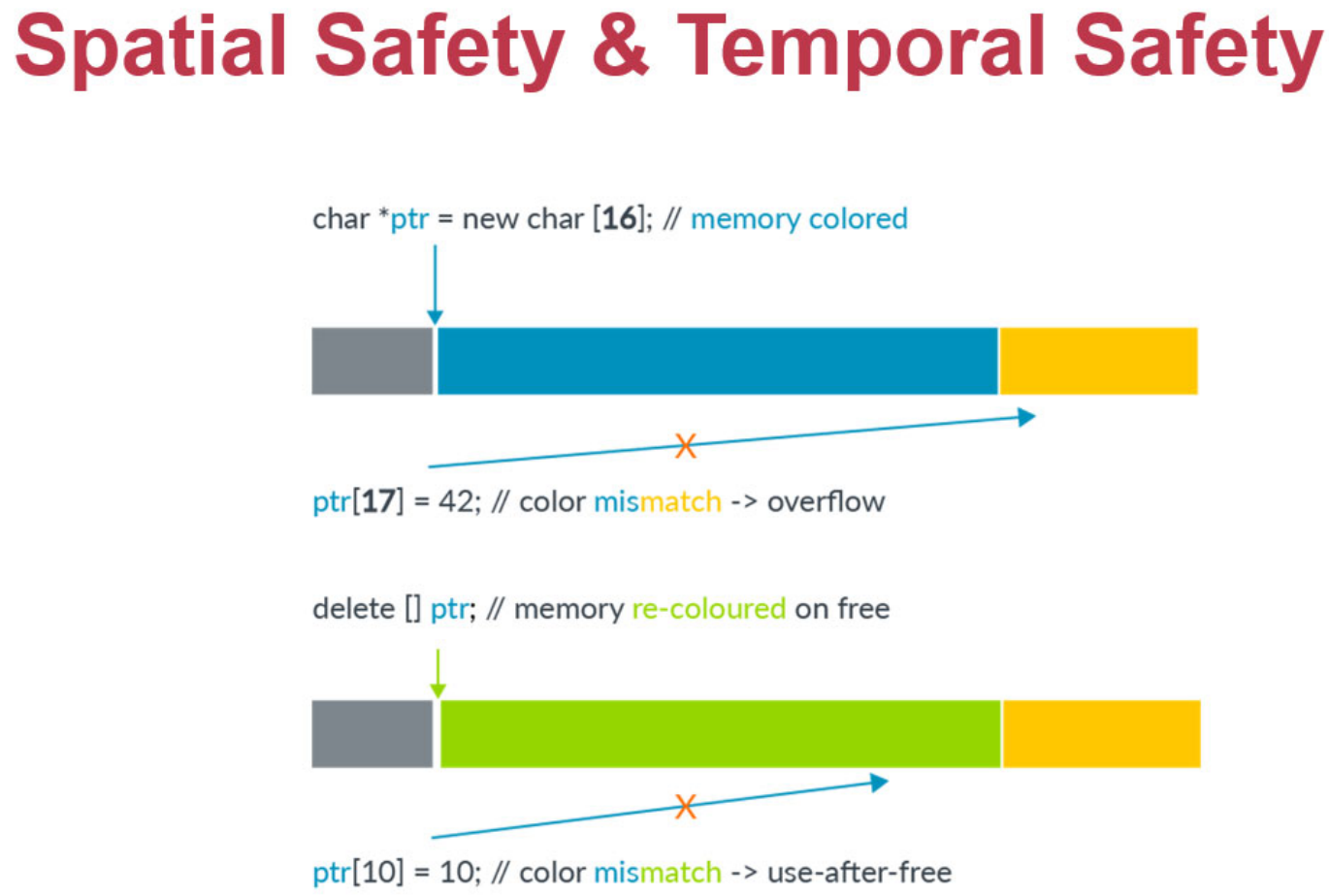
- With MPK, every page belongs to one of 16 domains. A domain is determined by 4 bits in every page-table entry (referred to as the protection key)
- For every domain, there are two bits in a special register (pkru), which denote whether pages associated with that key can be read or written
- Only the kernel can change the key of a page, application can read and write the pkru register using the rdpkru and wrpkru instructions respectively
    - use case 1: protect critical data within one address space
    - use case 2: prevent data corruption (In-memory database prevents writes most of the time, only enable changing data when needs to change)

