高级操作系统

论文阅读报告

2015310652 高伟

June 1, 2018

Contents

1 GPUvm: Why Not Virtualizing GPUs at the Hypervisor

本文是 USENIX ATC 2014 的一篇关于 GPU 虚拟化的文章，来自庆应义塾大学计算机系的 Kenji Kono 研究小组。第一作者是该系的一个博士生 Yusuke SUZUKI，主要研究浏览器、操作系统、加速器等。通讯作者是 Kenji Kono，研究方向涉及操作系统、网络安全、分布式和并行系统、编程语言系统等。

1.1 背景

注： 本文发表于 2014 年，彼时 GPU 发展与现在的情况差别较大，因而文章部分地方的观点与现今实际情况有所不同。

GPU 是一种被广泛使用的数据并行处理器，包括上千个计算核心，双精度计算峰值性能可以超过 1TFLOPS，每瓦特的计算性能也超过 CPU，是一款计算性能和绿色性能都比 CPU 强的处理器。在服务器和云计算环境中，GPU 并不是第一类的计算资源，往往是作为计算加速部件使用，大多数的 GPGPU 应用也停留在研究层面。其原因主要是 GPU 及其相关的软件系统不支持虚拟化，无法为多个用户提供资源隔离。

目前的 GPU 虚拟化方法有三类：

- I/O Pass-through（图1），GPU 直接对 VM 的驱动可见，可以最大化的减少虚拟化开销，但是缺陷是每个 GPU 只能对应一个固定的 VM，不能进行复用。
- API remoting（图2），面向多任务设计，并且相对容易实现。host 拦截并整合 VM 的 API 调用完成虚拟化，但是需要重写 driver。其缺陷是需要所有 VM 的库处于同一版本，限制了自由度。
- Para-virtualization 则是需要 Hypervisor 实现一个 GPU 的完整模型，然后 VM 修改相应的驱动支持该模型。

以上三种方法均很难或者不能为多个 VM 提供 GPU 资源隔离，并且需要修改驱动。

本文提供了一种新的虚拟化实现 GPUvm，基于 Xen Hypervisor 设计，通过虚拟存储映射 MMIO、GPU 影子通道、GPU 影子页表和虚拟调度器的方式，实现了多个 VM 的 GPU 资源隔离，并且不需要用户修改任何已有的驱动。

