Virtual Machines

"A virtual machine (VM) is an efficient, isolated duplicate of a real machine"

- Duplicate: VM should behave identically to the real machine
  - Programs cannot distinguish between execution on real or virtual hardware
  - Except for:
    - Fewer resources available (and potentially different between executions)
    - Some timing differences (when dealing with devices)
- Isolated: Several VMs execute without interfering with each other
- Efficient: VM should execute at a speed close to that of real hardware
  - Requires that most instruction are executed directly by real hardware

Virtual Machines, Simulators and Emulators

Simulator
- Provides a functionally accurate software model of a machine
  - May run on any hardware
- Is typically slow (order of 1000 slowdown)

Emulator
- Provides a behavioural model of hardware (and possibly S/W)
  - Not fully accurate
  - Reasonably fast (order of 10 slowdown)

Virtual machine
- Models a machine exactly and efficiently
  - Minimal slowdown
  - Needs to be run on the physical machine it virtualizes (more or less)

Types of Virtual Machines

- Contemporary use of the term VM is more general
- Call virtual machines even if there is nor correspondence to an existing real machine
  - E.g. Java virtual machine
  - Can be viewed as virtualizing at the ABI level
  - Also called process VM
- We only concern ourselves with virtualizing at the ISA level
  - ISA = instruction-set architecture (hardware-software interface)
  - Also called system VM
  - Will later see subclasses of this

Virtual Machine Monitor (VMM), aka Hypervisor

- Program that runs on real hardware to implement the virtual machine
- Controls resources
  - Partitions hardware
  - Schedules guests
  - Mediates access to shared resources
  - E.g. console
  - Performs world switch
- Implications:
  - Hypervisor executes in privileged mode
  - Guest software executes in unprivileged mode
  - Privileged instructions in guest cause a trap into hypervisor
  - Hypervisor interprets and emulates them
  - Can have extra instructions for hypervisors
Why Virtual Machines?

- Historically used for easier sharing of expensive mainframes
  - Run several (even different) OIs on same machine
  - Each on a subset of physical resources
  - Legacy support
  - "World switch" between VMs
- Gone out of fashion in 80's
  - Time-sharing OIs common-place
  - Hardware too cheap to worry...

Renaissance in recent years for improved isolation
- Server/desktop virtual machines
  - Improved QoS and security
  - Uniform view of hardware
  - Complete encapsulation
    - Replication
    - Migration
    - Checkpointing
    - Debugging
  - Different concurrent OIs
    - e.g. Linux and Windows
  - Total mediation
  - Would be mostly unnecessary
    - if OIs were doing their job...

Native vs. Hosted VMM

Native/Classic/Bare-metal/Type-I
- Hosted VMM can run besides native apps
  - Sandbox untrusted apps
  - Run second OS
  - Less efficient:
    - Guest privileged instruction traps into OS, forwarded to hypervisor
    - Return to guest requires a native OS system call
  - Return to guest requires a native OS system call
  - Convenient for running alternative OS environment on desktop

Hosted/Type-II
- Hosted VMs can run besides native apps

VMM Types

Classic: as above
- e.g. VMware Player/Fusion

Hosted: run on top of another operating system
- e.g. Virtual PC (for Macintosh)

Whole-system: Virtual hardware and operating system
- Ready an emulation
  - E.g. Virtual PC (for Macintosh)

Physically partitioned: allocate actual processors to each VM
- Logically partitioned: time-share processors between VMs

Co-designed: hardware specifically designed for VMM
- E.g. Transmeta Crusoe, IBM i-Series

Pseudo: no enforcement of partitioning
- Guests at same privilege level as hypervisor
- Really abuse of term "virtualization"
- E.g. products with "optional isolation"

Virtualization Mechanics

- Traditional "trap and emulate" approach:
  - Guest attempts to access physical resource
  - Hardware raises exception (trap), invoking hypervisor’s exception handler
  - Hypervisor emulates result, based on access to virtual resource
- Most instructions do not trap
  - Makes efficient virtualization possible
  - Requires that VM ISA is (almost) same as physical processor ISA