ARM PA: Pointer Authentication

- ARM只使用了64位中的40位，可以用key对这40位地址进行加密，保存在前24位，从而作为验证，防止地址被篡改
- PA defines five keys
    - Four keys for PAC* and AUT* instructions (combination of instruction/data and A/B keys),
    - One key for use with the general purpose PACGA instruction

- Keys are stored in internal registers and are not accessible by EL0 (user mode)
- New instructions:
  - PAC value creation: Write the value to the uppermost bits in a destination register alongside an address pointer value
  - Authentication: Validate a PAC and update the destination register with a correct or corrupt address pointer. If the authentication fails, an indirect branch or load that uses the authenticated, and corrupt, address will cause an exception

ARM MTE: Memory Tag Extension



- 引入新的内存类型：Normal Tagged Memory，只允许指针访问相同tag的内存
- tag占4位，即0~15
- 不是所有内存访问都需要tag checking，如instruction fetches,translation table walks等
- MTE and PA可结合使用

Intel CET: Control-flow Enforcement Technology

- Shadow stack: a second stack for the program
  - 该栈只记录控制数据(return address), 与原有的stack同时push和pop，如果RET时两个地址不一样，则引发control protection exception
  - 被页表保护
- Indirect Branch Tracking: New instruction: ENDBRANCH
  - call/jmp跳转到的指令必须以ENDBRANCH开头，否则无效
  - cpu维护了一个状态机跟踪indirect call/jmp, 当出现这些指令，状态由IDLE转为WAIT_FOR_ENDBRANCH, 该状态下下一条指令必须为ENDBRANCH，否则报错

