University of Glasgow | School of Computing Science

Virtualisation and wrap-up

Advanced Operating Systems
Tutorial 10

Tutorial Outline

- Review of lectured material
- Key points
- Discussion
- Wrap-up

Review of Lectured Material

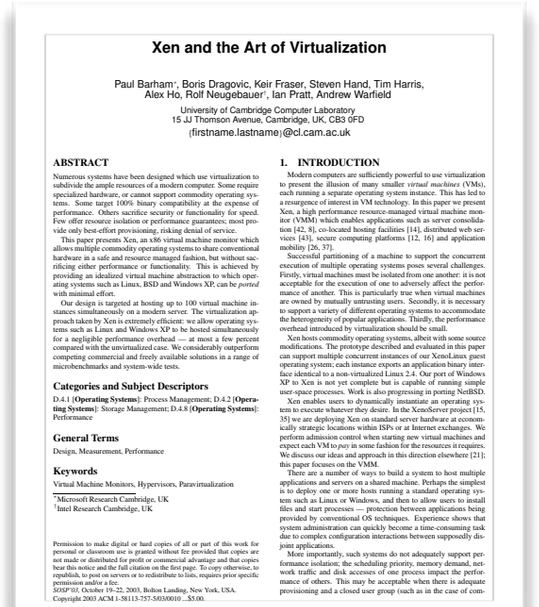
- Virtualisation concepts
- Full system virtualisation
  - Complete- vs. para-virtualisation
  - Types of hypervisor
  - Hardware features required to enable complete virtualisation
  - Xen
- Isolation of services
  - FreeBSD jails, Linux containers, etc.
  - Benefits of isolation and virtualisation within an operating system
  - Limitations and disadvantages

Key Points

- Flexibility/management/performance trade-off
- Application of full system virtualisation to hosting and cloud computing environments
- Benefits of process-level isolation and imperfect virtualisation

Discussion

- P. Barham *et al*, “Xen and the art of virtualization”, Proc. ACM Symposium on Operating Systems Principles, Oct. 2003. DOI:10.1145/945445.945462
  - Trade-offs of paravirtualisation vs. full virtualisation?
  - What needs to be done to port an OS to Xen?
  - Is paravirtualisation worthwhile, compared to full system virtualisation?
  - Is performance reasonable?



Xen and the Art of Virtualization

Paul Barham[1], Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer[2], Ian Pratt, Andrew Warfield

University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK, CB3 0FD
{firstname.lastname}@cl.cam.ac.uk

ABSTRACT

Numerous systems have been designed which use virtualization to subdivide the ample resources of a modern computer. Some require specialized hardware, or cannot support commodity operating systems. Some target 100% binary compatibility at the expense of performance. Others sacrifice security or functionality for speed. Few offer resource isolation or performance guarantees; most provide only best-effort provisioning, risking denial of service.

This paper presents Xen, an x86 virtual machine monitor which allows multiple commodity operating systems to share conventional hardware in a safe and resource managed fashion, but without sacrificing either performance or functionality. This is achieved by providing an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP, can be *ported* with minimal effort.

Our design is targeted at hosting up to 100 virtual machine instances simultaneously on a modern server. The virtualization approach taken by Xen is extremely efficient: we allow operating systems such as Linux and Windows XP to be hosted simultaneously for a negligible performance overhead — at most a few percent compared with the unvirtualized case. We considerably outperform competing commercial and freely available solutions in a range of microbenchmarks and system-wide tests.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Measurement, Performance

Keywords

Virtual Machine Monitors, Hypervisors, Paravirtualization

[1]Microsoft Research Cambridge, UK
[2]Intel Research Cambridge, UK

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

1. INTRODUCTION

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller *virtual machines* (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation [42, 8], co-located hosting facilities [14], distributed web services [43], secure computing platforms [12, 16] and application mobility [26, 37].

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our XenoLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.4. Our port of Windows XP to Xen is not yet complete but is capable of running simple user-space processes. Work is also progressing in porting NetBSD.

Xen enables users to dynamically instantiate an operating system to execute whatever they desire. In the XenoServer project [15, 35] we are deploying Xen on standard server hardware at economically strategic locations within ISPs or at Internet exchanges. We perform admission control when starting new virtual machines and expect each VM to *pay* in some fashion for the resources it requires. We discuss our ideas and approach in this direction elsewhere [21]; this paper focuses on the VMM.

There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes — protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of com-

- P.-H. Kamp & R. Watson, “Jails: Confining the omnipotent root”, Proc. System Administration & Network Engineering Conference, May 2000. http://www.sane.nl/events/sane2000/papers/kamp.pdf
  - Trade-offs vs. complete system virtualisation?
  - Overheads vs. flexibility vs. ease of management?
  - Jails vs. Docker vs. …?



Jails: Confining the omnipotent root.

Poul-Henning Kamp <phk@FreeBSD.org>
Robert N. M. Watson <rwatson@FreeBSD.org>
The FreeBSD Project

ABSTRACT

The traditional UNIX security model is simple but inexpressive. Adding fine-grained access control improves the expressiveness, but often dramatically increases both the cost of system management and implementation complexity. In environments with a more complex management model, with delegation of some management functions to parties under varying degrees of trust, the base UNIX model and most natural extensions are inappropriate at best. Where multiple mutually untrusting parties are introduced, ‘‘inappropriate’’ rapidly transitions to ‘‘nightmarish’’, especially with regards to data integrity and privacy protection.

The FreeBSD ‘‘Jail’’ facility provides the ability to partition the operating system environment, while maintaining the simplicity of the UNIX ‘‘root’’ model. In Jail, users with privilege find that the scope of their requests is limited to the jail, allowing system administrators to delegate management capabilities for each virtual machine environment. Creating virtual machines in this manner has many potential uses; the most popular thus far has been for providing virtual machine services in Internet Service Provider environments.

