阅读论文

刘喆 2017210813

第一篇：The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors

本文作者认为新的多核的结构和繁多复杂的workload造成了软件的可拓展性收到限制，并猜想真正限制软件可拓展性的问题可能出现在接口设计上。因此，作者考察了接口(如system call)对软件扩展性的影响。作者认为，接口的可拓展性就是当接口操作交换时，能够以拓展的方式实现。这个接口可拓展规则，支持在整个软件设计过程中对可拓展性进行论证。这样就能够指导接口的设计、明确设计目标，并可以系统地、脱离具体workload进行可拓展性测试。作者进而将这个可拓展性规则形式化并证明其正确性。接着设计了一种自动测试可拓展性的工具：Commuter。进而设计了一种可拓展的类似于POSIX的内核。

作者通过实验证明，一条简单的cache line竞争会对系统的可拓展性造成很大的影响。只有两个操作没有竞争关系时，我们才能确定的说这两个操作可以并行执行，是可拓展的。这两个操作之间应该是没有依赖关系，没有相互的通信，进而在执行的过程中也就没有竞争冲突。比如创建文件，在同一个文件夹下创建相同的文件就会存在竞争问题，在不同的文件目录下面创建就不存在冲突问题，不存在冲突问题的操作可以并行执行，我们就认为这些操作是可拓展的。作者通过形式化的表述，分析出了一些可以交换的规则，这个分析过程较为复杂，是一项大工作量的项目。作者同时应用设计的交换规则，对当前的linux系统进行了测试，发现了系统中大部分均是存在冲突的设计，比如文件目录锁、地址空间锁、文件描述符等冲突情况。于是，作者重新设计了sv6，一种类POSIX内核，文件系统和虚拟内存系统依据交换规则进行设计，应用使用Commuter提供的指导进行设计。经过进一步的测试，能够发现系统中99%的设计是不存在冲突的，这就是说sv6这个内核的可拓展性非常好，这就是启发了工业界在设计内核系统等时，需要按照文章提供的思路，在接口设计上多下功夫，能够获得系统良好的可拓展性，提高系统的性能。作者进一步对系统中运行的多线程邮件服务进行了测试，测试结果也是有力的证明了当前sv6系统能够很好的给予app可拓展性上的支持。文章解决的核心问题是, 如果把接口和实现良好的分离开来, 那接口的设计哪些是天然的限制, 使得无论怎样实现, 都无法突破这些天然限制，而哪些是我们可以通过一定的规则设计可以解决的这些限制的。

这篇文章的工作量很大，他们在文中实现的SV6应该是他们组一直在用的一个名为XV6的本科生教学操作系统。Commuter的实现工作也是不易。

第二篇：Hyperkernel: Push-Button Verification of an OS Kernel

本文针对操作系统内核安全性和正确性提出了自己的保证方法。操作系统内核是系统中重要的组建，对于保障应用程序正确运行具有重要作用，内核中如果出现bug将直接影响整个系统运行。当前的主流验证安全性的方法是预测一些系统行为，并根据这些行为设计对应的实现规则，通过分析操作是否符合规则来确保系统是正确安全运行的。这种验证方法存在一个问题，就是验证带来的负载过重，直接影响系统自身运行。作者使用类Unix的基于xv6的操作系统内核，实现了全自动的正确性验证。可以保证系统调用的正确性和进程的隔离性等。Hyperkernel设计了有限的接口来解决xv6的系统调用需要写循环不变式的问题，分离了用户和内核空间，并使用内核映射技术解决内核指针问题。作者使用LVM 核心库提供了与编译器相关的支持（可以作为多种语言编译器的后台来使用）。能够进行程序语言的编译期优化、链接优化、在线编译优化、代码生成。LLVM的项目是一个模块化和可重复使用的编译器和工具技术的集合。作者证明了LLVM的中间表示。作者使用声明的规范来验证状态机规范是否安全的执行，程序是否被安全的调度，进程之间是否安全的隔离。对于进程p中使用的虚拟地址，该虚拟地址对应的page必须由进程p独占，这样保证了进程的隔离性和安全性。如果LLVM操作在执行过程中没有按照声明规范执行，则认为系统中出现了bug，不能保证系统正确、安全的运行。为减轻运行验证的负担，符号化执行测试时是全自动执行验证程序，但需要保证程序没有循环且状态有限。作者采用有限的借口设计，显式地调用资源，分解复杂的系统调用。将原来的系统调用程序更加细致化操控。这样内核只需要记住哪些资源属于哪些进程，由进程主动申请资源，内核只是进行确认该进程能否申请使用该资源即可。保证了进程的隔离性，也减轻了内核的负担和风险。这篇文章的主要工作集中在规范的设计，以及如何将LLVM的数据结构与这些规范建立对应关系。取得的成果是在添加和证明系统调用时效率很高，本文跟第一篇论文均是从接口设计入手，考虑如何改进系统属性，可见，系统接口是一个很好的研究领域。