Trusted Execution Environment

XOM: eXecute-Only Memory

- 代码和数据在内存中加密
- 存储加密值的哈希

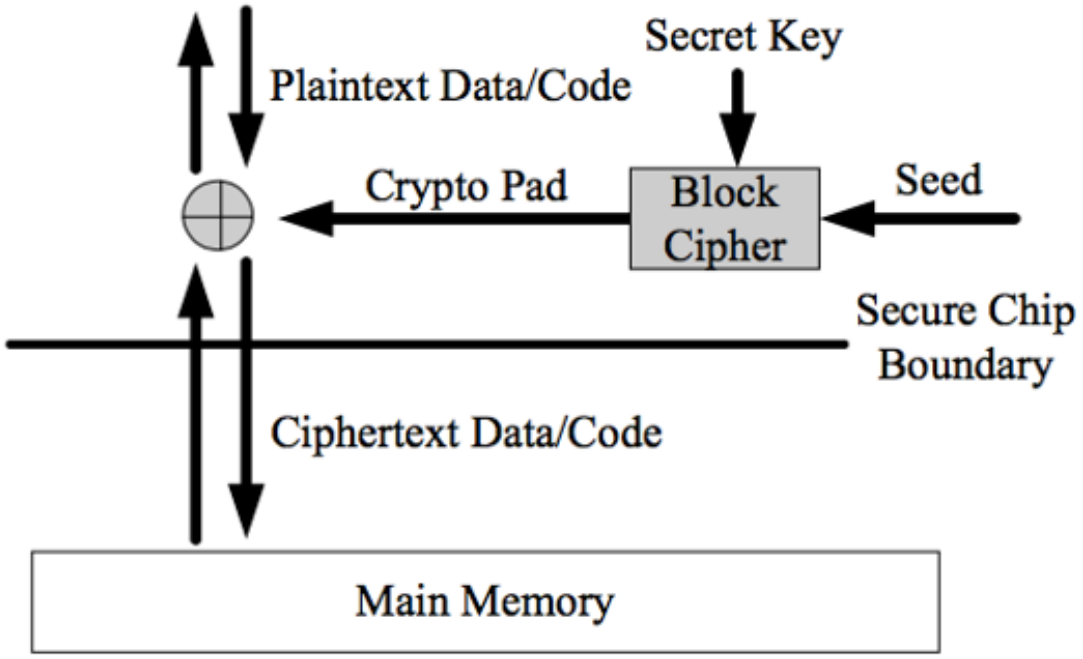
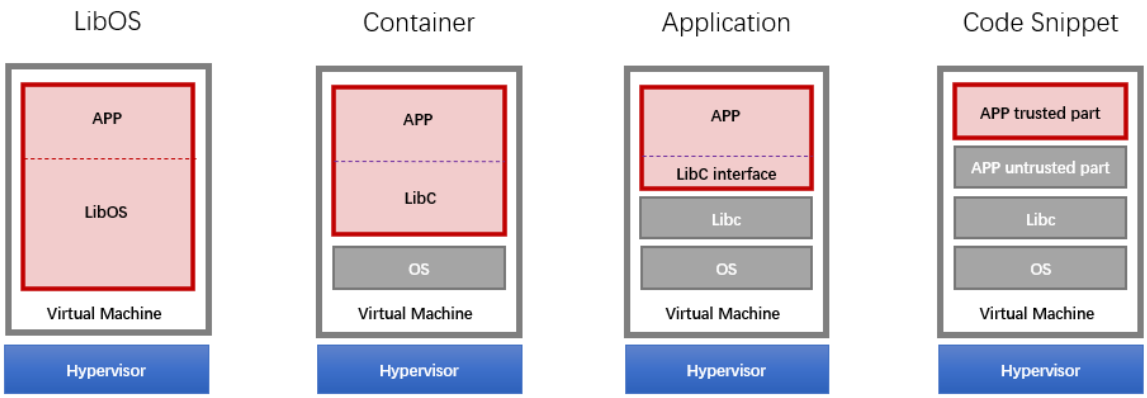


Figure 1. Counter-mode based memory encryption.

Intel SGX

- SGX Execution Flow
  - App built with trusted and untrusted parts
  - App runs & creates the enclave which is placed in trusted memory
  - Trusted function is called, execution transitioned to the enclave
  - Enclave sees all process data in clear; external access to enclave data is denied
  - Trusted function returns; enclave data remains in trusted memory
  - Application continues normal execution