1. Introduction

The UNIX access control mechanism is designed for an environment with two types of users: those with, and without administrative privilege. Within this framework, every attempt is made to provide an open system, allowing easy sharing of files and inter-process communication. As a member of the UNIX family, FreeBSD inherits these security properties. Users of FreeBSD in non-traditional UNIX environments must balance their need for strong application support, high network performance and functionality,

This work was sponsored by `http://www.servetheweb.com/` and donated to the FreeBSD Project for inclusion in the FreeBSD OS. FreeBSD 4.0-RELEASE was the first release including this code. Follow-on work was sponsored by Safeport Network Services, `http://www.safeport.com/`

- Is there a place for both types of system?

Wrap-up

- Real-time scheduling of periodic, aperiodic, and sporadic tasks
- Resource management
- Programming real-time systems
- Garbage collection
- Message passing, transactions, and region-based memory management
- General purpose GPU programming

Real-time Scheduling of Periodic Tasks

- Introduction and system model
  - Tasks, jobs, processors, resources
  - Timing constraints and scheduling algorithms
  - Periodic, aperiodic, and sporadic tasks
  - Hard and soft real-time systems
- Scheduling periodic tasks
  - Types of scheduler: clock-driven vs. priority-driven
  - Scheduling algorithms; approaches to proving correctness
  - Rate monotonic: non-optimality, time-demand analysis & critical instants, simply periodic systems, maximum utilisation tests
  - Earliest deadline first: optimality, maximum utilisation test, density test
  - Choice of rate monotonic vs. earliest deadline first
  - Other algorithms: deadline monotonic and least slack time

Scheduling Aperiodic and Sporadic Tasks

- Aperiodic and sporadic tasks; acceptance tests
- Scheduling aperiodic jobs
  - Background execution
  - Periodic servers: polling, deferrable, and sporadic
    - Critical instant analysis for fixed-priority deferrable server; maximum utilisation test for deferrable server in EDF systems
    - Sporadic server budget consumption/replenishment; proofs of correctness
- Scheduling sporadic jobs
  - Acceptance test in EDF systems: density of intervals
  - Acceptance test in rate monotonic systems: maximum usage over periods
- Implementation choices

Resource Management

- Resource management protocols
  - Priority inheritance protocol – simple, but transitive blocking and potential deadlock
  - Priority ceiling protocol – reduced blocking and no transitive blocking, but requires a-priori knowledge of resource usage; must track system priority ceiling; avoidance blocking prevents deadlock
  - Stack-based priority ceiling protocol – further reduction in blocking if jobs never self-suspend; blocks jobs from starting until resources available
  - Maximum duration of blocking; operation in dynamic priority systems

Programming Real-time Systems

- Real-time and embedded systems programming
  - Ensuring predictable timing
  - Device drivers – hardware interactions; options for improving robustness
  - System longevity; desire to improve robustness through alternate system implementation techniques

Garbage Collection

- Automatic memory management
  - Stack allocation
- Reference counting
  - Simple, incremental, problems with cycles
- Garbage collection
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational collectors
  - Real-time collectors
- Practical factors

Message Passing

- Implications of multicore systems
  - Hardware trends; NUMA and heterogeneity in multicore systems
  - Challenges of NUMA systems – is a shared memory model appropriate?
  - Multi-kernel systems – distributed operating systems for multicore
- Message passing systems
  - Limitations of threads and lock-based concurrency
  - Multicore memory models; composition of lock-based code
  - Concepts of message passing systems
    - Interaction models; communication and the type system; naming communications
    - Message handling; immutability; linear types; use of an exchange heap
    - Pattern matching and state machines
    - Error handling; let-it-crash philosophy; supervision hierarchies; case study
  - Erlang and Scala+Akka as examples

Transactions

- Concepts of transactions
  - ACID properties
  - Concurrent execution
  - Possible to compose transactions
- Implementation challenges
  - Controlling I/O operations
  - Controlling memory access – rollback and recovery
  - Implementation using monadic concepts
- Integration into Haskell
- Integration challenges for other languages

Region-based Memory Management

- Relation to stack-based management of local variables
- Ownership and tracking changes to ownership
- Borrow – shared references to immutable data vs. unique references to mutable data
- Safety guarantees
- Limitations – cyclic data structures and shared ownership

General Purpose GPU Programming

- Heterogeneous instruction set systems
- Heterogeneous multi-kernel systems – Helios
- Main core with heterogenous offload
  - Graphics offload hardware – GPGPU
  - Programming model
  - OpenCL
  - Integration with operating systems
- Heterogenous virtual machines – Hera JVM
- Hybrid models – Accelerator
  - Lazy encoding of SIMD-style operations and JIT compilation into type system

Discussion

- Wide spectrum of research ideas and concepts
- Which are seeing widespread use?
  - Functional languages and message passing concurrency
  - Garbage collection – potential for integration with kernels
  - Increased use of static code analysis tools, to debug the limitations of C
- Opportunities for dependable kernels
  - New implementation frameworks and safe programming languages
  - Approaches similar to Singularity have large potential

Examination

- Final examination:
  - Worth 80% of marks for the course
  - 2 hours; answer 3-out-of-4 questions
  - Sample exam and past papers available on Moodle, and on the website
- All material covered in the lectures, tutorials, and papers is examinable
  - Aim is to test your understanding of the material, not simply to test your memory of all the details – in particular, *read papers to understand the concepts*, not details
  - Explain why, don't just recite what – are looking for your reasoned and justified technical opinion about the material

The End

https://csperkins.org/teaching/2015-2016/adv-os/