Dune: Safe User-level Access to Privileged CPU Features

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, Christos Kozyrakis
Stanford University

Abstract

Dune is a system that provides applications with direct but safe access to hardware features such as ring protection, page tables, and tagged TLBs, while preserving the existing OS interfaces for processes. Dune uses the virtualization hardware in modern processors to provide a *process*, rather than a *machine* abstraction. It consists of a small kernel module that initializes virtualization hardware and mediates interactions with the kernel, and a user-level library that helps applications manage privileged hardware features. We present the implementation of Dune for 64-bit x86 Linux. We use Dune to implement three user-level applications that can benefit from access to privileged hardware: a sandbox for untrusted code, a privilege separation facility, and a garbage collector. The use of Dune greatly simplifies the implementation of these applications and provides significant performance advantages.

1 Introduction

A wide variety of applications stand to benefit from access to "kernel-only" hardware features. As one example, Azul Systems demonstrates significant speedups to garbage collection through use of paging hardware [15, 36]. As another example, process migration, though implementable within a user program, can benefit considerably from access to page faults [40] and system calls [32]. In some cases, it might even be appropriate to replace the kernel entirely to meet the needs of a particular application. For example, IBOS improves browser security by moving browser abstractions into the lowest OS layers [35].

Such systems require changes in the kernel because hardware access in userspace is restricted for security and isolation reasons. Unfortunately, modifying the kernel is not ideal in practice because kernel changes can be fairly intrusive and, if done incorrectly, affect whole system stability. Moreover, if multiple applications require kernel changes, there is no guarantee that the changes will compose.

Another strategy is to bundle applications into virtual machine images with specialized kernels [4, 14]. Many modern CPUs contain virtualization hardware with which guest operating systems can safely and efficiently access kernel hardware features. Moreover, virtual machines provide failure containment similar to that of processes—*i.e.,* buggy or malicious behavior should not bring down the entire physical machine.

Unfortunately, virtual machines offer poor integration with the host operating system. Processes expect to inherit file descriptors from their parents, spawn other processes, share a file system and devices with their parents and children, and use IPC services such as Unix-domain sockets. Moving a process to a virtual machine for the purposes of, say, speeding up garbage collection is likely to break many assumptions and may simply not be worth the hassle. Moreover, producing a kernel for an application-specific virtual machine is no small task. Production kernels such as Linux are complex and hard to modify. Yet implementing a special-purpose kernel with a simple virtual memory layer is also challenging. In addition to virtual memory, one must support a file system, a networking stack, device drivers, and a bootstrap process.

This paper introduces a new approach to application use of kernel hardware features: using virtualization hardware to provide a *process*, rather than a *machine* abstraction. We have implemented this approach for Linux on 64-bit Intel CPUs in a system called *Dune*. Dune provides a loadable kernel module that works with unmodified Linux kernels. The module allows processes to enter "Dune mode," an irreversible transition in which, through virtualization hardware, safe and fast access to privileged hardware features is enabled, including privilege modes, virtual memory registers, page tables, and interrupt, exception, and system call vectors. We provide a user-level library, libDune, to facilitate the use of these features.

For applications that fit its paradigm, Dune offers several advantages over virtual machines. First, a Dune process is a normal Linux process, the only difference being that it uses the `VMCALL` instruction to invoke system calls. This means that Dune processes have full access to the rest of the system and are an integral part of it, and

Dune：安全用户级访问特权CPU功能

亚当·贝莱，安德烈亚·比特奥，阿里·马什蒂扎德，大卫·泰雷，大卫·马齐雷斯，克里斯托斯·科济拉基斯 斯坦福大学

摘要

Dune是一个系统，它为应用程序提供直接但安全的硬件功能访问，如环保护、页表和带标签的TLB，同时保留现有的操作系统接口供进程使用。Dune利用现代处理器的虚拟化硬件来为进程提供抽象，而不是机器抽象。它由一个小内核模块组成，该模块初始化虚拟化硬件并调解与内核的交互，以及一个用户级库，该库帮助应用程序管理特权硬件功能。我们介绍了Dune在64位x86 Linux上的实现。我们使用Dune实现了三个可以受益于特权硬件访问的用户级应用程序：一个用于不受信任代码的沙盒、一个特权分离设施和一个垃圾回收器。使用Dune极大地简化了这些应用程序的实现，并提供了显著的性能优势。

1 简介

各种应用程序都可能受益于访问"内核专用"硬件功能。例如，Azul Systems 通过使用分页硬件显著提升了垃圾回收的速度 [15,36]。另一个例子是进程迁移，尽管可以在用户程序中实现，但访问页错误 [40] 和系统调用 [32] 可以显著受益。在某些情况下，甚至可能完全替换内核以满足特定应用程序的需求。例如，IBOS 通过将浏览器抽象移至最低操作系统层来提升浏览器安全性 [35]。

此类系统需要在内核中进行修改，因为出于安全和隔离的原因，用户空间中的硬件访问受到限制。不幸的是，在实践中修改内核并不理想，因为内核变更可能相当侵入性，如果操作不当，会影响整个系统的稳定性。此外，如果多个应用程序需要内核变更，无法保证这些变更能够协同工作。

另一种策略是将应用程序打包到具有专用内核 [4, 14] 的虚拟机镜像中。许多
现代CPU包含虚拟化硬件，客户机操作系统可以通过它安全高效地访问内核硬件功能。此外，虚拟机提供了类似于进程的故障隔离——也就是说，有问题的或恶意的操作不应导致整个物理机崩溃。

不幸的是，虚拟机与主机操作系统集成不佳。进程期望从其父进程继承文件描述符、创建其他进程、与父进程和子进程共享文件系统和设备，并使用 Unix 域套接字等 IPC 服务。将进程迁移到虚拟机以加速垃圾回收等目的很可能破坏许多假设，并且可能根本不值得麻烦。此外，为特定应用程序的虚拟机开发内核绝非易事。像 Linux 这样的生产内核复杂且难以修改。然而，实现具有简单虚拟内存层的专用内核也是一项挑战。除了虚拟内存之外，还必须支持文件系统、网络堆栈、设备驱动程序和引导进程。

本文介绍了一种新的内核硬件特性应用方法：通过虚拟化硬件为进程提供抽象，而非机器抽象。我们已在名为Dune的系统上，为64位英特尔CPU上的Linux实现了该方法。Dune提供一个可加载的内核模块，可与未修改的Linux内核协同工作。该模块允许进程进入"Dune模式"，这是一种不可逆的转换，通过虚拟化硬件，可安全快速地访问特权硬件特性，包括特权模式、虚拟内存寄存器、页表以及中断、异常和系统调用向量。我们提供了一个用户级库libDune，以方便使用这些特性。

对于符合其范式的应用程序，Dune 相比虚拟机提供了几个优势。首先，Dune进程是一个正常Linux进程，唯一的不同之处在于它使用VMCALL指令来调用系统调用。这意味着Dune进程可以完全访问系统的其他部分，并且是其不可分割的一部分，和

that Dune applications are easy to develop (like application programming, not kernel programming). Second, because the Dune kernel module is not attempting to provide a machine abstraction, the module can be both simpler and faster. In particular, the virtualization hardware can be configured to avoid saving and restoring several pieces of hardware state that would be required for a virtual machine.

With Dune we contribute the following:

- We present a design that uses hardware-assisted virtualization to safely and efficiently expose privileged hardware features to user programs while preserving standard OS abstractions.
- We evaluate three hardware features in detail and show how they can benefit user programs: exceptions, paging, and privilege modes.
- We demonstrate the end-to-end utility of Dune by implementing and evaluating three use cases: sandboxing, privilege separation, and garbage collection.

2 Virtualization and Hardware

In this section, we review the hardware support for virtualization and discuss which privileged hardware features Dune is able to expose. Throughout the paper, we describe Dune in terms of x86 CPUs and Intel VT-x. However, this is not fundamental to our design, and in Section 7, we broaden our discussion to include other architectures that could be supported in the future.

2.1 The Intel VT-x Extension

In order to improve virtualization performance and simplify VMM implementation, Intel has developed VT-x [37], a virtualization extension to the x86 ISA. AMD also provides a similar extension with a different hardware interface called SVM [3].

The simplest method of adapting hardware to support virtualization is to introduce a mechanism for trapping each instruction that accesses privileged state so that emulation can be performed by a VMM. VT-x embraces a more sophisticated approach, inspired by IBM's interpretive execution architecture [31], where as many instructions as possible, including most that access privileged state, are executed directly in hardware without any intervention from the VMM. This is possible because hardware maintains a "shadow copy" of privileged state. The motivation for this approach is to increase performance, as traps can be a significant source of overhead.

VT-x adopts a design where the CPU is split into two operating modes: *VMX root* and *VMX non-root* mode. VMX root mode is generally used to run the VMM and does not change CPU behavior, except to enable access to new instructions for managing VT-x. VMX non-root mode, on the other hand, restricts CPU behavior and is intended for running virtualized guest OSes.

Transitions between VMX modes are managed by hardware. When the VMM executes the `VMLAUNCH` or `VMRESUME` instruction, hardware performs a *VM entry*; placing the CPU in VMX non-root mode and executing the guest. Then, when action is required from the VMM, hardware performs a *VM exit*, placing the CPU back in VMX root mode and jumping to a VMM entry point. Hardware automatically saves and restores most architectural state during both types of transitions. This is accomplished by using buffers in a memory resident data structure called the VM control structure (VMCS).

In addition to storing architectural state, the VMCS contains a myriad of configuration parameters that allow the VMM to control execution and specify which type of events should generate VM exits. This gives the VMM considerable flexibility in determining which hardware is exposed to the guest. For example, a VMM could configure the VMCS so that the `HLT` instruction causes a VM exit or it could allow the guest to halt the CPU. However, some hardware interfaces, such as the interrupt descriptor table (IDT) and privilege modes, are exposed implicitly in VMX non-root mode and never generate VM exits when accessed. Moreover, a guest can manually request a VM exit by using the `VMCALL` instruction.

Virtual memory is perhaps the most difficult hardware feature for a VMM to expose safely. A straw man solution would be to configure the VMCS so that the guest has access to the page table root register, `%CR3`. However, this would place complete trust in the guest because it would be possible for it to configure the page table to access any physical memory address, including memory that belongs to the VMM. Fortunately, VT-x includes a dedicated hardware mechanism, called the *extended page table* (EPT), that can enforce memory isolation on guests with direct access to virtual memory. It works by applying a second, underlying, layer of address translation that can only be configured by the VMM. AMD's SVM includes a similar mechanism to the EPT, referred to as a nested page table (NPT).

2.2 Supported Hardware Features

Dune uses VT-x to provide user programs with full access to x86 protection hardware. This includes three privileged hardware features: exceptions, virtual memory, and privilege modes. Table 1 shows the corresponding privileged

Dune 应用程序易于开发（例如应用程序编程，而不是内核编程）。其次，因为 Dunekernel 模块没有试图提供机器抽象，所以该模块可以既更简单也更快速。特别是，虚拟化硬件可以配置为避免保存和恢复几块硬件状态，这些状态对于虚拟机来说是必需的。

使用 Dune 我们贡献了以下内容：

- 我们提出了一种使用硬件辅助虚拟化的设计化以安全高效地暴露特权硬将 ware 功能特性提供给用户程序，同时保留标准 OS 抽象。
- 我们详细评估了三种硬件特性，并展示了它们如何使用户程序受益：异常、分页和特权模式。
- 我们通过实现和评估三个用例——沙盒、特权分离和垃圾回收——来展示 Dune 的端到端实用性。

2 虚拟化和硬件

在本节中，我们回顾了虚拟化的硬件支持，并讨论了Dune能够暴露哪些特权硬件特性。全文中，我们以x86 CPU和Intel VT-x的视角描述Dune。但这并非我们设计的根本，在第七节中，我们将讨论扩展到其他架构，这些架构未来可能得到支持。

2.1 Intel VT-x扩展

为了提高虚拟化性能并简化VMM实现，Intel开发了x86 ISA的VT-x [37]，虚拟化扩展。AMD也提供了一种具有不同硬件接口的类似扩展，称为SVM [3]。

将硬件适配以支持虚拟化最简单的方法是引入一种机制，用于捕获每个访问特权状态的指令，以便VMM进行模拟。VT-x采用了一种更复杂的方法，灵感来自IBM的解释执行架构 [31]，，尽可能多的指令，包括大多数访问特权状态的指令，都在硬件中直接执行，而无需VMM的任何干预。这是可能的，因为硬件维护了特权状态的“影子副本”。这种方法的动机是提高性能，因为陷阱可能是显著的性能开销来源。

VT-x采用了一种设计，将CPU分为两种操作模式：VMX根模式和VMX非根模式。VMX根模式通常用于运行VMM，并且不会改变CPU行为，除了启用对管理VT-x的新指令的访问。而VMX非根模式则限制CPU行为，并旨在运行虚拟化客户机操作系统。

VMX模式之间的转换由硬件管理。当VMM执行VMLAUNCH或VMRESUME指令时，硬件执行虚拟机进入；将CPU置于VMX非根模式并执行客户机。

然后，当VMM需要采取行动时，硬件执行虚拟机退出，将CPU恢复到VMX根模式并跳转到VMM入口点。硬件在两种类型的转换过程中自动保存和恢复大部分架构状态。这是通过使用驻留在称为虚拟机控制结构（VMCS）的内存驻留数据结构中的缓冲区来实现的。

除了存储架构状态外，VMCS还包含大量配置参数，允许VMM控制执行并指定应生成虚拟机退出的事件类型。这为VMM在确定哪些硬件暴露给客户机方面提供了相当大的灵活性。例如，VMM可以配置VMCS，以便HLT指令导致虚拟机退出，或者它可以允许客户机停止CPU。然而，某些硬件接口，如中断描述符表（IDT）和特权模式，在VMX非根模式下隐式暴露，并且在访问时永远不会生成虚拟机退出。此外，客户机可以通过使用VMCALL指令手动请求虚拟机退出。

虚拟内存或许是一个VMM最难安全暴露的硬件特性。一个简单的解决方案是配置VMCS，让客户机可以访问页表根寄存器，%CR3。然而，这会完全信任客户机，因为它可以配置页表来访问任何物理内存地址，包括属于VMM的内存。幸运的是，VT-x包含一个专用的硬件机制，称为扩展页表（EPT），它可以在客户机直接访问虚拟内存的情况下强制执行内存隔离。它的工作原理是应用一个第二层、底层的地址转换，这只能由VMM配置。AMD的SVM包含一个与EPT类似的机制，称为嵌套页表（NPT）。

2.2 支持的硬件特性

Dune使用VT-x为用户程序提供对x86保护硬件的完全访问权限。这包括三个特权硬件功能：异常、虚拟内存和特权模式。表1显示了每个功能对应的特权指令。

Mechanism	Privileged Instructions
Exceptions	LIDT, LTR, IRET, STI, CLI
Virtual Memory	MOV CRn, INVLPG, INVPCID
Privilege Modes	SYSRET, SYSEXIT, IRET
Segmentation	LGDT, LLDT

Table 1: Hardware features exposed by Dune and their corresponding privileged x86 instructions.

instructions made available for each feature. Dune also exposes segmentation, but we do not discuss it further, as it is primarily a legacy mechanism on modern x86 CPUs.