Virtualization Mechanics: Address Translation
Impure Virtualization

- Used for two reasons:
  - unvirtualizable architectures
  - performance problems of virtualization
- Change the guest OS, replacing sensitive instructions
  - by trapping code (hypervisor)
  - by in-line emulation code
- Two standard approaches:
  - binary translation: modifies binary
  - para-virtualization: changes ISA

Virtualization Overheads

- VMM needs to maintain virtualized privileged machine state
  - processor status
  - addressing context
  - device state
- VMM needs to emulate privileged instructions
  - translate between virtual and real privileged state
  - e.g. guest -- real page tables
  - Virtualization traps are be expensive on modern hardware
    - can be 100s of cycles (x86)
    - Some OS operations involve frequent traps
      - STI/CLI for mutual exclusion
        - frequent page table updates during fork()
    - MIPS KSEG address used for physical addressing in kernel

Unvirtualizable Architectures

- x86: lots of unvirtualizable features
  - e.g. sensitive PUSH of PSW is not privileged
  - segment and interrupt descriptor tables in virtual memory
    - segment description expose privileged level
  - Itanium: mostly virtualizable, but
    - Interrupt vector tables in virtual memory
    - THASH instruction exposes hardware page tables address
  - MIPS: mostly virtualizable, but
    - THASH instruction exposes hardware page tables address
    - performance issue with virtualizing KSEG addresses
  - ARM: mostly virtualizable, but
    - some instructions undefined in user mode (banked registers, CPSR)
    - PC is a GPR, exception return in MOVS to PC, doesn’t trap
  - Most others have problems too
  - Recent architecture extensions provide virtualization support hacks

Trap-and-Emulate Requirements

- An architecture is virtualizable if all sensitive instructions are privileged
- Can then achieve accurate, efficient guest execution
  - by simply running guest binary on hypervisor
  - VMM controls resources
  - Virtualized execution is indistinguishable from native, except:
    - Resources more limited (running on smaller machine)
    - Timing is different (if there is an observable time source)
  - Recursively virtualizable machine:
    - VMM can be built without any timing dependence

Definitions:

- Privileged instruction: executes in privileged mode, traps in user mode
  - Note: trap is required, NO-OP is insufficient!
- Privileged state: determines resource allocation
  - Includes privilege mode, addressing context, exception vectors, ...
- Sensitive instruction: control-sensitive or behaviour-sensitive
  - control sensitive: changes privileged state
  - behaviour sensitive: exposes privileged state
    - includes instructions which are NO-OPs in user but not privileged mode
- Innocuous instruction: not sensitive

Note:

- Some instructions are inherently sensitive
  - e.g. TLB load
- Others are sensitive in some context
  - e.g. store to page table

Virtualization

- XEN: para-virtualization, uses in-line code
  - binary translation: modifies binary
    - inline emulation code
  - para-virtualization: changes ISA
- paravirtualization: changes ISA
- Virtualization: changes ISA
- Virtualization: changes ISA
- para-virtualization: changes ISA

Binary Translation

- Locate sensitive instructions in guest binary and replace on-the-fly by
  - emulation code or hypercall
  - pioneered by VMware
  - can also detect combinations of sensitive instructions and replace by simple
    - emulation doesn’t require source, uses unmodified native binary
      - in this respect appears like pure virtualization!
      - very tricky to get right (especially on x86)
  - needs to make some assumptions on same behaviour of guest

Requirements for Virtualization

- Note:
  - Some instructions are inherently sensitive
  - Others are sensitive in some context
  - Includes instructions which are NO-OPs in user but not privileged mode
- Innocuous instruction: not sensitive
- Sensitive instruction: control-sensitive or behaviour-sensitive
  - control sensitive: changes privileged state
  - behaviour sensitive: exposes privileged state
    - includes instructions which are NO-OPs in user but not privileged mode
- Privileged instruction: executes in privileged mode, traps in user mode
  - Note: trap is required, NO-OP is insufficient!
Virtualization Techniques