Software Architectures of SGX

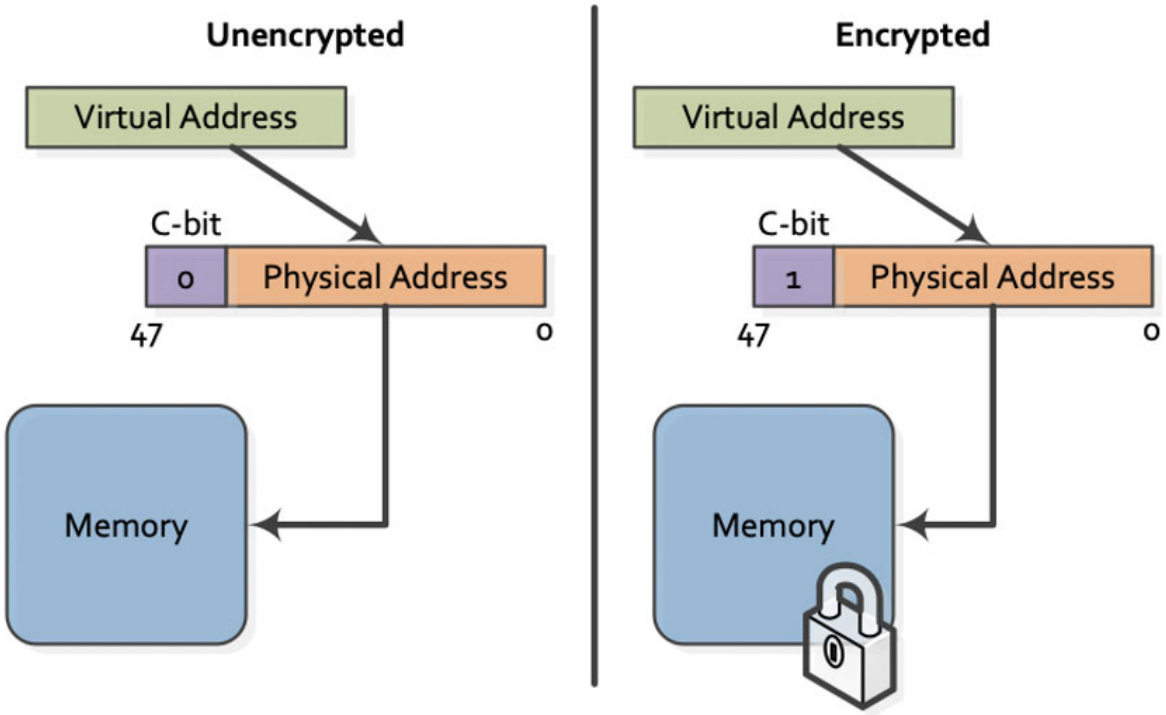


	Compatibility	TCB Size	Ocall Num	Attack Surface	Protect OS
LibOS	Part	Large	Few	Small	✔
Container	✔	Mid	Mid	Mid	✘
Application	✔	Mid	Many	Large	✘
Code Snippet	✘	Small	Few	Small	✘

-

AMD's SME/SEV: Secure Memory Encryption/Secure Encrypted Virtualization

- Features



  -
  - Hardware AES engine located in the memory controller performs inline encryption and decryption of DRAM
  - Minimal performance impact: Extra latency only taken for encrypted pages
  - No application changes required
  - Encryption keys are managed by the AMD Secure Processor and are hardware isolated, not known to any software on the CPU
- Comparing with Intel SGX
  - SME不会防范内核，用来防御cold-boot attacks，以及非易失内存的数据泄露

    - SEV专注于虚拟机，可以防御其他虚拟机以及宿主机
- 类似技术: Intel MKTME: Multi-Key Total Memory Encryption, 支持多个key加密

ARM TrustZone

- Two modes: Normal world (REE, rich execution environment) and secure world(TEE, trusted execution environment), SMC instruction to switch
- 总线上增加1位，外设可以区分请求来自哪个world
- 应用: 手机指纹识别、交通工具、无人机禁飞区

RISC-V PMP/sPMP: Physical Memory Protection

- 通过一组PMP registers将物理内存划分为互相隔离的区域，每一段属于一个enclave
- 由于PMP registers数量有限，enclave数量也受限

Penglai

- Enclave on RISC-V ISA
- 为了实现细粒度内存划分，在DRAM增加一个bitmap，每一位表示一个页是否安全
- 所有不安全的页存储在一个隔离的内存区域PT_AREA
- 使用cache partition防御侧信道攻击
- 通过签名证明确实运行在enclave

Hardware Features Not Designed for Security

Intel TSX: Transactional Synchronization eXtensions

- Programming with RTM(restricted transactional memory)
    - If transaction starts successfully, do work protected by RTM, and then try to commit
    - If abort, system rollback to _xbegin, return an abort code
    - Manually abort inside a transaction
- 使用HTM保护数据: 将数据放在transaction中，利用HTM保证的原子性防止其他并发访问
- 利用HTM攻击KASLR: KASLR技术用来随机化内核地址。用户态随机访问一个地址，有两种情况，一是未映射，二是内核地址空间，两者返回segmentation fault有时间差。将这种试探代码放在transaction中，abort速度极快，可以快速试探出内核地址

Intel CAT: Cache Allocation Technology

The "Noisy Neighbor" Problem: 某些应用使用大量内存，但本身对cache需求不高（例如流媒体播放，之前的帧没有用处），反而占据了其他需要利用cache的应用的cache容量，影响性能。