Efficient support for exceptions is important in a variety of use cases such as emulation, debugging, and performance tracing. Normally, reporting an exception to a user program requires privilege mode transitions and an upcall mechanism (*e.g.,* signals). Dune can reduce exception overhead because it uses VT-x to deliver exceptions directly in hardware. This does not, however, allow a Dune process to monopolize the CPU, as timer interrupts and other exceptions intended for the kernel will still cause a VM exit. The net result is that software overhead is eliminated and exception performance is determined by hardware efficiency alone. As just one example, Dune improves the speed of delivering page fault exceptions, when compared to SIGSEGV in Linux, by more than 4×. Several other types of exceptions are also accelerated, including breakpoints, floating point overflow and underflow, divide by zero, and invalid opcodes.

User programs can also benefit from fast and flexible access to virtual memory [5]. Use cases include checkpointing, garbage collection (evaluated in this paper), data-compression paging, and distributed shared memory. Dune improves virtual memory access by exposing page table entries to user programs directly, allowing them to control address translations, access permissions, global bits, and modified/accessed bits with simple memory references. In contrast, even the most efficient OS interfaces [17] add extra latency by requiring system calls in order to perform these operations. Letting applications write their own page tables does not affect security because the underlying EPT exposes only the normal process address space, which is equally accessible without Dune.

Dune also gives user programs the ability to manually control TLB invalidations. As a result, page table updates can be performed in batches when permitted by the application. This is considerably more challenging to support in the kernel because it is difficult to defer TLB invalidations when general correctness must be maintained. In addition, Dune exposes TLB tagging by providing access to Intel's recently added process-context identifier (PCID) feature. This permits a single user program to switch between multiple page tables efficiently. All together, we show that using Dune results in a 7× speedup over Linux in the Appel and Li user-level virtual memory benchmarks [5]. This figure includes the use of exception hardware to reduce page fault latency.

Finally, Dune exposes access to privilege modes. On x86, the most important privilege modes are ring 0 (supervisor mode) and ring 3 (user mode), although rings 1 and 2 are also available. Two motivating use cases for privilege modes are privilege separation and sandboxing of untrusted code, both evaluated in this paper. Dune can support privilege modes efficiently because VMX non-root mode maintains its own set of privilege rings. Hence, Dune allows hardware-enforced protection within a process in exactly the way kernels protect themselves from user processes. The supervisor bit in the page table is available to control memory isolation. Moreover, system call instructions trap to the process itself, rather than to the kernel, which can be used for system call interposition and to prevent untrusted code from directly accessing the kernel. Compared to *ptrace* in Linux, we show that Dune can intercept a system call with 25× less overhead.

Although the hardware features Dune exposes suffice in supporting our motivating use cases, several other hardware features, such as cache control, debug registers, and access to DMA-capable devices, could also be safely exposed through virtualization hardware. We leave these for future work and discuss their potential in Section 7.

3 Kernel Support for Dune

The core of Dune is a kernel module that manages VT-x and provides user programs with greater access to privileged hardware features. We describe this module here, including a system overview, a threat model, and a comparison to an ordinary VMM. We then explore three key aspects of the module's operation: managing memory, exposing access to privileged hardware, and preserving access to kernel interfaces. Finally, we describe the Dune module we implemented for the Linux kernel.

3.1 System Overview

Figure 1 shows a high-level view of the Dune architecture. Dune extends the kernel with a module that enables VT-x, placing the kernel in VMX root mode. Processes using Dune are granted direct but safe access to privileged hardware by running in VMX non-root mode. The Dune module intercepts VM exits, the only means for a Dune pro-

机制	特权指令
异常	LIDT、LTR、IRET、STI、CLI
虚拟内存	MOV CRn、INVLPG、INVPCID
特权模式	SYSRET、SYSEXIT、IRET
分段	LGDT、LLDT

表1：Dune暴露的硬件特性及其对应的特权x86指令。

Dune还暴露了分段机制，但我们不再进一步讨论它，因为它是现代x86 CPU上的一个主要遗留机制。

高效支持异常在各种用例（如模拟、调试和性能跟踪）中非常重要。通常，向用户程序报告异常需要特权模式转换和向上调用机制（例如，信号）。Dune可以减少异常开销，因为它使用VT-x将异常直接在硬件中传递。然而，这并不允许Dune进程独占CPU，因为计时器中断和其他针对内核的异常仍会导致虚拟机退出。最终结果是消除了软件开销，异常性能仅由硬件效率决定。例如，与Linux中的SIGSEGV相比，Dune将页面错误异常的传递速度提高了 4×以上。其他几种类型的异常也得到了加速，包括断点、浮点溢出和下溢、除零和无效操作码。

用户程序也能从虚拟内存的快速灵活访问中受益 [5]。用例包括检查点、垃圾回收（本文评估）、数据压缩分页和分布式共享内存。Dune通过将页表条目直接暴露给用户程序来改进虚拟内存访问，允许它们通过简单的内存引用来控制地址转换、访问权限、全局位和修改/访问位。相比之下，即使是最高效的操作系统接口 [17] 也通过要求系统调用执行这些操作而增加了额外的延迟。让应用程序自己编写页表不会影响安全性，因为底层的EPT只暴露了正常进程地址空间，而无需Dune即可同等访问。

Dune 还允许用户程序手动控制 TLB 失效。因此，当应用程序允许时，页表更新可以批量执行。由于在内核中维护通用正确性时难以延迟 TLB 失效，因此支持这一点要困难得多。此外，Dune 通过提供 ac-

cess到英特尔最近添加的进程上下文标识符（PCID）功能。这允许单个用户程序在多个页表之间高效切换。所有要-

gether，我们展示了在使用 Dune 时，在 Appel 和 Li 用户级虚拟内存基准测试中，与 Linux 相比实现了 7× 加速。此图表包括了使用异常硬件来降低页面错误延迟。

最后，Dune暴露了对特权模式的访问。在x86架构上，最重要的特权模式是环0（监督模式）和环3（用户模式），尽管环1和环2也可用。特权模式的两个主要用例是特权分离和不信任代码沙箱，本文都进行了评估。Dune可以高效地支持特权模式，因为VMX非根模式维护着自己的特权环集。因此，Dune允许在进程内部进行硬件强制保护，就像内核保护自己免受用户进程攻击的方式一样。页表中的监督位可用于控制内存隔离。此外，系统调用指令会陷阱到进程本身，而不是到内核，这可用于系统调用拦截，并防止不信任代码直接访问内核。与Linux中的ptrace相比，我们表明Dune拦截系统调用时的开销更小 25× 。

尽管Dune暴露的硬件特性足以支持我们的主要用例，但缓存控制、调试寄存器以及DMA设备访问等几个其他硬件特性，也可以通过虚拟化硬件安全地暴露出来。我们将这些留待未来工作，并在第7节讨论它们的潜在价值。

3 内核对Dune的支持

Dune的核心是一个内核模块，该模块管理VT-x并为用户程序提供对特权硬件特性的更大访问权限。我们在此描述该模块，包括系统概述、威胁模型以及与普通VMM的比较。然后，我们探讨该模块操作的三个方面：管理内存、暴露特权硬件访问权限以及保留对内核接口的访问权限。最后，我们描述了为Linux内核实现的Dune模块。

3.1 系统概述

图1展示了Dune架构的高层视图。Dune通过一个启用VT-x的模块扩展内核，将内核置于VMX根模式。使用Dune的进程在VMX非根模式下运行，从而获得直接但安全的特权硬件访问权限。Dune模块拦截VM退出，这是Dune进程访问内核的唯一方式，并执行任何必要的操作，例如处理页错误、调用系统调用或在HLT指令后释放CPU。Dune还包含一个名为libDune的库，用于在用户空间管理特权硬件功能，详见第4节。

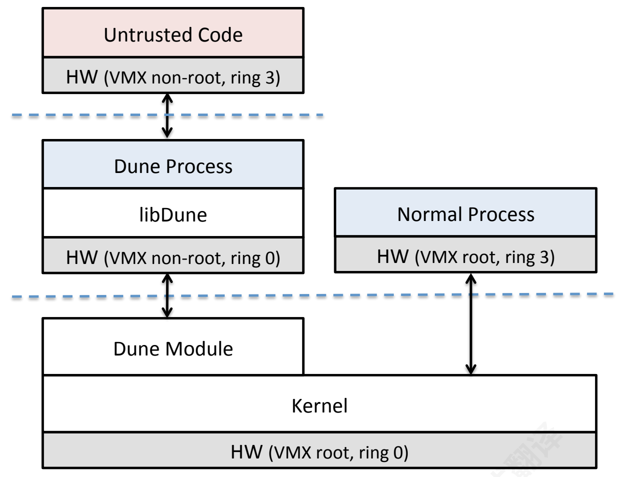

Figure 1: The Dune system architecture.

cess to access the kernel, and performs any necessary actions such as servicing a page fault, calling a system call, or yielding the CPU after a `HLT` instruction. Dune also includes a library, called *libDune*, to assist with managing privileged hardware features in userspace, discussed further in Section 4.

We apply Dune selectively to processes that need it; processes that do not use Dune are completely unaffected. A process can enable Dune at any point by initiating a transition through an *ioctl* on the `/dev/dune` device, but once in Dune mode, a process cannot exit Dune mode. Whenever a Dune process forks, the child process does not start in Dune mode, but can re-enter Dune if the use case requires it.

The Dune module requires VT-x. As a result, it cannot be used inside a VM unless there is support for nested VT-x [6]; the performance characteristics of such a configuration are an interesting topic of future consideration. On the other hand, it is possible to run a VMM on the same machine as the Dune module, even if the VMM requires VT-x, because VT-x can be controlled independently on each core.

3.2 Threat Model

Dune exposes privileged CPU features without affecting the existing security model of the underlying OS. Any external effects produced by a Dune-enabled process could be produced without Dune through the same series of system calls. However, by exposing hardware privilege modes, Dune enables additional privilege-separation techniques within a process that would not otherwise be practical.

We assume that the CPU is free of defects, although we acknowledge that in rare cases exploitable hardware flaws have been identified [26, 27].

3.3 Comparing to a VMM

Though all software using VT-x shares a common structure, Dune's use of VT-x deviates from that of standard VMMs. Specifically, Dune exposes a process environment instead of a machine environment. As a result, Dune is not capable of supporting a normal guest OS, but this permits Dune to be lighter weight and more flexible. Some of the most significant differences are as follows:

- Hypercalls are a common way for VMMs to support paravirtualization, a technique in which the guest OS is modified to use interfaces that are more efficient and less difficult to virtualize. In Dune, by contrast, the hypercall mechanism invokes normal Linux system calls. For example, a VMM might provide a hypercall to register an interrupt handler for a virtual network device, whereas a Dune process would use a hypercall to call *read* on a TCP socket.

- Many VMMs emulate physical hardware interfaces in order to support unmodified guest OSes. In Dune, only hardware features that can be directly accessed without VMM intervention are made available; in cases where this is not possible, a Dune process falls back on the OS. For example, most VMMs go to great lengths to present a virtual graphics card interface in order to support a frame buffer. By contrast, Dune processes employ the normal OS display service, usually an X server accessed over a Unix-domain socket and shared memory.

- A typical VMM must save and restore all state that is necessary to support a guest OS. In Dune, we can limit the differences in guest and host state because processes using Dune have a narrower hardware interface. This results in reductions to the overhead of performing VM entries and VM exits.

- VMMs place each VM in a separate address space that emulates flat physical memory. In Dune, we configure the EPT to reflect process address spaces. As a result, the memory layout can be sparse and memory can be coherently shared when two processes map the same memory segment.

Despite these differences, the Dune module could be considered a type-2 hypervisor [22] because it runs on top of an existing OS kernel.