第三篇：My VM is Lighter (and Safer) than your Container

本文采用容器技术，提出了LightVM这样一款虚拟机，实现了轻量、安全的虚拟。容器技术具有轻量、高效的特性，如今已广受欢迎。与传统的虚拟机技术相比，由于容器不存在硬件虚拟及运行完整操作系统等额外开销，因此对系统资源的利用率更高。容器的应用执行速度、文件存储速度，都要比传统虚拟机技术更高效。由于系统资源利用率更高，且容器的镜像通常较小，相比虚拟机技术，一个相同配置的主机，往往可以运行更多数量的应用。但容器技术之所以没有取代虚拟机，一个很重要的原因在于容器的隔离性不足，在安全上，容器无法与虚拟机想提并论。容器技术相比虚拟机技术存在着更多的安全性问题。本文是基于hypervisor的轻量级的虚拟化技术，能够实现快速启动（容器通常可达到秒级甚至毫秒级的启动时间，而虚拟机往往需要数十秒甚至数分钟才能启动）、大应用量并行（在一台主机上，同时运行成百上千的容器实例）、暂停/恢复（容器实例拥有快速暂停/恢复的能力，相比于虚拟机快照技术更加高效）。为此，作者从缩小虚拟机的镜像和内存占用大小出发。采用容器共用系统内核，实例所占用的内存很小，进而实现实例的高密度。

作者通过观察发现，多数VM和Container都是运行单应用的，所以不需要VM具有超强复杂的功能，只要剪裁到够用即可。为此作者探究了Unikernel和Tinyx两种方式来创建适合单应用运行的最小镜像。实验表明，利用Unikernel能够创建性能最好的镜像，但开发成本较高；而利用Tinyx，可以使VM镜像的创建类似于创建容器镜像一样简单，性能比Unikernel差，优势是效率可以同容器相比。作者进一步工作对Xen进行了改进。通过性能分析，作者发现影响VM启动的主要性能瓶颈是XenStore和device creation。XenStore是Xen提供的一个域间共享的存储系统，基于共享内存页与事件通道来实现虚拟机通信。XenStore中存储了各个虚拟机的配置信息，例如Domain ID，Domain Name，前后端设备，启动时间，虚拟机状态等。XenStore中存储了Domain之间共享的配置信息，Domain 0作为管理域，可以查看整个结构的内容，而Domain U只能查看自身的信息，这种设计方法提高了Xenstore的安全性。带来的问题就是维护XenStore的开销比较大，随着VM实例数量的增加而开销显著增长。为此，作者修改了Xen的架构，提出noxs（no XenStore）并扩展了hypervisor。使得前后端驱动使用共享内存交互实现，由此减少了domain切换的开销。作者进一步分解VM启动过程中的toolstack为prepare和execute两个phase，prepare phase中包含所有VMs 通用的功能，这部分可交给守护进程在后台周期完成，执行较为迅速。execute phase主要负责实例化过程中每个VM差异性的事务。通过对toolstack的分割，提高了启动的速度。本文实际上是对VM镜像的裁剪及Xen的改造，作者文章的题目非常抓人眼球。但是这样的设计也有一定的局限性，应用的场景目前来看也较为有限，能否大规模地推广还有待于是否有需求。

第四篇：Safely and Efficiently Multiprogramming a 64kB Computer

本文作者关注低功耗微控制器不能提供多可编程环境的特点，设计了面向低功耗平台的全新操作系统Tock，可以提供硬件保护、软件故障隔离、内存保护与隔离等功能。随着当前技术的进步和发展，嵌入式应用已经不再是单一的功能，而是趋向于软件平台提供各种应用支持。现在的嵌入式软件由于目标单一，组件之间设计均为深耦合、内存分配单一、模块之间也不够隔离。由于嵌入式的内存资源有限，一般在64Kb，功耗消耗低，内存在硬件上的保护也十分有限。作者着手解决嵌入式组件隔离、替换、并发等问题。由于不能保证代码的正确性，即不能保证程序不存在bug，所以，作者放弃了这个想法，设计了Rust编程语言的类型安全特性的。Tock的内核由称之为胶囊（capsule）的组件构建而成。这个胶囊是Rust结构的实例，包括自身字段、关联方式和可访问的静态变量等。进程无法访问授予的内存，转而使用了一种动态规定的MPU规则。当上下文切换时，MPU可以访问进程放在flash中的代码空间以及SRAM栈或堆中的数据。当进入内核执行后，MPU将被禁止。为在有限的空间里满足动态需求，作者设计了grants工具。通过对API进行限制，比如授予区域的内存只能在闭包中访问，指向该内存区域的指针不能越界等。可以用来在各个进程之间划分内核空间。它可以积极响应内存需求，而不影响其他正在运行的进程或内核。实验表明，Tock的线程资源中22%均可重复利用，这样从某种程度上所有的内存资源均可以被利用起来，因为只要线程不是并发执行，其内存资源均可重复利用。在安全性上，Tock也有较好的设计，设计内核时，避免进程数据结构的交叉，维护Rust的类型安全特性。是在嵌入式系统安全上的一个尝试。可以预想，未来嵌入式设备和相应的软件硬件将会越来越普及，但是低配置、低功耗、低成本的要求将一直都在，嵌入式设备由于其使用范围就决定了它不可能像服务器一样具有充足的资源配置，但它对性能、安全的要求，并不比普通的系统低，所以，如何解决安全性、多编程环境、动态需求等方面，本文提供了一个不错的思路。