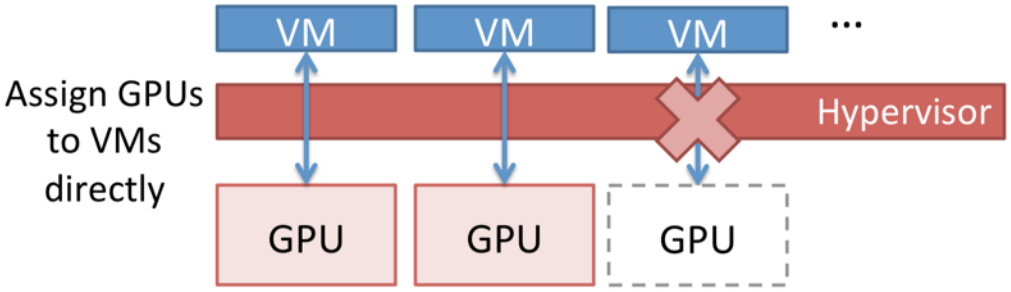

Figure 1: I/O pass-through

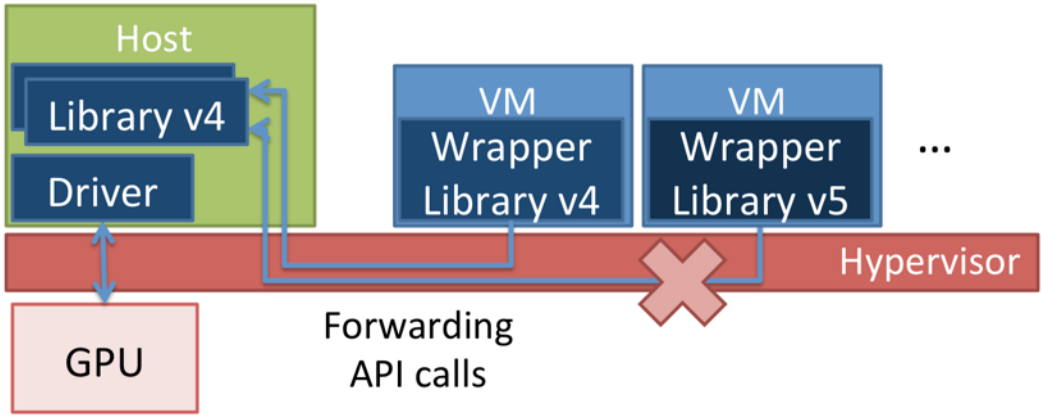

Figure 2: API remoting

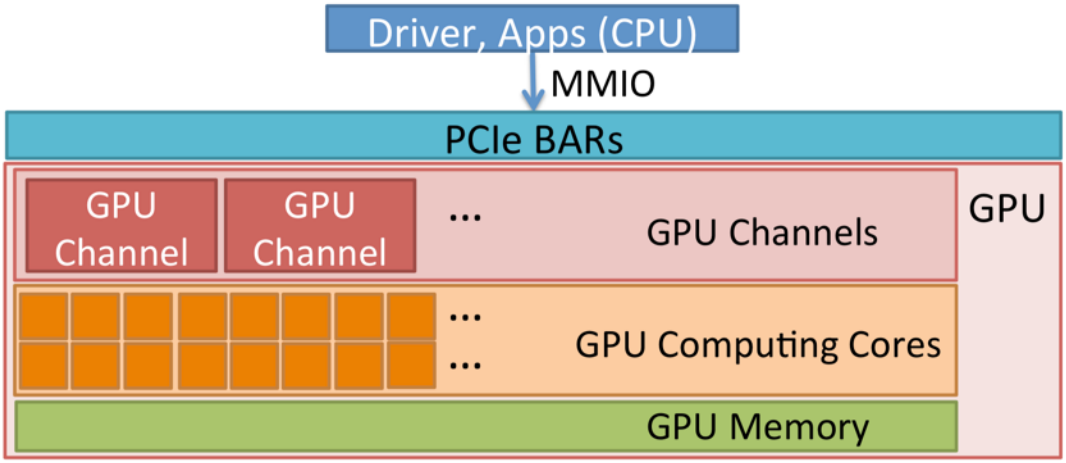

Figure 3: GPU architecture

1.2 主要工作

GPU 内部（图3）由 cores（计算核心）、channel（通道）和 memory 组成。channel 是用于向 cores 发送命令的通道，channel 数目固定，多个通道可以并发工作。memory 由 GPU 页表管理。

系统整体架构如图4所示。其设计思路是，当 VM 加载 GPU 驱动时，为每个 VM 提供一个本地的 GPU 模型，也就是虚拟的 GPU。当 VM 对本地的 GPU 模型进行操作时，并不直接操作实体 GPU 设备，而是被重定向到 Hypervisor，Hypervisor 转发到 GPUvm 进行处理。GPU Access Aggregator 将 VM 请求对应到 GPU 上专属本 VM 的资源部分，其通信采用 client-server 的模式，从而使得多个 VM 可以相互隔离的使用同一个 GPU 设备。

GPUvm 的内部架构如图5所示，其核心包含四个部分：Resouce Partitioning，GPU Shadow Page Tables，GPU Shadow Channels，GPU Fair-Share Scheduler。

1. Resource Patitioning 将物理内存和 MMIO 空间通过 PCIe BARs 划分成多个连续的地址空间部分，每个部分分配一个单独的 VM。

2. GPU Shadow Page Tables 负责将 VM 的 GPU 虚拟地址转换为 GPU 或者主机的物理地址。设备驱动在每次一个页被更新的时候需要刷新 TLB。而 GPUvm 可以拦截 TLB 刷新请求，从而更新相关的 GPU Shadow page table 条目。