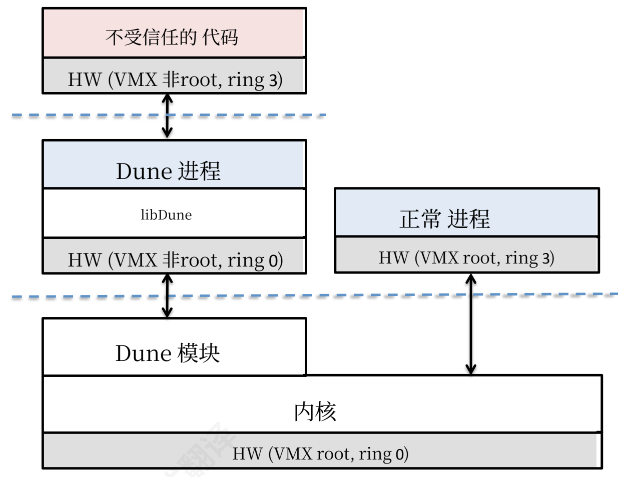

图1：Dune系统架构。

Dune还包含一个名为libDune的库，用于在用户空间管理特权硬件功能，详见第4节。

我们将Dune选择性地应用于需要它的进程；不使用Dune的进程完全不受影响。任何进程都可以通过在/dev/dune设备上发起ioctl来启用Dune，但在Dune模式下，一旦进入Dune模式，进程就无法退出Dune模式。每当Dune进程进行fork时，子进程不会以Dune模式启动，但如果用例需要，它可以重新进入Dune模式。

Dune模块需要VT-x。因此，除非支持嵌套VT-x [6]，否则它不能在虚拟机中使用；这种配置的性能特征是未来值得考虑的一个有趣话题。另一方面，即使VMM需要VT-x，也可以在Dune模块所在的同一台机器上运行VMM，因为VT-x可以在每个核心上独立控制。

3.2 威胁模型

Dune 暴露特权 CPU 功能，而不会影响底层操作系统的现有安全模型。任何由 Dune 启用进程产生的外部效果，都可以通过相同的系统调用序列在不使用 Dune 的情况下产生。然而，通过暴露硬件特权模式，Dune 能够在进程内部启用额外的特权隔离技术，而这些技术在不使用 Dune 的情况下则不切实际。

我们假设 CPU 没有缺陷，尽管我们承认在罕见情况下已被识别出可利用的硬件漏洞 [26, 27]。

3.3 与VMM的比较

尽管所有使用VT-x的软件都共享一个共同的结构，但Dune对VT-x的使用与标准VMM有所不同。具体来说，Dune暴露的是一个进程环境，而不是一个机器环境。因此，Dune无法支持正常的客户机操作系统，但这允许Dune更轻量级和更灵活。一些最重要的区别如下：

- 超调用是VMM支持虚拟化的一种常见方式，虚拟化是一种修改客户机操作系统以使用更高效且更易于虚拟化的接口的技术。相比之下，在Dune中，超调用机制调用正常的Linux系统调用。例如，VMM可能会提供一个超调用来注册虚拟网络设备的中断处理程序，而Dune进程则会使用超调用来对TCP套接字调用read。

- 许多VMM模拟物理硬件接口，以便支持未经修改的客户机操作系统。在Dune中，只有那些无需VMM干预即可直接访问的硬件功能才会被启用；在无法实现这一点的情况下，Dune进程会回退到操作系统。例如，大多数VMM会费尽心思呈现虚拟显卡接口，以支持帧缓冲区。相比之下，Dune进程采用正常的操作系统显示服务，通常是通过Unix域套接字和共享内存访问的X服务器。

- 典型的VMM必须保存和恢复所有支持客户机操作系统所需的状态。在Dune中，我们可以限制客户机和主机状态之间的差异，因为使用Dune的进程具有更窄的硬件接口。这导致执行VM入口和VM退出的开销减少。

- VMM将每个虚拟机置于一个模拟扁平物理内存的独立地址空间中。在Dune中，我们配置EPT以反映进程地址空间。因此，内存布局可以是稀疏的，当两个进程映射相同的内存段时，内存可以一致地共享。

尽管存在这些差异，Dune模块可以被认为是一个类型2虚拟机管理程序 [22]，因为它运行在现有的OS内核之上。

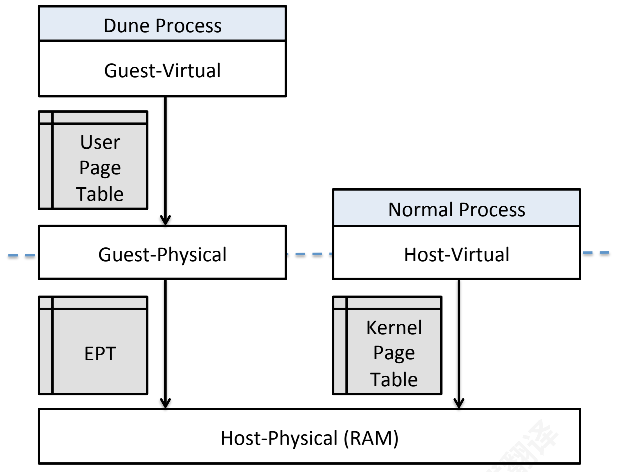

Figure 2: Virtual memory in Dune.

3.4 Memory Management

Memory management is one of the biggest responsibilities of the Dune module. The challenge is to expose direct page table access to user programs while preventing arbitrary access to physical memory. Moreover, our goal is to provide a normal process memory address space by default, permitting user programs to add just the functionality they need instead of completely replacing kernel-level memory management.

Paging translations occur in three separate cases in Dune, shown in Figure 2. One translation is specified by the kernel's standard page table. In virtualization terminology this is the host-virtual to host-physical (*i.e.,* raw memory) translation. Host-virtual addresses are ordinary virtual addresses, but they are only used by the kernel and normal processes. For processes using Dune, a user controlled page table maps guest-virtual addresses to guest-physical. Then the EPT, managed by the kernel, performs an additional translation from guest-physical to host-physical. All memory references made by processes using Dune can only be guest-virtual, allowing for isolation and correctness to be enforced in the EPT while application-specific functionality and optimizations can be applied in the user page table.

Ideally, we would like to match the EPT to the kernel's page table as closely as possible because of our goal to give processes using Dune access to the same address space they would have as normal processes. If it were permitted by hardware, we would simply point the EPT and the kernel's page table to the same page root. Unfortunately, two limitations make this impossible. First, the EPT requires a different binary format from the standard x86 page table. Second, Intel x86 processors limit the address width of guest-physical addresses to be the same as host-physical addresses. In a standard virtual machine environment this would not be a concern because any machine being emulated would have a realistically bounded amount of RAM. For Dune, however, the problem is that we want to expose the full host-virtual address space and yet the guest-physical address space is limited to a smaller size (*e.g.,* a 36-bit physical limit vs. a 48-bit virtual limit on many contemporary Intel processors). We note that this issue is not present when running in 32-bit protected mode, as physical addresses are at least as large as virtual addresses.

Our solution to EPT format incompatibility is to query the kernel for process memory mappings and to manually update the EPT to reflect them. We start with an empty EPT. Then, we receive an EPT fault (a type of VM exit) each time a missing EPT entry is accessed. The fault handler crafts a new EPT entry that reflects an address translation and permission reported by the kernel's page fault handler. Occasionally, address ranges will need to be unmapped. In addition, the kernel requires page access information, to assist with swapping, and page dirty status, to determine when write-back to disk is necessary. Dune supports all of these cases by hooking into an MMU notifier chain, the same approach used by KVM [30]. For example, when an address is unmapped, the Dune module receives an event. It then evicts affected EPT entries and sets dirty bits in the appropriate Linux page structures.

We work around the address width issue by allowing only some address ranges to be mapped in the EPT. Specifically, we only permit addresses from the beginning of the process (*i.e.,* the heap, code, and data segments), the mmap region, and the stack. Currently, we limit each of these regions to 4GB, allowing us to compress the address space to fit in the first 12GB of the EPT. Typically the user's page table will then expand the addresses to their original layout. This could result in incompatibilities in programs that use nonstandard portions of the address space, though such cases are rare. A more sophisticated solution might pack each virtual memory area into the guest-physical address space in arbitrary order and then provide the user program the additional information required to remap the segment to the correct guest-virtual address in its own page table, thus avoiding the possibility of unaddressable memory regions.

3.5 Exposing Access to Hardware

As discussed previously, Dune exposes access to exceptions, virtual memory, and privilege modes. Exceptions and privilege modes are implicitly available in VMX non-

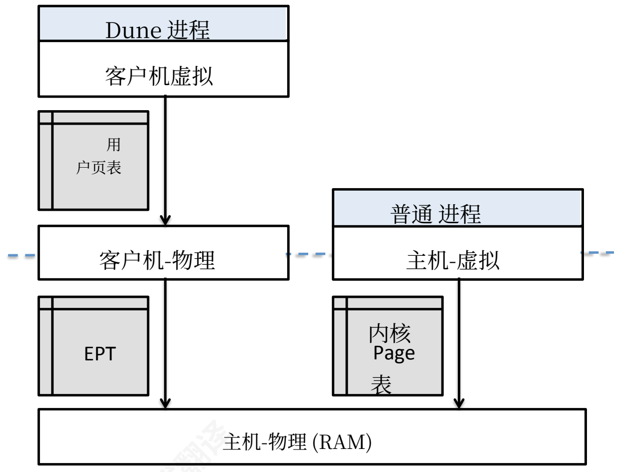

图2：Dune中的虚拟内存。

3.4 内存管理

内存管理是 Dune 模块的一项重大职责。挑战在于向用户程序暴露直接的页表访问，同时防止对物理内存的任意访问。此外，我们的目标是通过默认提供普通进程的内存地址空间，允许用户程序仅添加他们需要的功能，而不是完全替换内核级的内存管理。

在Dune中，分页转换发生在三种独立的情况下，如图2所示。其中一种转换由内核的标准页表指定。在虚拟化术语中，这是宿主机虚拟到宿主机物理（即原始内存）的转换。宿主机虚拟地址是普通虚拟地址，但它们仅由内核和普通进程使用。对于使用Dune的进程，用户控制的页表将客户机虚拟地址映射到客户机物理地址。然后由内核管理的EPT执行从客户机物理到宿主机物理的附加转换。所有使用Dune的进程只能进行客户机虚拟内存引用，这允许在EPT中强制执行隔离和正确性，同时可以在用户页表中应用特定于应用程序的功能和优化。

理想情况下，我们希望尽可能将 EPT 与内核的页表进行匹配，因为我们的目标是为使用 Dune 的进程提供与普通进程相同的地址空间。如果硬件允许，我们只需将 EPT 和内核的页表指向相同的页根。不幸的是，两个限制使这成为不可能。首先，EPT 需要不同于标准 x86 页表的二进制格式。其次，英特尔 x86 处理器将客户机物理地址的地址宽度限制为主机物理地址的地址宽度。在标准的虚拟机环境中，这不会成为问题，因为任何被模拟的机器都会有实际可限制的 RAM 量。然而，对于 Dune，问题在于我们希望暴露完整的宿主机虚拟地址空间，但客户机物理地址空间的大小有限（例如，在许多当代英特尔处理器上，物理限制为 36 位，而虚拟限制为 48 位）。我们注意到，在 32 位保护模式下运行时，这个问题不会出现，因为物理地址至少与虚拟地址一样大。

理想情况下，我们希望尽可能将 EPT 与内核的页表进行匹配，因为我们的目标是为使用 Dune 的进程提供与普通进程相同的地址空间。如果硬件允许，我们只需将 EPT 和内核的页表指向相同的页根。不幸的是，两个限制使这成为不可能。首先，EPT 需要不同于标准 x86 页表的二进制格式。其次，英特尔 x86 处理器将客户机物理地址的地址宽度限制为主机物理地址的地址宽度。在标准的虚拟机环境中，这不会成为问题，因为任何被模拟的机器都会有实际可限制的 RAM 量。然而，对于 Dune，问题在于我们希望暴露完整的宿主机虚拟地址空间，但客户机物理地址空间的大小有限（例如，在许多当代英特尔处理器上，物理限制为 36 位，而虚拟限制为 48 位）。我们注意到，在 32 位保护模式下运行时，这个问题不会出现，因为物理地址至少与虚拟地址一样大。

我们解决EPT格式不兼容问题的方案是查询内核获取进程内存映射，并手动更新EPT以反映这些映射。我们从空的EPT开始。然后，每当访问缺失的EPT条目时，我们都会收到一个EPT错误（一种虚拟机退出）。错误处理程序会创建一个新的EPT条目，该条目反映内核页面错误处理程序报告的地址转换和权限。偶尔需要取消映射地址范围。此外，内核需要页面访问信息，以协助交换，以及页面脏状态，以确定何时需要将数据回写到磁盘。Dune通过挂钩MMU通知链支持所有这些情况，这与KVM [30]使用的方法相同。例如，当地址被取消映射时，Dune模块会收到一个事件。然后它会驱逐受影响的EPT条目，并在适当的Linux页面结构中设置脏位。

我们通过仅允许部分地址范围映射到EPT中，来规避地址宽度问题。具体来说，我们仅允许来自进程开头的地址（即堆、代码段和数据段）、mmap区域以及栈的地址。目前，我们将这些区域的大小限制为4GB，从而能够将地址空间压缩以适应EPT的前12GB。通常，用户的页表会将地址扩展到其原始布局。这可能导致在那些使用非标准地址空间部分的程序中存在不兼容性，尽管这种情况很少见。一个更复杂的解决方案可能是将每个虚拟内存区域任意顺序地打包到客户机物理地址空间中，然后向用户程序提供额外的信息，以便其将段重新映射到其页表中的正确客户机虚拟地址，从而避免出现无法寻址的内存区域的可能性。

3.5 暴露硬件访问

如前所述，Dune 暴露了对异常、虚拟内存和特权模式的访问。异常和特权模式在 VMX 非

root mode and do not require any special configuration. On the other hand, virtual memory requires access to the %CR3 register, which can be granted in the VMCS. We maintain a separate VMCS for each process in order to allow for per-process configuration of privileged state and to support context switching more easily and efficiently.

x86 includes a variety of control registers that determine which hardware features are enabled (*e.g.,* floating point, SSE, no execute, *etc.*) Although we could have permitted Dune processes to configure these directly, we instead mirror the configuration set by the kernel. This allows us to support a normal process environment; permitting many configuration changes would break compatibility with user programs. For example, it makes little sense for a 64-bit process to disable long mode. There are, however, a couple of important exceptions to this rule. First, we allow user programs to disable paging because it is the only method available on x86 to clear global TLB entries. Second, we give user programs some control over floating point hardware in order to allow for support of lazy floating point state management.

