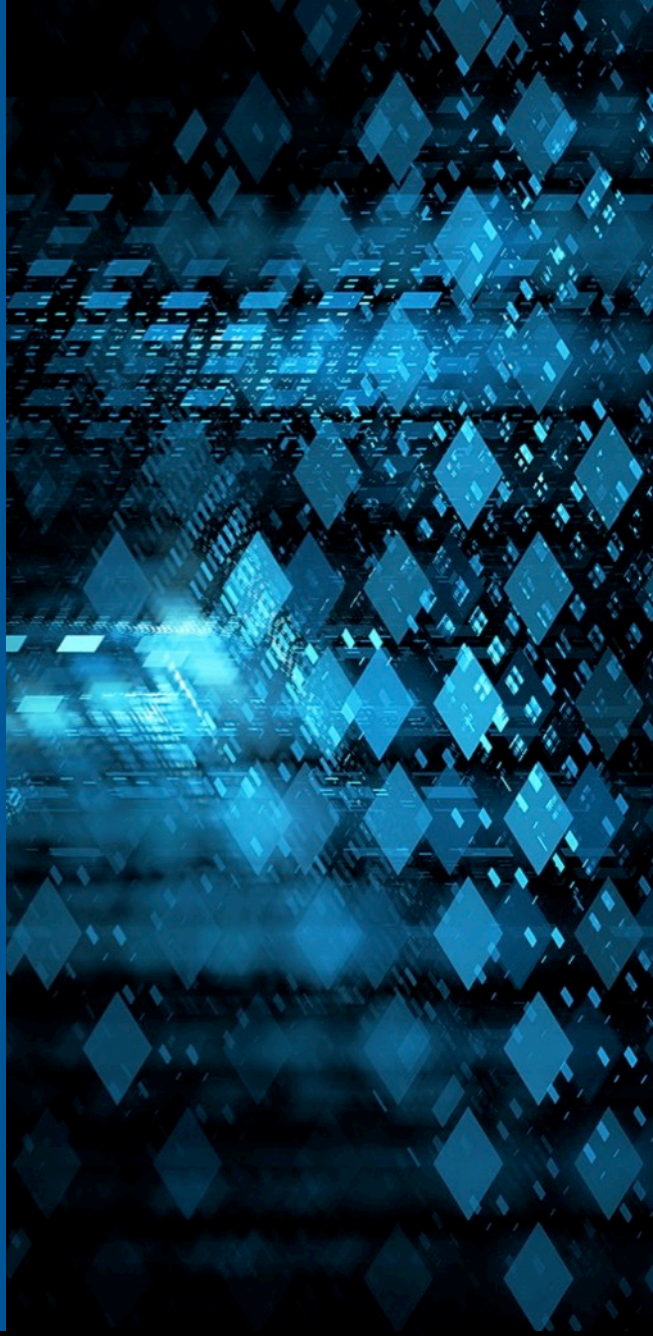
Multicore Processing, Virtualization, and Containerization:
Similarities, Differences, and Challenges

Donald Firesmith

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM18-0001

Topics

Big Picture Up Front (BPUF)

Multicore Processing (MCP)

- Definition
- Current Trends
- Pros and Cons
- Safety/Security Ramifications

Virtualization (V)

- Definition
- Current Trends
- Pros and Cons
- Safety/Security Ramifications

Containerization (C)

