Software Engineering at VMware

Dan Scales
May 2008
The Challenge

• Suppose that you have a very popular software platform:
  • … that includes hardware-level and OS code that can easily crash a machine
  • … that must run on many different hardware platforms
  • … that big corporations use all the time and must be very reliable
  • … that customers really like and constantly want new features for

• How do you do very rapid development of large new features and yet produce high-quality releases
  • Requires concurrent development via many developers
  • But can you really do high-quality releases with such concurrent development?
The Challenge (cont.)

- Big part of the solution:
  - Good developers
  - Coding practices that allow lots of testing while code is running
  - Reliable (and fast!) tools for building, testing, and merging code
  - Constant automated QA (testing)
Outline

• Background on VMware
• Scale of development and releases at VMware, and some of our problems
  • Rapid concurrent development
  • Rapidly evolving interfaces
  • Many product releases sharing same code
  • Many hardware platforms
• Some software techniques for keeping OS reliable
• Some more automated techniques for keeping code base robust and ensuring quality of releases
What is Virtualization?

• VMware provides hardware virtualization that presents a complete x86 platform to the virtual machine
• Allows multiple applications to run in isolation within virtual machines on the same physical machine
• Virtualization provides direct access to the hardware resources to give you much greater performance than software emulation
Why are virtual machines useful

• Run Windows on Linux (or Mac OS)
• Consolidate many different workloads/guest OSes on one big machine ("server consolidation")
• We can migrate running virtual machines (VMs) run one host to another, so we can do automatic workload balancing across a cluster of machines
• VMs also simplify high availability, disaster recovery, etc.
VMware Virtualization Architectures

VMware Workstation

Hosted design
- Runs on Windows, Linux, MacOS
- Device support is inherited from host operating system
- Virtualization installs like an application rather than like an operating system

VMware ESX Server

Hypervisor
- Virtualization supported via small kernel (VMkernel)
- Highly efficient direct I/O pass-through architecture for network and disk
- Excellent management/scheduling of hardware resources
• We touch every level of the software stack – drivers, CPU code, OS code, user-level code, UI, management agents, management applications
• We keep adding features that can span the entire stack
• Strong interfaces – management API and DDK
• Most intertwined code – VMX, VMM, and vmkernel
Customers constantly want new features

- 64-bit support
- 10-Gigabit ethernet
- Support for new features in Intel/AMD chips (including virtualization support)
- NUMA support
- Storage migration
- Network boot
- …..
OUR SERVERS ARE USING TOO MUCH ELECTRICITY. WE NEED TO VIRTUALIZE.

I DID MY PART BY READING ABOUT VIRTUALIZATION IN A TRADE JOURNAL. NOW YOU DO THE SOFTWARE PART.

WHY IS YOUR PART TAKING SO LONG?

© Scott Adams, Inc./Dist. by UFS, Inc.
Outline

• Background on VMware

• Scale of development and releases at VMware, and some of our problems
  • Rapid concurrent development
  • Rapidly evolving interfaces
  • Many product releases sharing same code
  • Many hardware platforms

• Some software techniques for keeping OS reliable

• Some more automated techniques for keeping code base robust and ensuring quality of releases
Goal

- Goal: fast development of new features and support for new platforms while maintaining performance and reliability
- Problems:
  - Highly concurrent development
  - Rapidly evolving interfaces
  - Frequent and concurrent releases
  - Many hardware platforms
Highly concurrent development

- Biggest areas of inter-dependence: vmkernel, VMX, and VMM
  - Largely written in C
- Recently, for the vmkernel (hypervisor):
  - 15-19 checkins per day
  - 150 developers
- For the vmkernel, vmx, and vmm:
  - 36 checkins per day
  - 230 developers
- Need to minimize problems with checkins, so one checkin with issues doesn’t affect lots of developers
Many releases

- VMware Workstation, VMware Server, and ESX Server that are released independently
- All share common code (VMX and VMM)
- Many different platforms:
  - Windows
  - Linux
  - MacOS
  - VMkernel
    - variety of server hardware
    - Variety of storage hardware (SCSI, FibreChannel, iSCSI, NFS) and network hardware (1 Gb Ethernet, 10 Gb Ethernet, TCP offload, …)
  - 32-bit vs. 64-bit
Concurrent Development and Releases

• You need to integrate a variety of features and bug fixes into the code base (**Main**)

• You need to create stable snapshots of the source code for product release
  • Typically, you create branches of the code for product releases, so development for future releases can continue on **Main**
  • Code will have to be “cross-ported” from product branch to **Main**
Outline

• Background on VMware
• Scale of development and releases at VMware, and some of our problems
  • Rapid concurrent development
  • Rapidly evolving interfaces
  • Many product releases sharing same code
  • Many hardware platforms
• Some software techniques for keeping OS reliable
• Some more automated techniques for keeping code base robust and ensuring quality of releases
Software development for vmkernel

• Mostly written in C
• Includes base kernel, ~40 drivers, ~25 other modules
• Modules do things such as:
  • Virtual switch
  • Storage multipathing
  • VM migration
  • Distributed file system (VMFS)

• Important to continually verify that new features/sub-systems will work together
Static Checking

• Maintain the strictest possible static checking, so we check “easy” bugs as soon as possible
  • Important to do from the start, else it becomes hard to fix later
  • E.g. all those warning messages that come from linux drivers….