3. GPU Shadow Channel 用于隔离从 VM 发出的 GPU 访问。GPU channels 的物理索引对 VM 不可见，但是每个 VM 可以拥有自己的 channel 虚拟索引，GPUvm 维护虚

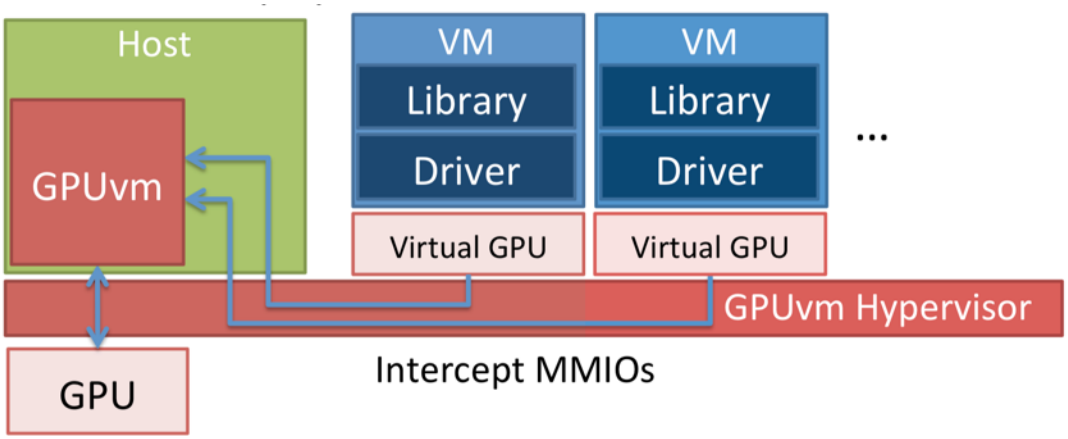

Figure 4: 整体架构

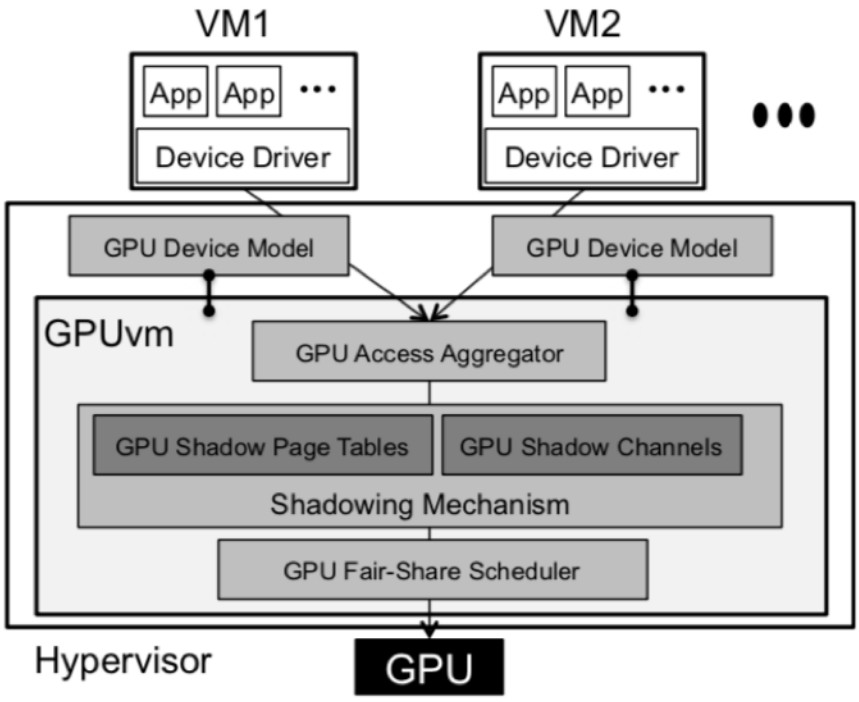

Figure 5: GPUvm 内部逻辑架构

拟索引到物理索引的映射。GPUvm 管理所有的 VM 使用的 channels，由于每个 VM 的 channel 使用请求都可以被 GPUvm 拦截，所以 GPUvm 可以在完成虚实转换之后，使用物理索引激活相应的 channels。

4. GPU Fair-Share Scheduler 基于非抢占的调度模式，使用 BAND Scheduling 算法完成虚拟请求的调度。

GPUvm 还采用了多种优化技术以减少其带来的开销，比如 BAR Remap，Lazy Shadowing，Para-virtualization 等。

1.3 实验以及局限性

通过实验验证，相对于没有虚拟化的情况，GPUvm 带来较大的开销，即使是经过多级优化，其 Para-virtualization 性能降低依然有 2-3 倍，全虚拟化性能降低达到了 40 倍。而且对于 8 个 VM 的情况，其开销增加相对于 4 个 VM 十分明显。可见，GPUvm 尚不适合部署使用，而且横向扩展性能也较差。

2 Hyperkernel: Push-Button Verification of an OS Kernel

本文是 SOSP2017 的一篇关于自动验证的文章，来自华盛顿大学的系统实验室 Xi Wang 老师小组，第一作者是 Luke Nelson，为该实验室的一名博士生，从事 HyperKernel 的设计开发工作。

2.1 背景

系统内核是计算机系统的关键部分之一，为顶层的应用提供底层抽象和服务。如果内核中存在 BUG，则会严重影响整个系统的正确性和安全性，导致应用错误甚至系统宕机，所以如何保证系统内核的正确性成为研究的重点之一。先前的研究采用形式化验证的方式，首先构建一个正确操作的 *specification*，然后使用机器证明代码的实现逻辑符合 *specification*，但是额外开销很大。作者提出了一种既可以自动化验证 OS kernel(HyperKernel, 基于 *xv6* 开发) 的正确性，同时开销又可以接受，只需要十几分钟的时间。

作者以 *xv6* 系统为基础，针对于自动化验证做了诸多删减和修改，开发了 HyperKernel。HyperKernel 既可以通过 state-machine SPEC 和 declarative SPEC 的方式验证 50 多个系

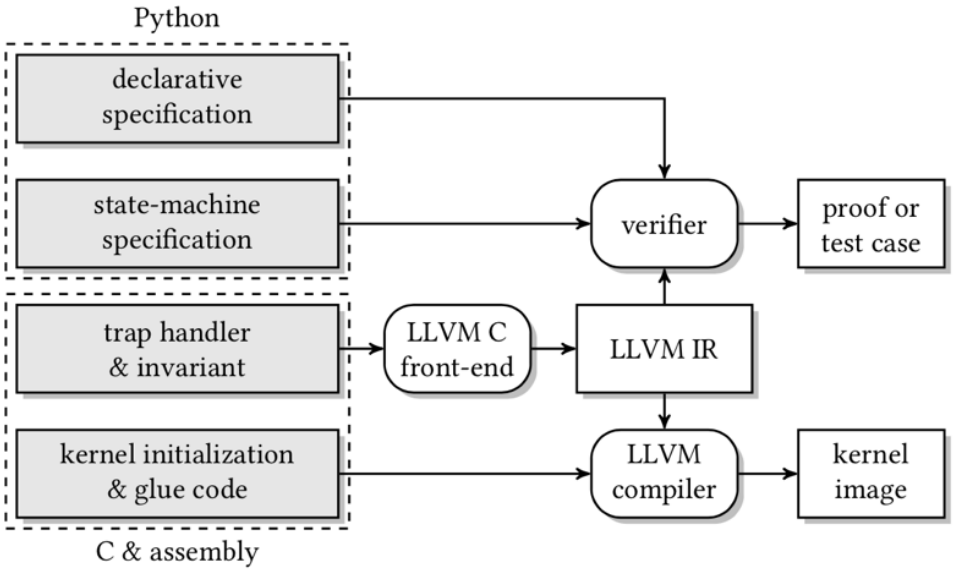

Figure 6: The Hyperkernel development flow. Rectan- gular boxes denote source, intermediate, and output files; rounded boxes denote compilers and verifiers. Shaded boxes denote files written by programmers