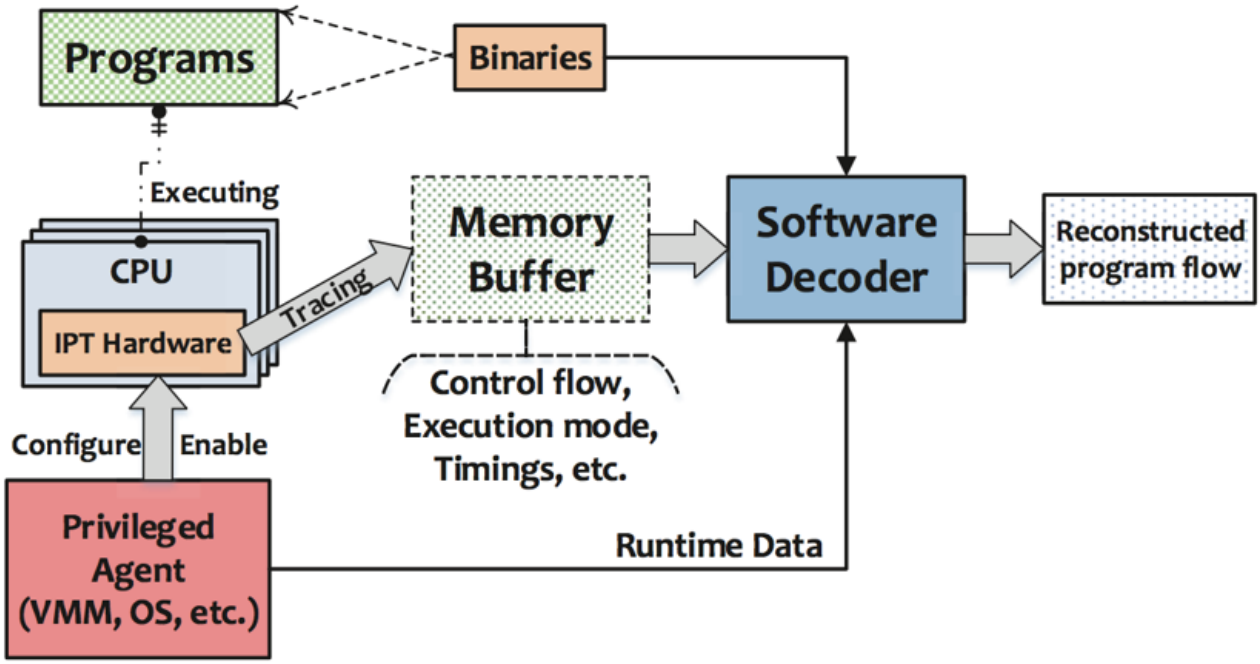
- CAT将thread / app / VM / container通过Class of Service (CLOS)分组，每个CLOS有相应的resource capacity bitmasks (CBMs)，表示可以使用cache的哪一部分
- 通过CAT防御基于cache的侧信道攻击（The PRIME+PROBE Attack，先用随机数据将cache填满，然后触发加密程序，最后重新访问原数据，通过cache miss情况推算加密行为），将不可信应用隔离，只能使用部分cache

PMU: Performance Monitor Unit

- BTS: Branch Trace Store, 记录程序所有地址跳转
- Motivation: Code Injection Attack, Code Reuse Attack
- 做法：利用PMU监测CFI
    - Offline phase: 记录所有可能的分支跳转
    - Online phase: 将实际跳转与合法跳转对比，发现恶意行为
    - 记录3种合法跳转
        - ret_set: all the addresses next to a call
        - call_set: all the first addresses of a function

- train_sets: all the target addresses that once happened

Intel PT: Intel Processor Tracing



- 增加了硬件对Trace进行压缩，使tracing很快，但decode很慢
- FlowGuard: 将压缩的实际trace与压缩的可信trace直接对比，如果无法判断再解压缩

16. Data Privacy

ZKP: Zero-Knowledge Proof

Problem: Alice想向Bob证明她has the answer A of the problem P，如果直接发送A，则Bob也知道了A

Zero-Knowledge Proof:

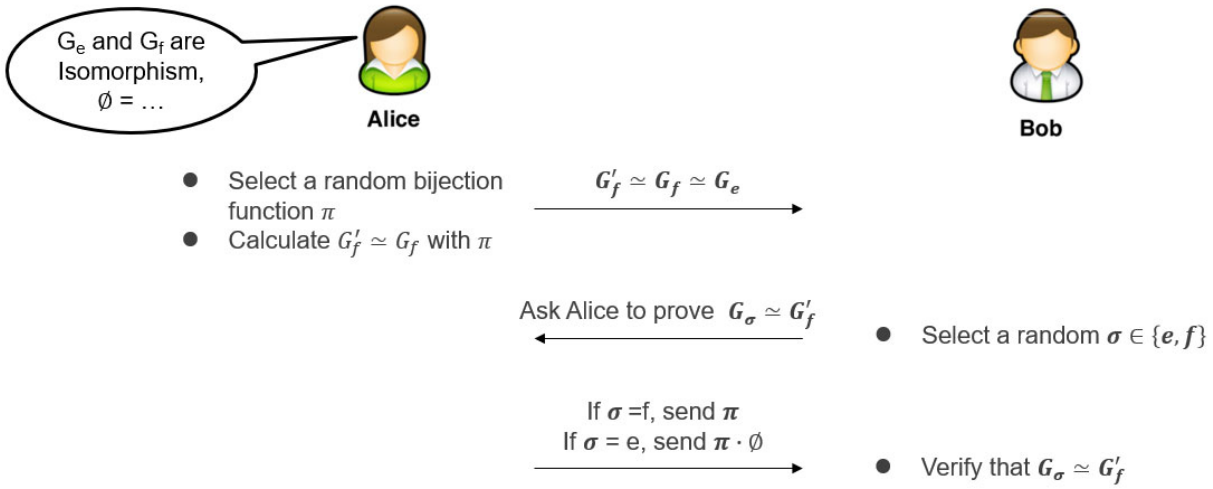
- Completeness: Alice can construct the proof if she has A
- Soundness: Alice cannot construct the proof if she doesn't have A
- Zeroknowledge: Bob knows nothing about A

Interactive Zero-Knowledge Proof: P has answer $\boldsymbol{x}$ of a problem $\boldsymbol{L}$, and tries to prove it with > 1 iterations:

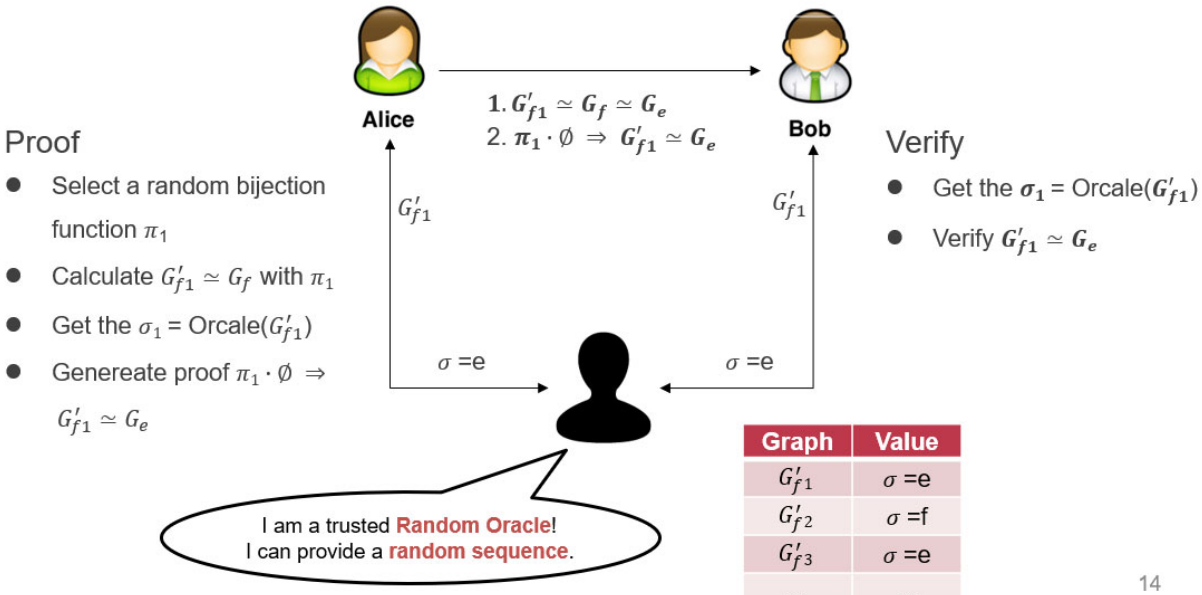
- Step-1: P transfers $\boldsymbol{L}$ to $\boldsymbol{L}'$, and promises that $\boldsymbol{L}'$ is transferred from $\boldsymbol{L}$ and she has the answer $\boldsymbol{x}'$
- Step-2: V challenges P
- Step-3: P shows the proof of the answer $\boldsymbol{x}'$, which will not leak $\boldsymbol{x}$
- V trusts that P has $\boldsymbol{x}$ when P always meets the challenge