- Definition
- Current Trends
- Pros and Cons
- Safety/Security Ramifications

Recommendations

Conclusion

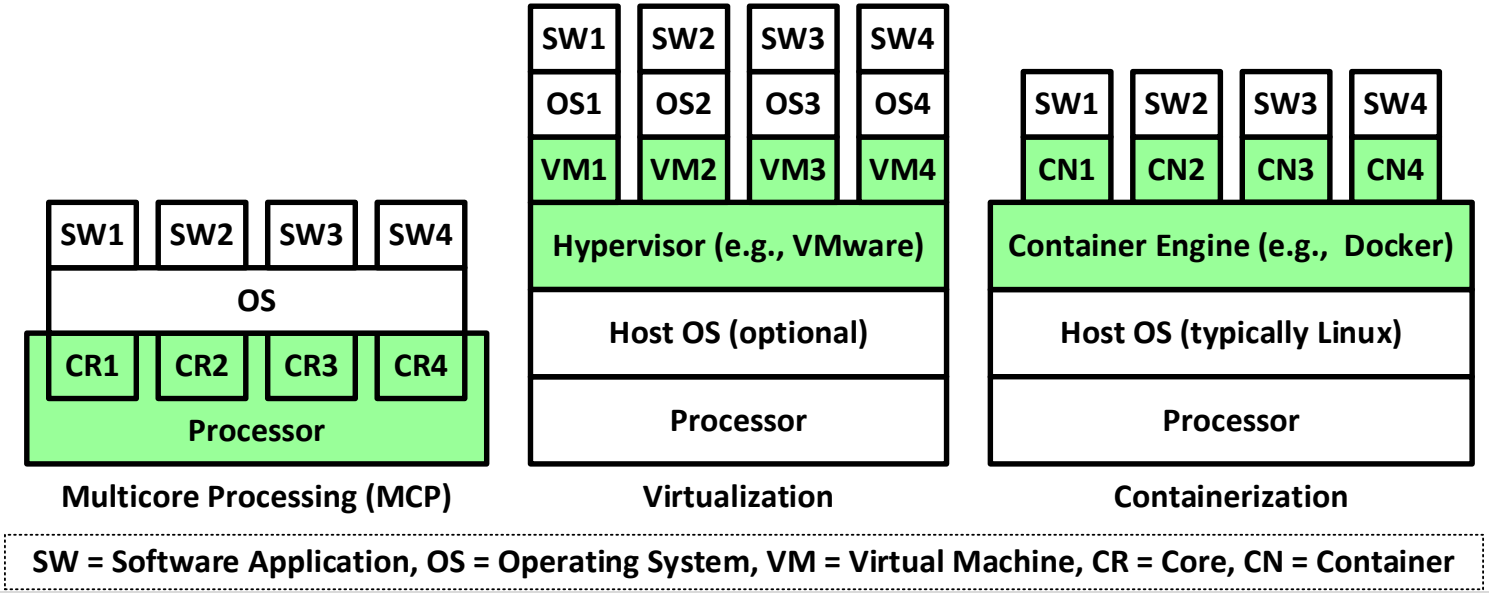
MCP, Virtualization, and Containerization

Big Picture Up Front (BPUF)

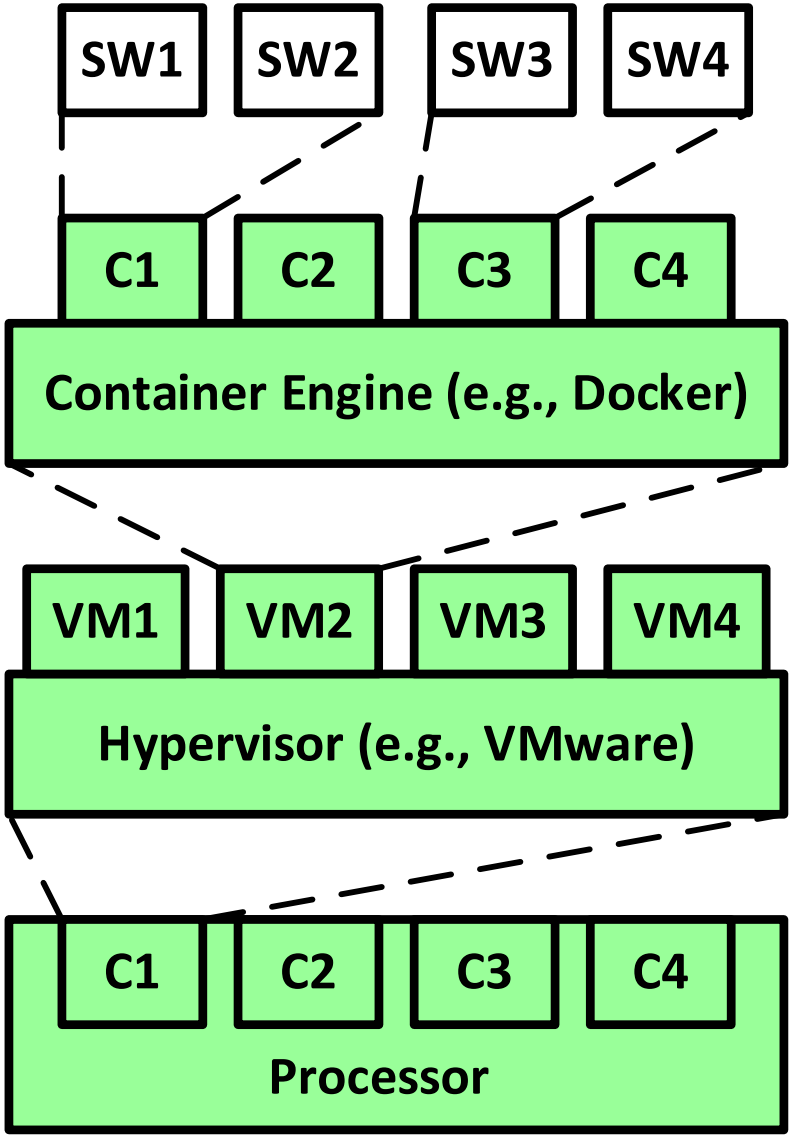
BPUF - Concepts

Mission-critical cyber-physical systems are beginning to be built using some combination of the following three technologies that have significant related ramifications on performance, reliability, robustness, safety, and security:

- Multicore Processing (MCP) – via multicore processors
- Virtualization (V) – via virtual machines (VMs)
- Containerization (C) – via containers



Three Technologies at Three Levels



Multiple Software Applications

Virtualization (via VMs)
(Multiple *Virtual* OSs)

Containerization
(Multiple *Virtual* Hardware)

Multicore Processing (MCP)
(Multiple Actual Hardware)

BPUF – Key Points

Multicore processing, virtualization, and containerization are:

- **Ubiquitous** and largely becoming unavoidable, even in real-time safety-critical systems
- **Different** than traditional architectures in terms of complexity, interference, and non-determinism

Multicore processing, virtualization, and containerization may require:

- **Additional analysis and testing**
- **Changes in safety/security certification policy**

BPUF – Pros

Support for concurrency

Improved reliability and robustness by:

- Limiting fault/failure propagation
- Supporting failover and recovery

Improved SWAP-C

Hardware/OS isolation

- Supports reuse and technology refresh

Decreased hardware costs (due to multicore):

- Fewer computers/processors
- Sharing of underutilized computers/processors

BPUF – Cons

Additional complexity

- Architecture
- Analysis (e.g., performance, safety, and security)
- Testing

Layers of shared resources
(e.g., caches, memory controllers, I/O controllers, and buses):

- Sources of interference
- Added single points of failure

Sources of non-determinism

Increased hardware costs (due to virtualization overhead)

Changes to safety and security accreditation and certification

BPUF– Policies

New ways are needed to verify and certify real-time safety-critical systems using multicore processing, virtualization, and containerization.

Existing policies for ensuring that the related quality requirements (especially reliability, robustness, safety, and security) are met:

- Are often based on assumptions that are no longer true
- Often mandate traditional architectural patterns that are inconsistent with processing, virtualization, and containerization technologies.

MCP, Virtualization, and Containerization

Multi-Core Processing (MCP)

Definition

A **multicore processor** is a single integrated circuit (a.k.a., chip multiprocessor or CMP) that contains multiple core processing units (CPUs), more commonly known as cores.