统调用（即文中的 trap handler) 的正确性，又保证了较低的验证开销，自动化验证通过 Z3 SMT 求解器实现。

2.2 主要工作

自动验证主要有 3 个挑战：1）接口设计需要平衡可用性和方便自动证明，所以 HyperKernel 的设计中，将接口约束为有限接口，即除去无界循环和递归操作，从而可以方便的使用 Z3 求解器验证；2）虚拟内存管理使得内核和用户空间会存在重叠的情况，在验证时需要将虚拟地址转换为物理地址，大大增加了验证的复杂性。作者通过借助于硬件虚拟化技术，使得内核地址和用户地址分离，从而简化了验证过程；3）HyperKenel 使用 C 语言编写，而 C 语言的复杂丰富的语义特征致使难以进行准确的验证，作者采用对 LLVM IR 中间表示的验证绕开这个问题。

除此之外，HyperKernel 运行于单核系统，并且系统初始化和胶水代码并没有进行验证，采用设计针对性 checker 的方式保证正确性。

HyperKernel 开发过程如图6所示。编程人员编写两个 SPEC，一个是用于函数正确性验证的 state-machine SPEC，另一个是 declarative SPEC。

有限接口设计是 HyperKernel 设计的一大特点。其目的是保证所有的系统调用（HyperKernel 的系统调用为原子操作）在可预计、可接受的时间内完成，而不能无限阻塞下去。其思路是将所有的可能无限循环的情况进行改良，改为确定的、有限的实现。HyperKernel 以 xv6 和 POSIX 为基础，设计了适合横切属性的有限接口，主要采用三种方法：

1. HyperKernel 使用类似 exokernel 的方式，需要用户显式资源调用，因而自然存在资源泄露的风险。作者借用 xvf6 的进程结构和参数，采用引用基数的方式，保证进程释放的同时，起占用的资源也已经回收。

2. 传统的进程创建过程涉及复杂的过程处理，作者借鉴了 exokernel 的方法，提供基本的创建进程结构的系统调用，繁杂的资源复制等工作通过用户库实现。类似的，HyperKernel 也实现了细粒度的虚拟存储管理。

3. HyperKernel 内部主要采用数组的数据结构，但是也采用链表存储空页和准备好的进程，可以简化用户空间代码的实现。

在有限接口之上，即是 SPEC 验证部分。对于 state-machine SPEC，由抽象状态和抽象状态转化组成。通过使用 python 代码，编写可以转换为 S3 求解器输入的 SPEC。为了进一步增加正确性执行度，在 state-machine SPEC 之上，可以继续增加 declarative SPEC，用于验证 state-machine SPEC 的正确性，其思路是使用横切属性（crosscutting properties，即不变条件），即保证某些条件（比如进程内文件描述符与文件引用匹配）恒成立。

验证过程依据两个定理 (2.1，2.2)，从而完成正确性验证。一方面通过 state-machine SPEC 保证了代码的实现与设计初衷是相同的，另一方面采用 declarative SPEC 保证了设计初衷是正确的。

定理 2.1. *内核的实现基本与 state-machine SPEC 相符*

定理 2.2. *state-machine SPEC 满足 declarative SPEC*

为了验证上面两条定理，验证器（verifier）的需要将 LLVM IR 和横切属性编码为 SMT 表示。作者将两个定理表达为具体的可以通过 S3 求解器推理验证的形式，具体方法包括去除无定义行为、LLVM IR 到 SMT 编码、明确专有属性、明确引用计数共享属性等。

HyperKernel 的设计使用了硬件虚拟化技术（Intel VT-x, AMD-V），内核运行在 host 状态，用户进程运行在 guest 的 ring0 态，一方面有利于分离地址空间，另一方面在捕获中断的同时，可以把内核不关注的异常留给用户处理。设计中也采用了 exokernel 的思想，用户进程需要显式的资源管理。

2.3 局限性

HyperKernel 有其局限性。首先，初始化代码繁多且复杂，相对于原来的系统调用，HyperKernel 的 hypercall 开销较大。其次，验证器依据 LLVM IR 表示，所以可能会丢掉部分 C 语言的未定义行为 BUG。还有就是 HyperKernel 针对于单核 CPU，不支持线程、cow fork、页共享、UNIX 权限等。

3 Locating Cache Performance Bottlenecks Using Data Profiling

本文是 EuroSys 2010 的一篇文章，来自 MIT 的计算机科学和 AI 实验室（Massachusetts Institute of Technology Computer Science and Artificial Intelligence Lab）的并行和分布式计算小组（Parallel & Distributed Operating Systems Group ）。第一作者是 Aleksey Pesterev，第二作者和第三作者是分别是该小组的 Nickolai Zeldovich 和 Robert T. Morris 教授。