- Impure virtualization methods enable new optimisations
  - due to the ability to control the ISA
  - E.g. maintain some virtual machine state inside VMM:
    - e.g. interrupt-enable bit (in virtual PSR)
    - guest can update without (expensive) hypervisor invocation
    - requires changing guest’s idea of where this bit lies
    - hypervisor knows about VMM-local virtual state and can act accordingly
      - e.g. queue virtual interrupt until guest enables in virtual PSR

- Para-virtualization:
  - Idea: manually port the guest OS to modified ISA
    - Use more high-level API to reduce the number of traps
    - Remove un-virtualizable instructions
  - Drawbacks:
    - Significant engineering effort
    - Needs to be repeated for each guest-ISA-hypervisor combination
    - Requires source

- Full virtualization of device:
  - Hypervisor implements device-sharing policies
    - Hypervisor contains real device driver
    - Native guest driver accesses device as usual
    - implies virtualizing all device registers etc.
    - trap on every device access
    - Virtualizes at device interface
    - Hypervisor implements device-sharing policies
    - Drawbacks:
      - must re-implement/port all drivers in hypervisor
        - unfeasible for contemporary hardware
        - very expensive (frequent traps)
      - will not work on most devices
        - timing constraints violated

Page table implementation options

- Strict shadowing of virtual page table
  - write protect PTs ⇒ force trap into hypervisor on each update
  - can combine multiple updates in single hypervisor
    - e.g. during fork()

- Lazy shadowing of virtual page table
  - identify synchronisation points
  - possible due to TLB semantics
    - real PT updates only become effective once loaded into TLB
  - explicit TLB loads and flushes are natural synchronisation points
  - PTs are big ⇒ need to tell hypervisor which part to sync
  - guest must deal with PT reads differing from what was written

Device Virtualization Techniques

- Memory reclamation using ballooning
  - load pseudo device driver into guest, colludes with hypervisor
  - to reclaim memory, hypervisor instructs driver to request memory
  - hypervisor can re-use memory hoarded by ballooning driver
  - guest controls which memory it gives up
### Device Virtualization Techniques

**Virtual device drivers**
- Guest OS contains virtual drivers
- forwards guest I/O requests to real driver via hypercalls
- very simple driver
- Need only support small number of different virtual devices
  - e.g. one type of virtual NIC, one type of virtual disk
- Virtualizes at driver interface
- Hypervisor implements device-sharing policies
- **Drawbacks:**
  - must re-implement/port all drivers in hypervisor
  - unfeasible for contemporary hardware
  - ... unless a complete OS becomes the hypervisor (KVM)

**Device Virtualization Techniques**

**Device-driver OS**
- Special guest OS contains real drivers
  - XEN "Own guest"
- Hypervisor passes requests from virtual driver through to driver OS
- Can re-use driver guest's native drivers unchanged
- **Drawbacks:**
  - driver invocation requires full context switch
  - driver OS = all drivers become part of VMM
  - very large TCB
- Can improve TCB by running each driver in its own guest OS instance
  - full encapsulation of drivers [LeVasseur et al 04]

**Advantage:**
- actual binary is
- Hypervisor not involved in I/O
- In general insecure and thus infeasible
- **Possible for:**
  - simple devices not doing DMA
  - load sharing is an issue
  - with hardware support
    - virtualization-friendly devices
      - e.g. IBM channel architecture
      - maps IO space to RAM
    - under control of hypervisor
      - e.g. Intel VT-d

**Drawback:**
- can expand code for virtualization
  - more space than virtualized instruction
  - jump (to virtualization code) may need
    - hook onto macros etc
    - no need to keep addresses invariant
    - can expand code for virtualization
  - can do much virtualization in-line
    - avoid branches to virtualization code
- **Disadvantage:** needs source (at least assembler output of compiler)

**Automated pre-virtualization**
- pre-virtualizes ("afterburn") code
  - Link in hypervisor-specific user-level VMM code ("wedge")
  - can be patched (at load time)
  - no-ops have very little performance effect (0.15%)
  - has almost properties of pure virtualization
  - except for much improved performance
  - pre-virtualization doesn’t have to be perfect
  - can run on bare hardware without any patches
  - no-ops have very little performance effect (0.15%)
  - has almost properties of pure virtualization
  - as long as the instruction traps
  - e.g. page table updates (PTs are write-protected)