图的同构：If G1 = (V1, E1) and G2 = (V2, E2) are isomorphic, there exist a bijection function（双射，即一一对应函数） $\boldsymbol{\phi}$, that for any $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E\_1}$, exist $\boldsymbol{\phi}(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E\_2}$，即两张图的每条边都一一对应

- 基于图同构的Interactive ZKP
  - Alice生成一个与两张图都同构的新图，反复询问迭代



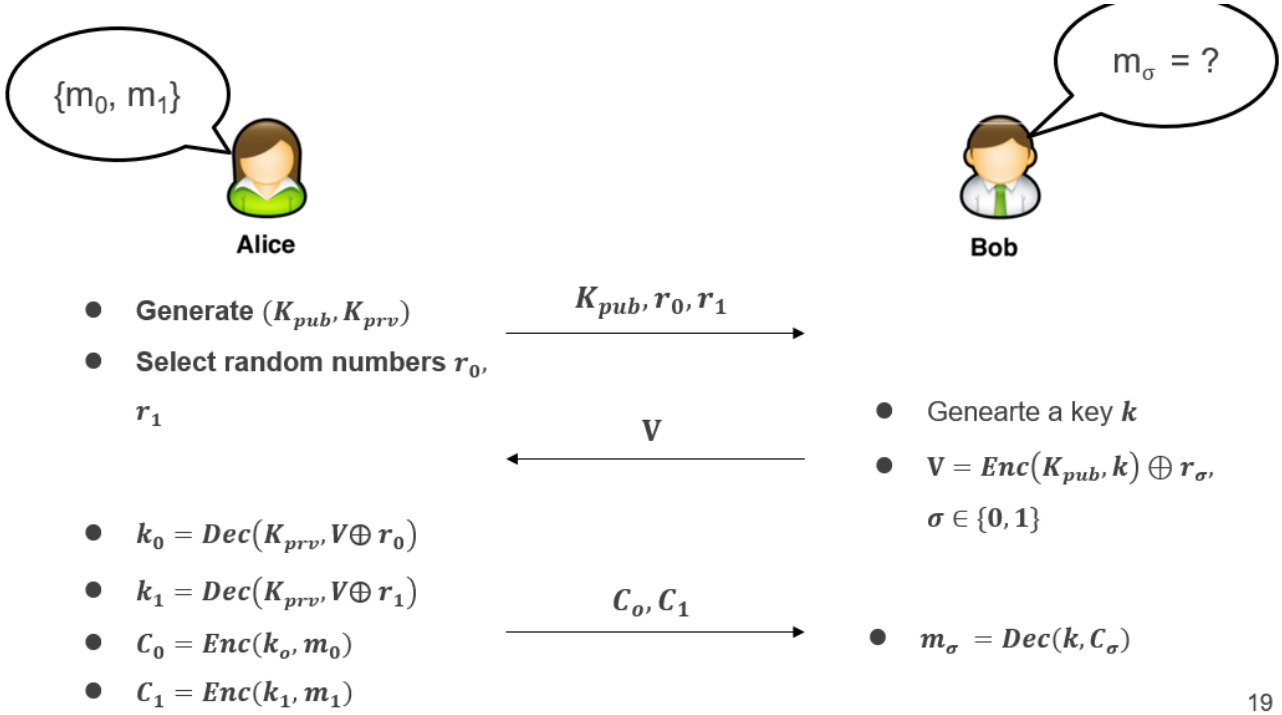
  -
- Non-Interactive ZKP
  - 有一个可信的第三方生成随机序列，让Alice一次性证明



