

操作系统研究前沿

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

操作系统研究

- **操作系统是一个持续发展的领域**
 - 操作系统的研究持续火热，如今是系统研究“最好的时代”
- **操作系统研究受到上层应用和底层硬件双重驱动**
 - 互联网、网络搜索、大数据、人工智能、智能驾驶、云计算等
 - 持久性内存、GPU、智能网卡、AI芯片、硬件Enclave等
- **两个核心问题**
 - 如何为上层应用提供更快、更安全、更易用的接口？
 - 如何为底层硬件建立高效、安全、高利用率的抽象？

操作系统方向的重要学术会议

- **顶级学术会议**

- SOSP & OSDI : 操作系统影响力最大的会议, 许多内容进入OS教科书

- **重要学术会议**

- EuroSys : 基本在欧洲召开
- APSys : 在亚太地区召开的系统会议
- ChinaSys : 在中国召开的系统会议
- USENIX ATC : USENIX的技术年会, 议题较多, 偏工业界
- HotOS : 专门讨论OS前沿的workshop, 奇数年召开

- **特定方向的学术会议**

- ASPLOS : 偏重体系结构、编译与OS的结合
- USENIX NSDI : 网络系统方向
- USENIX FAST : 文件系统与存储方向
- ...

操作系统的八个前沿研究领域

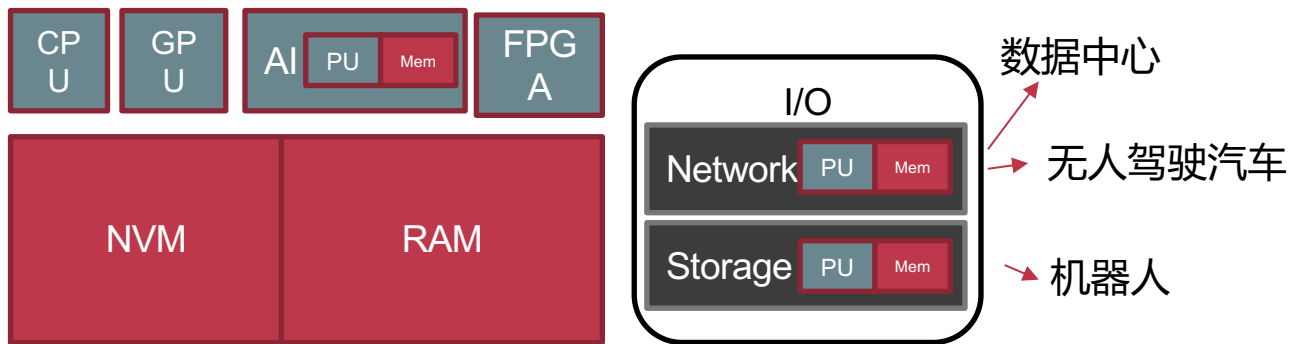
1. 异构操作系统
2. 新的应用接口
3. 同步原语
4. 持久性内存
5. 智能网卡
6. 系统安全
7. 操作系统测试
8. 形式化证明

Heterogeneous OS

1、异构操作系统

硬件发展的趋势：多样化与异构化

- 异构计算：CPU、GPU、FPGA、AI加速器等
- 异构存储：DRAM、NVRAM、PIM等
- 异构I/O：智能网卡、智能SSD等



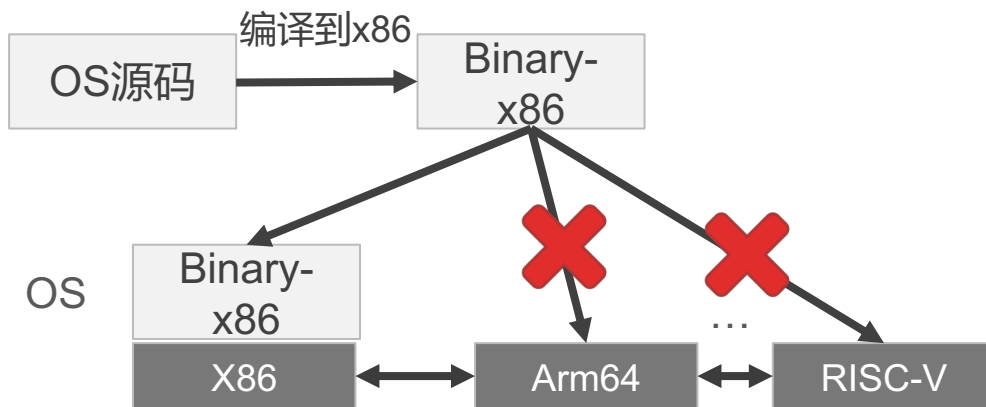
为什么需要异构硬件？

- 硬件能力很难再提升单个CPU核性能（摩尔定律的结束）
- 上层应用对架构的性能提出了更高的要求
- CPU无法满足AI计算、图形处理等场景的计算需求
 - AI加速器、GPU等异构计算和CPU并存
- DRAM容量受限 → NVM可以提供更大的内存容量
 - DRAM和NVM等多种内存并存
- 数据中心和云需要更大的IO带宽和更低的时延
 - RDMA、以太网卡、智能I/O设备等并存

异构硬件的涌现对OS带来了全新的挑战

挑战-1：需要同时支持多种指令集 (ISA)

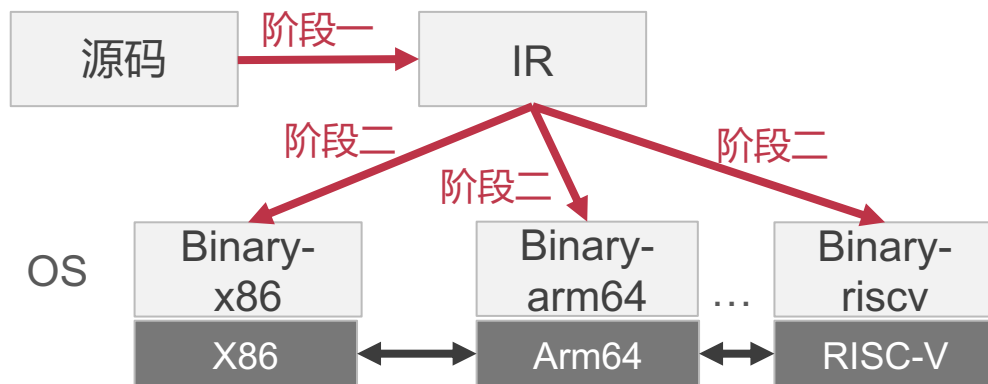
- 不同的计算单元可能使用不同的指令集



思路：两阶段编译解决异构ISA问题

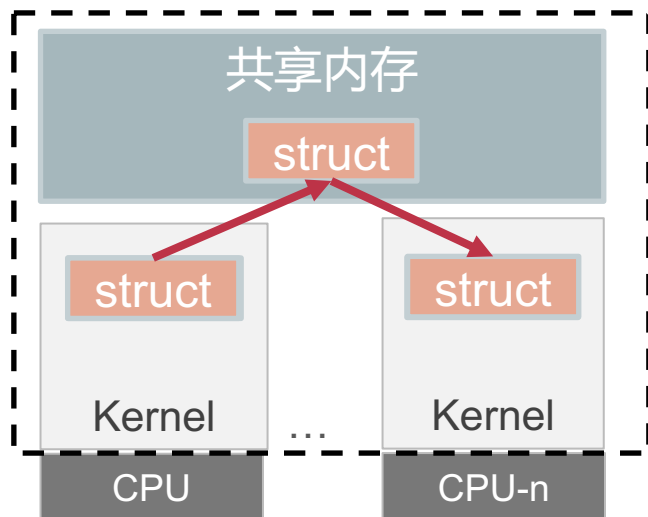
- 阶段1：从源码编译得到一个中间表示 (IR)，分发IR给使用者
- 阶段2：当决定在具体的平台上运行时，将IR生成binary
 - 根据具体部署的计算节点编译出不同指令集的二进制