**Native real driver in guest**
- Guest allowed to "own device"
- Hypervisor not involved in I/O
- In general insecure and thus infeasible

**Soft Layering aka Pre-Virtualization**
- Combines advantages of pure and para-virtualization [LeVasseur et al, 08]
- **Core idea:** Post-process ("afterburn") assembly code (compiler output)
  - prepares ("Pre-virtualization") code
  - more flexible than binary rewriting
  - use semantic info from compiler
  - replace instruction sequences by hypercalls
  - hook onto macros etc
  - no need to keep addresses invariant
    - jump (to virtualization code) may need
      - more space than virtualized instruction
      - linker will fix up address changes
  - can expand code for virtualization
  - can do much virtualization in-line
    - avoid branches to virtualization code
  - **Disadvantage:** needs source (at least assembler output of compiler)
Soft Layering aka Pre-Virtualization

- ARM TrustZone extensions introduce:
  - new processor mode: monitor
  - similar to VT-x root mode
  - banked registers (PC, LR)
  - can run unmodified guest OS binary in non-monitor
  - new privileged instruction: BMI enters monitor mode
  - new processor status: secure
  - partitioning of resources
    - memory and devices marked secure or insecure
    - in secure mode, processor has access to all resources
    - in insecure mode, processor has access to insecure resources only
    - monitor switches world (secure → insecure)
  - only supports one virtual machine (guest in insecure mode)
  - need another hypervisor and para-virtualization for multiple guests

Uses of Virtual Machines

- Multiple (identical) OSes on same platform
  - the original raison d'être
  - these days driven by server consolidation
  - interesting variants of this:
    - different OSes (Linux + Windows)
    - used for OS debugging (most likely uses Type-II VMs)
  - Checkpoint/restore
    - minimise lost work in case of crash
    - useful for debugging, incl. going backwards in time
  - Ship application with complete OS
    - dependency on environment
  - Java done right?
  - How about embedded systems?
### Why Virtualization for Embedded Systems?

#### Use case 1: Mobile phone processor consolidation
- **High-end phones run high-level OS (Linux) on app processor**
  - supports complex UI software
- **Base-band processing supported by real-time OS (RTOS)**
- **Medium-range phone needs less grunt**
  - can share processor
  - two VMs on one physical processor
  - hardware cost reduction

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>

#### Use case 1a: License separation
- **Linux desired for various reasons**
  - familiar, high-level API
  - large developer community
  - free
- **Other parts of system contain proprietary code**
- **Manufacturer doesn’t want to open-source**
- **User VM to contain Linux + GPL**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>

#### Use case 1b: Software-architecture abstraction
- **Support for product series**
  - range of related products of varying capabilities
- **Same low-level software for high- and medium-end devices**
- **Benefits:**
  - time-to-market
  - engineering cost

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>

#### Use case 1c: Dynamic processor allocation
- **Allocate share of base-band processor to application OS**
  - Provide extra CPU power during high-load periods (media play)
  - Better processor utilisation => higher performance with lower-end hardware
  - HW cost reduction

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>

#### Use case 2: Certification re-use
- **Phones need to be certified to comply with communication standards**
- **Any change that (potentially) affects comms needs re-certification**
- **UI part of system changes frequently**
- **Encapsulation of UI**
  - provided by VM
  - avoids need for costly re-certification

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>

#### Use case 2a: Open phone with user-configured OS
- **Give users control over the application environment**
  - perfect match for Linux
  - Requires strong encapsulation of application environment
  - without undermining performance!

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>

#### Use case 2b: Certification re-use
- **Phones need to be certified to comply with communication standards**
- **Any change that (potentially) affects comms needs re-certification**
- **UI part of system changes frequently**
- **Encapsulation of UI**
  - provided by VM
  - avoids need for costly re-certification

<table>
<thead>
<tr>
<th>Processor</th>
<th>Hypervisor</th>
<th>Linux</th>
<th>Baseband Software</th>
<th>RTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
<td>Processor</td>
</tr>
</tbody>
</table>
Why Virtualization for Embedded Systems?