In some cases, Dune restricts access to hardware registers for performance reasons. For instance, Dune does not allow modification to MSRs in order to avoid the relatively high overhead of saving and restoring them during each system call. The FS and GS base registers are exceptions because they are not only used frequently but are also saved and restored by hardware automatically. MSR_LSTAR, which contains the address of the system call handler, is a special case where Dune allows read-only access. This allows a user process to map code for a system call handler at the existing address (by manipulating its page table) instead of changing the register to a new address and, as a result, harming performance.

Dune exposes raw access to the time stamp counter (TSC). By contrast, most VMMs virtualize the TSC in order to avoid confusing guest kernels, which tend to make timing assumptions that could be violated if time spent in the VMM is made visible.

3.6 Preserving OS Interfaces

In addition to exposing privileged hardware features, Dune preserves access to OS system calls. Normal system call invocation instructions will only trap within the process itself and do not cause a VM exit. Instead, processes must use VMCALL, the hypercall instruction, to make system calls. The Dune module vectors hypercalls through the kernel's system call table. In some cases, it must perform extra actions before calling the system call handler. For example, during an *exit* system call, Dune performs cleanup tasks.

Dune completely changes how signal handlers are invoked. Some signals are obviated by more efficient direct hardware support. For example, hardware page faults largely subsume the role of SIGSEGV. For other signals (*e.g.,* SIGINT), the Dune module injects fake hardware interrupts into the process. This is not only an efficient mechanism, but also has the advantage of correctly composing with privilege modes. For example, if a user process were running in ring 3 to sandbox untrusted code, hardware would automatically transition it to ring 0 in order to service the signal securely.

3.7 Implementation

Dune presently supports Linux running on Intel x86 processors in 64-bit long mode. Support for AMD CPUs and 32-bit mode are possible future additions. In order to keep changes to the kernel as unintrusive as possible, we developed Dune as a dynamically loadable kernel module. Our implementation is based partially on KVM [30]. Specifically, it shares code for managing low-level VT-x operations. However, high-level code is not shared with KVM because Dune operates differently from a VMM. Furthermore, our Dune module is simpler than KVM, consisting of only 2,509 lines of code.

In Linux, user threads are supported by the kernel, making them nearly identical to processes except they have a shared address space. As a result, it was easiest for us to create a VMCS for each thread instead of merely each process. One interesting consequence is that it is possible for both threads using Dune and threads not using Dune to belong to the same process.

Our implementation is capable of supporting thousands of processes at a time. The reason is that processes using Dune are substantially lighter-weight than full virtual machines. Efficiency is further improved by using virtual-processor identifiers (VPIDs). VPIDs enable a unique TLB tag to be assigned to each Dune process, and, as a result, hypercalls and context switches do not require TLB invalidations.

One limitation in our implementation is that we cannot efficiently detect when EPT pages have been modified or accessed, which is needed for swapping. Intel recently added hardware support for this capability, so it should be easy to rectify this limitation. For now, we take a conservative approach and always report pages as modified and accessed during MMU notifications in order to ensure correctness.

如前所述，Dune 提供了对异常、虚拟内存和特权模式的访问。异常和特权模式在 VMX 非root模式下隐式可用，无需任何特殊配置。另一方面，虚拟内存需要访问 %CR3 寄存器，这可以在 VMCS 中授予权限。我们为每个进程维护一个单独的 VMCS，以便允许对特权状态进行进程级配置，并更轻松高效地支持上下文切换。

x86 包含多种控制寄存器，用于确定启用的硬件特性（例如，浮点数、SSE、不可执行等）。尽管我们可以允许 Dune 进程直接配置这些寄存器，但我们选择镜像内核配置的设置。这使我们能够支持普通进程环境；允许许多配置更改将破坏与用户程序的兼容性。例如，对于 64 位进程禁用长模式几乎没有意义。然而，这条规则存在一些重要的例外。首先，我们允许用户程序禁用分页，因为 x86 上这是清除全局 TLB 条目的唯一方法。其次，我们赋予用户程序对浮点硬件的一些控制权，以支持惰性浮点状态管理。

在某些情况下，Dune 为性能原因限制对硬件寄存器的访问。例如，Dune 不允许修改 MSR，以避免在每次系统调用时保存和恢复它们的高开销。FS 和 GS 基寄存器是例外，因为它们不仅使用频繁，而且由硬件自动保存和恢复。包含系统调用处理程序地址的 MSR LSTAR 是一个特殊情况，Dune 允许只读访问。这允许用户进程通过操作其页表将系统调用处理程序的代码映射到现有地址（而不是将寄存器更改为新地址），从而避免损害性能。

Dune提供了对时间戳计数器（TSC）的原始访问权限。相比之下，大多数VMM会虚拟化TSC，以避免混淆客户机内核，这些内核往往会做出时间假设，而如果VMM中的时间消耗变得可见，这些假设可能会被破坏。

3.6 保留操作系统接口

除了暴露特权硬件功能外，Dune还保留了访问操作系统调用的能力。正常的系统调用指令仅在进程内部触发陷阱，不会导致虚拟机退出。相反，进程必须使用VMCALL（超调用指令）来执行系统调用。Dune模块通过内核的系统调用表来转发超调用。在某些情况下，它必须在调用系统调用处理程序之前执行额外的操作。例如，在退出系统调用期间，Dune会执行清理任务。

Dune完全改变了信号处理器的调用方式。一些信号由于更高效的直接硬件支持而被取消。例如，硬件页错误在很大程度上取代了SIGSEGV的作用。对于其他信号（例如SIGINT），Dune模块向进程注入伪造的硬件中断。这不仅是高效的机制，而且具有与特权模式正确组合的优势。例如，如果用户进程在环3中运行以沙盒不受信任的代码，硬件将自动将其切换到环0以安全地处理信号。

3.7 实现

Dune目前支持在Intel x86处理器上的64位长模式下运行的Linux。对AMD CPU和32位模式的支持是可能的未来添加。为了尽可能减少对内核的更改，我们将Dune开发为动态可加载的内核模块。我们的实现部分基于KVM [30]。具体来说，它共享管理低级VT-x操作代码。然而，高级代码没有与KVM共享，因为Dune与VMM的操作方式不同。此外，我们的Dune模块比KVM更简单，仅由2,509行代码组成。

在Linux中，用户线程由内核支持，使其几乎与进程相同，只是它们拥有共享的地址空间。因此，对我们来说，为每个线程创建VMCS比仅为每个进程创建要容易得多。一个有趣的结果是，使用Dune的线程和未使用Dune的线程都有可能属于同一个进程。

我们的实现能够同时支持数千个进程。原因是使用Dune的进程比完整的虚拟机要轻得多。通过使用虚拟处理器标识符（VPIDs），效率得到了进一步提升。VPIDs为每个Dune进程分配了唯一的TLB标签，因此，超调用和上下文切换不需要TLB失效。

我们实现中的一个限制是，无法高效检测 EPT 页面是否被修改或访问，而这对于交换操作是必要的。英特尔最近为这项功能添加了硬件支持，因此应该很容易解决这个问题。目前，我们采取保守策略，在 MMUnotifications 中始终报告页面为已修改和已访问，以确保正确性。

4 User-mode Environment

The execution environment of a process using Dune has some differences from a normal process. Because privilege rings are an exposed hardware feature, one difference is that user code runs in ring 0. Despite changing the behavior of certain instructions, this does not typically result in any incompatibilities for existing code. Ring 3 is also available and can optionally be used to confine untrusted code. Another difference is that system calls must be performed as hypercalls. To simplify supporting this change, we provide a mechanism that can detect when a system call is performed from ring 0 and automatically redirect it to the kernel as a hypercall. This is one of many features included in libDune.

libDune is a library created to make it easier to build user programs that make use of Dune. It is completely untrusted by the kernel and consists of a collection of utilities that aid in managing and configuring privileged hardware features. Major components of libDune include a page table manager, an ELF loader, a simple page allocator, and routines that assist user programs in managing exceptions and system calls. libDune is currently 5,898 lines of code.

We also provide an optional, modified version of libc that uses `VMCALL` instructions instead of `SYSCALL` instructions in order to get a slight performance benefit.

4.1 Bootstrapping

In many ways, transitioning a process into Dune mode is similar to booting an OS. The first issue is that a valid page table must be provided before enabling Dune. A simple identity mapping is insufficient because, although the goal is to have process addresses remain consistent before and after the transition, the compressed layout of the EPT must be taken into account. After a page table is created, the Dune entry *ioctl* is called with the page table root as an argument. The Dune module then switches the process to Dune mode and begins executing code, using the provided page table root as the initial `%CR3`. From there, libDune configures privileged registers to set up a reasonable operating environment. For example, it loads a GDT to provide basic flat segmentation and loads an IDT so that hardware exceptions can be captured. It also sets up a separate stack in the TSS to handle double faults and configures the GS segment base in order to easily access per-thread data.

4.2 Limitations

Although we are able to run a wide variety of Linux programs, libDune is still missing some functionality. First, we have not fully integrated support for signals despite the fact that they are reported by the Dune module. Applications are required to use *dune_signal* whereas a more compatible solution would override several libc symbols like *signal* and *sigaction*. Second, although we support pthreads, some utilities in libDune, such as page table management, are not yet thread-safe. Both of these issues could be resolved with further implementation.

One unanticipated challenge with working in a Dune environment is that system call arguments must be valid host-virtual addresses, regardless of how guest-virtual mappings are setup. In many ways, this parallels the need to provide physical addresses to hardware devices that perform DMA. In most cases we can work around the issue by having the guest-virtual address space mirror the host-virtual address space. For situations where this is not possible, walking the user page table to adjust system call argument addresses is necessary.

Another challenge introduced by Dune is that by exposing greater access to privileged hardware, user programs require more architecture-specific code, potentially reducing portability. libDune currently provides an x86-centric API, so it is already compatible with AMD machines. However, it should be possible to modify libDune to support non-x86 architectures in a fashion that parallels the construction of many OS kernels. This would require libDune to provide an efficient architecture independent interface, a topic worth exploring in future revisions.

5 Applications

Dune is generic enough that it lets us improve on a broad range of applications. We built two security-related applications, a sandbox and privilege separation system, and one performance-related application, a garbage collector. Our goals were simpler implementations, higher performance, and where applicable, improved security.

5.1 Sandboxing

Sandboxing is the process of confining code so as to restrict the memory it can access and the interfaces or system calls it can use. It is useful for a variety of purposes, such as running native code in web browsers, creating secure OS containers, and securing mobile phone applications. In order to explore Dune's potential for these types

4 用户模式环境

使用 Dune 的进程的执行环境与普通进程存在一些差异。由于权限环是暴露的硬件特性，一个差异是用户代码运行在环0中。尽管修改了某些指令的行为，但这通常不会导致现有代码的不兼容。环3也是可用的，并且可以选择性地用于隔离不受信任的代码。另一个差异是系统调用必须作为超调用执行。为了简化支持这一变化，我们提供了一种机制，可以检测系统调用是否从环0执行，并自动将其重定向为内核的超调用。这是 libDune 包含的众多功能之一。

libDune 是一个库，旨在简化构建使用 Dune 的用户程序。它完全不受内核信任，由一系列用于管理和配置特权硬件特性的工具组成。libDune 的主要组件包括一个页表管理器、一个 ELF 加载器、一个简单的页分配器，以及帮助用户程序管理和处理异常和系统调用的例程。libDune 目前包含 5,898 行代码。

我们还提供了一个可选的、修改过的 libc 版本，它使用 VMCALL 指令而不是 SYSCALL 指令，以获得轻微的性能提升。

4.1 引导

在很多方面，将进程切换到Dune模式与启动操作系统类似。第一个问题是，在启用Dune之前必须提供有效的页表。简单的身份映射是不够的，因为尽管目标是在转换前后保持进程地址的一致性，但必须考虑EPT的压缩布局。创建页表后，会调用Dune入口ioctl，并将页表根作为参数。然后Dune模块将进程切换到Dune模式并开始执行代码，使用提供的页表根作为初始%CR3。从那里开始，libDune配置特权寄存器以设置一个合理的操作系统环境。例如，它加载一个GDT以提供基本的平坦分段，并加载一个IDT以便捕获硬件异常。它还在TSS中设置了一个单独的栈来处理双重故障，并配置了GS段基以轻松访问每个线程的数据。

4.2 限制

尽管我们能够运行各种Linux程序，但libDune仍然缺少一些功能。首先，尽管Dune模块报告了信号，但我们还没有完全集成对信号的支持。应用程序必须使用dune signal，而一个更兼容的解决方案将覆盖几个libc符号，如signal和sigaction。其次，尽管我们支持pthread，但libDune中的一些工具，如页表管理，还不是线程安全的。这两个问题都可以通过进一步实现来解决。

在 Dune 环境中工作的一个未预料的挑战是，系统调用参数必须是有效的宿主机虚拟地址，无论客户机虚拟映射如何设置。在很多方面，这类似于向执行 DMA 的硬件设备提供物理地址的需要。在大多数情况下，我们可以通过让客户机虚拟地址空间镜像宿主机虚拟地址空间来解决这个问题。对于无法这样做的场景，需要遍历用户页表来调整系统调用参数地址。

Dune 带来的另一个挑战是，通过提供对特权硬件的更大访问权限，用户程序需要更多与架构相关的代码，这可能会降低可移植性。libDune 目前提供以 x86 为主的 API，因此它已经与 AMD 机器兼容。然而，应该可以通过修改 libDune 来支持非 x86 架构，其方式类似于许多操作系统内核的构建。这将要求 libDune 提供一个高效的、与架构无关的接口，这是一个值得在后续版本中探索的主题。

5 应用程序

Dune 足够通用，使我们能够在广泛的应用领域进行改进。我们构建了两个与安全相关的应用程序：一个沙盒和一个特权分离系统，以及一个与性能相关的应用程序：一个垃圾回收器。我们的目标是实现更简单的实现、更高的性能，以及在适用的情况下提高安全性。

5.1 沙盒

沙盒化是将代码进行限制的过程，以限制其可访问的内存和可使用的接口或系统调用。它适用于多种用途，例如在网页浏览器中运行原生代码、创建安全的操作系统容器以及保护手机应用程序。为了探索 Dune 在这些类型的应用中的潜力，我们构建了一个支持原生 64 位 Linux 可执行文件的沙盒。

of applications, we built a sandbox that supports native 64-bit Linux executables.