不同的计算节点使用不同的ISA进行阶段2 → 解决异构ISA问题

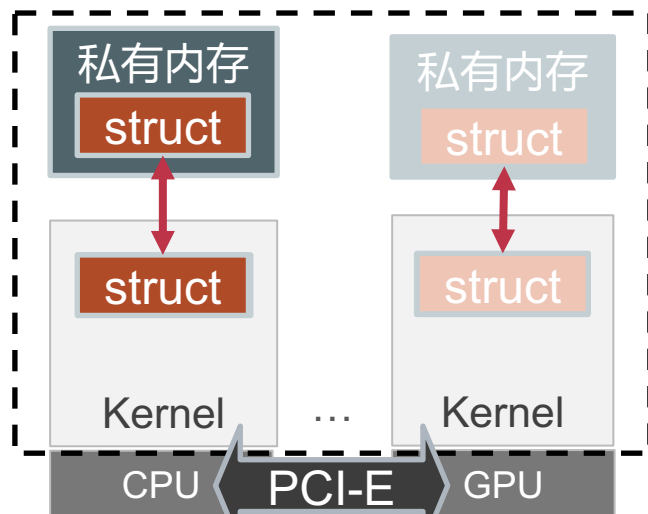


挑战-2：内核/应用在跨总线的环境下的同步和通信

- 异构硬件间通过多种总线连接，无共享内存和缓存一致性



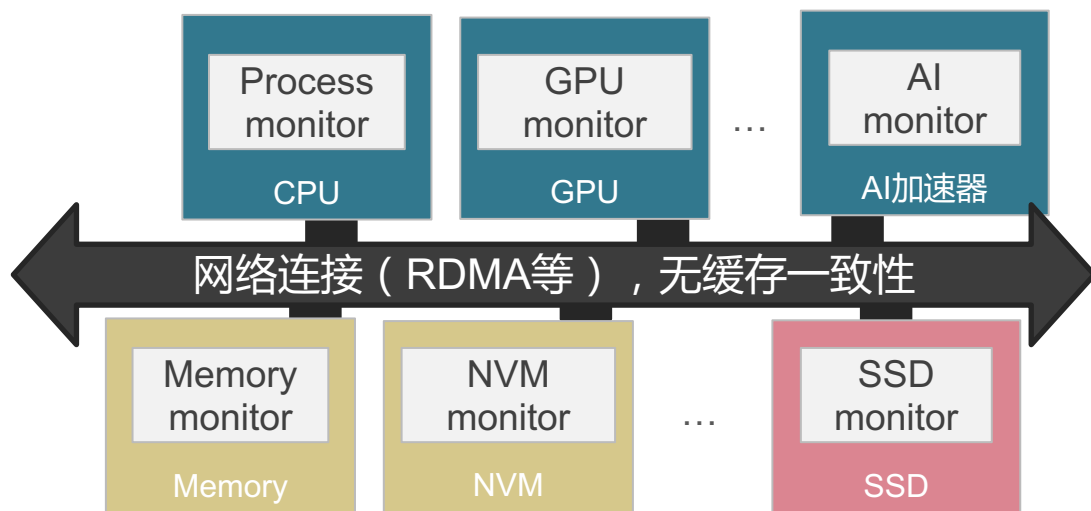
同构环境（共享内存）



异构环境（非共享内存）

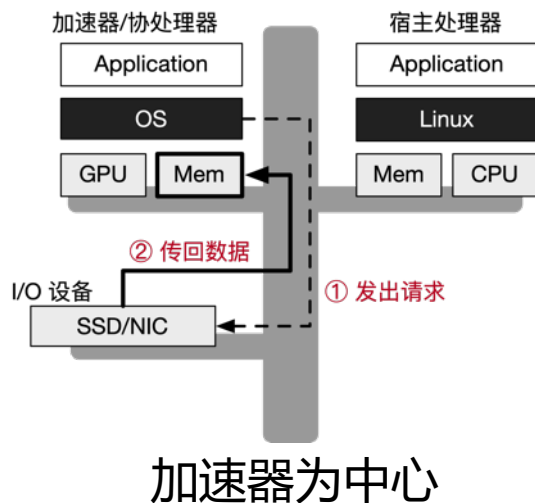
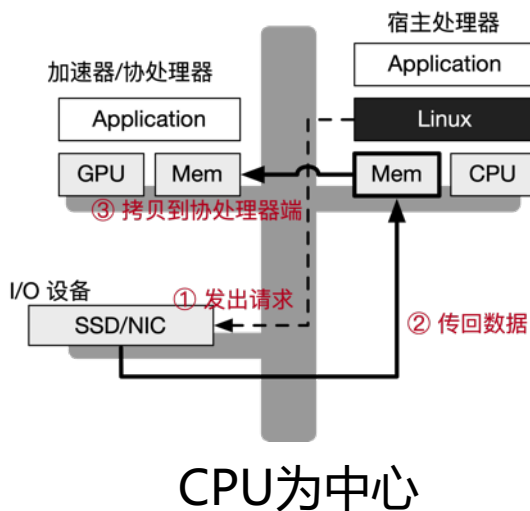
思路：多内核抽象——每个设备运行一个monitor

- OS也被拆分到不同设备上的monitor中，类似MultiKernel
 - CPU上负责OS进程管理
 - Memory monitor上负责OS内存管理
 - 通过RDMA等快速网络通信



挑战-3：CPU提供OS服务容易成为系统性能瓶颈

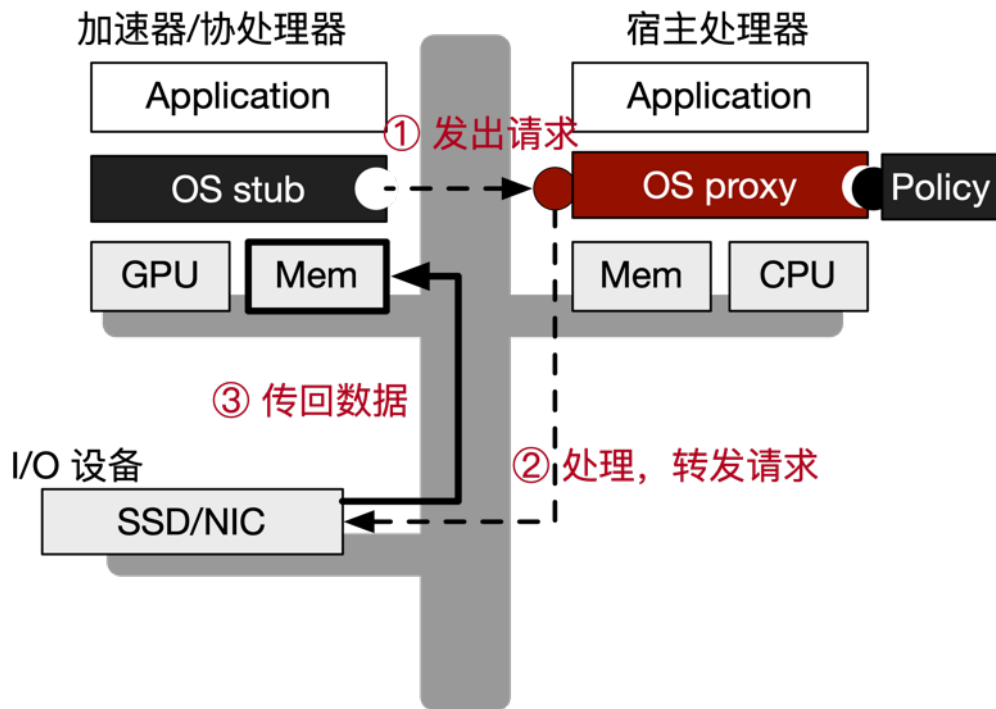
- 各种加速器等异构计算单元依赖于CPU提供OS服务
 - 以CPU为中心的OS服务：需要两次拷贝，性能开销大
 - 以加速器为中心的OS服务：加速器不适合执行如网络协议栈等OS服务



思路：CPU负责控制流，加速器负责数据流

- 解耦数据流和控制流

- 控制流由CPU负责
- 数据流由异构加速器负责
- 避免OS成为性能瓶颈



其他挑战与待解决的问题

- 跨计算节点的通信通常使用PCI-E，性能较差，如何优化？
- 如何将OS的服务以及应用程序分布在不同的计算节点？
- 异构硬件非常多样，OS需要为应用提供怎样的抽象？
- 如何支持跨ISA的应用程序迁移？
- 智能I/O设备上的计算单元容易出现Failure，如何容错？
- 如何调度和共享各种异构设备给不同用户/应用使用？
- 异构系统的安全性如何保证？

New OS Interface for Applications

2、新的应用接口

极低时延应用对操作系统的新要求



amazon

数据来源: <http://www.telx.com/blog/the-cost-of-latency/>

每500ms延迟



→\$35亿



数据来源: <http://qlinden.blogspot.jp/2006/11/marissa-mayer-at-web-20.html/>

任天堂总裁岩田聪：

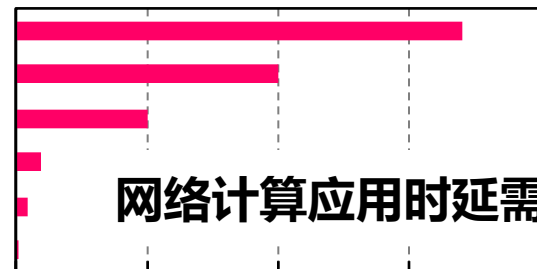


- ▶ 电子娱乐交互体验中最重要的是用户能够得到**快速响应**
- ▶ 目前云计算还难以满足游戏平台的**低时延需求**

应用时延需求

从秒级走向**微秒级**

实时音/视频
电子商务
在线游戏
股票交易
高频交易
增强虚拟现实



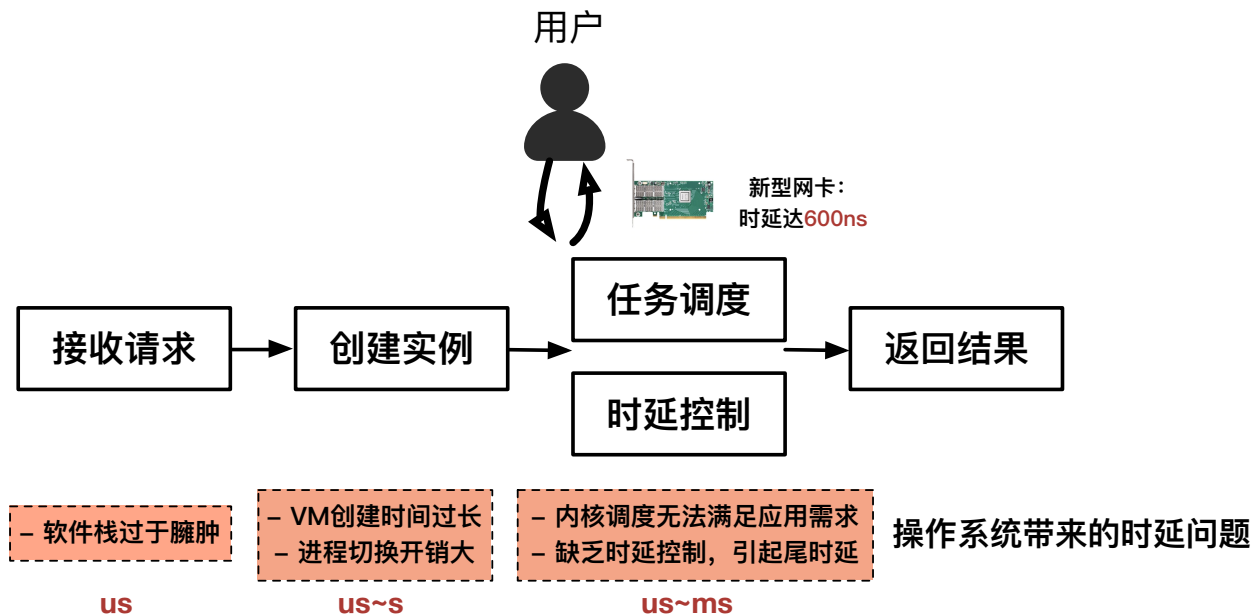
网络计算应用时延需求

微秒 0 50 100 150 200



极低时延应用对操作系统的新要求

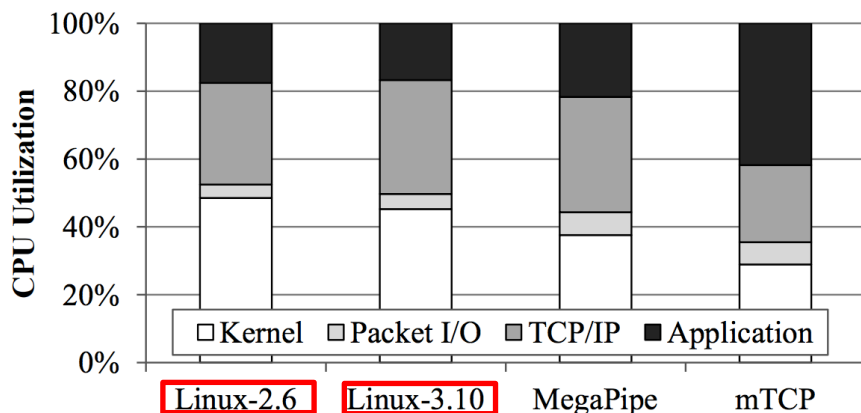
- 随着网络时延的降低，操作系统提供的服务和抽象成为了瓶颈
 - 操作系统应当如何重新设计，以解决这些问题？