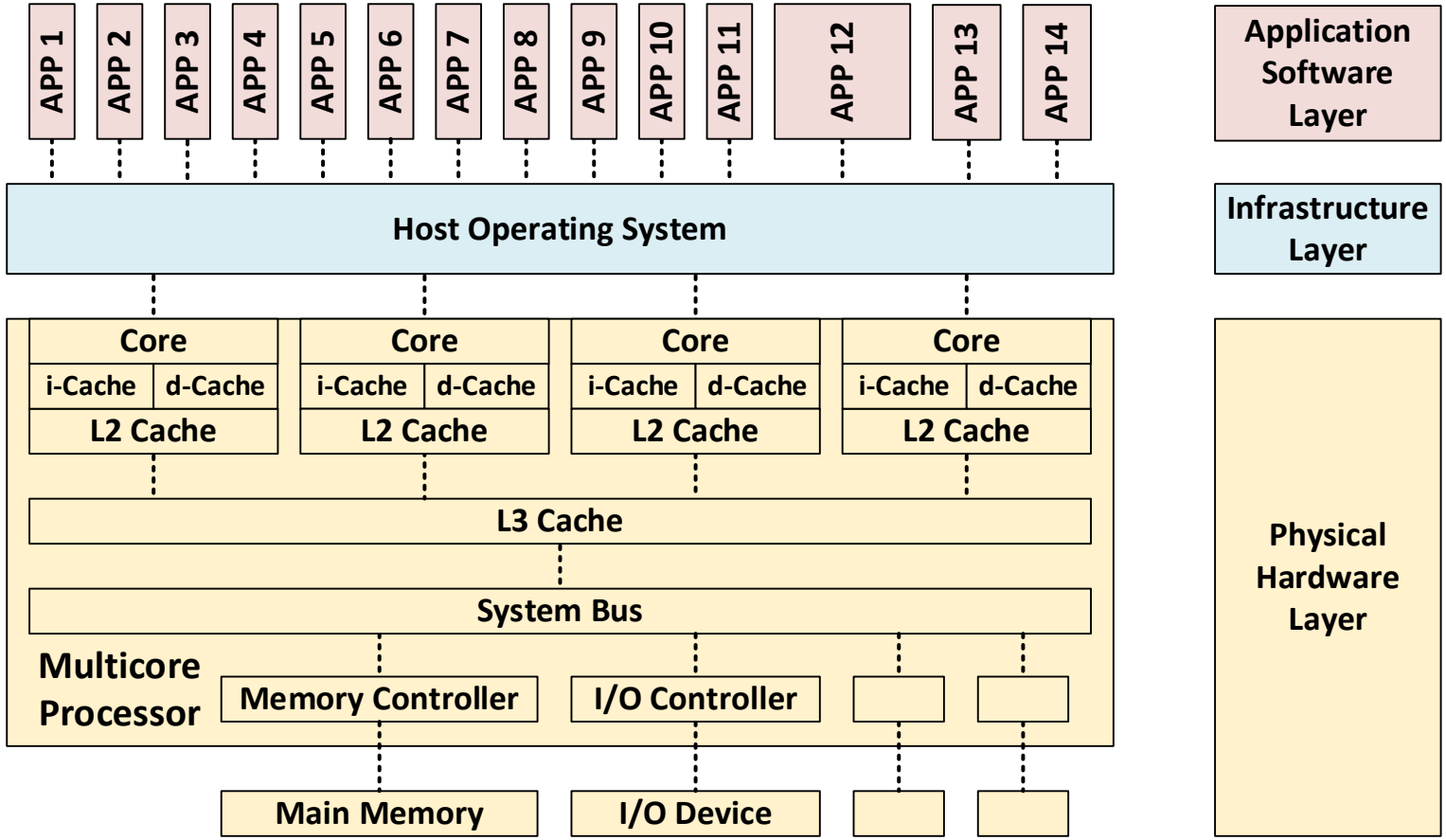
Many different multicore processor architectures exist in terms of:

- Number of cores
- Homogeneous or heterogeneous cores
- Number and level of caches
  (relatively small and fast pools of local memory)
- How the cores are interconnected
- Minimal in-chip support for spatial and temporal isolation of cores:
  - **Physical isolation** ensures that different *cores* cannot access the same physical hardware (e.g., memory locations: caches and RAM).
  - **Temporal isolation** ensures that the execution of software on one *core* does not impact the temporal behavior of software running on another *core*.

Symmetric Multiprocessing (SMP)

Homogeneous cores (typically general purpose)