The sandbox enforces security through privilege modes by running a trusted sandbox runtime in ring 0 and an untrusted binary in ring 3, both operating within a single address space (on the left of Figure 1, the top and middle boxes respectively). Memory belonging to the sandbox runtime is protected by setting the supervisor bit in appropriate page table entries. Whenever the untrusted binary performs an unsafe operation such as trying to access the kernel through a system call or attempting to modify privileged state, libDune receives an exception and jumps into a handler provided by the sandbox runtime. In this way, the sandbox runtime is able to filter and restrict the behavior of the untrusted binary.

While we rely on the kernel to load the sandbox runtime, the untrusted binary must be loaded in userspace. One risk is that it could contain maliciously crafted headers designed to exploit flaws in the ELF loader. We hardened our sandbox against this possibility by using two separate ELF loaders. First, the sandbox runtime uses a minimal ELF loader (part of libDune), that only supports static binaries, to load a second ELF loader into the untrusted environment. We choose to use *ld-linux.so* as our second ELF loader because it is already used as an integral and trusted component in Linux. Then, the sandbox runtime executes the untrusted environment, allowing the second ELF loader to load an untrusted binary entirely from ring 3. Thus, even if the untrusted binary is malicious, it does not have a greater opportunity to attack the sandbox during ELF loading than it would while running inside the sandbox normally.

So far our sandbox has been applied primarily as a tool for filtering Linux system calls. However, it could potentially be used for other purposes, including providing a completely new system call interface. For system call filtering, a large concern is to prevent execution of any system call that could corrupt or disable the sandbox runtime. We protect against this hazard by validating each system call argument, checking to make sure performing the system call would not allow the untrusted binary to access or modify memory belonging to the sandbox runtime. We do not yet support all system calls, but we support enough to run most single-threaded Linux applications. However, nothing prevents supporting multi-threaded programs in the future.

We implemented two policies on top of the sandbox. Firstly, we support a null policy that allows system calls to pass through but still validates arguments in order to protect the sandbox runtime. It is intended primarily to demonstrate raw performance overhead. Secondly, we support a userspace firewall. It uses system call interposition to inspect important network system calls, such as *bind* and *connect*, and prevents communication with undesirable parties as specified by a policy description.

To further demonstrate the flexibility of our sandbox, we also implemented a checkpointing system that can serialize an application to disk and then restore execution at a later time. This includes saving memory, registers, and system call state (*e.g.,* open file descriptors).

5.2 Wedge

Wedge [10] is a privilege separation system. Its core abstraction is an *sthread* which provides *fork*-like isolation with *pthread*-like performance. An sthread is a lightweight process that has access to memory, file descriptors and system calls as specified by a policy. The idea is to run risky code in an sthread so that any exploits will be contained within it. In a web server, for example, each client request would run in a separate sthread to guarantee isolation between users. To make this practical, sthreads need fast creation (*e.g.,* one per request) and context switch time. Fast creation can be achieved through sthread *recycling*. Instead of creating and killing an sthread each time, an sthread is checkpointed on its first creation (while still pristine and unexploited) and restored on exit so that it can be safely reused upon the next creation request. Doing so reduces sthread creation cost to the (cheaper) cost of restoring memory.

Wedge uses many of Dune's hardware features. Ring protection is used to enforce system call policies; page tables limit what memory sthreads can access; dirty bits are used to restore memory during sthread recycling; and the tagged TLB is used for fast context switching.

5.3 Garbage Collection

Garbage collectors (GC) often utilize memory management hardware to speed up collection [28]. Appel and Li [5] explain several techniques that use standard user level virtual memory protection operations, whereas Azul Systems [15, 36] went to the extent of modifying the kernel and system call interface. By contrast, Dune provides a clean and efficient way to access relevant hardware directly. The features provided by Dune that are of interest to garbage collectors include:

- **Fast faults**. GCs often use memory protection and fault handling to implement read and write barriers.
- **Dirty bits**. Knowing what memory has been touched since the last collection enables optimizations and can be a core part of the algorithm.

为了探索 Dune 在这些类型的应用中的潜力，我们构建了一个支持原生 64 位 Linux 可执行文件的沙盒。

沙盒通过特权模式强制执行安全策略，在环0中运行受信任的沙盒运行时，在环3中运行不受信任的二进制文件，两者都在单个地址空间内运行（如图1左侧，分别为顶部和中间框）。沙盒运行时的内存通过在适当的页表条目中设置监督位来保护。每当不受信任的二进制文件执行不安全操作，例如尝试通过系统调用访问内核或尝试修改特权状态时，libDune会接收异常并跳转到沙盒运行时提供的处理器。通过这种方式，沙盒运行时能够过滤和限制不受信任的二进制文件的行为。

虽然我们依赖内核加载沙盒运行时，但不受信任的二进制文件必须在用户空间加载。一个风险是它可能包含恶意构造的头部，旨在利用ELF加载器的漏洞。我们通过使用两个独立的ELF加载器来加固我们的沙盒，以防范这种可能性。首先，沙盒运行时使用一个极简的ELF加载器（libDune的一部分），该加载器仅支持静态二进制文件，将第二个ELF加载器加载到不受信任的环境中。我们选择使用ld-linux.so作为第二个ELF加载器，因为它已经是Linux中一个整体且受信任的组件。然后，沙盒运行时执行不受信任的环境，允许第二个ELF加载器从环3完全加载不受信任的二进制文件。因此，即使不受信任的二进制文件是恶意的，它在ELF加载期间攻击沙盒的机会也不会比在沙盒中正常运行时更大。

到目前为止，我们的沙盒主要被用作过滤 Linux 系统调用的工具。然而，它有可能被用于其他目的，包括提供一个全新的系统调用接口。对于系统调用过滤，一个主要的问题是防止执行任何可能破坏或禁用沙盒运行时的系统调用。我们通过验证每个系统调用参数来防范这种风险，确保执行系统调用不会让不受信任的二进制程序访问或修改属于沙盒运行时的内存。我们目前还不支持所有系统调用，但我们支持足够多的系统调用，可以运行大多数单线程 Linux 应用程序。然而，未来没有任何东西可以阻止我们支持多线程程序。

我们在沙盒之上实现了两种策略。首先，我们支持一种空策略，它允许系统调用通过，但仍然验证参数，以保护沙盒运行时。它主要旨在展示原始性能开销。其次，我们支持一种用户空间防火墙。它使用系统调用拦截来检查重要的网络系统调用，例如bind和connect，并根据策略描述阻止与不希望通信的方的通信。

为了进一步展示我们沙盒的灵活性，我们还实现了一个检查点系统，可以将应用程序序列化到磁盘，并在稍后时间恢复执行。这包括保存内存、寄存器和系统调用状态（例如，打开的文件描述符）。

5.2 Wedge

Wedge [10] 是一个特权分离系统。它的核心抽象是一个 sthread，它提供了类似 fork 的隔离性和类似 pthread 的性能。一个 sthread 是一个轻量级进程，它可以访问根据策略指定的内存、文件描述符和系统调用。这个想法是在 sthread 中运行有风险的代码，以便任何利用都会被限制在其内部。例如，在一个 Web 服务器中，每个客户端请求都会在一个单独的 sthread 中运行，以保证用户之间的隔离。为了使这一点实用化，sthreads 需要快速创建（例如，每个请求一个）和上下文切换时间。快速创建可以通过 sthread 循环利用来实现。每次不是创建和销毁 sthread，而是在第一次创建时（在仍然原始且未被利用时）对 sthread 进行检查点，并在退出时将其存储起来，以便在下一个创建请求时安全地重用。这样做将 sthread 创建成本降低到（更便宜）恢复内存的成本。

Wedge 采用了 Dune 的许多硬件特性。环保护用于执行系统调用策略；页表限制 sthreads 可以访问的内存；脏位用于在 sthread 回收期间恢复内存；带标签的 TLB 用于快速上下文切换。

5.3 垃圾回收

垃圾收集器（GC）通常利用内存管理硬件来加速收集 [28]。Appel 和 Li [5] 解释了若干种使用标准用户级虚拟内存保护操作的技术，而 Azul Systems [15, 36] 则进一步修改了内核和系统调用接口。相比之下，Dune 提供了一种干净且高效的方式来直接访问相关硬件。对垃圾收集器感兴趣的 Dune 提供的功能包括：

- 快速故障。GC 通常使用内存保护和故障处理来实现读写屏障。
- 脏位。知道自上次收集以来哪些内存已被访问，可以实现优化，并且可能是算法的核心部分。

- **Page table**. One optimization in a moving GC is to free the underlying physical frame without freeing the virtual page it was backing. This is useful when the data has been moved but references to the old location remain and can still be caught through page faults. Remapping memory can also be performed to reduce fragmentation.
- **TLB control**. GCs often manipulate memory mappings at high rates, making control over TLB invalidation very useful. If it can be controlled, mapping manipulations can be effectively batched, rendering certain algorithms more feasible.

We modified the Boehm GC [12] to use Dune in order to improve performance. The Boehm GC is a robust mark-sweep collector that supports parallel and incremental collection. It is designed either to be used as a conservative collector with C/C++ programs, or by compiler and run-time backends where the conservativeness can be controlled. It is widely used, including by the Mono project and GNU Objective C.

An important implementation question for the Boehm GC is how dirty pages are discovered and managed. The two original options were (i) utilizing *mprotect* and signal handlers to implement its own dirty bit tracking; or (ii) utilizing OS provided dirty bit read methods such as the Win32 API call *GetWriteWatch*. In Dune we support and improve both methods.

A direct port to Dune already gives a performance improvement because *mprotect* can directly manipulate the page table and a page fault can be handled directly without needing an expensive `SIGSEGV` signal. The GC manipulates single pages 90% of the time, so we were able to improve performance further by using the `INVPLG` instruction to flush only a single page instead of the entire TLB. Finally, in Dune, the Boehm GC can access dirty bits directly without having to emulate this functionality. Some OSes provide system calls for reading page table dirty bits. Not all of these interfaces are well matched to GC—for instance, SunOS examines the entire virtual address space rather than permit queries for a particular region. Linux provides no user-level access at all to dirty bits.

The work done on the Boehm GC represents a straightforward application of Dune to a GC. It is worth also examining the changes made by Azul Systems to Linux so that they could support their C4 GC [36] and mapping this to the support provided by Dune:

- **Fast faults**. Azul modified the Linux memory protection and mapping primitives to greatly improve performance, part of this included allowing hardware exceptions to bypass the kernel and be handled directly by usermode.
- **Batched page table**. Azul enabled explicit control of TLB invalidation and a shadow page table to expose a prepare and commit style API for batching page table manipulation.
- **Shatter/Heal**. Azul enabled large pages to be ‘shattered’ into small pages or a group of small pages to be ‘healed’ into a single large page.
- **Free physical frames**. When the Azul C4 collector frees an underlying physical frame, it will trap on accesses to the unmapped virtual pages in order to catch old references.

All of the above techniques and interfaces can be implemented efficiently on top of Dune, with no need for any kernel changes other than loading the Dune module.

6 Evaluation

In this section, we evaluate the performance and utility of Dune. Although using VT-x has an intrinsic cost, in most cases, Dune's overhead is relatively minor. On the other hand, Dune offers significant opportunities to improve security and performance for applications that can take advantage of access to privileged hardware features.

All tests were performed on a single-socket machine with an Intel Xeon E3-1230 v2 (a four core Ivy Bridge CPU clocked at 3.3 GHz) and 16GB of RAM. We installed a recent 64-bit version of Debian Linux that includes Linux kernel version 3.2. Power management features, such as frequency scaling, were disabled.

6.1 Overhead from Running in Dune

Performance in Dune is impacted by two main sources of overhead. First, VT-x increases the cost of entering and exiting the kernel—VM entries and VM exits are more expensive than fast system call instructions or exceptions. As a result, both system calls and other types of faults (*e.g.,* page faults) must pay a fixed cost in Dune. Second, using the EPT makes TLB misses more expensive because, in some cases, the hardware page walker must traverse two page tables instead of one.

We built synthetic benchmarks to measure both of these effects. Table 2 shows the overhead of system calls, page faults, and page table walks. For system calls, we manually performed *getpid*, an essentially null system call (worst case for Dune), and measured the round-trip latency. For page faults, we measured the time it took to fault in a pre-zeroed memory page by the kernel. Finally, for page table walks, we measured the time spent filling a TLB miss.