3.1 背景

程序的执行性能与数据缓存的利用情况息息相关。现有的主流的 Profiler 工具主要统计各行代码的 cache miss 频率，但是 cache miss 可能散布在整个程序中，很容易忽视那些平均 miss 次数较少但是指令数目很多的情况。cache miss 的原因主要包括无效缺失、容量缺失和关联缺失（冲突缺失）三种，针对不同的缺失情况，减少缺失率的策略不同，因而准确的识别 cache miss 的类型很重要。

本文设计实现了 DProf 工具，旨在帮助编程人员发现、理解并消除 cache miss。DProf 通过性能检测硬件获取 cache miss 的信息，并进一步分析缺失的原因。识别缺失原因之后，DProf 列出造成缺失的具体数据及其类型，而不是代码，帮助编程人员消除缺失。

3.2 主要工作

DProf 使用 AMD 基于指令采样的硬件（instruction-based sampling(IBS) hareware）和 x86 调试寄存处（debug registers）实现访存路径跟踪，使用 LInux kernel allocator 帮助识别缺失类型，DProf 已经在 Linux 内核中实现。

DProf 收集的程序数据分为路径数据和地址集合两种。

Field	Description
type	The data type containing this data.
offset	This data's offset within the data type.
ip	Instruction address responsible for the access.
cpu	The CPU that executed the instruction.
miss	Whether the access missed.
level	Which cache hit.
lat	How long the access took.

Figure 7: An access sample that stores information about a memory access

- **路径数据**。DProf 随机选择一个数据对象集合，并记录每个数据对象从分配到回收的整个生命周期，包括在每一级 cache 的命中率、平均访问时间、从分配到访问的平均时间和标识访问的核是否发生了变化的标志。DProf 整合具有相同数据路径的数据对象，并记录对给定数据类型，不同数据路径的出现频率。

- **地址集合**。地址集合包括数据对象的地址和数据类型，DProf 使用地址计算数据对象在 cache 中的关联组。

DProf 首先通过性能检测硬件采样得到原始数据，原始数据包括数据访问样本（如图7）和对象访问历史（如图8）两种。数据访问样本使用 AMD 的 IBS 收集，然后通过访问地址解析得到数据类型，解析的过程需要考虑到静态分配和动态分配（使用 linux kernel allocator）两种情况。对象的方位历史，则直接采用 AMD 或者 Intel 的调试寄存器收集得到。原始数据收集得到之后，则根据数据类型、路径信息进行整合，计算统计信息。

DProf 运行程序之后，会得到如下几个视图：

- **数据概况**。DPorf 展示程序的数据概况，即根据 cache miss 数目排好序的数据类型列表。数据概况反映了数据的 cache miss rate 和 CPU 变化信息多用于分析不同数据类型的缺失原因。

- **缺失分类**。缺失分类视图展示了每种数据类型的主要缺失原因，DProf 中的具体分类包括：无效缺失、冲突缺失和容量缺失三类。

- **工作集**。工作集视图展示了程序中哪些数据类型最为活跃，在任意给定时刻每种数据

Field	Description
offset	Offset within data type that's being accessed.
ip	Instruction address responsible for the access.
cpu	The CPU that executed the instruction.
time	Time of access, from object allocation.

Figure 8: An element from an object access history for a given type, used to record a single memory access to an offset within an object of that type

类型有多少处于活跃状态, 以及每种数据类型使用的 cache 关联组 (cache associativity sets)。DProf 通过运行一个 cache 模拟器，获取信息并完成统计。主要用于跟踪容量缺失和冲突缺失的原因。

- **工作集**。工作集视图展示了程序中哪些数据类型最为活跃，在任意给定时刻每种数据类型有多少处于活跃状态, 以及每种数据类型使用的 cache 关联组 (cache associativity sets)。主要用于跟踪容量缺失和冲突缺失的原因。

作者是用 memcached 内存数据库和 Apache 进行实验测试，达到了 16-57% 的性能提升。

3.3 局限性

DProf 受限于 AMD 和 Intel 的性能检测硬件的数据情况以及调试寄存器的数目，难以后的精确的统计数据。

4 My VM is Lighter (and Safer) than your Container

本文是 SOSP 2017 的一篇关于虚拟操作系统的文章，是欧洲 NEC 实验室（NEC Laboratories Europe）和布加勒斯特理工大学（Univ. Politehnica of Bucharest）合作完成的。第一作者是欧洲 NEC 实验室的软件工程师 Filipe Manco。

4.1 背景

轻量级的虚拟化技术（比如 docker、LXC 等容器）相对于虚拟机有着实例化时间短、内存占用小、单机密度高的优势，近年来发展迅速，得到了广泛的使用。另一方面，虚拟机相对于容器有着更好的隔离安全性。作者基于 Xen 开发了 LightVM，既有虚拟机的隔离安全性，又有着容器的性能。

4.2 主要工作

本文首先缩减虚拟机的镜像和内存占用。大多数的容器和虚拟机用来运行单个应用，因此通过缩减 VM 不必要的功能，只保留针对特定应用的功能模块可以大幅度减少存储使用。作者借鉴了 Unikernel 和 Tinyx 的技术。