第五篇：Addressing Shared Resource Contention in Multicore Processors via Scheduling

本文围绕多核竞争共享资源的问题，提出了一种高效的调度方法，能够充分利用系统资源，提高系统性能。普遍地观点认为对于运行在多核上面的多线程程序，为了追求系统运行效率和资源利用率，必须要做到负载均衡，充分利用每一个核的计算能力，达到高负载、快计算的效果。但是，很不幸，越来越多的研究发现事实上这种调度策略，有一个根本的难点无法解决，那就是每一个核上运行的不是一个独立程序，很可能这个核上的程序需要跟其他核上的程序共享共用相同的资源，这个弊端将导致了运行效率的损失，尤其是共享的资源是最快最高级cache（LLC）的时候。作者通过实验证明，两个不同的线程共享资源时，使用不同的调度方法，实现的效率时存在差异的。作者指出之前对多核系统性能的研究主要聚焦内存竞争（如使用cache分区和页染色等技术），事实上，内存控制器、内存总线等也是线程竞争等对象。作者提出了监控竞争的调度算法，先将应用根据竞争方式进行分类，而后根据分类决定如何分配内核。作者尝试了栈距离竞争、动物类、单独缺失率等方法，用来测试哪一种分类能够获得最好的调度效果。作者通过对比发现，最高级cache的缺失率能够比较清晰地表明应用所收到竞争的影响，所以记录下这些数据，根据这些数据设计了分布强度（DI）的调度方法，并可以在线获取这些数据，并修正调度方法，作者取名为DIO。作者证明有竞争监督地线程分配可以提高性能、稳定性，使用性能计数技术就能生成一个实用的线程调度器。文章中的亮点是作者在分析最佳调度的时候使用了图理论，有了理论的支持，不只是通过实验测试。本文的工作量很大，充分尝试了各种调度分类方法和调度效果。

第六篇：kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels

作者针对全虚拟化的策略降低模糊测试的效率的缺点，提出了基于硬件加速的模糊测试。kAFL同时利用了Intel提供的两个硬件特色： Intel VT-X 以及 Intel Process Trace。利用Hypervisor通过改造以后的 Qemu 和 KVM 实现内核态的路径反馈模糊测试。kAFL具有良好的跨平台特性，并且由于Intel PT的支持，可以将模糊测试的开销降到最低，同时解决了由于内核崩溃带来的效率和恢复的困难。作者在不同的操作系统上进行了测试评估，发现了新的内核漏洞，并且编写了测试用的驱动加入内核模块，横向比较了几款内核模糊测试方法的效能，证明kAFL在模糊测试方面具有较高的性能。基于实现的kAFL，评估跨平台的kAFL的模糊性能。作者在测试时，为了避免I/O性能差导致的失真，所有的基准测试都在RAM磁盘上执行。与AFL类似，作者使用bitmap记录基础块的变化。不同于AFL的是，kAFL使用大量的多线程并行技术，性能更高。作者使用kAFL发现了许多错误，并对其进行了进一步的评估测试，发现Linux，Windows和MacOS等所有测试操作系统存在多个安全漏洞。到目前为止，已经有八个漏洞被报告，其中三个漏洞被维护人员证实。除了找出新的漏洞，作者还验证了在keyctl接口上使用kAFL，keyctl允许用户空间程序存储和管理内核中的各种密钥材料，具有一个DER解析器来加载证书。使用kAFL也能发现已知的DER漏洞。作者又将kAFL的模糊测试性能与

Triforce进行比较，Triforce是与操作系统无关的反馈模糊器。TriforceAFL基于QEMU的仿真后端，而不是硬件辅助虚拟化和Intel PT。与TriforceAFL的QEMU CPU仿真相比，kAFL的性能提高了54倍。 单进程执行的性能（性能提升48倍）。足以证明，使用硬件辅助加速模糊测试在测试结果和测试速度上更佳。研究者在找bug的过程中，往往只依赖一些软件，很少有靠硬件辅助进行模糊测试的，一方面原因是之前的硬件或者现在的大部分硬件，不能支持模糊测试的特殊需求，另一个原因是做模糊测试的人未必懂得硬件。作者在这个方面做了很好的尝试，值得后来的研究者借鉴。