- **页表**。　　移动GC中的一个优化是
释放底层物理帧，但不释放它所支持的虚拟页。这在进行
数据迁移后但旧位置TLB控制尚未清除时很有用。
数据已被移动但引用到旧位置
操作仍然存在，并且仍然可以通过页错误来捕获。
还可以通过重映射内存来减少碎片。
- TLB控制。GC通常以高频率操作内存映射，这使得TLB失效控制非常实用。如果可以控制，映射操作可以有效地批量处理，使某些算法更可行。
某些算法更可行。

我们修改了 Boehm GC [12] 以使用 Dune，目的是提升性能。Boehm GC 是一种稳健的标记-清除收集器，支持并行和增量收集。它设计为可用于 C/C++ 程序作为保守收集器，或由编译器和运行时后端使用，其中保守性可以控制。它被广泛使用，包括 Mono 项目和 GNU Objective C。

Boehm GC 的重要实现问题之一是如何发现和管理脏页。最初的两个选项是 (i) 利用 mprotect 和信号处理器实现自己的脏位跟踪；或 (ii) 利用操作系统提供的脏位读取方法，如 Win32 API 调用 GetWriteWatch。在 Dune 中，我们支持并改进了这两种方法。

直接将 Boehm GC 移植到 Dune 已经带来了性能提升，因为 mprotect 可以直接操作页表，且页错误可以直接处理，无需昂贵的 SIGSEGV 信号。收集器 90% 的时间操作单页，因此我们通过使用 INVPLG 指令仅刷新单页而不是整个 TLB，进一步提升了性能。最后，在 Dune 中，Boehm GC 可以直接访问脏位，无需模拟此功能。一些操作系统提供读取页表脏位的系统调用。并非所有这些接口都与收集器匹配——例如，SunOS 检查整个虚拟地址空间，而不是允许查询特定区域。Linux 完全没有用户级对脏位的访问。

对 Boehm GC 的工作代表了一种将 Dune 应用于 GC 的直接应用。也值得考察 Azul Systems 对 Linux 所做的修改，以便他们能够支持他们的 C4 GC [36]，并将其映射到 Dune 提供的支持：

- 快速故障。Azul修改了Linux内存保护和映射原语，以大幅提升性能，其中一部分包括允许硬件异常绕过内核，并由用户模式直接处理
用户模式。
- 批量页表。Azul启用了对TLB失效的显式控制，并引入了影子页表，以提供一种准备和提交风格的 API，用于批量页表操作。
- 碎分/修复。Azul启用了大页可以被‘碎分’成小页，或一组小页可以被‘修复’成一个单独的大页。
- 释放物理帧。当Azul C4收集器　　r
释放一个底层物理帧时，它会在访问
进程访问未映射的虚拟页，以便捕获旧的引用。

所有上述技术和接口都可以在 Dune 之上高效实现，除了加载 Dune 模块外，无需任何内核更改。

6 评估

在本节中，我们评估了Dune的性能和实用性。尽管使用VT-x存在固有成本，但在大多数情况下，Dune的开销相对较小。另一方面，Dune为能够利用特权硬件功能的应用提供了显著的机会，以提升安全性和性能。

所有测试均在单插槽机器上进行，该机器配备了一颗Intel Xeon E3-1230 v2（四核Ivy Bridge CPU，主频3.3 GHz）和16GB内存。我们安装了一个包含Linux内核版本3.2的较新64位Debian Linux版本。电源管理功能（如频率缩放）已禁用。

6.1 Dune运行的开销

Dune中的性能受两种主要开销来源影响。首先，VT-x增加了进入和退出内核的成本——VM入口和VM退出比快速系统调用指令或异常更昂贵。因此，在Dune中，系统调用和其他类型的错误（例如页错误）都必须支付固定成本。其次，使用EPT使得TLB未命中成本更高，因为某些情况下，硬件页面遍历器必须遍历两个页表，而不是一个。

我们构建了合成基准测试来测量这两种效应。表2显示了系统调用、页错误和页表遍历的开销。对于系统调用，我们手动执行了getpid，这是一个本质上为空（对Dune而言最坏情况）的系统调用，并测量了往返延迟。

效率。对于页错误，我们测量了内核将一个预清零的内存页错误注入所需的时间。最后，对于页表遍历，我们测量了填充一个TLB未命中所花费的时间。

	getpid	page fault	page walk
Linux	138	2,687	35.8
Dune	895	5,093	86.4

Table 2: Average time (in cycles) of operations that have overhead in Dune compared to Linux.

	ptrace	trap	appel1	appel2
Linux	27,317	2,821	701,413	684,909
Dune	1,091	587	94,496	94,854

Table 3: Average time (in cycles) of operations that are faster in Dune compared to Linux.

Measuring TLB miss overhead required us to build a simple memory stress tool. It works by performing a random page-aligned walk across 2^{16} memory pages. This models a workload with poor memory locality, as nearly every memory access results in a last-level TLB miss. We then divided the total number of cycles spent waiting for page table walks, as reported by a performance counter, by the total number of memory references, giving us a cycle cost per page walk.

In general, the overhead Dune adds has only a small effect on end-to-end performance, as we show in Section 6.3. For system calls, the time spent in the kernel tends to be a larger cost than the fixed VMX mode transition costs. Page fault overhead is also not much of a concern, as page faults tend to occur infrequently during normal use, and direct access to exception hardware is available when higher performance is required. On the other hand, Dune's use of the EPT does impact performance in certain workloads. For applications with good memory locality or a small working set, it has no impact because the TLB hit rate is sufficiently high. However, for application with poor memory locality or a large working set, more frequent TLB misses result in a measurable slowdown. One effective strategy for limiting this overhead is to use large pages. We explore this possibility further in section 6.3.1.

6.2 Optimizations Made Possible by Dune

Access to privileged hardware features creates many opportunities for optimization. Table 3 shows speedups we achieved in the following OS workloads:

ptrace is a measure of system call interposition performance. This is the cost of a Linux process intercepting a system call (*getpid*) with *ptrace*, forwarding the system call to the kernel and returning the result. In Dune this is the cost of intercepting a system call directly using ring protection in VMX non-root mode, forwarding the system call through a `VMCALL` and returning the result. An additional scenario is where applications wish to intercept system calls but not forward them to the kernel and instead just implement them internally. `PTRACE_SYSEMU` is the most efficient mechanism for doing so since *ptrace* requires forwarding a call to the kernel. The latency of intercepting a system call with `PTRACE_SYSEMU` is 13,592 cycles. In Dune this can be implemented by handling the hardware system call trap directly, with a latency of just 180 cycles. This reveals that most of the Dune *ptrace* benchmark overhead was in fact forwarding the *getpid* system call via a `VMCALL` rather than intercepting the system call.

trap indicates the time it takes for a process to get an exception for a page fault. We compare the latency of a SIGSEGV signal in Linux with a hardware-generated page fault in Dune.

appel1 is a measure of user-level virtual memory management performance. It corresponds to the TRAP, PROT1, and UNPROT test described in [5], where 100 protected pages are accessed, causing faults. Then, in the fault handler, the faulting page is unprotected, and a new page is protected.

appel2 is another measure of user-level virtual memory management performance. It corresponds to the PROTN, TRAP, and UNPROT test described in [5], where 100 pages are protected. Then each is accessed, with the fault handler unprotecting the faulting page.

6.3 Application Performance

6.3.1 Sandbox

We evaluated the performance of our sandbox by running two types of workloads. First, we tested compute performance by running SPEC2000. Second, we tested IO performance by running lighttpd. The null sandbox policy was used in both cases.

Figure 3 shows the performance of SPEC2000. In general, the sandbox had very low overhead, averaging only 2.9% percent slower than Linux. However, the *mcf* and *ammp* benchmarks were outliers, with 20.9% and 10.1% slowdowns respectively. This deviation in performance can be explained by EPT overhead, as we observed a high TLB miss rate. We also measured SPEC2000 in VMware Player, and, as expected, EPT overhead resulted in very similar drops in performance.

We then adjusted the sandbox to avoid EPT overhead by backing large memory allocations with 2MB large

	getpid	页错误	页遍历
Linux	138	2,687	35.8
Dune	895	5,093	86.4

表2：与Linux相比，Dune中具有开销的操作的平均时间（以周期为单位）。

	ptrace	trap	appel1	appel2
Linux	27,317	2,821	701,413	684,909
Dune	1,091	587	94,496	94,854

表 3：与 Linux 相比，Dune 中操作的平均时间（以周期为单位），这些操作更快。

测量TLB未命中开销需要我们构建一个简单的内存压力工具。它的工作原理是在 2^{16} 内存页面上执行随机对齐的遍历。这模拟了一个内存局部性差的负载，因为几乎每个内存访问都会导致最后一级TLB未命中。然后，我们将性能计数器报告的用于等待页表遍历的总周期数除以内存引用的总数，从而得到每页遍历的周期成本。

一般来说，Dune带来的开销对端到端性能的影响很小，正如第6.3节所示。对于系统调用，内核中花费的时间往往比固定的VMX模式转换成本更高。页错误开销也不是什么大问题，因为页错误在正常使用期间通常很少发生，当需要更高性能时，可以直接访问异常硬件。另一方面，Dune对EPT的使用确实在某些工作负载中会影响性能。对于具有良好内存局部性或较小工作集的应用，它没有影响，因为TLB命中率足够高。然而，对于内存局部性差或工作集较大的应用，更频繁的TLB未命中会导致可测量的性能下降。限制这种开销的一种有效策略是使用大页。我们在第6.3.1节进一步探讨了这种可能性。

6.2 Dune带来的优化

对特权硬件功能的访问为优化提供了许多机会。表3显示了我们在以下操作系统工作负载中实现的加速效果:

ptrace 是系统调用拦截性能的衡量指标。这是 Linux 进程使用 ptrace 拦截系统调用（getpid），将系统调用转发给内核并返回结果的开销。在 Dune 中，这是在 VMX 非根模式下直接使用环保护拦截系统调用，通过 VMCALL 转发系统调用并返回结果的开销。另一种场景是应用程序希望拦截系统调用，但不想将它们转发给内核，而是直接在内部实现。PTRACE SYSEMU 是实现此目的的最高效机制，因为 ptrace 需要将调用转发给内核。使用 PTRACE SYSEMU 拦截系统调用的延迟为 13,592 周期。在 Dune 中，这可以通过直接处理硬件系统调用陷阱来实现，延迟仅为 180 周期。这表明 Dune ptrace 基准测试的大部分开销

这样做是因为ptrace需要将调用转发给内核。使用PTRACE SYSEMU拦截系统调用的延迟为13,592周期。在Dune中，这可以通过直接处理硬件系统调用陷阱来实现，其延迟仅为180周期。这揭示了大多数Dune ptrace基准测试都超

head 实际上是通过 VMCALL 而不是拦截系统调用来转发 getpid 系统调用的。

VMCALL 而不是拦截系统调用。

陷阱表示进程因页错误获得异常所需的时间。我们将 Linux 中 SIGSEGV 信号的延迟与 Dune 中硬件生成的页错误进行比较。

appel1 是用户级虚拟内存管理性能的衡量指标。它对应于 [5]，中描述的 TRAP、PROT1 和 UNPROT 测试，其中访问了 100 个受保护的页面，导致发生错误。然后在错误处理程序中，故障页面将不再受保护，而新页面将被保护。

appel2 是用户级虚拟内存管理性能的另一个衡量指标。它对应于 [5], 中描述的 PROTN、TRAP 和 UNPROT 测试，其中保护了 100 个页面。然后每个页面都被访问，错误处理程序将故障页面解除保护。

6.3 应用性能

6.3.1 沙盒

我们通过运行两种类型的负载评估了我们的沙盒性能。首先，我们通过运行SPEC2000测试计算性能。其次，我们通过运行lighttpd测试IO性能。在这两种情况下，都使用了空沙盒策略。

图3显示了SPEC2000的性能。总体而言，沙盒的开销非常低，平均比Linux慢2.9%。然而，mcf和ammp基准测试是异常值，分别慢了20.9%和10.1%。这种性能偏差可以用EPT开销来解释，因为我们观察到TLB未命中率高。我们还测量了VMware Player中的SPEC2000，正如预期的那样，EPT开销导致性能下降非常相似。

然后我们调整了沙盒，通过用2MB大页支持大内存分配来避免EPT开销

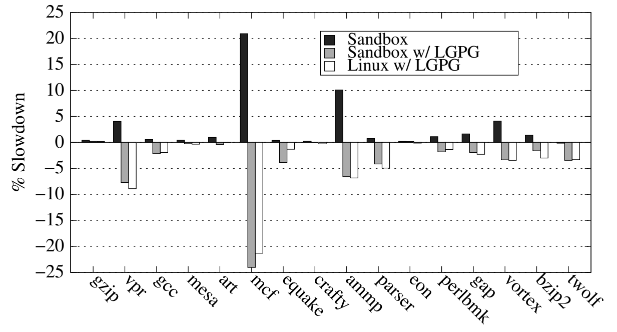

Figure 3: Average SPEC2000 slowdown compared to Linux for the sandbox, the sandbox using large pages, and Linux using large pages identically.

	1 client	100 clients
Linux	2,236	24,609
Dune sandbox	2,206	24,255
VMware Player	734	5,763

Table 4: Lighttpd performance (in requests per second).

	create	ctx switch	http request
fork	81	0.49	454
Dune sthread	2	0.15	362

Table 5: Wedge benchmarks (times in microseconds).

pages, both in the EPT and the user page table. Supporting this optimization was straightforward because we were able to intercept *mmap* calls and transparently modify them to use large pages. Such an approach does not cause much memory fragmentation because large pages are only used selectively. In order to perform a direct comparison, we tested SPEC2000 in a modified Linux environment that allocates large pages in an identical fashion using libhugetlbfs [1]. When large pages were used for both, average performance in the sandbox and Linux was nearly identical (within 0.1%).

Table 4 shows the performance of lighttpd, a single-threaded, single-process, event-based HTTP server. Lighttpd exercises the kernel to a much greater extent than SPEC2000, making frequent system calls and putting load on the network stack. Lighttpd performance was measured over Gigabit Ethernet using the Apache *ab* benchmarking tool. We configured *ab* to repeatedly retrieve a small HTML page over the network with different levels of concurrency: 1 client for measuring latency and 100 clients for measuring throughput.