1. Unikernel：Unikernel 是一个很小的与应用关联的操作系统，很低的存储占用，但是依然处于研究阶段。

2. Tinyx：Tinyx 可以自动构建一个满足某个应用最需求的最小的 VM 镜像，通过提供应用和目标平台，Tinyx 可以分别构建一个基础的发布版和相应的优化后的内核。

本文分析了 Xen 的性能，从而获得 Xen 的性能瓶颈。Xen 架构如图9使用 Debian jessie 最小化安装、Tinyx、daytime unikernel 三种类型的 VM，分别测试在 Xen 上的创建和启动的开销，如图10，总结发现不同大小的 VM 创建时间差别很大；随着 VM 的数目的减少，创建时间占得比例增大；对于轻量级的 VM，实例化时间是主要的瓶颈。本文进一步分析了具体操作的性能，如图11，发现 XenStore 和设备创建是主要的开销。

作者基于 Xen 控制平面重新设计、开发了轻量级虚拟化 LightVM，架构如图12。

Hypervisor 本身已经包含了很多 VM 的信息，因而可以通过扩展 Hypervisor 的功能，从而除去 XenStore。LightVM 在创建和启动过程中不再使用 XenStore，代之采用轻量级的 noxs 驱动。noxs 通过共享内存的方式通信，为每个 VM 分配一个页存储 VM 的信息，替代了原始的通信方式，解决了 XenStore 的扩展性问题。

VM 创建时的很多工作是不必要的，很多代码对所有或者配置相同的 VM 来说是一样的。LightVM 使用了分离的工具链，将 Xen 的工具链使用 libchaos 替代，并分为 prepare 和 execute 两部分：prepare 部分负责处理所有 VM 共性的工作，比如分配 CPU 资源、产生 ID 和其他管理信息，并床架了 VM shells 池；execute 部分负责其他的工作，首先获取一个 VM shells，然后完成加载内核镜像以及设备初始化等工作。chaos/libchaos 减少了 VM 创建过程的工作量，加快了创建过程。

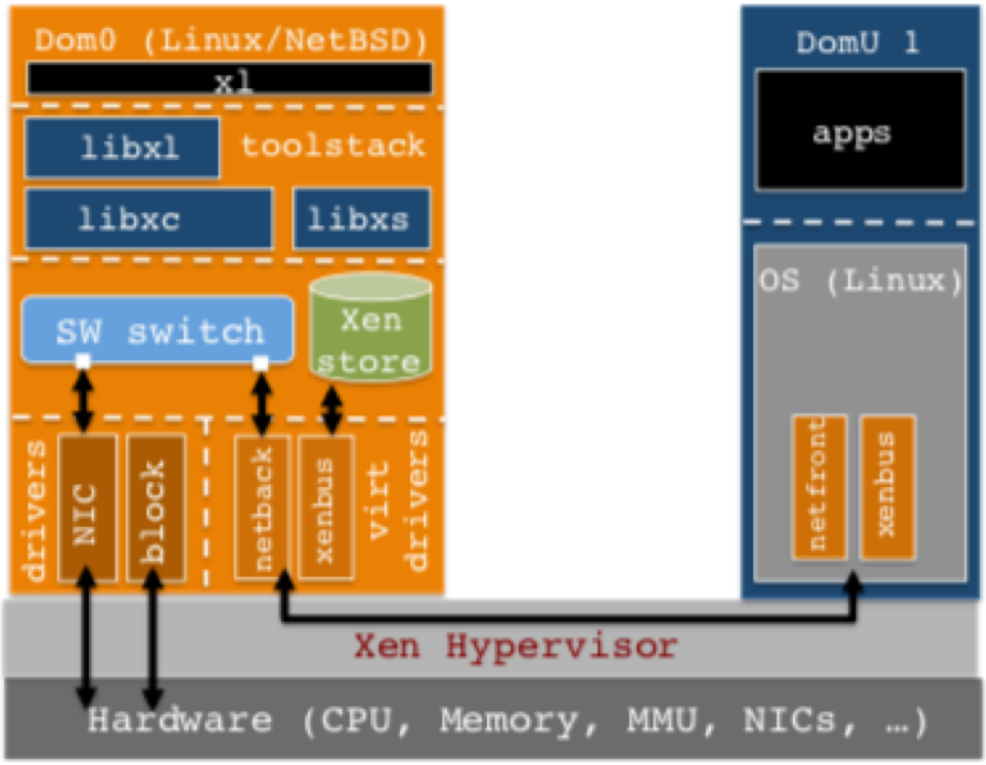

Figure 9: The Xen architecture including toolstack, the XenStore, software switch and split drivers between the driver domain (Dom0) and the guests (DomUs)