挑战-1：内核软件栈臃肿

- 内核软件栈涉及系统调用、内存管理、命名管理等开销，成为了性能瓶颈

内核占了网络包处理近一半的时间



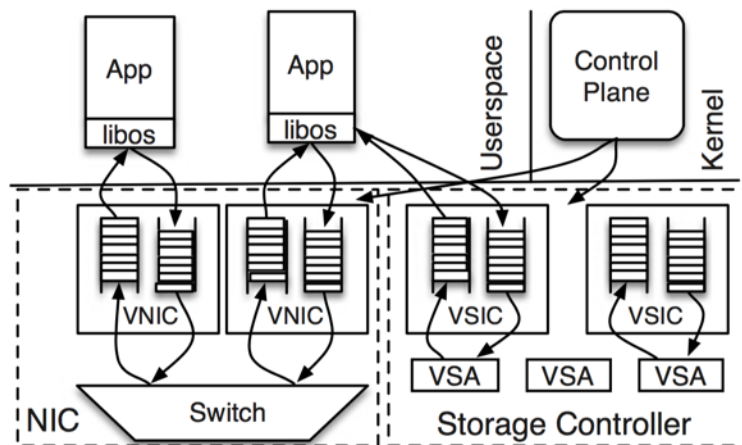
层层调用带来了微秒级的开销

Linux					
Network stack		Receiver running		CPU idle	
	in	1.26	(37.6%)	1.24	(20.0%)
	out	1.05	(31.3%)	1.42	(22.9%)
Scheduler		0.17	(5.0%)	2.40	(38.8%)
Copy	in	0.24	(7.1%)	0.25	(4.0%)
	out	0.44	(13.2%)	0.55	(8.9%)
Kernel crossing	return	0.10	(2.9%)	0.20	(3.3%)
	syscall	0.10	(2.9%)	0.13	(2.1%)
Total		3.36	($\sigma=0.66$)	6.19	($\sigma=0.82$)

mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems (NSDI'14)
Arrakis: The Operating System is the control plane (OSDI'14)

思路-1：通过内核旁路解决软件栈臃肿问题

- **机遇：新型硬件设备具备虚拟化能力**
 - 如一张网卡（NIC）可以虚拟为多张虚拟网卡（vNIC）
- **核心思想：操作系统将虚拟设备直接交给应用使用**
 - 应用可以直接访问设备，无需经过内核软件栈，大幅降低时延



思路-1：通过内核旁路解决软件栈臃肿问题

- **潜在问题**

- 设备映射过于死板，可能出现负载不均衡 / 资源浪费
- 应用直接访问设备，可能存在安全问题
- 生态碎片化

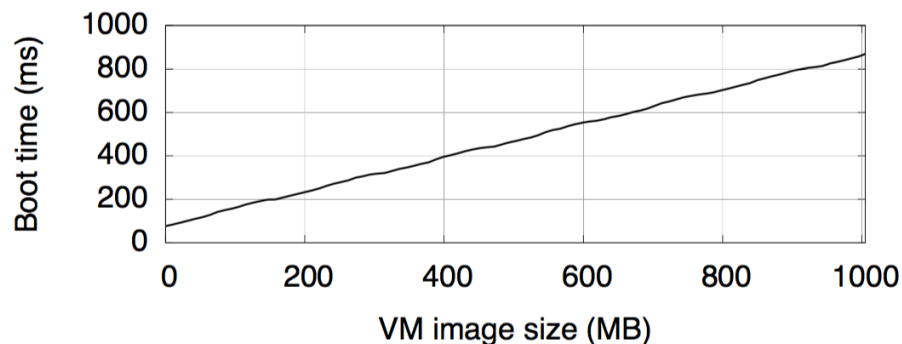
- **可能的解决思路**

- 支持任务偷窃机制，均衡负载
- 引入硬件保护机制

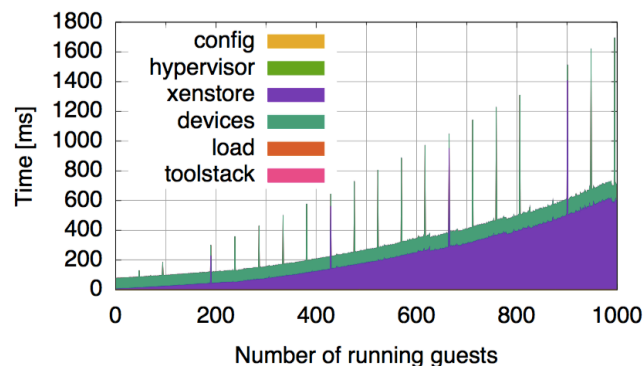
挑战-2：内核抽象过于笨重

- Linux内核代码逐年递增，极大影响了虚拟机的启动效率
- 虚拟机监控器（如Xen）在启动过程中引入了过大的软件开销

虚拟机启动时间随镜像大小增长



Xen中的软件模块（Xenstore）造成了昂贵的启动开销

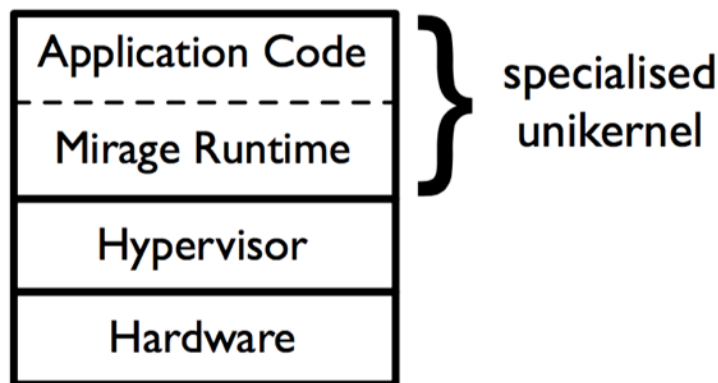


My VM is Lighter (and Safer) than your Container (SOSP'17)

思路-2：通过轻量级与定制化内核提高性能

- **核心思想：只保留与应用相关的功能**
 - Unikernel: 专为应用定制的特化内核
 - LightVM：移除VM启动过程中不需要的部分，实现微秒级启动

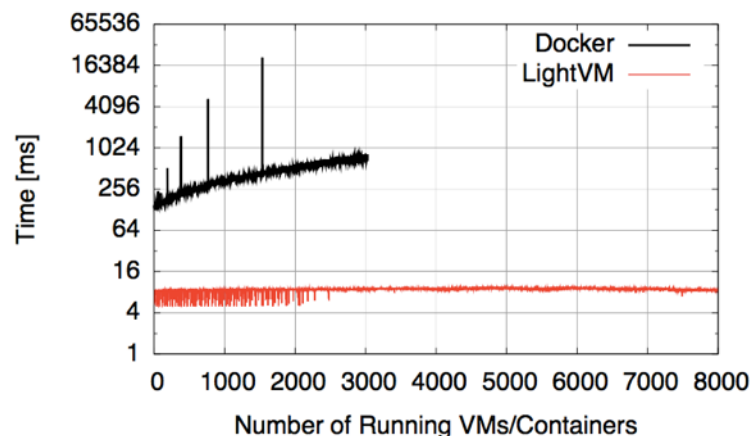
Unikernel只包括应用和必需的运行时代码，规模较小



Unikernels: Library Operating Systems for the Cloud (ASPLOS'13)

My VM is Lighter (and Safer) than your Container (SOSP'17)

LightVM启动时间远优于docker，接近进程创建时间



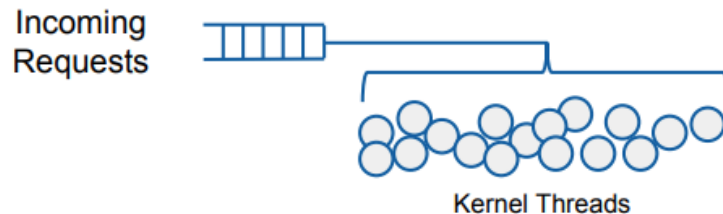
挑战-3：应用缺乏对于调度的控制

- 调度一定会进入内核，造成微秒级开销
 - 进程调度的开销是用户态线程调度开销的2倍
- 内核缺乏应用语义，可能无法做出最好的调度决策

进程切换开销明显高于用户态线程

process	k-thread	u-thread
4.25 (0.86)	4.12 (0.98)	1.71 (0.06)