Use case 2b: Phone with private and enterprise environment
- Work phone environment integrated with enterprise IT system
- Private phone environment contains sensitive personal data
- Mutual distrust between the environments: strong isolation needed

![Diagram showing processor, Linux, private phone environment, enterprise phone environment, hypervisor, Baseband Software, and RTOS.]

Why Virtualization for Embedded Systems?

Use case 2c: Security
- Protect against exploits
- Modern software attacked by UI exploits
  - Compromised application OS could compromise RT side
  - Could have serious consequences
    - e.g., jamming cellular network
- Virtualization protects
  - Separate apps and system code into different VMs

![Diagram showing UI SW, Attack, Hypervisor, OS, Core, Hypervisor, OS, Codec, Crypto, and Generalised Hypervisor.]

Why Virtualization for Embedded Systems?

Use case 3: Mobile internet device (MID) with enterprise app
- MID is open device, controlled by owner
- Enterprise app is closed and controlled by enterprise IT department
- Hypervisor provides isolation

![Diagram showing processor, Linux, Apps, Special-purpose OS, Hypervisor, and Critical code.]

Why Virtualization for Embedded Systems?

Use case 3a: Environment with minimal trusted computing base (TCB)
- Minimise exposure of highly security-critical service to other code
- Avoid even an OS, provide minimal trusted environment
  - need a minimal programming environment
  - goes beyond capabilities of normal hypervisor
  - requires basic OS functionality

![Diagram showing processor, Linux, Apps, Special-purpose OS, Hypervisor, and Critical code.]

Why Virtualization for Embedded Systems?

Use case 3b: Point-of-sale (POS) device
- May be stand-alone or integrated with other device (e.g., phone)
- Financial services providers require strong isolation
  - dedicated processor for PIN/key entry
  - use dedicated virtual processor to HW cost reduction

![Diagram showing processor, Linux, Apps, PIN entry, and Generalised Hypervisor.]

Why Virtualization for Embedded Systems?

Use case 4: DRM on open device
- Device runs Linux as app OS, uses Linux-based media player
- DRM must not rely on Linux
- Need trustworthy code that
  - loads media content into on-chip RAM
  - decrypts and decodes content
  - allows Linux-based player to display
  - Need to protect data from guest OS

![Diagram showing Apps, Linux, Codec, Crypto, Generalised Hypervisor, and Processor.]

Why Virtualization for Embedded Systems?
Why Virtualization for Embedded Systems?

Use case 4a: IP protection in set-top box
- STB runs Linux for UI, but also contains highly valuable IP
  - highly-efficient, proprietary compression algorithm
  - operates in hostile environment
    - reverse engineering of algorithms
    - loads code from Flash into on-chip RAM
    - decrypts code
    - runs code protected from interference
- Need highly-trustworthy code that loads code from Flash into on-chip RAM
  - decrypts code
  - runs code protected from interference

Hypervisor
Processor
Linux
Crypto, Secure loader
Decompression
Apps

Why Virtualization for Embedded Systems?

Use case 5: Automotive control and infotainment
- Trend to processor consolidation in automotive industry
  - top-end cars have > 100 CPUs!
  - cost, complexity and space pressures to reduce by an order of magnitude
  - AUTOSAR OS standard addressing this for control/convenience function
- Increasing importance of infotainment
  - not addressed by AUTOSAR
  - eg park-distance control using infotainment display
  - benefits from being located on same CPU

Hypervisor
Processor
Linux
AUTOSAR
Apps

Why Virtualization for Embedded Systems?

Future use case: multicore resource management (esp. power)
- Hypervisor is virtualization layer that allows turning off idle resources

Hypervisor
Processor
Guest 1
Guest 2
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU
Virtual CPU

Enterprise vs Embedded Systems VMs

Homogenous vs heterogeneous guests
- Enterprise: many similar guests
  - hypervisor size irrelevant
  - VMs scheduled round-robin
- Embedded: 1 HLOS + 1 RTOS
  - hypervisor resource-constrained
  - interrupt latencies matter