• We use the strictest type checking options from the compiler
  • Sometimes use C++ compiler, since it does stricter checking
  • Catch 64-bit VMotion bug

• Coverity is very useful
Verify/test code while running

- Make it easy to enable/disable the verification code
  - Development and release builds, all extra code disappears in release builds

- Assertions are crucial
  - Simple and very general assertions are invaluable – e.g. don’t ever block while you have a spin lock or in ISR
  - Document and verify invariants at beginning of functions – e.g. the scheduling lock must be held
  - Allows a module/function to protect against a drastic change in its usage
  - Catch compiler and processor bugs!
    - Page table change (CR3) doesn’t immediately take effect
Verify/test code while running (cont.)

• Lock ranking
  • Establish global order for all locks, locks must be acquired in that order to guarantee no deadlock
  • E.g. buffer cache lock must always be acquired before file system lock
  • Extra benefit – discourages modules from creating too many locks

• Stress options
  • Used for causing unusual, but recoverable conditions
    
    if (STRESS_OPTION(disk_error)) return DISK_FAILURE;

  • Enabled in devel builds, returns TRUE in (say) 1 out of 1000 calls
  • Allows testing of cases that would be very hard for QA to reproduce
Verify/test code while running (cont.)

- POST (Power-on self test)
  - Tests that run when module is initialized
  - Hard to get developers to write them
- Heap memory poisoning
• Make it more likely that one module can’t affect another

• Resource usage – verify that a bottom-half or kernel thread doesn’t run for more than reasonable limit
  • Can lead to very mysterious networking performance…. (serial port bug)

• Separate heap for each module
  • So memory bug with one module is less likely to affect another
• Verify DMA is to appropriate memory
  • DMA goes around MMU, can write arbitrary memory
  • IO MMUs are coming

• Use address-space protection
  • Add extra address space protections in devel builds
  • Separate address space for drivers (Nooks work)
  • Move code out to user space
• Coding conventions
• Get error codes right from the start
  • Don’t use 0 and –1!
  • Even Linux error codes don’t carry enough information
  • Want more general error codes that carry arbitrary other information
  • Probably want requirement to never ignore error codes

• Interestingly, many of these kernel issues apply to other multi-threaded, modular applications:
  • Hostd – multi-threaded management daemon on ESX host
  • Web-servers
  • App servers
Outline

• Background on VMware
• Scale of development and releases at VMware, and some of our problems
  • Rapid concurrent development
  • Rapidly evolving interfaces
  • Many product releases sharing same code
  • Many hardware platforms
• Some software techniques for keeping OS reliable
• Some more automated techniques for keeping code base robust and ensuring quality of releases
Large-scale Concurrent Development

• How do you keep source code base (Main) stable for developers and product releases?

→

• Make it easy to test/verify code while running
• Good developers who are careful and follow responsibilities
  • Code reviews
  • Pre-checkin build
  • Pre-checkin testing
• Policy for determining how code is integrated into Main
Concurrent development

- Modular code with well-defined interfaces
  - develop in separate branches and merge
- Most changes span modules and interfaces always changing
  - free-for-all!
- Most changes confined to modules, but interfaces change – “component branching”
  - More stable than free-for-all
  - Requires sync back-and-forth

- Some features are huge and touch every part of the system:
  - major features developed on “feature branch”
What to use?

• Feature branches are painful
  • Because you must always merge with changes from other developers until you are ready to check in
  • But useful for really big features that can be incrementally added

• VMware uses component branching
  • Requires merging between the component branches
  • Most other OSes seem to do this as well (Microsoft, Linux, have hierarchy of branches with human mergers)
  • Big changes may use feature branch

• Component branching requires frequent sync’ing between branches after really good testing
Fast automation is key!

• We need to be able to do automated testing
  • We have many levels of automatic tests: pre-checkin, nightly tests, longer tests before merging between branches…..

• We need automatic merging of code from one branch to another
  • Alternatively, have a “gatekeeper” that manually does merge, verifies functionality – e.g. Linux integration trees

• For all this, we need a very fast build system
  • So developers will build on all possible platforms before checking in
  • So we can verify checkins to branches, merges to other branches very quick
  • So we can quickly check if checked-in code compiles on all platforms
Concurrent Development

• Problem: rapid concurrent development of code with rapidly changing interfaces

• Solutions:
  • Developer responsibilities, including pre-checkin tests
  • Component branching
  • Automated build of tree after checkin
  • Automated test of build after checkin
  • Should we have automatic backout if build or tests fail? Must be quick to isolate exact change that caused problem….
  • Continuous manual and automated QA to find bugs early….
Concurrent releases

- Problem: releases of many products on many platforms
- Solutions
  - Build all products on all branches in order to catch problems early
  - Create “product branches” when nearing release of a product
Concurrent Releases (cont.)

- Product branches automatically merge changes to main development tree
  - Need to do continuous builds on main tree to check for problem because of automerges
  - Need to remind developers if automerges fail
  - Should really do merging via human

![Diagram showing main, feature branches, and releases over time with automerges and bug-fixes](image-url)
Conclusion

• Rapid development of new features on a large code base that is changing at all layers is hard!

• Solutions:
  • Good developers with good development practices
  • Coding practices that allow lots of testing while code is running
  • Reliable (and fast!) tools for building, testing, and merging code
  • Continuous testing (manual and automated)

• Developers are important, but a good build/tools team is also crucial