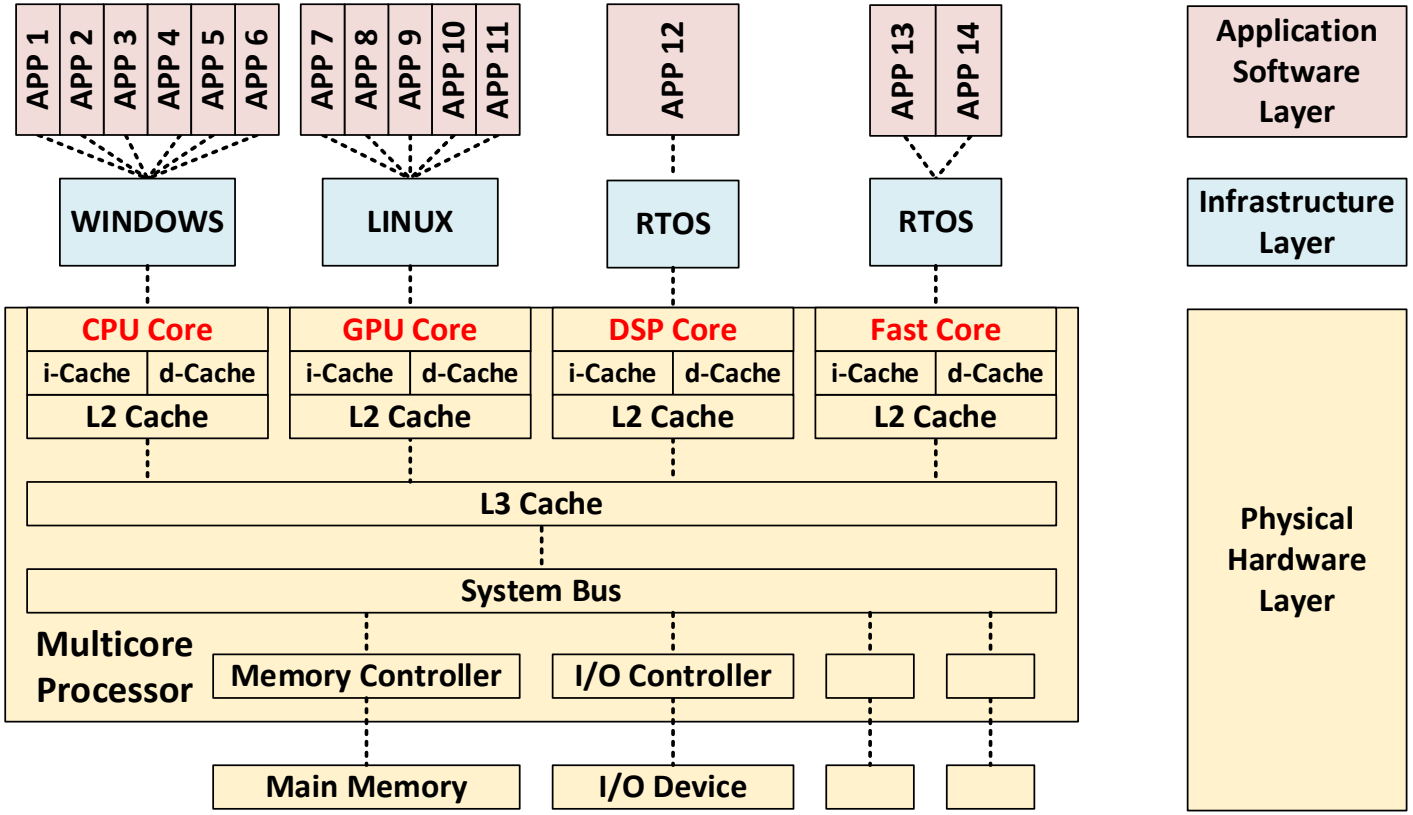
Typically uses a single homogeneous operating system



Asymmetric Multiprocessing (ASP)

Heterogeneous cores

May have heterogeneous operating systems / kernels



Current Trends

Multicore processors are replacing traditional single core processors:

- Fewer single core processors are being produced and supported.
- Single-core processors are increasingly technologically obsolete.

The number of cores continues to increase.

Asymmetric (e.g., computer on a chip) processors becoming more common.

User demand for significantly-increased performance in SWAP-C constrained environments increases need for multicore processing.

Multicore processors are starting to be used in real-time, safety- and security-critical, cyber-physical systems.

Pros – Increased Energy Efficiency

Decrease number of separate embedded computers

Overcomes increased heat generation due to Moore's Law

- Reduces the need for cooling

Reduces power consumption

- Increases battery life

Reduces SWAP-C (size, weight, and power and cooling)

Pros – True Concurrency

Increased intrinsic support for *actual* (as opposed to virtual) parallel processing of:

- Individual software applications
- Multiple SW applications (server and cloud computing)

Pros – Increased Performance

Depends on number of cores, level of real concurrency (multithreading) of the software, and use of shared resources

Decreased distance between cores on integrated chips enable shorter resource access latency and higher cache speeds

- Compared to having separate processors/computers

Pros – Improved Isolation

May somewhat improve (but does *not* guarantee) special and temporal isolation (segregation) compared to single core architectures:

- SW running on one core less likely to affect SW on another core than if both are executing on same single core
  - Spatial isolation of data in core-specific cashes
  - Temporal isolation because thread on one core is not delayed by thread on another core
- May improve robustness by localizing impact of defects to single core

This increased isolation is particularly important in the "independent" execution of mixed-criticality applications (mission-critical, safety-critical, and security-critical).

Cons – Shared Resources

Cores share:

- *Processor-internal* resources (L3 cache, system bus, memory controller, I/O controllers, and interconnects)
- *Processor-external* resources (main memory, I/O devices, and networks)

Shared resources imply:

- Single points of failure
- Two applications running on *same* core can interfere with each other.
- Software running on one core can impact software running on *another* core (i.e., interference can violate spatial and temporal isolation because multicore support for isolation is limited).

Cons – Interference

Interference occurs when software executing on one core impacts the behavior of software executing on other cores in the same processor:

- Failure of spatial isolation (due to shared memory access)
- Failure of temporal isolation (due to interference delays/penalties)

Multicore processors may have special hardware that can be used to enforce spatial isolation to prevent software running on different cores from accessing the same processor-internal memory.