内核线程数量多，做出最优决策极其困难



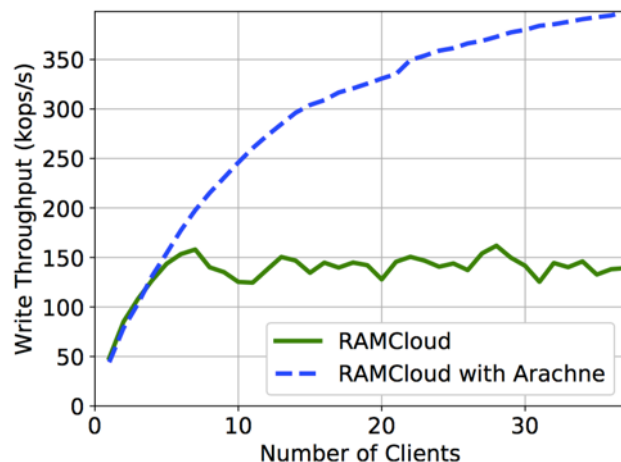
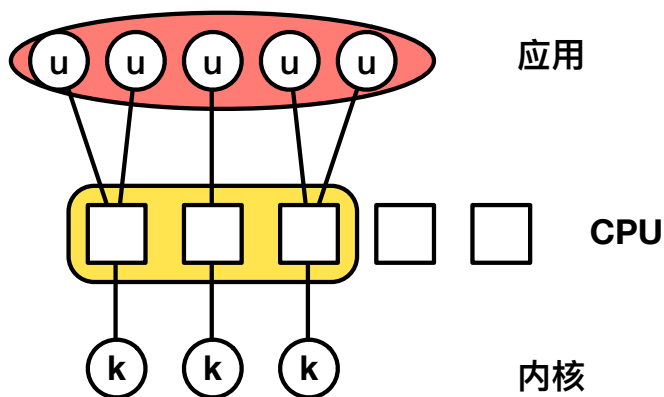
思路-3：让应用做更多决策，如自主调度

- 例如：将物理CPU核直接交给应用

- 应用使用核与内核交互，而不是线程
- 避免内核调度器与应用调度策略冲突造成的“双调度问题”

以内核线程的形式将物理核交给应用，用户态线程可自主调度

免除调度开销使应用具有了更好的可扩展性



小结：设计新型操作系统的关键词

- **放权**

- 将内核从应用的关键路径上移除，绕开臃肿的软件栈
- 将更多功能（如调度）交由应用完成，减少进入内核的次数

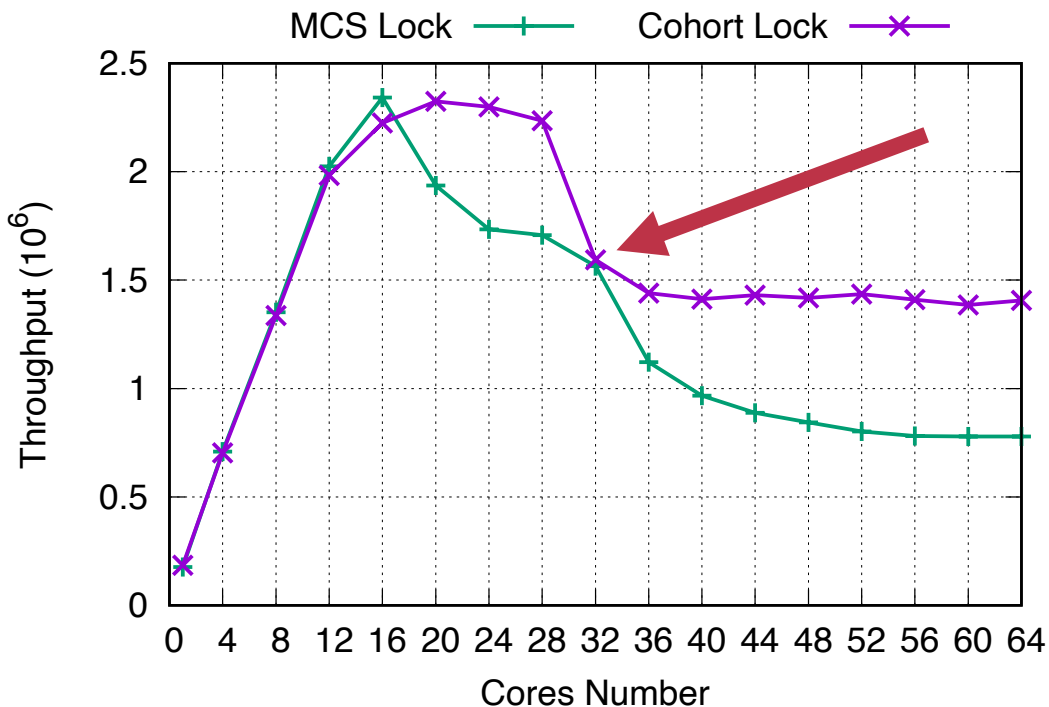
- **裁剪**

- 只保留内核抽象的必需功能，最小化抽象带来的开销

Synchronization

3、多核同步原语

挑战-1：CPU核越来越多，锁的可扩展性遇到瓶颈



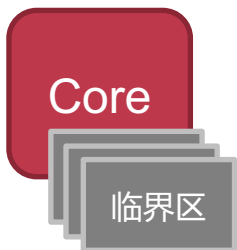
鲲鹏服务器 锁微基准测试

回顾：之前课程介绍的Cohort Lock

- Cohort Lock：将竞争限制在本地
依旧**无法避免**跨节点的开销
- **能否进一步提升**NUMA环境下同步原语的可扩展性？

挑战-1：CPU核越来越多，锁的可扩展性遇到瓶颈

思考：多核环境下锁的**理想性能上限**在哪？

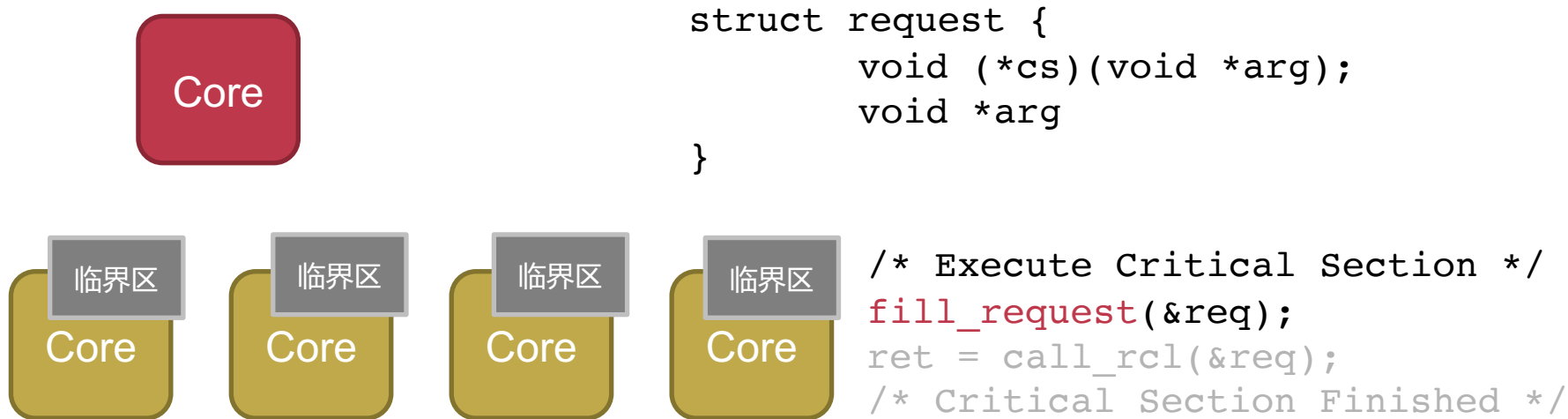


在**一个**核心上，**连续执行临界区**能达到的吞吐率

没有由于**其他核心竞争**、**数据迁移**导致的性能开销



思路：将所有竞争者的临界区集中到一个核上处理



**Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications (ATC' 12)*

思路：将所有竞争者的临界区集中到一个核上处理



```
for req in req_list:  
    ret = req->cs(arg);  
    send_resp(ret);
```



```
/* Execute Critical Section */  
fill_request(&req);  
ret = call_rcl(&req);  
/* Critical Section Finished */
```

**Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications (ATC' 12)*

思路：将所有竞争者的临界区集中到一个核上处理



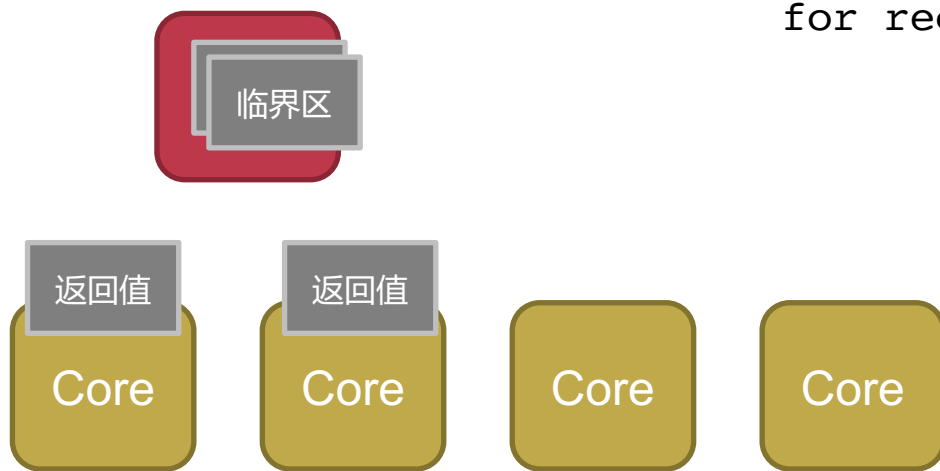
```
for req in req_list:
    ret = req->cs(arg);
    send_resp(ret);
```



```
/* Execute Critical Section */
fill_request(&req);
ret = call_rcl(&req);
/* Critical Section Finished */
```

**Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications (ATC' 12)*

思路：将所有竞争者的临界区集中到一个核上处理



```
for req in req_list:  
    ret = req->cs(arg);  
    send_resp(ret);
```

```
/* Execute Critical Section */  
fill_request(&req);  
ret = call_rcl(&req);  
/* Critical Section Finished */
```

**Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications (ATC' 12)*

思路：将所有竞争者的临界区集中到一个核上处理

利用**batch**思想对迁移锁进一步优化**返回值的写回**

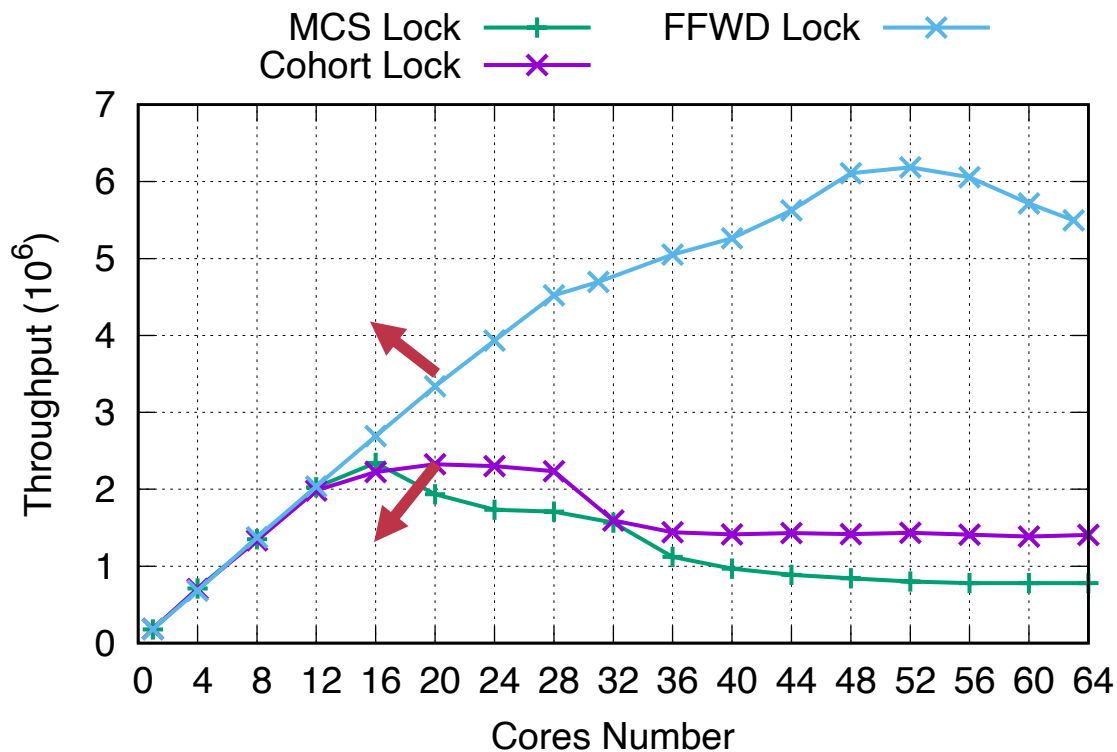


RCL每次处理一个请求结束后写返回值
会由于处理器**写缓存**导致性能瓶颈

FFWD：等待处理结束一个节点所有请求后，
一次性将同一个节点的**所有请求**返回值放入
一个缓存行

**ffwd: delegation is (much) faster than you think (SOSP' 17)*

迁移锁的效果：在跨NUMA节点时依然保持可扩展



鲲鹏服务器 锁微基准测试

迁移锁是多核锁的终极方案么？No

问题-1：适应性 -- 逻辑复杂、单独的锁服务器使得迁移锁在**低竞争程度**时开销巨大

问题-2：易用性 -- 锁暴露的接口发生改变，无法直接使用在现有系统中

```
/* Execute Critical Section */  
lock(&glock);  
cs(arg);  
unlock(&glock);  
/* Critical Section Finished */
```



```
struct request {  
    void (*cs)(void *arg);  
    void *arg  
}  
  
/* Execute Critical Section */  
fill_request(&req);  
ret = call_rcl(&req);  
/* Critical Section Finished */
```

一种解决思路：通过自动化转换工具进行


挑战-2：如何在提升性能的同时，维持锁的易用性？

如何使用尽可能少的额外存储空间、适应遗留应用（如Linux内核）的编程接口？

Cohort锁 与 MCS锁 锁的**存储空间** 对比

N个NUMA节点，指针8 bytes，缓存行 64 bytes

Cohort锁 $(N * 8 + 1)$ 倍开销 MCS锁



- 每个锁需要一个指针指向全局锁
- 每个NUMA节点都需要一个指针指向本地锁，且必须**在不同的缓存行**避免竞争

- 每个锁只需要一个指针指向队列尾

Linux内核使用类似MCS的队列锁

无法直接放入Linux内核使用

除了性能可扩展性与易用性之外的挑战

挑战#3：新的硬件会给同步带来什么样的影响？

异构系统不同设备之间如何同步/通讯？

采用弱内存模型的处理器如何耗费最小开销保证正确性？

挑战#4：同步原语能否和操作系统其他模块更好的协作？

如调度器，以维持调度公平；或避免调度在忙等的线程提升性能。

Persistent Memory

4、持久性内存

持久性内存（PM）正在变得越来越流行

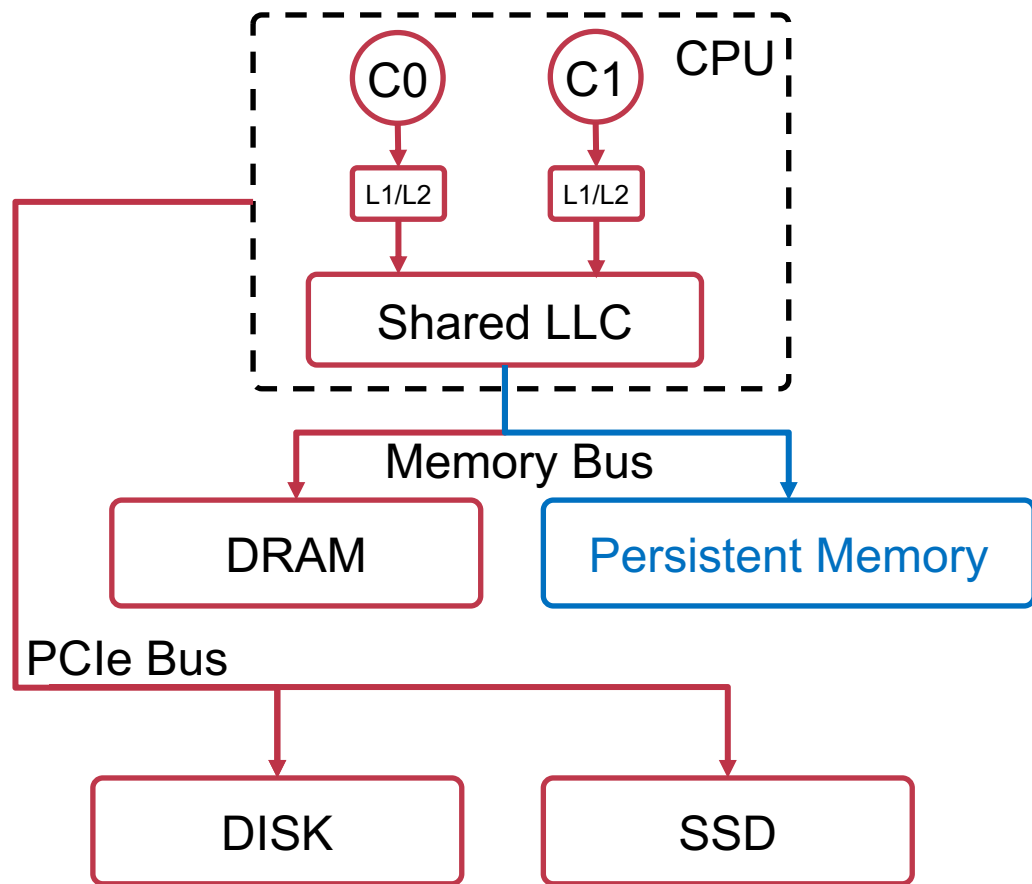
- 新型的内存/存储设备
 - 又称非易失性内存（NVM）
- 结合内存与存储各自的特点



性能	与 DRAM 接近，高于 SSD	功耗	无需刷新，低于 DRAM
访问	load/store 指令，字节粒度	价格	\$/GB 低于 DRAM
位置	内存插槽，使用内存总线	持久性	持久保存数据
密度	目前最高 512GB / 条	耐磨度	高于Flash

Figure from <https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html>

挑战-1：操作系统该如何为持久内存提供抽象？



思路：直接将持久性内存当做存储设备管理

- 使用内核文件系统管理持久内存
 - 优势：对应用透明（复用 FS 的接口）
 - 问题：持久内存持久化粒度为字节（缓存行）
 - 如何保证崩溃一致性？
 - 🤔回想讲过的崩溃一致性方法
- 新的挑战：持久性内存远快于硬盘，内核成为新的瓶颈
 - 之前文件系统瓶颈在于硬盘访问速度

挑战-2：如何避免内核带来的开销？

- **发现：持久性内存可以在用户态访问**
 - 硬盘是I/O设备，必须由内核统一管理
 - 内存则本来就是用户态可以访问的
- **思路：将 FS 功能移到用户态，从而避免进入内核**
 - 即 kernel bypassing
- **新挑战：如何保障数据安全性和隔离性？**
 - 例如：若inode可以由用户直接访问，会发生什么？
 - 想法：需要用户态和内核态进行分工

思路-1：用户FS记录日志，内核FS整理日志

- **写操作**

- 用户态 FS 只能（只需）在 Per-process 的 NVM 空间中记录日志
- 内核 FS 定期查看日志，并将数据更新到全局位置
- 在应用写完后，内核整理前，新的内容全局不可见

- **读操作**

- 经过内核 FS 检查后，用户态 FS 可以直接读取 NVM 部分区域

- **优势**

- 用户 FS 只需记录 NVM 日志即可保证操作完成

- **问题：需要定期去进行整理，多进程同步、写放大问题**

思路-2：用户FS处理数据，内核FS管理元数据

- **元数据安全是安全与隔离的关键问题**
 - 用户 FS 将文件 mmap 到用户态后，直接在用户态处理数据请求
 - 内核 FS 只负责处理文件的元数据操作
- **优势**
 - 数据修改只需写入一次，且无需内核参与
 - 元数据的安全和一致由内核保证
- **问题：文件元数据依然需要内核的参与**
 - 元数据操作本身较快，因此进出内核时间占比更大