We found that the sandbox incurred only a slight slowdown, less than 2% for both the latency and throughput test. This slowdown can be explained by Dune's higher system call overhead. Using *strace*, we determined that lighttpd was performing several system calls per connection, causing frequent VMX transitions. However, VMware Player, a conventional VMM, experienced much greater overhead: 67% for the latency test and 77% for the throughput test. Although VMware Player pays VMX transition costs too, the primary reason for the slowdown is that each network request must traverse two network stacks, one in the guest and one in the host.

We also found that the sandbox provides an easily extensible framework that we used to implement checkpointing and our firewall. The checkpointing implementation consisted of approximately 450 SLOC with 50 of those being enhancements to the sandbox loader. Our firewall was around 200 SLOC with half of that being the firewall rules parser.

6.3.2 Wedge

Wedge has two main benchmarks: sthread creation and context switch time. These are compared to *fork*, the system call used today to implement privilege separation. As shown in Table 5, sthread creation is faster than *fork* because instead of creating a new process each time, an sthread is reused from a pool and "recycled" by restoring dirty memory and state. Context switch time in sthreads is low because TLB flushes are avoided by using the tagged TLB. In Dune sthreads are created 40× faster than processes and the context switch time is 3× faster. In previous Wedge implementations sthread creation was 12× faster than *fork* with no improvement in context switch time [9]. Dune is faster because it can leverage the tagged TLB and avoid kernel calls to create sthreads. The last column of Table 5 shows an application benchmark of a web server serving a static file on a LAN where each request runs in a newly *fork*ed process or sthread for isolation. Dune sthreads show a 20% improvement here.

The original Wedge implementation consisted of a 700-line kernel module and a 1,300-line library. A userspace-only implementation of Wedge exists, though the authors lamented that POSIX did not offer adequate APIs for memory and system call protection, hence the result was a very complicated 5,000-line implementation [9]. Dune instead exposes the hardware protection features needed for a simple implementation, consisting of only 750 lines of user code.

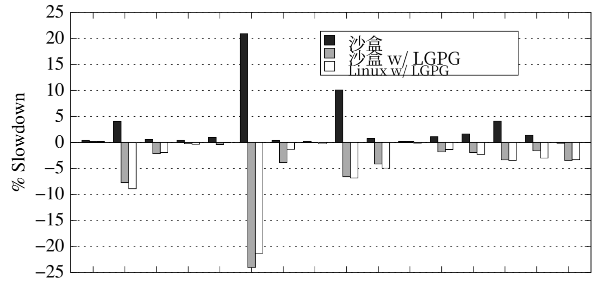

图 3：沙盒、使用大页的沙盒以及使用大页的 Linux 相对于 Linux 的平均 SPEC2000 减速。

	1 个客户端	100 个客户端
Linux	2,236	24,609
Dune沙盒	2,206	24,255
VMware Player	734	5,763

表4：lighttpd性能（每秒请求数）。

	创建	上下文切换	HTTP请求
fork	81	0.49	454
Dune sthread	2	0.15	362

表 5：Wedge 基准测试（时间以微秒为单位）。

同时在EPT和用户页表中。支持这种优化很简单，因为我们能够拦截mmap调用并透明地修改它们以使用大页。这种方法不会引起太多内存碎片化，因为大页只是被选择性地使用。为了进行直接比较，我们在一个使用libhugetlbfs [1]以相同方式分配大页的修改版Linux环境中测试了SPEC2000。当两者都使用大页时，沙盒和Linux的平均性能几乎相同（在0.1%以内）。

表4展示了lighttpd的性能，它是一款单线程、单进程、基于事件的HTTP服务器。与SPEC2000相比，lighttpd对内核的调用程度要高得多，频繁进行系统调用，并对网络栈施加负载。lighttpd的性能通过Apache ab基准测试工具在千兆以太网上进行测量。我们将ab配置为以不同的并发级别重复从网络上获取小型HTML页面：1个客户端用于测量延迟，100个客户端用于测量吞吐量。

我们发现沙盒仅出现了轻微的延迟，延迟和吞吐量测试均小于2%。这种延迟可以归因于Dune更高的系统调用开销。通过strace，我们确定lighttpd每个连接会执行多个系统调用，导致频繁的VMX转换。然而，传统的VMM VMware Player则承受了更大的开销：延迟测试为67%，吞吐量测试为77%。尽管VMware Player也需要支付VMX转换开销，但导致延迟的主要原因是每个网络请求必须穿越两个网络栈——一个在客户机，一个在主机。

我们还发现沙盒提供了一个易于扩展的框架，我们利用它实现了检查点和我们的防火墙。检查点实现大约包含 450 行 SLOC，其中 50 行是对沙盒加载器的改进。我们的防火墙大约有 200 行 SLOC，其中一半是防火墙规则解析器。

6.3.2 Wedge

Wedge 有两个主要基准测试：sthread 创建和上下文切换时间。这些测试与 fork 进行了比较，fork 是当今用于实现特权分离的系统调用。如表 5 所示，sthread 创建比 fork 更快，因为每次不是创建新进程，而是从池中重用 sthread 并通过恢复脏内存和状态来“回收”它。由于使用带标签的 TLB，sthread 中的上下文切换时间较低，避免了 TLB 冲刷。在 Dune 中，sthread 的创建 40× 比进程更快，上下文切换时间 3× 也更快。在之前的 Wedge 实现中，sthread 创建 12× 比 fork 更快，但上下文切换时间 [9] 没有改进。Dune 更快是因为它可以利用带标签的 TLB 并避免内核调用来创建 sthread。表 5 的最后一列显示了一个在局域网中服务静态文件的服务器应用程序基准测试，其中每个请求都在一个新的 fork 进程或 sthread 中运行以实现隔离。Dune sthread 在这里提高了 20%。

原始的 Wedge 实现，由一个 700 行的内核模块和一个 1,300 行的库组成。虽然存在仅运行在用户空间的 Wedge 实现，但作者们抱怨 POSIX 没有提供足够的内存和系统调用保护 API，因此最终结果是一个非常复杂的 5,000 行实现 [9]。Dune 则暴露了硬件保护功能，这些功能对于简单的实现是必需的，仅由 750 行用户代码组成。

	GCBench	LinkedList	HashMap	XML
Collections	542	33,971	161	10
Memory use (MB)				
Allocation	938	15,257	10,352	1,753
Heap	28	1,387	27	1,737
Execution time (ms)				
Normal	1,224	15,983	14,160	6,663
Dune	1,176	16,884	13,715	7,930
Dune TLB	933	14,234	11,124	7,474
Dune dirty	888	11,760	8,391	6,675

Table 6: Performance numbers of the GC benchmarks.

6.3.3 Garbage Collector

We implemented three different sets of modifications to the Boehm GC. The first is the simplest port possible with no attempt to utilize any advanced features of Dune. This benefits from Dune's fast memory protection and fault handling but suffers from the extra TLB costs. The second version improves the direct port by carefully controlling when the TLB is invalidated. The third version avoids using memory protection altogether, instead it reads the dirty bits directly. The direct port required changing 52 lines, the TLB optimized version 91 lines, and the dirty bit version 82 lines.

To test the performance improvements of these changes we used the following benchmarks:

- GCBench [11]. A microbenchmark written by Hans Boehm and widely used to test garbage collector performance. In essence, it builds a large binary tree.
- Linked List. A microbenchmark that builds increasingly large linked lists of integers, summing each one after it is built.
- Hash Map. A microbenchmark that utilizes the Google sparse hash map library [23] (C version).
- XML Parser. A full application that uses the Mini-XML library [34] to parse a 150MB XML file containing medical publications. It then counts the number of publications each author has using a hash map.

The results for these benchmarks are presented in Table 6. The direct port displays mixed results due to the improvement to memory protection and the fault handler but slowdown of EPT overhead. As soon as we start using more hardware features, we see a clear improvement over the baseline. Other than the XML Parser, the TLB version improves performance between 10.9% and 23.8%, and the dirty bit version between 26.4% and 40.7%.

The XML benchmark is interesting as it shows a slowdown under Dune for all three versions: 19.0%, 12.2% and 0.2% slower for the direct, TLB and dirty version respectively. This appears to be caused by EPT overhead, as the benchmark does not create enough garbage to benefit from the modifications we made to the Boehm GC. This is indicated in Table 6; the total amount of allocation is nearly equal to the maximum heap size. We verified this by modifying the benchmark to instead take a list of XML files, processing each sequentially so that memory would be recycled. We then saw a linear improvement in the Dune versions over the baseline as the number of files was increased. With ten 150MB XML files as input, the dirty bit version of the Boehm GC showed a 12.8% improvement in execution time over the baseline.

7 Reflections on Hardware

While developing Dune, we found VT-x to be a surprisingly flexible hardware mechanism. In particular, the fine-grained control provided by the VMCS allowed us to precisely direct how hardware was exposed. However, some hardware changes to VT-x could benefit Dune. One noteworthy area is the EPT, as we encountered both performance overhead and implementation challenges. Hardware modifications have been proposed to mitigate EPT overhead [2, 8]. In addition, modifying the EPT to support the same address width as the regular page table would reduce the complexity of our implementation and improve coverage of the process address space. Further reductions to VM exit and VM entry latency could also benefit Dune. However, we were able to aggressively optimize hypercalls, and VMX transition costs had only a small effect on the performance of the applications we evaluated.

There are a few hardware features that we have not yet exposed, despite the fact that they are available in VT-x and possible to support in Dune. Most seem useful only in special situations. For example, a user program might want to have control over caching in order to prevent information leakage. However, this would only be effective if CPU affinity could be controlled. As another example, access to efficient polling instructions (*i.e.,* `MONITOR` and `MWAIT`) could be useful in reducing power consumption for userspace messaging implementations that perform cache line polling. Finally, exposing access to debug registers could allow user programs to more efficiently set up memory watchpoints.

It may also be useful to provide Dune applications with direct access to IO devices. Many VT-x systems include support for an IOMMU, a device that can be used to make DMA access safe even when it is available to untrusted software. Thus, Dune could be modified to safely expose certain hardware devices. A potential benefit could be reduced IO latency. The availability of SR-IOV makes this

	GCBench	链表	哈希映射	XML
集合	542	33,971	161	10
内存使用（MB）				
分配	938	15,257	10,352	1,753
Heap	28	1,387	27	1,737
执行时间 (毫秒)				
正常	1,224	15,983	14,160	6,663
Dune	1,176	16,884	13,715	7,930
Dune TLB	933	14,234	11,124	7,474
Dune 污染	888	11,760	8,391	6,675

表 6：GC 基准测试的性能数据。

6.3.3 垃圾回收器

我们对 Boehm GC 实现了三种不同的修改方案。第一种是最简单的移植，没有尝试利用 Dune 的任何高级功能。这受益于 Dune 的快速内存保护和错误处理，但会遭受额外的 TLB 成本。第二个版本通过仔细控制 TLB 使其失效来改进直接移植。第三个版本完全避免使用内存保护，而是直接读取脏位。直接移植需要修改 52 行代码，TLB 优化版本需要修改 91 行代码，脏位版本需要修改 82 行代码。

为了测试这些更改带来的性能提升，我们使用了以下基准测试：

- GCBench [11]。这是一个由汉斯·博姆编写的小型基准测试，广泛用于测试垃圾回收器的性能。本质上，它会构建一个大的二叉树。
- 链表。这是一个小型基准测试，它会构建越来越大的整数链表，并在构建后对每个元素求和。
- 哈希表。这是一个小型基准测试，它利用了 Google 稀疏哈希表库 [23](C 版本)。
- XML 解析器。一个完整的应用程序，使用 Mini-XML 库 [34] 解析一个包含 150MB XML 文件的医疗出版物。然后它使用哈希表统计每位作者发表的文章数量。

这些基准测试的结果展示在表 6 中。
表 6。 直接移植由于{{...}}原因表现不一。
内存保护和错误处理器的改进，但EPT开销有所减缓。一旦我们开始使用更多硬件特性，就明显看到相比基线有显著改进。除了XML解析器，TLB版本的性能提升了10.9%到23.8%，而脏位版本提升了26.4%到40.7%。

XML基准测试很有趣，因为它显示了在Dune下所有三个版本中都比较缓慢：

19.0%、12.2%
并且直接、TLB和脏版本重试慢了0.2%

这从表6中可以看出；分配的总量几乎等于最大堆大小。我们通过修改基准测试来验证这一点，使其改为取一组XML文件，按顺序处理每个文件，以便内存可以回收。然后，随着文件数量的增加，我们看到Dune版本相对于基线线性改进。使用十个150MB的XML文件作为输入时，Boehm GC的脏位版本在执行时间上比基线提高了12.8%。

硬件的七次反射

在开发Dune的过程中，我们发现VT-x是一种出乎意料的灵活的硬件机制。特别是VMCS提供的细粒度控制，让我们能够精确地指导硬件的暴露方式。然而，一些对VT-x的硬件变更可能对Dune有益。一个值得注意的领域是EPT，因为我们遇到了性能开销和实现挑战。已经提出了硬件修改来缓解EPT开销 [2, 8]。此外，将EPT修改为支持与常规页表相同的地址宽度将减少我们实现的复杂性，并提高进程地址空间的覆盖范围。进一步降低虚拟机进入和虚拟机退出延迟也可能对Dune有益。然而，我们能够积极优化超调用，并且VMX转换开销对我们在评估的应用程序性能只有较小的影响。

尽管这些硬件特性在VT-x中可用，并且有可能在Dune中支持，但我们尚未暴露它们。大多数特性似乎只在特殊情况下才有用。例如，用户程序可能希望控制缓存，以便

防止信息泄露。然而，这只有在能够控制CPU亲和性时才会有效。作为另一个例子，

访问高效的轮询指令（即MONITOR和MWAIT）可能有助于减少执行缓存行轮询的用户空间消息实现的功耗。最后，暴露对调试寄存器的访问可能允许用户程序更有效地设置内存断点。提供Dune应用程序直接访问IO设备也可能有用。许多VT-x系统包括对IOMMU的支持，这是一种设备，即使在它对不受信任的

调试寄存器可以允许用户程序更有效地设置内存断点。它还可以用于向Dune应用程序提供对IO设备的直接访问。许多VT-x系统包括对IOMMU的支持，IOMMU是一种设备，即使在它对不受信任的可用时，也可以用来使DMA访问安全。

软件。因此，Dune 可以修改以安全地暴露某些硬件设备。潜在的好处可能是降低 IO 延迟。SR-IOV 的可用性使这一点

possibility more practical because it allows a single physical device to be partitioned across multiple guests.