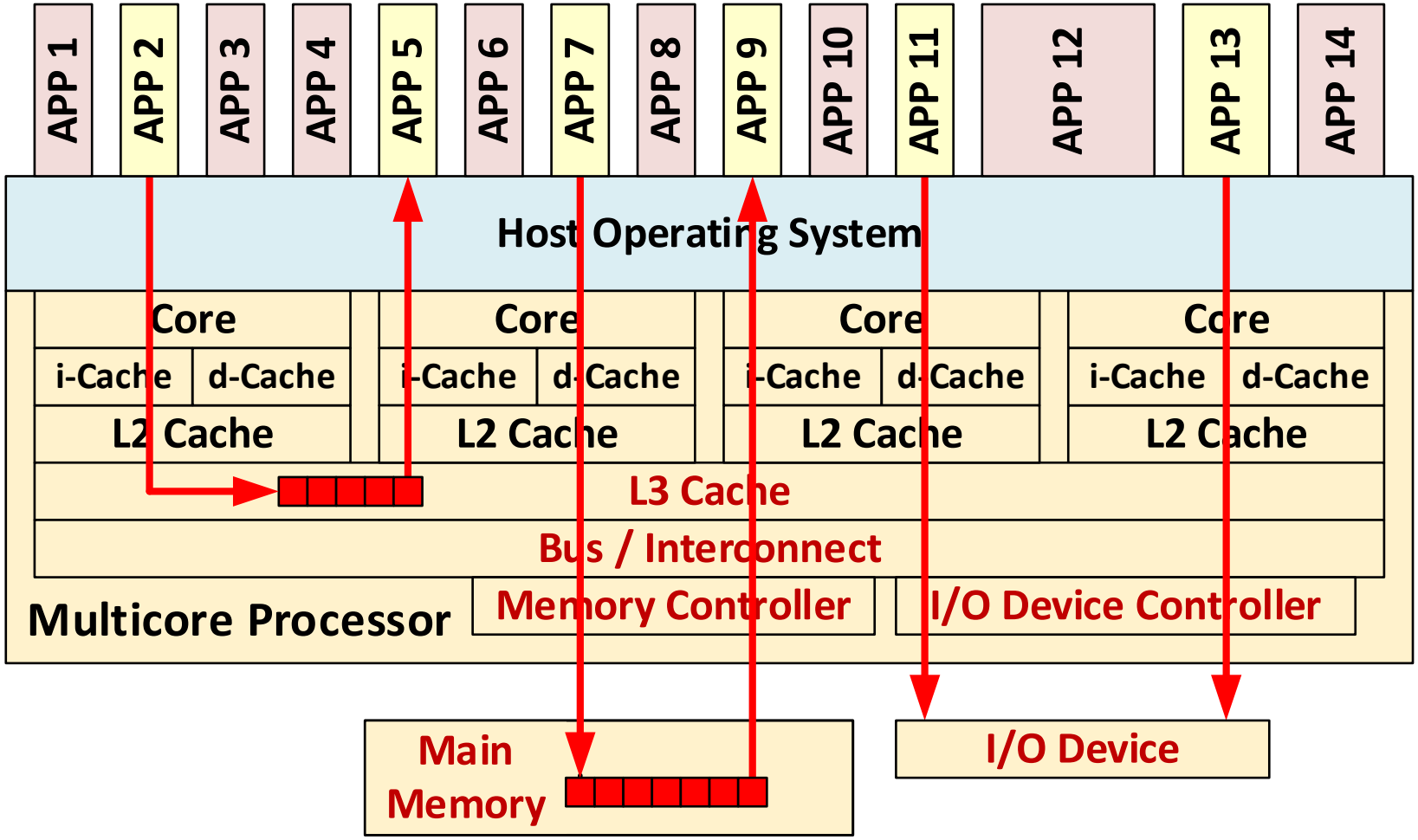
- Temporal isolation is a bigger problem than spatial isolation.

The number of interference paths increase very rapidly with number of cores.

- Exhaustive analysis of all interference paths is often impossible.
- Representative selection of paths is necessary.

Cons – Example Interference Paths

Three example interference paths with shared resources indicated:



Cons – Concurrency Defects

Cores execute concurrently, creating the potential for concurrency defects including deadlock, livelock, starvation, suspension, (data) race conditions, priority inversion, order violations, and atomicity violations.