思路-3：用户FS处理请求，内核FS提供隔离

- **发现：相同权限的文件具有聚集性**
 - 在验证权限后，用户 FS 能够处理相同权限文件上的所有请求
 - 内核 FS 只提供权限验证、粗粒度空间管理和强隔离保障
 - 读写权限控制由页表实现（别忘了PM是内存！）
- **优势**
 - 文件操作基本不需要内核参与，包括数据和元数据
 - 文件权限和隔离性被内核进行保障
- **问题：权限相关的操作需要内核处理**

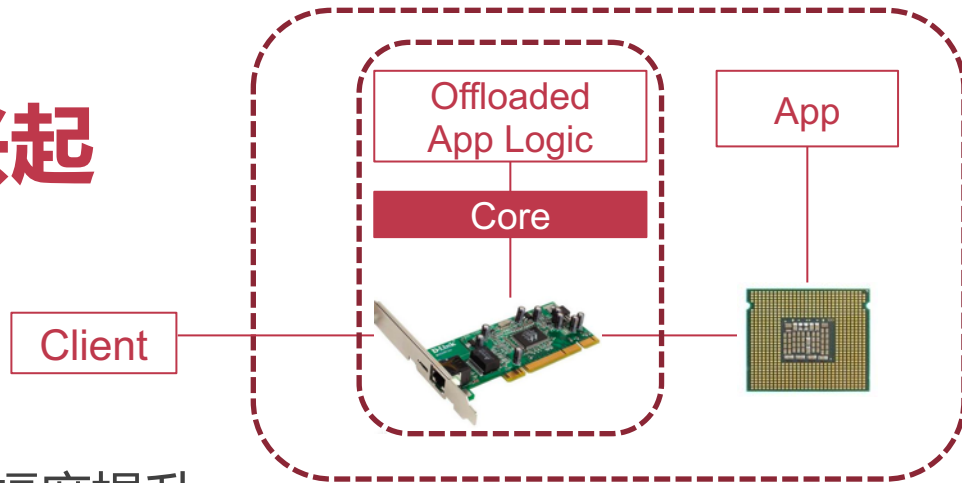
未来：PM上是否还需要文件系统？

- **在持久性内存的场景下，是否还需要 FS 进行管理？**
 - 命名（naming）问题谁来处理？
 - 管理员如何对存储进行维护？
- **除了 FS 接口之外，是否有更好的（存储）编程抽象？**
 - 持久性内存大量使用 mmap() 接口
 - ioctl 被使用的越来越频繁
 - 标准文件接口以及 POSIX 标准是否已经过时？

SmartNIC

5、智能网卡

背景：智能网卡的兴起



- **摩尔定律逐渐失效**

- 处理器频率和核数难以大幅度提升
- 网卡速度稳步提升：处理器速度难以跟随网络速度进一步提升

- **智能网卡的方案**

- 允许将一部分软件功能卸载至智能网卡之中
 - 卸载的软件可直接处理网络数据，减少与Host处理器交互

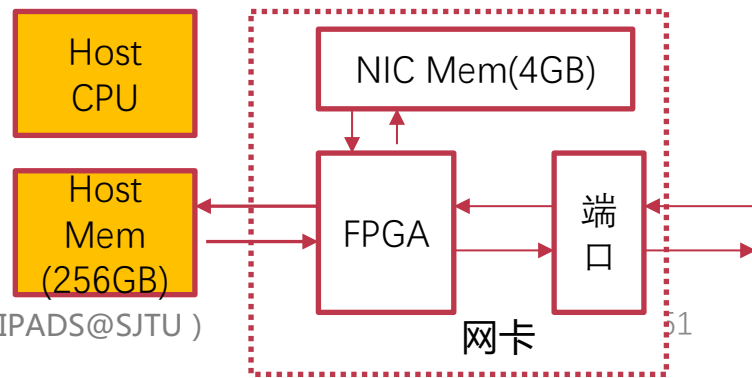
- **研究思路：思考将何种功能或应用卸载至设备之中**

挑战：OS如何使用智能网卡优化网络应用？

- **方向一：卸载部分应用逻辑**
 - KV-Direct (SOSP'17), NetCache (SOSP'17)
 - NICA (ATC'19), E3 (ATC'19), iPipe (SIGCOMM'19)
- **方向二：卸载操作系统功能**
 - 卸载网络虚拟化功能 (NSDI'18)
 - 卸载部分网络协议栈功能: AccelTCP (NSDI'20)

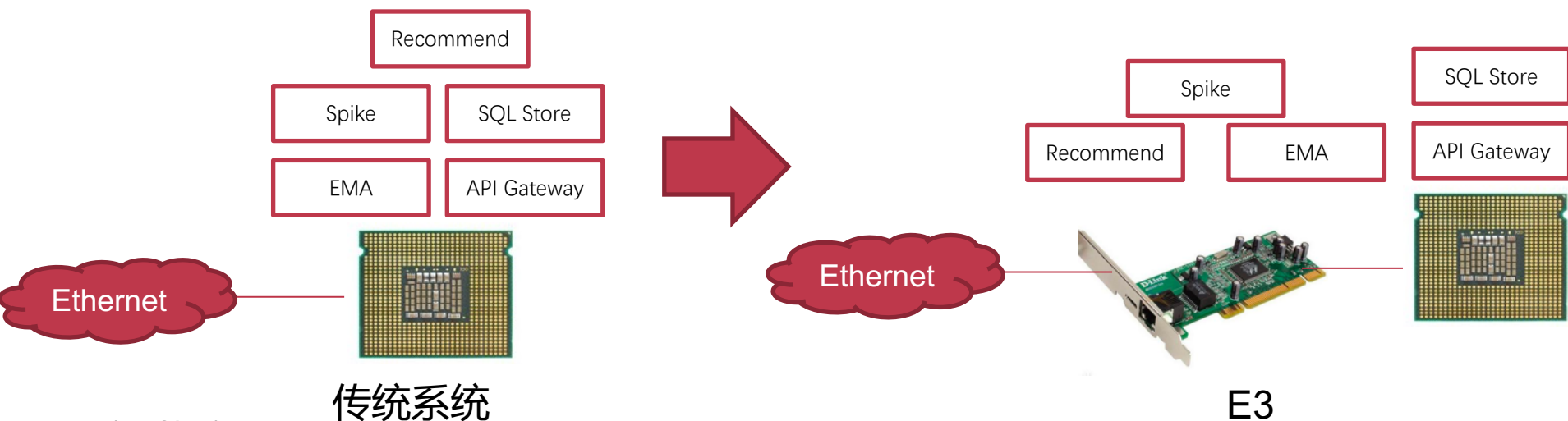
思路-1：卸载部分应用逻辑至网卡

- 目的：使用智能网卡提高应用的吞吐量并降低时延
- 核心想法：将部分Key-Value操作卸载至网卡内的FPGA
- 利用on-path网卡的能力
 - FPGA直接读取端口的数据
 - FPGA处理完数据后直接返回，绕开Host CPU
 - 尽可能地利用网卡中的内存



思路-2：卸载部分微内核服务至网卡

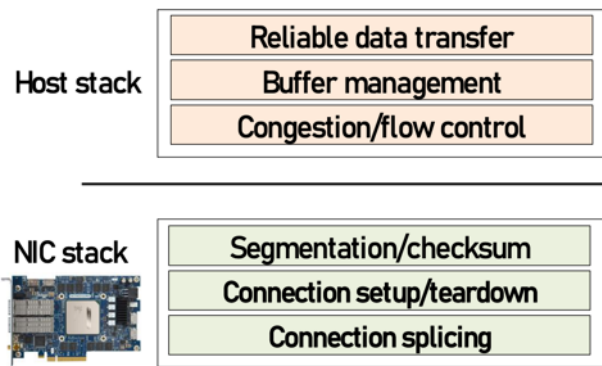
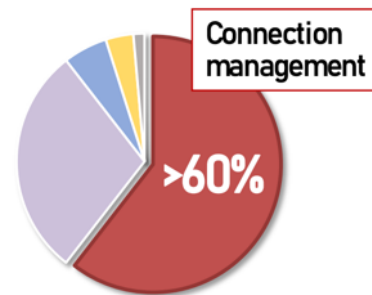
- 目的：利用智能网卡降低能耗、同时降低时延
- 核心想法：将数据中心中的部分微服务直接运行在网卡内



E3 (ATC'19)

思路-3：卸载部分网络协议栈至网卡

- 卸载操作系统功能：对应用提供通用的性能提升
- 目的：优化Short TCP Flows和L7 Proxy性能
 - 开销主要来自于Short Flow连接的管理
 - L7 Proxy引起大量设备与Host Memory的DMA操作
- 核心想法：双网络协议栈的设计
 - 将简单的TCP网络协议栈操作卸载至网卡
 - 优化Short TCP的管理以及数据拷贝



更多智能网卡的研究挑战

- **设备内难以隔离多个用户卸载的计算任务**
 - 安全性和可用性之间的权衡
- **设备与处理器之间异构性较大**
 - 需要同时支持不同的指令集(ISA)
 - 编程模型存在较大差异
- **卸载计算任务与网卡本身任务之间的竞争**
 - 卸载的计算任务可能会对网卡本身性能造成较大影响