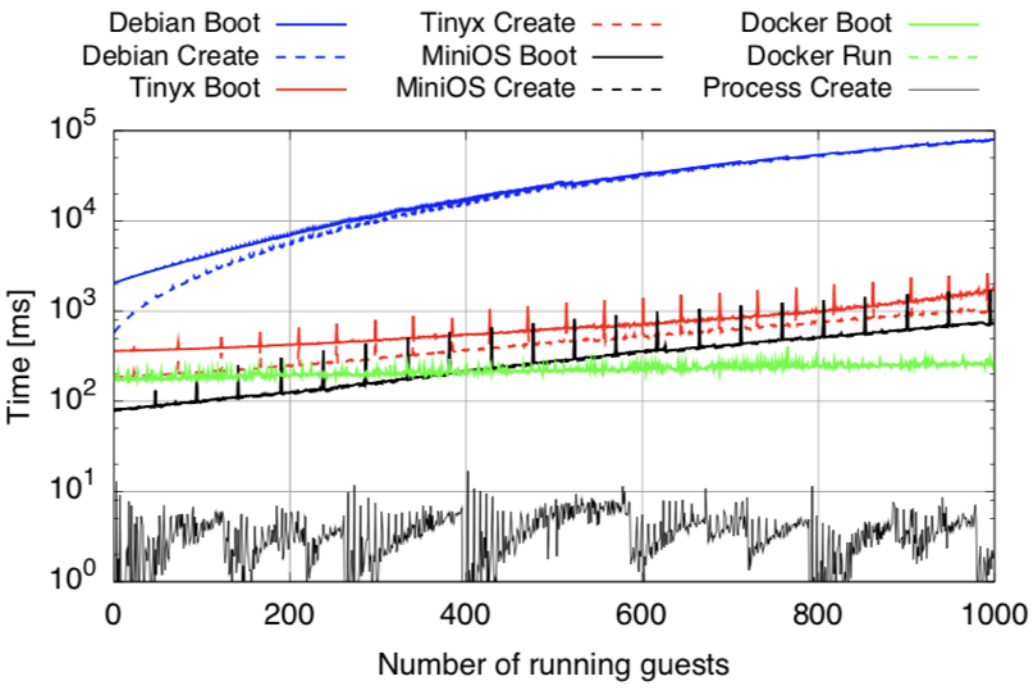

Figure 10: Comparison of domain instantiation and boot times for several guest types. With small guests, instantiation accounts for most of the delay when bringing up a new VM

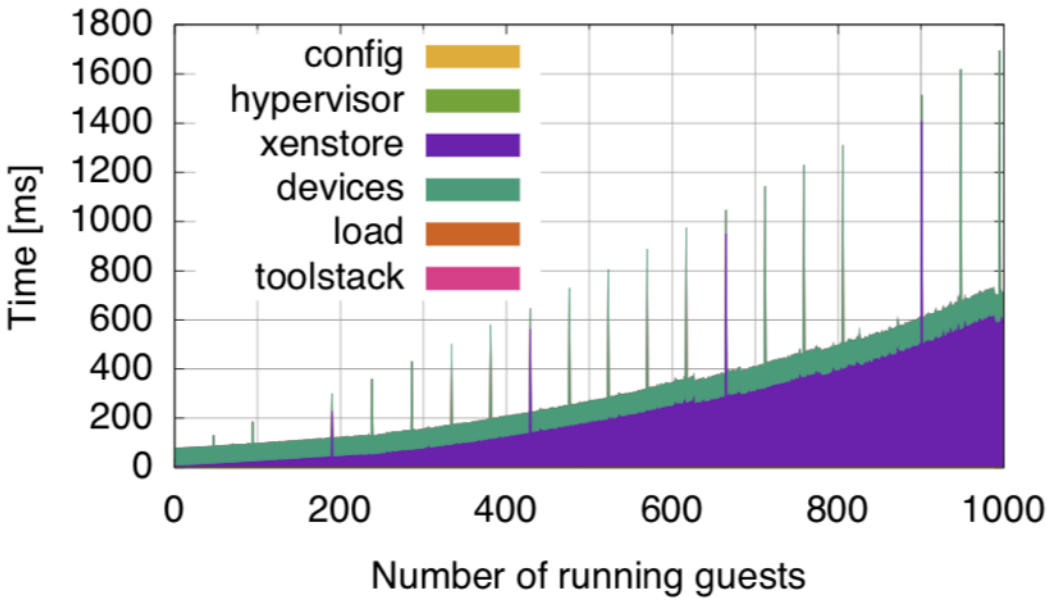

Figure 11: Breakdown of the VM creation overheads shows that the main contributors are interactions with the XenStore and the creation of virtual device

在创建虚拟设备驱动方面，LightVM 使用 xendevd 替代原来比较耗时的脚本执行方式，加快了设置速度。

4.3 测试和总结

经过测试 LightVM 创建 VM 的时间在 100ms 以内，启动最小化的 VM 只需要 2.3ms，与执行 fork/exec 耗时相当。不论有多少 VM，LightVM 的创建和启动时间基本为常数，使得性能可以与容器媲美。