Note that these concurrency defects may also apply to:

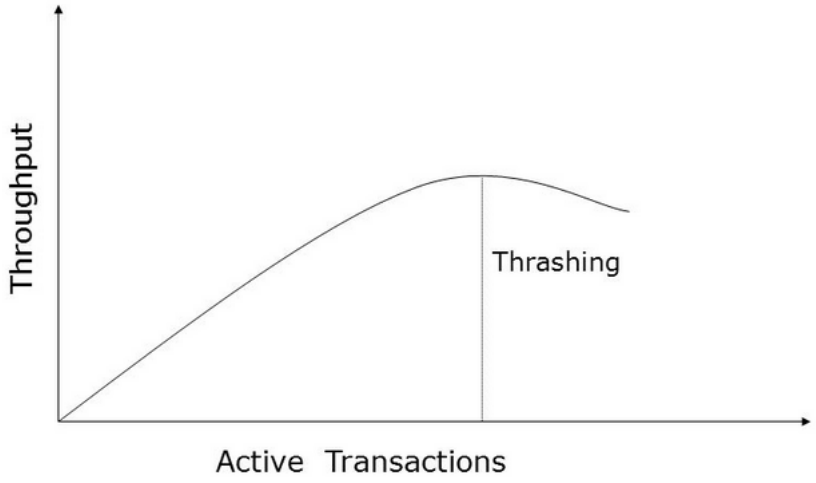
- Multiple threads on single core
- Multiple virtual machines on one or more cores
- Multiple containers on one or more cores

Cons – Increased Non-Determinism

I/O Interrupts have top-level hardware priority

- Note that this is also a problem with single core processors.

Lock thrashing is excessive lock conflicts due to simultaneous access of kernel services by different cores, resulting in decreased concurrency and performance.



The resulting behavior is non-deterministic, unpredictable, and the source of related failures.

Cons – Analysis

Real concurrency requires:

- Different memory consistency models than virtual interleaved concurrency
- Breaks traditional analysis approaches that work on single core processors

Analysis of maximum time limits is:

- More difficult
- May be overly conservative

Although interference analysis becomes more complex as the number of cores per processor increases, overly restricting core number may not provide adequate performance.

Cons – Safety Ramifications

Moving to a multicore architecture may require recertification.

Interference between cores can cause missed deadlines and excessive jitter:

- Can cause faults (hazards) and failures (accidents)
- Requires:
  - Proper real-time scheduling and timing analysis and/or
  - Specialized performance testing

Safety policy guidelines are based on obsolete assumptions.

Safety policy guidelines need to be updated based on the guidelines in the recommendations section.

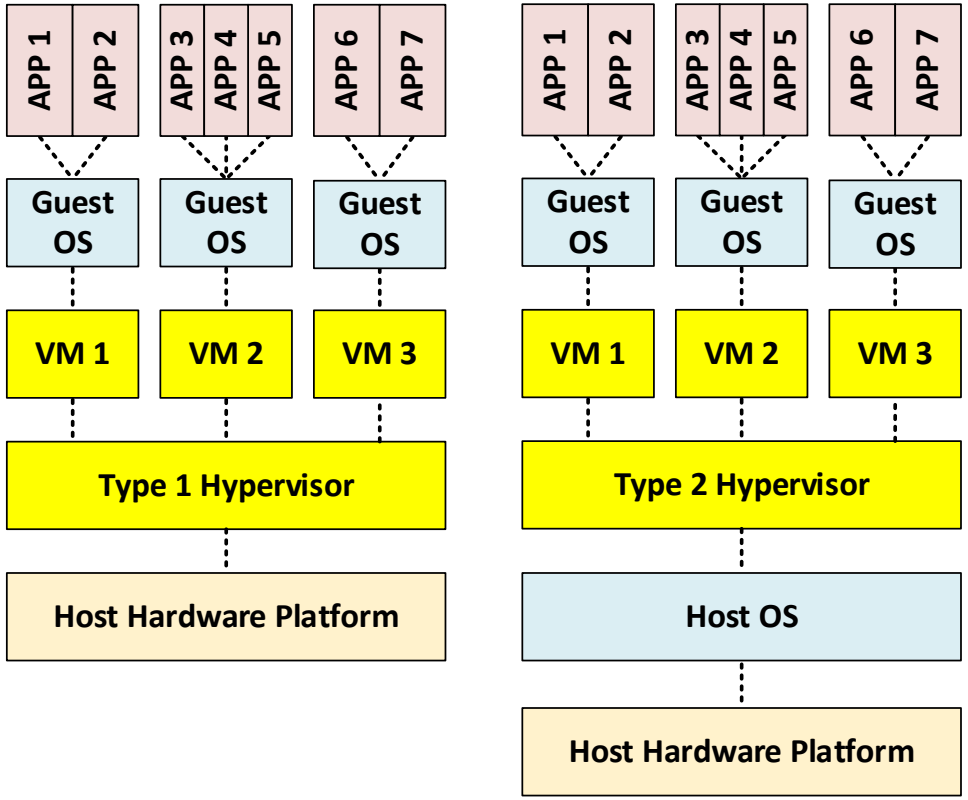
MCP, Virtualization, and Containerization