System Security

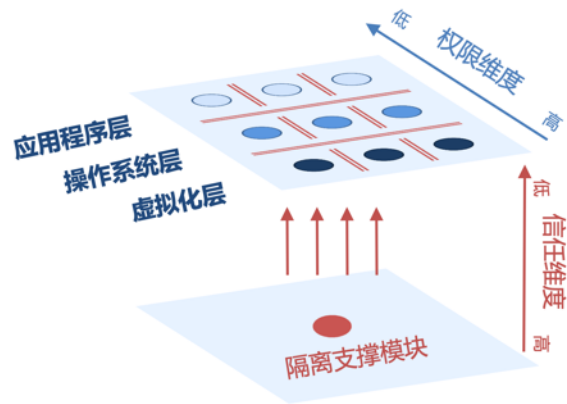
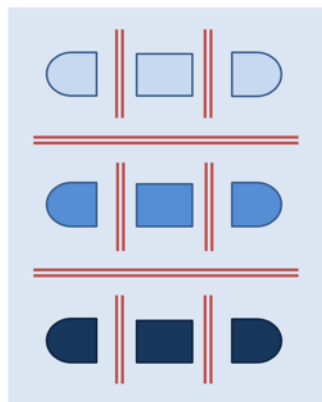
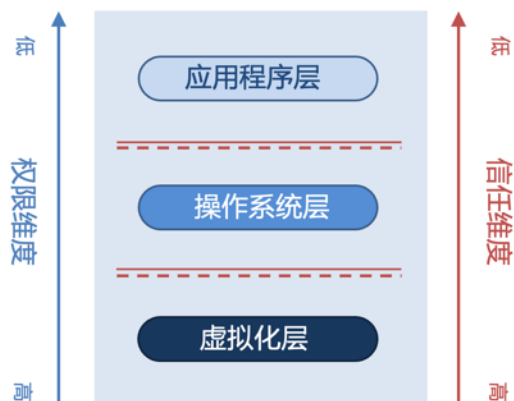
6、系统安全隔离

挑战：如何保证越来越复杂系统的安全

在互不信任的多用户云计算环境中，系统软件的安全性显得尤为重要



系统隔离的演变：层次化到“九宫格”



- 自底向上权限由高到低，信任同样由高到低，隔离面为**单向**
- 越底层，漏洞危害越大

- 隔离面由**单向变为双向**，信任与权限解耦
- 每层内部**细粒度隔离**

- **隔离支撑模块**：硬件原语支持更高层次语义
- **软硬结合成为趋势**



为什么要假设操作系统是恶意的？

- **系统的复杂性**

- 软件：恶意软件，OS本身可能存在漏洞
- 硬件：外设越来越智能，本身可能存在漏洞，甚至是恶意构造
- 环境：云计算环境、IoT设备，面临这更复杂多变的
- 人：运维外包（如云计算等）导致接触计算机的人更复杂

- **一种更简单的威胁模型**

- “除了应用，别的都不可信”，即“自包含的信任”

恶意操作系统如何攻击应用？

- **应用的攻击面**
 - 同层：其他应用程序（如通过IPC）
 - 底层：操作系统、Hypervisor、硬件
- **操作系统窃取应用的数据**
 - 操作系统控制着页表，可直接映射应用的内存并读取数据
- **操作系统改变应用的执行**
 - 操作系统控制着页表，可直接在应用内部新映射一段恶意代码
 - 操作系统可任意改变程序的RIP，劫持其执行流

现实中的类似攻击

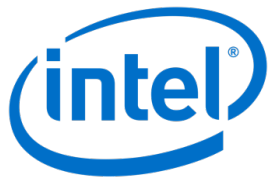
- 系统对应用的攻击

- 攻击者获取具有root权限的bash（如利用sticky位程序的漏洞）
- 攻击者获得`insmod`的权限，在内核中插入恶意的模块
- 攻击者篡改内核的文件，下次启动后加载
- 攻击者获得`kexec`的权限，动态执行另一个内核
- 攻击者获得hypervisor的权限，从更底层发起攻击
- 攻击者能够控制某个设备（如智能网卡），直接访问物理内存
-

一种新的威胁模型：硬件Enclave

- **不信任CPU外的硬件**
 - 包括内存（DRAM）、设备、网络
- **仅信任CPU**
 - 包括cache、所有计算逻辑（Anyway，总得信任CPU吧...）
- **Enclave（飞地）**
 - 又称为可信执行环境，TEE（Trusted Execution Environment）
 - 什么是“可信”？信什么呢？

TEE在工业界发展迅速



※ Intel SGX/TDX



※ AMD SEV



※ ARM TrustZone



※ Keystone, 蓬莱

- 多家云厂商利用TEE/Enclave保护数据

- 2018年, Microsoft Azure 基于Intel SGX推出Confidential Computing
- 2019年, Amazon 推出了Nitro Enclave保护用户的关键数据
- 2020年, Google Cloud 基于AMD SEV推出加密虚拟机Secure VM

- 2019年8月, 多家公司成立机密计算联盟

- 国内公司包括: 华为、阿里云、腾讯、百度、字节跳动等
- 国外公司包括: Arm、AMD、Intel、Redhat、Facebook、Google等



Enclave/TEE：可信执行环境

- **Enclave/TEE的定义**

- Enclave，又称"可信执行环境"（TEE，Trusted Execution Environment），是计算机系统中一块通过底层软硬件构造的安全区域，通过保证加载到该区域的代码和数据的完整性和隐私性，实现对代码执行与数据资产的保护 —— *Wikipedia*

- **Enclave的两个主要功能**

- 远程认证：验证远程节点是否为加载了合法代码的Enclave
- 隔离运行：Enclave外无法访问Enclave内部的数据

- **Enclave带来的能力：限制访问数据的软件**

- 可保证数据只在提前被认证的合法节点间流动
 - 合法节点：部署了合法软件的节点

基于Enclave的不同粒度隔离环境

应用片段抽象



2015年 Intel SGX
由硬件提供高层次接口，但交互漏洞频频

应用抽象



传统的操作系统隔离环境抽象，高度依赖OS语义，硬件不直接参与

容器抽象

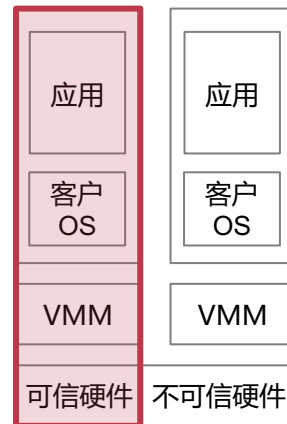


虚拟机抽象



2013年 HyperCoffer
在内部复杂性与交互复杂性间寻找平衡点

物理机抽象



2003年 ARM TrustZone
由于层次结构复杂，导致内部的软件漏洞不断

← 粒度细：内部简单，但交互复杂 ————— 粒度粗：交互简单，但内部复杂 →

操作系统安全隔离的其他挑战

- **性能问题**

- SGX安全内存大小受限，大内存访问性能差
- 启动性能、系统调用性能 ...
- 研究思路：避免触发硬件安全内存换页，flexSC ...

- **安全问题**

- 攻击者使用硬件可执行环境保护恶意软件
- 恶意硬件与硬件漏洞
- 层出不穷的侧信道攻击

Bug Finding in OS

7、操作系统新型测试方法

操作系统测试的重要性

- 操作系统代码规模庞大，难以避免BUG和漏洞
 - Linux 5.8 内核约包含2870万行代码
 - CVE：2019年Linux内核超过170个CVE
 - 实际数量更多，并非所有漏洞都被公开
 - Bug数量更多
- 操作系统正确性直接影响上层应用

Kernel.org Bugzilla – Bug List

Status: RESOLVED

Creation date: (is greater than or equal to) 2019-01-01

Creation date: (is less than or equal to) 2019-12-31

531 bugs found.

操作系统测试面临的挑战

- **测试的方法**

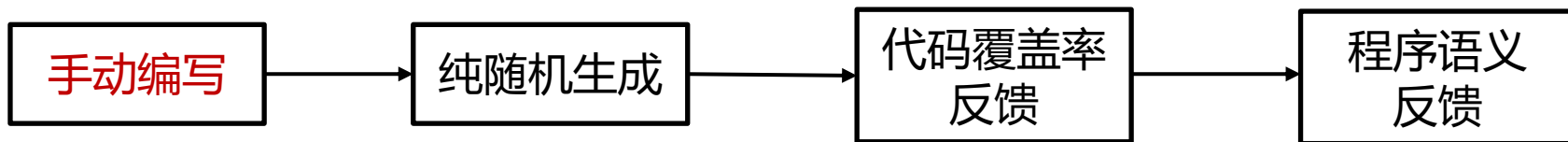
- 静态：在不运行代码前提下，通过语法、控制流等分析代码异常
- 动态：运行测试程序，判断执行是否符合预期
- 近年来动态测试工作较多

- **动态测试的挑战**

- 如何构造有效的测试样例
- 测试中如何判定出现异常
- 如何有效地复现BUG
-

挑战-1：如何构造测试样例

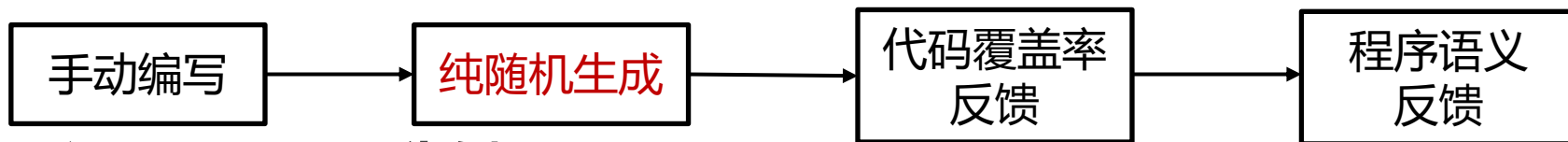
- 测试样例构造发展



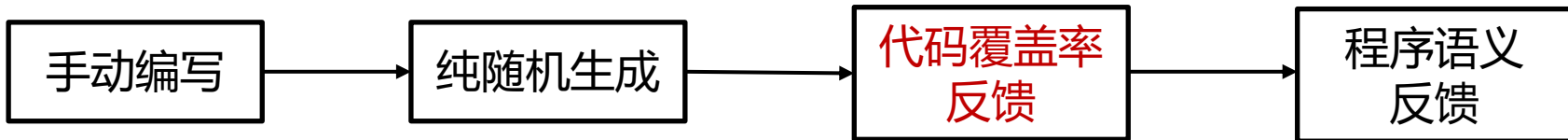
- 手动编写

- 较传统的测试方法，例如Linux Test Project
- 优点：可实现针对性较强的测试
- 缺点：测试样例数量有限导致测试不完全
编写针对性的测试人力成本较高