5 The Scalable Commutativity Rule Designing Scalable Software for Multicore Processors

5.1 背景

在多核下，接口的设计与可拓展性有密切的联系。传统的检验多核软件的可拓展性的方法通常需要在实现完软件后才开始验证并多次迭代，作者试图在接口设计阶段就充分考

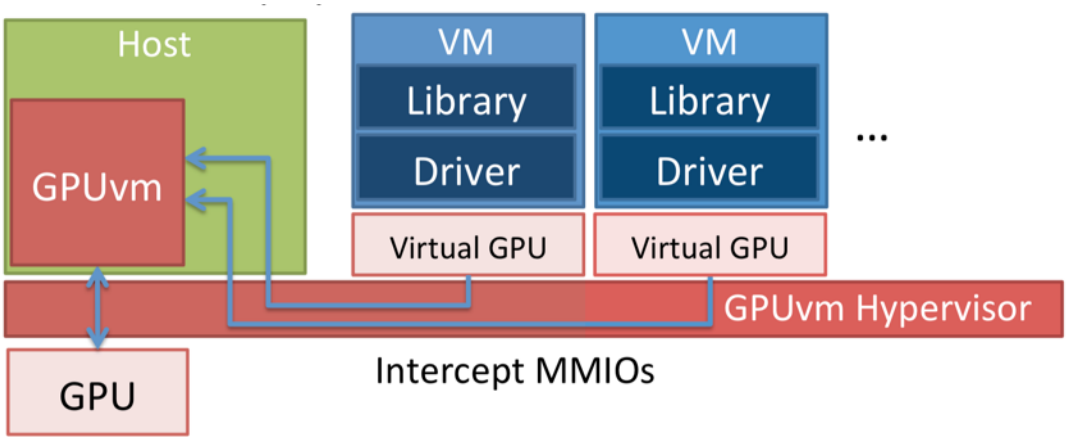

Figure 12: LightVM architecture showing noxs, xl' s re- placement (chaos), the split tool-stack and accompany- ing daemon, and xendevd in charge of quickly adding virtual interfaces to the software switch

虑可拓展性和操作的可交换性间的关系，利用其是否会产生 cache line 的争用来判断可交换性，并利用 COMMUTER 工具来寻找软件设计内部存在的制约可拓展性的因素，推导出为实现，可拓展性应满足的条件，和测试这些条件的测试用例，为多核软件这在设计初期就提供可拓展性的指导，提高了开发效率。

5.2 主要工作

文章主要贡献之一是 SIM 可交换性。即在一段操作序列/历史中，如果不同线程的操作间交叉 (interleaving) 执行而最后结果和状态一样，则这段历史是可以避免冲突的，也即没有 cache 的争用，可以开发并行性。通过找到一组特定的条件使得满足 SIM 交换性后，则比存在保证可拓展性的实现。

通过利用 SIM 交换性的启发性，作者在设计接口时提供了建议。首先是分解复合操作。POSIX API 通常组合包含不可交换的几个子操作。如果可能，这些操作应该分开。第二个是确保规范非确定性，这允许较少的同步和发挥更多的并发性。第三，许多接口操作限制了必须采用严格排序，在当弱有序就能满足应用需求是限制了可拓展性。最后，异步释放资源意味着昂贵的同步可以被放弃，并且可以再次改善交换性。

本文最后的主要贡献是 COMMUTER 系统。该系统允许开发人员在实现过程之前指定接口，COMMUTER 通过抽象接口的定义，用符号执行的方法寻找能满足可交换性的条件，并生成测试用例测试是否满足可交换性，这些都能指导开发者实现可拓展性的关键点，最后系统还能对实际实现进行测试，判断不同操作间是否有冲突。

5.3 实验

作 MTRACE 工具识别出，在 13,664 个 Linux 测试用例中，只有 9,389 个是无冲突的。在许多情况下，公共引用计数器是主要原因。为了改善这一点，他们的 sv6 内核使用 COMMUTER 系统进行了修改后，13,528 个相同的测试用例能实现避免冲突。此外，还进行了实际运行测试以确定实际的拓展性改善。使用 sv6 系统，statbench 和 openbench 都表明新的可交换 API 比传统的 Linux 版本更具可扩展性。

6 RID Finding Reference Count Bugswith Inconsistent Path Pair Checking

6.1 背景

引用计数广泛使用于操作系统内核以对资源进行管理。然而，在大型系统中正确使用引用计数是非常重要的，因为一个引用计数的增加与对应的减少是由开发人员来管理的。而这很容易由于人的原因而出错。例如，在操作系统内核的子系统中，其通常用于同步管理、动态内存管理和动态电力管理。对引用计数错误使用会引发各自情况，包括内存资源耗尽、死锁、异常高电力消耗和安全漏洞。

6.2 主要工作

这篇文章提出了一个非一致性路径对检测 (inconsistent path pair checking) 机制，用来静态检测 OS 内核中的引用错误。

一对非一致性路径 (简称 IPP) 由两个路径组成，其满足以下条件:

1. 在同一个函数中 (Both paths are in the same function)，
2. 都从一个函数入口开始结束于函数的结尾 (Both paths start from the entry of the function and end at the exit)，
3. 对一个引用有不同的修改 (The two paths have different changes to a refcount)，
4. 在运行时，函数的参数和返回值相同 (at runtime, indistinguishable outside the function by checking arguments and the return value)。

非一致性路径检测不依赖于函数的上下文，而是像库和驱动一样，是操作系统内核的一部分。并且它对引用计数在函数中的使用不做任何假设，它仅仅需要引用计数的 API 规格即可。

检测非一致性路径分为三个步骤：

1. 计算所有路径；

2. 总结每个路径的独立性；

3. 检测每个路径的一致性，报告非一致性路径，并将所有结果综合。

基于非一致性路径分析、程序抽象和总结性地过程分析，作者实现了一个静态检测器，叫做 RID。

6.3 实验

用 RID 检测 Linux 内核，实验表明 RID 对于检测开发者对内核 API 规格误解和查找引用错误非常有效，并找到了 83 个被开发者确认的新错误。用 RID 检测 Python/C 程序，检测出 100+ 个错误。

1.0