Recently, a variety of non-x86 hardware platforms have gained support for hardware-assisted virtualization, including ARM [38], Intel Itanium [25], and IBM Power [24]. ARM is of particular interest because of its prevalence in mobile devices, making the ARM Virtualization Extensions an obvious future target for Dune. ARM's support for virtualization is similar to VT-x in some areas. For example, ARM is capable of exposing direct access to privileged hardware features, including exceptions, virtual memory, and privilege modes. Moreover, ARM provides a System MMU, which is comparable to the EPT. ARM's most significant difference is that it introduces a new deeper privilege mode call *Hyp* that runs underneath the guest kernel. In contrast, VT-x provides separate operating modes for the guest and VMM. Another difference from VT-x is that ARM does not automatically save and restore architectural state when switching between a VMM and a guest. Instead, the VMM is expected to manage state in software, perhaps creating an opportunity for optimization.

8 Related Work

There have been several efforts to give applications greater access to hardware. For example, The Exokernel [18] exposes hardware features through a low-level kernel interface that allows applications to manage hardware resources directly. Another approach, adopted by the SPIN project [7], is to permit applications to safely load extensions directly into the kernel. Dune shares many similarities with these approaches because it also tries to give applications greater access to hardware. However, Dune differs because its goal is not extensibility. Rather, Dune provides access to privileged hardware features so that they can be used in concert with the OS instead of a means of modifying or overriding it.

The Fluke project [20] supports a nested process model in software, allowing OSes to be constructed "vertically." Dune complements this approach because it could be used to efficiently support an extra OS layer between the application and the kernel through use of privilege mode hardware. However, the hardware that Dune exposes can only support a single level instead of the multiple levels available in Fluke.

A wide range of strategies have been employed to support sandboxing, such as ptrace [16], dedicated kernel modifications [16, 21, 33], binary translation [19], and binary verification [39]. To our knowledge, Dune is the first system to support sandboxing entirely through user-level access to hardware protection, improving performance and reducing code complexity. For example, Native Client [39] reports an average SPEC2000 overhead of 5% with a worst case performance of 12%—anecdotally, we observed higher overheads on modern microarchitectures. By contrast, we were able to achieve nearly zero average overhead (1.4% worst case) for the same benchmarks in Dune. Our sandbox is similar to Native Client in that it creates a secure subdomain within a process. However, Native Client is more portable than Dune because it does not require virtualization hardware or kernel changes.

Like Dune, some previous work has used hardware virtualization for non-traditional purposes. For example, VT-x has been suggested as a tool for creating rootkits [29] that are challenging to detect. Moreover, IOMMU hardware has been used to safely isolate malicious device drivers by running them in Linux processes [13].

9 Conclusion

Dune provides ordinary applications with efficient and safe access to privileged hardware features that are traditionally available only to kernels. It does so by leveraging modern virtualization hardware, which enables direct execution of privileged instructions in unprivileged contexts. Our implementation of Dune for Linux uses Intel's VT-x virtualization architecture and provides application-level access to exceptions, virtual memory, and privilege modes. Our evaluation shows both performance and security benefits to Dune. For instance, we built a sandbox that approaches zero overhead, modified a garbage collector to improve performance by up to 40.7%, and created a privilege separation system with 3× less context switch overhead than without Dune.

In an effort to spur adoption, we have structured Dune as a module that works with unmodified Linux kernels. We hope the applications described in this paper are just the first of many uses people will find for the system. The hardware mechanisms exposed by Dune are at the core of many operating systems innovations; their new accessibility from user-level creates opportunities to deploy novel systems without kernel modifications. Dune is freely available at `http://dune.scs.stanford.edu/`.

Acknowledgments

We wish to thank our shepherd, Timothy Roscoe, for his guidance and valuable suggestions. We would also like to thank Edouard Bugnion for feedback on several itera-

变得更加实用，因为它允许单个物理设备被划分到多个客户机。

近来，多种非x86硬件平台已获得硬件辅助虚拟化支持，包括ARM [38], 英特尔安腾 [25], 和IBM Power [24]。ARM特别引人关注，因为它在移动设备中广泛使用，使得ARM虚拟化扩展成为Dune未来明确的目标。ARM的虚拟化支持在某些方面与VT-x相似。例如，ARM能够直接暴露特权硬件功能，包括异常、虚拟内存和特权模式。此外，ARM提供系统MMU，这与EPT相当。ARM最显著的区别是它引入了一种新的更深特权模式Hyp，该模式在客户机内核下方运行。相比之下，VT-x为客户机和VMM提供独立的操作模式。与VT-x的另一区别是，ARM在VMM和客户机之间切换时不会自动保存和恢复架构状态。相反，期望VMM通过软件管理状态，这可能为优化创造机会。

8 相关工作

为了让应用程序获得更大的硬件访问权限，已经有一些尝试。例如，The Exoker- nel [18]通过一个低级内核接口暴露硬件特性，允许应用程序直接管理硬件资源。另一种方法，被SPIN项目 [7], 采用，是允许应用程序直接将扩展安全地加载到内核中。Dune与这些方法有许多相似之处，因为它也试图让应用程序获得更大的硬件访问权限。然而，Dune有所不同，因为它的目标不是可扩展性。相反，Dune提供对特权硬件特性的访问权限，以便它们可以与操作系统协同使用，而不是作为修改或覆盖它的手段。

The Fluke项目 [20] 支持软件中的嵌套进程模型，允许操作系统“垂直”构建。Dune补充了这种方法，因为它可以通过使用特权模式硬件，高效地支持应用程序和内核之间的额外操作系统层。然而，Dune暴露的硬件只能支持单个级别，而不是Fluke中可用的多个级别。

为了支持沙盒，人们已经采用了多种策略，例如ptrace [16], 专用的内核修改 [16, 21, 33], 二进制翻译 [19], 和二进制验证 [39]。据我们所知，Dune是第一个完全通过用户硬件保护级别的访问，提高了性能并降低了代码复杂度。例如，原生客户端 [39] 报告平均 SPEC2000 开销为 5%，最坏情况性能为 12%—据传闻，我们在现代微架构上观察到了更高的开销。相比之下，我们在 Dune 中为相同的基准测试实现了接近零的平均开销（最坏情况为 1.4%）。我们的沙盒与原生客户端类似，它在一个进程中创建了一个安全的子域。然而，原生客户端比 Dune 更便携，因为它不需要虚拟化硬件或内核更改。

像《沙丘》一样，一些先前的工作已将硬件虚拟化用于非传统目的。例如，VT-x曾被建议作为一种创建难以检测的rootkit的工具 [29]。此外，IOMMU硬件已被用于通过在Linux进程中运行它们来安全地隔离恶意设备驱动程序 [13]。

9 结论

《沙丘》为普通应用程序提供了对特权硬件特征的高效且安全的访问，这些特征传统上只有内核才能使用。它通过利用现代虚拟化硬件来实现这一点，该硬件允许在无特权上下文中直接执行特权指令。我们对Linux的《沙丘》实现使用了英特尔的VT-x虚拟化架构，并提供应用程序级对异常、虚拟内存和特权模式的访问。我们的评估表明，《沙丘》在性能和安全性方面都有优势。例如，我们构建了一个接近零开销的沙盒，修改了垃圾收集器以提高性能高达40.7%，并创建了一个特权分离系统，其上下文切换开销比没有《沙丘》时更小 3× 。

为推动采用，我们将Dune构建为一个可与未经修改的Linux内核协同工作的模块。我们希望本文中描述的应用只是人们发现该系统众多用途中的第一个。Dune暴露的硬件机制是许多操作系统创新的核心；它们从用户级获得的新访问能力创造了无需内核修改即可部署新系统的机会。Dune可在http://dune.scs.stanford.edu/免费获取。

致谢

我们感谢我们的指导者，蒂莫西·罗斯科，感谢他的指导和宝贵建议。我们也想感谢爱德华·布吉尼翁对几个迭代提出的反馈。

tions of this paper and for his valuable discussions during the early phases of Dune. Finally, we thank Richard Uhlig, Jacob Leverich, Ben Serebrin, and our anonymous reviewers for suggestions that greatly shaped our paper. This work was funded by DARPA CRASH under contract #N66001-10-2-4088 and by a gift from Google. Adam Belay is supported by a Stanford Graduate Fellowship.

References

[1] Libhugetlbfs. `http://libhugetlbfs.sourceforge.net`, Apr. 2012.

[2] J. Ahn, S. Jin, and J. Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 476–487, Piscataway, NJ, USA, 2012.

[3] AMD. *Secure Virtual Machine Architecture Reference Manual*.

[4] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 13–15, 2007.

[5] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Apr. 1991.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, 1995.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

[9] A. Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, 2009.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, 2008.

[11] H. Boehm. GC Bench. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`, Apr. 2012.

[12] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, 1991.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, pages 9–9, 2010.

[14] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, 1997.

[15] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, 2005.

[16] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation*, OSDI'08, pages 339–354, 2008.

[17] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, Orcas Island, Washington, May 1995.

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995.

[19] B. Ford and R. Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 293–306, 2008.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, 1996.

[21] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[22] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[23] Google. sparsehash. `http://code.google.com/p/sparsehash/`, Apr. 2012.

[24] IBM. *Power ISA, Version 2.06 Revision B*.

[25] Intel. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*.

[26] Intel Corporation. Invalid Instruction Erratum Overview. `http://www.intel.com/support/processors/pentium/sb/cs-013151.htm`, Apr. 2012.

[27] K. Kaspersky and A. Chang. Remote Code Execution thorugh Intel CPU Bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[28] H. Kermany and E. Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, 2006.

[29] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, 2006.

[30] A. Kivity. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[31] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA. *IBM Syst. J.*, 30(1):34–51, Feb. 1991.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[33] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, SSYM'03, 2003.

[34] M. Sweet. Mini-XML: Lightweight XML Library. `http://www.minixml.org/`, Apr. 2012.

[35] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[36] G. Tene, B. Iyengar, and M. Wolf. C4: the Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, 2011.

[37] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48 – 56, May 2005.

[38] P. Varanasi and G. Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, 2011.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, 2009.

[40] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 13–24, 1987.

我们感谢我们的指导者蒂莫西·罗斯科，感谢他的指导和宝贵建议。我们还要感谢爱德华·布吉尼翁，感谢他对本文多个版本的反馈以及他在《沙丘》早期阶段的有价值讨论。最后，我们感谢理查德·乌利希、雅各布·莱维里奇、本·塞雷布林以及我们的匿名审稿人为我们的论文提出的宝贵建议。这项工作由DARPA CRASH项目资助，合同号为N66001-10-2-4088，并得到了谷歌的资助。亚当·贝莱由斯坦福研究生奖学金支持。

参考文献

[1] Libhugetlbfs. `http://libhugetlbfs.sourceforge.net`, Apr. 2012.

[2] J. Ahn, S. Jin, and J. Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 476–487, Piscataway, NJ, USA, 2012.

[3] AMD. *Secure Virtual Machine Architecture Reference Manual*.

[4] G. Ammons, D. D. Silva, O. Krieger, D. Grove, B. Rosenburg, R. W. Wisniewski, M. Butrico, K. Kawachiya, and E. V. Hensbergen. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 13–15, 2007.

[5] A. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Apr. 1991.

[6] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, 1995.

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–35, 2008.

[9] A. Bittau. *Toward Least-Privilege Isolation for Software*. PhD thesis, 2009.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, 2008.

[11] H. Boehm. GC Bench. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/`, Apr. 2012.

[12] H. Boehm, A. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 157–164, 1991.

[13] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, pages 9–9, 2010.

[14] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 143–156, 1997.

[15] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, 2005.

[16] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging Legacy Code to Deploy Desktop Applications on the Web. In *Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation*, OSDI'08, pages 339–354, 2008.

[17] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, Orcas Island, Washington, May 1995.

[18] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995.

[19] B. Ford and R. Cox. Vx32: Lightweight User-Level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference*, ATC'08, pages 293–306, 2008.

[20] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, 1996.

[21] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[22] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[23] Google. sparsehash. `http://code.google.com/p/sparsehash/`, Apr. 2012.

[24] IBM. *Power ISA, Version 2.06 Revision B*.

[25] Intel. *Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i)*.

[26] Intel Corporation. Invalid Instruction Erratum Overview. `http://www.intel.com/support/processors/pentium/sb/cs-013151.htm`, Apr. 2012.

[27] K. Kaspersky and A. Chang. Remote Code Execution thorugh Intel CPU Bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*.

[28] H. Kermany and E. Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, 2006.

[29] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 314–327, 2006.

[30] A. Kivity. KVM: the Linux Virtual Machine Monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[31] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 Interpretive-Execution Architecture, Foundation for VM/ESA. *IBM Syst. J.*, 30(1):34–51, Feb. 1991.

[32] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[33] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, SSYM'03, 2003.

[34] M. Sweet. Mini-XML: Lightweight XML Library. `http://www.minixml.org/`, Apr. 2012.

[35] S. Tang, H. Mai, and S. T. King. Trust and Protection in the Illinois Browser Operating System. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[36] G. Tene, B. Iyengar, and M. Wolf. C4: the Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, 2011.

[37] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48 – 56, May 2005.

[38] P. Varanasi and G. Heiser. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, 2011.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, 2009.

[40] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 13–24, 1987.