挑战-1：如何构造测试样例



- 以B3 (OSDI 18)为例
 - 针对Linux文件系统崩溃一致性的黑盒测试
- 构造测试样例
 - 构造**顺序调用**的POSIX文件系统接口
 - 问题：无穷的可能的接口调用顺序
 - 解决：在**限制**接口调用情况下，**穷举**可能出现的调用
 - 限制接口调用数量，文件数量，目录深度等
- 随机生成的缺点
 - 生成产生大量测试输入，很可能包含大量重复测试



- **以HYDRA (SOSP 19)为例**

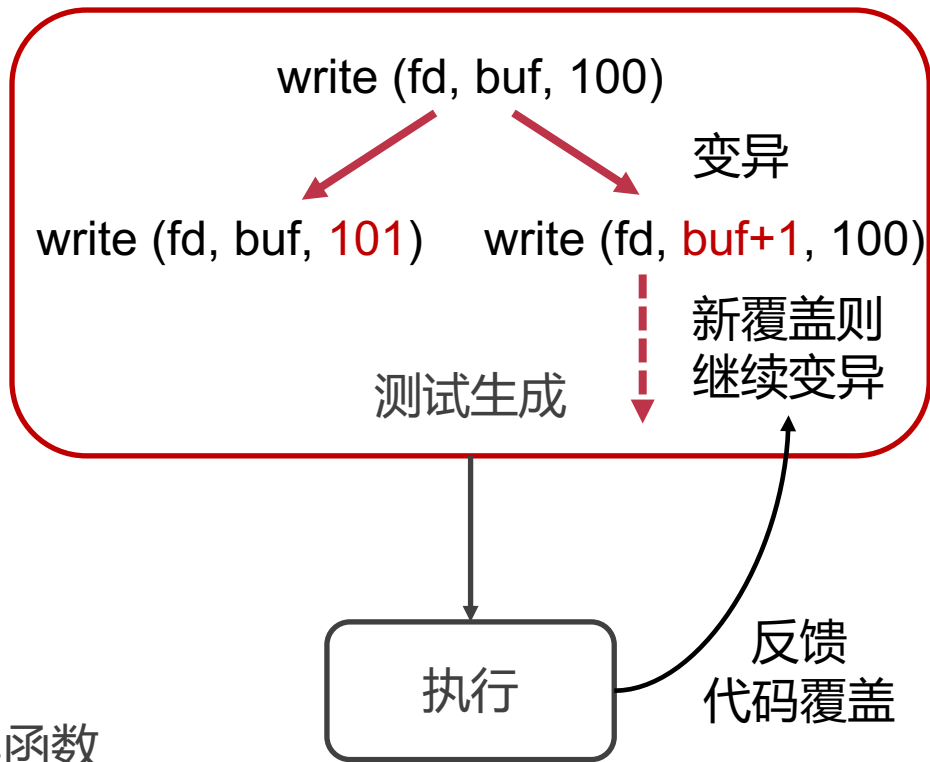
- Linux 文件系统 Fuzzing 测试

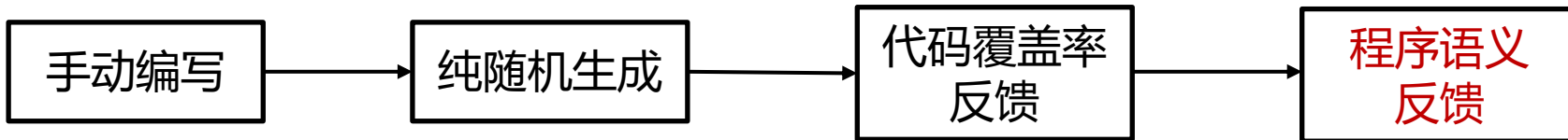
- **Fuzzing测试**

- **变异和反馈**指导测试样例生成
 - 参数变异后进行系统调用
 - 产生新的代码覆盖则继续变异
- 有效生成高代码覆盖率的测试

- **基于代码覆盖生成测试缺点**

- 较难触发边界条件，例如错误处理函数





- 以FIFUZZ (Security 20)为例

- 针对错误处理进行测试

- 针对错误处理

- 使用错误注入 (fault injection) 作为输入
 - 例如故意将kmalloc返回的ptr设为NULL
- 变异过程：配置各个错误注入打开或关闭
 - 如果能够触发新的错误处理序列则继续变异
- 有效覆盖错误处理相关代码

```
ptr = kmalloc(...);  
/* 错误处理 */  
if (ptr == NULL)  
    return -ENOMEM;
```


操作系统测试的其他挑战

- **Fuzzing测试输入生成**
 - 如何针对各类场景提升生成测试的有效性
- **测试中如何判定出现异常**
 - 部分错误不会引起可观测的异常 (silent error)
- **如何有效地复现BUG**
 - 在非确定性和并发环境下，触发的BUG不便于复现

.....

Formal Verification

8、形式化证明

系统软件的保障性前所未有的重要

- 1994年，在英特尔故障中，召回错误芯片导致4.75亿美元损失
- 1996年，阿丽亚娜5型火箭由于整数溢出漏洞导致升空爆炸
- 2014年，Mt.Gox比特币交易所遭遇黑客攻击，申请破产保护
- ...



无人驾驶



电脑系统



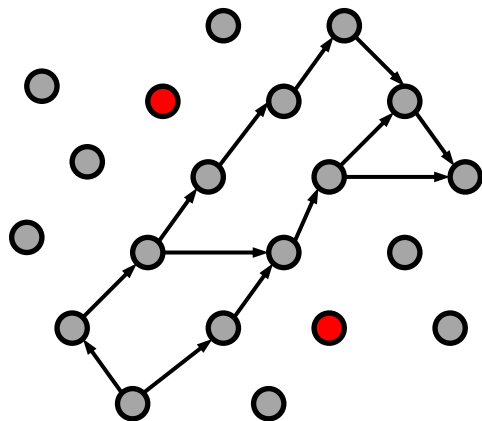
手机系统

背景：形式化证明

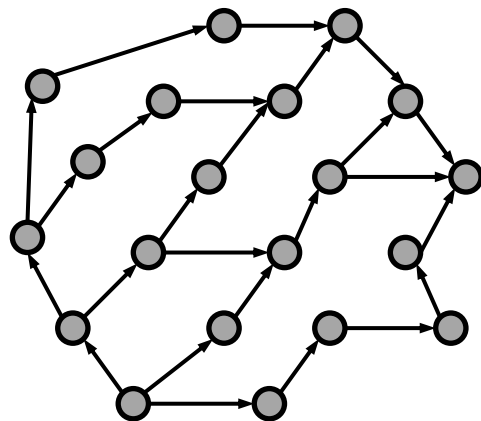
- “形式化证明是已知唯一一个保证软件没有编程错误的手段”

— seL4, SOSP'09

- 基于数学证明，完全覆盖软件所有可能执行情况



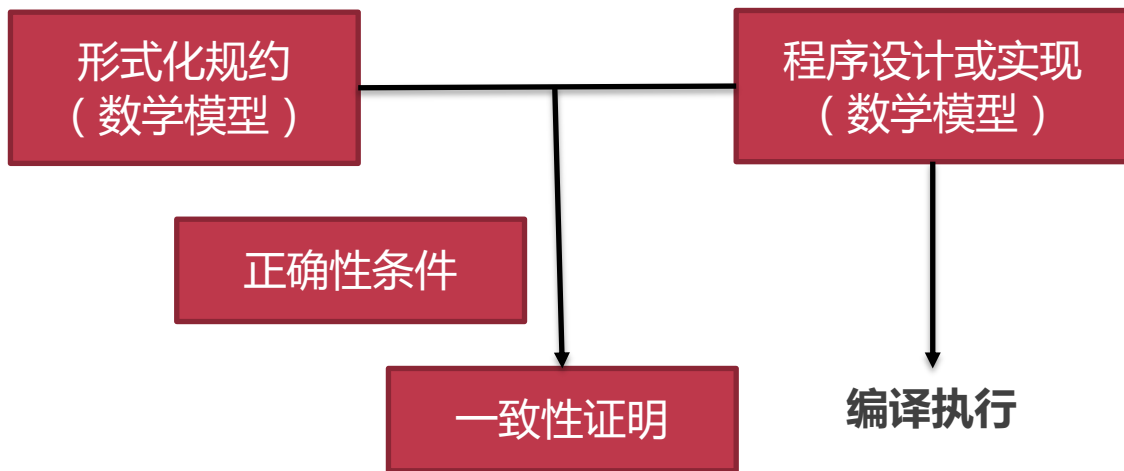
软件测试



形式化证明

形式化证明的组成

- 形式化规约：精确描述期待的软件行为——用户视角的软件
- 实现：软件系统的一个实现——实现视角的软件
- 正确性条件：定义实现与规约相一致的含义
- 一致性证明：表明实现与规约相一致的数学证明



形式化证明成功案例

- **操作系统**

- seL4 , 首个形式化验证的微内核 , SOSP'09最佳论文
- CertiKOS , 耶鲁大学开发的支持并发的内核[OSDI'16]

- **文件系统**

- FSCQ , 具有崩溃安全性的串行文件系统[SOSP'15]
- AtomFS , 接口具有原子性的并发文件系统[SOSP'19]

- **分布式协议**

- Amazon , 广泛应用TLA+用于保障软件协议设计的正确性
- 证明分布式共识协议Paxos与Raft联系与优化迁移[PODC'19]

形式化证明的挑战与现状

- **技术性挑战**

- 如何描述系统（及系统级性质）的正确性定义
 - 如崩溃一致性，安全性（noninterference），进展性（liveness）等
- 证明理论：如何有效证明正确性定义？
 - 并发系统的证明依然非常困难

- **工程性挑战**

- 可扩展性：随着系统规模变大，如何控制证明成本？
- 可维护性：随着系统持续开发，如何维护证明？

- **形式化证明现状**

- 仅自动化验证技术在工业界推广
- 交互式证明的开销仍然较高，仅有部分应用
- 依然存在许多开放性问題——很多机遇与挑战！