OP: Oblivious Transfer 不经意传输

Problem: sender不想让receiver拿到所有数据，receiver不想让sender知道自己想要哪个数据

1-out-of-2 OT:



- Bob把自己的key用Alice的公钥加密，并与r_i作异或
- Alice分别用两个解密结果加密她的两份消息，将加密结果送给Bob（只有Bob选择的i才能解密出Bob的key，且Alice不知道Bob选择了哪个）
- Bob用k解密两份密文，得到需要的消息，而另一份无法解密

HE: Homomorphic Encryption 同态加密

Problem: 想把数据放在云端计算，但想加密

Solution: 密文可以直接运算，返回结果后用户解密即可。但目前算法难以支持所有密文运算，例如RSA只支持乘法

SMPC: Secure Multi-Party Computing

Problem: 多方拥有不同数据，想用这些数据共同计算而又不将数据泄露给其他方

Yao's Protocol: GC(Garbled Circuits)+OT(Oblivious Transfer), 姚期智的混淆电路+不经意传输算法

TEE: Trusted Execution Environmnet

- Software TEE
  - VM-based TEE
  - Same privilege protection
- ARM TrustZone
- Intel SGX
- AMD SME/SEV
- Penglai, SANCTUM

DP: Differential privacy 差分隐私

Problem: 想要拒绝用户访问数据库单个条目，但不能简单限制，否则可以通过sum(*) - sum(where != xx)反推出

Solution: （若两个数据集有且仅有一条数据不一样，则称此二者为相邻数据集）如果某算法作用于任何相邻数据集，得到一个特定输出的概率应差不多，那么我们就说这个算法能达到差分隐私的效果。也就是说，观察者通过观察输出结果很难察觉出数据集一点微小的变化，从而达到保护隐私的目的。差分度越低，安全性越高。

- 实现方式：向计算函数中加噪声

FL: Federated Learning 联邦学习

Problem: 多方训练同一个模型，需要保证各方数据隐私

- 横向：一方拥有一个样本的全部数据
  - 服务器分发子模型给每个用户，用户提交update，会泄露个人隐私
  - 用户向update中添加噪声，并保证总体对称，相互抵消，不影响总模型更新
  - 如果有用户掉线，服务器必须向其他用户询问该用户的噪声偏移量；则服务器可以伪造用户掉线，通过询问计算出该用户原本的update
  - Secret Sharing & Double Masking算法
    - Share a secret S among N nodes, T nodes can reconstruct the secret，基于k次函数方程求解需要k以上个点的原理
    - Each user u generates a_u and $S_{u,v}$ (for each user pair (u,v))
    - Each user u updates y_u，是关于a和S的函数
    - Server计算$\sum y_i$
      - For online node u, server asks other nodes to get the secret share of the a_u
      - For offline node u, server asks other nodes to get the secret share of the $S_{u,i}$
      - 所有节点防止server同时得到a_u and $S_{u,i}$
- 纵向：每一方拥有同一个样本的部分数据
  - 每方分别单独训练，在全连接层汇总，由有label的一方算出最终模型

后面介绍了3篇论文，是对以上技术的具体应用

- Oblivious Multi-Party Machine Learning on Trusted Processors (Security'2016)
- BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning (ATC'20)
- Privacy Accounting and Quality Control in the Sage Differentially Private ML Platform (SOSP'19)