Hypervisor
Processor
Linux
Baseband Software
RTOS

Enterprise vs Embedded Systems VMs

Devices in enterprise-style virtual machines
- Hypervisor owns all devices
- Drivers in hypervisor
  - need to port all drivers
  - huge TCB
- Drivers in privileged guest OS
  - can leverage guest's driver support
  - need to trust driver OS
  - still huge TCB

Hypervisor
Driver
Trusted
Processor

Core Difference: Isolation vs Cooperation

Enterprise
- independent services
- Emphasis on isolation
- Inter-VM communication is secondary
- VMs connected to Internet (and thus to each other)

Embedded
- integrated system
- Cooperation with protection
- Inter-VM communication is critically important
- performance crucial
- VMs are subsystems accessing shared (but restricted) resources

Hypervisor
Driver
Driver
Driver
Processor

Isolation vs Cooperation: Scheduling

Enterprise
- Round-robin scheduling of VMs
- Guest OS schedules its apps

Embedded
- Global view of scheduling
- Schedule threads, not VMs

→ Similar for energy management:
  - energy is a global resource
  - optimal per-VM energy policies are not globally optimal

Enterprise vs Embedded Systems VMs

Devices in embedded virtual machines
- Some devices owned by particular VM
- Some devices shared
- Some devices too sensitive to trust any guest
- Driver OS too resource hungry
- Use isolated drivers
  - protected from other drivers
  - protected from guest O/Ss

Modern Inter-VM Communication Control

Modern embedded systems are multi-user devices!
- Eg a phone has three classes of “users”:
  - the network operator(s)
    - assets: cellular network
    - content providers
    - media content
  - the owner of the physical device
    - assets: private data, access keys
- They are mutually distrusting
  - need to protect integrity and confidentiality against internal exploits
  - need control over information flow
    - strict control over who has access to what
    - strict control over communication channels

Inter-VM Communication Control

Modern embedded systems are multi-user devices!
- Eg a phone has three classes of “users”:
  - the network operator(s)
    - assets: cellular network
    - content providers
    - media content
  - the owner of the physical device
    - assets: private data, access keys

Isolation vs Cooperation: Scheduling

Enterprise
- Round-robin scheduling of VMs
- Guest OS schedules its apps

Embedded
- Global view of scheduling
- Schedule threads, not VMs

→ Similar for energy management:
  - energy is a global resource
  - optimal per-VM energy policies are not globally optimal
Inter-VM Communication Control

- Different "users" are mutually distrusting
- Need strong protection / information-flow control between them
- Isolation boundaries # VM boundaries
  - some are much smaller than VMs
  - individual buffers, programs
  - some contain VMs
  - some overlap VMs
- Need to define information flow between isolation domains

![Diagram showing inter-VM communication control](image)

High Safety/Reliability Requirements

- Software complexity is mushrooming in embedded systems too
  - millions of lines of code
- Some have very high safety or reliability requirements
- Need divide-and-conquer approach to software reliability
  - Highly componentised systems to enable fault tolerance

![Diagram showing high safety/reliability requirements](image)

Componentisation for IP Blocks

- Match HW IP blocks with SW IP blocks
- HW IP owner provides matching SW blocks
  - encapsulate SW to ensure correct operation
  - Stable interfaces despite changing HW/SW boundary

![Diagram showing componentisation for IP blocks](image)

Componentization for Security — MILS

- MILS architecture: multiple independent levels of security
- Approach to making security verification of complex systems tractable
- Separation kernel provides strong security isolation between subsystems
- High-grade verification requires small components

![Diagram showing componentization for security — MILS](image)

Embedded Systems Requirements

- Sliding scale of isolation from individual program to VM running full-blown OS
  - isolation domains, information-flow control
- Global scheduling and power management
  - no strict VM-hypervisor hierarchy
  - increased hypervisor-guest interaction
- High degree of sharing is essential and performance-critical
  - high bandwidth, low latency communication, subject to security policies
- Real-time response
  - fast and predictable switches to device driver / RT stack
- High safety/security requirements
  - need to maintain minimal TCB
  - need to support componentized software architecture / MILS

Virtualization in embedded systems is good, but different from enterprise
- requires more than just a hypervisor, also needs general OS functionality
- perfect match for good microkernel, such as OKL4...