Virtualization (V)

Definition

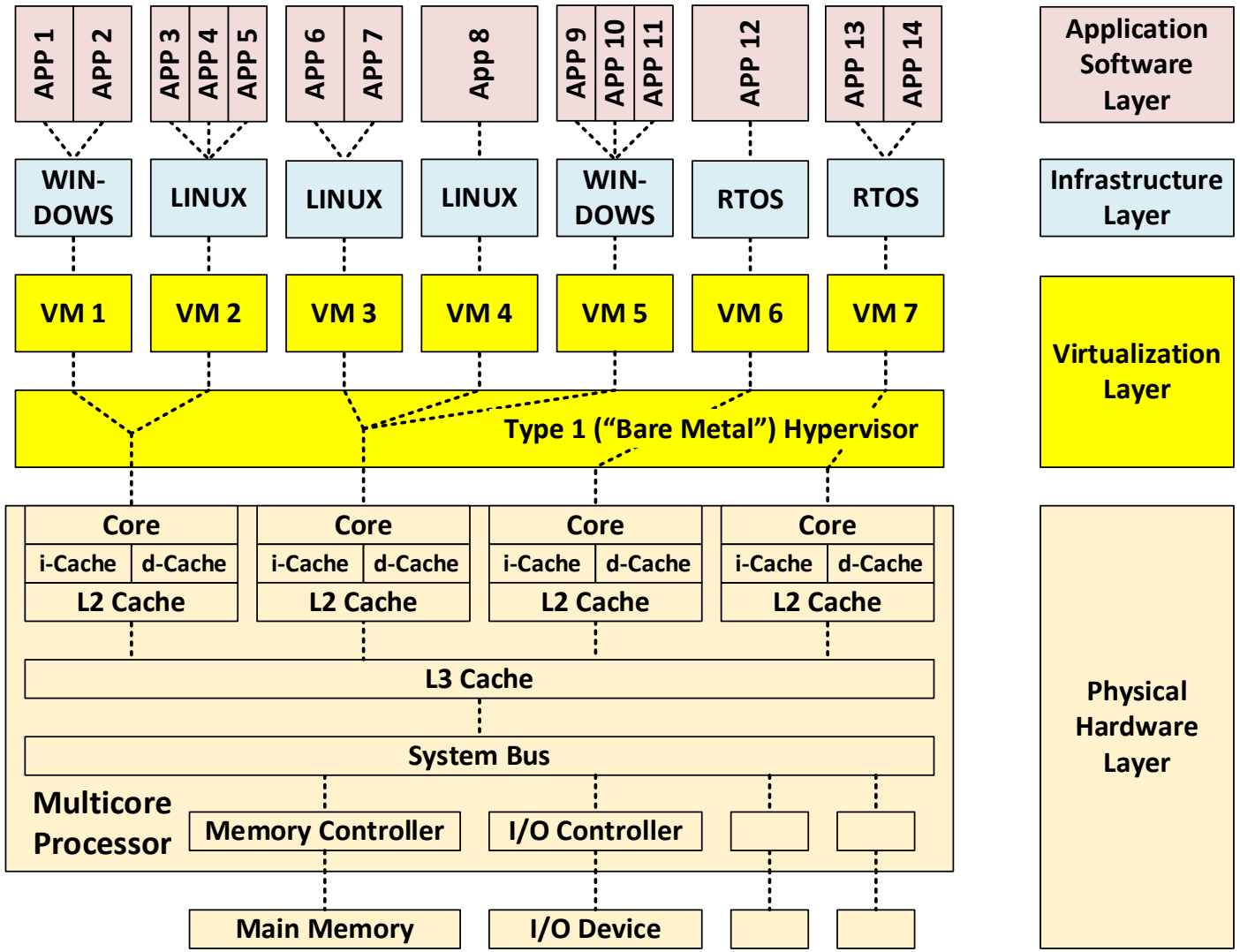
A **virtual machine (VM)**, also called a guest machine, is a software simulation of a hardware platform that provides a virtual operating environment for guest operating systems.

A **hypervisor**, also called a virtual machine monitor (VMM), is a software program that runs on an actual host hardware platform and supervises the execution of the guest operating systems on the virtual machines.



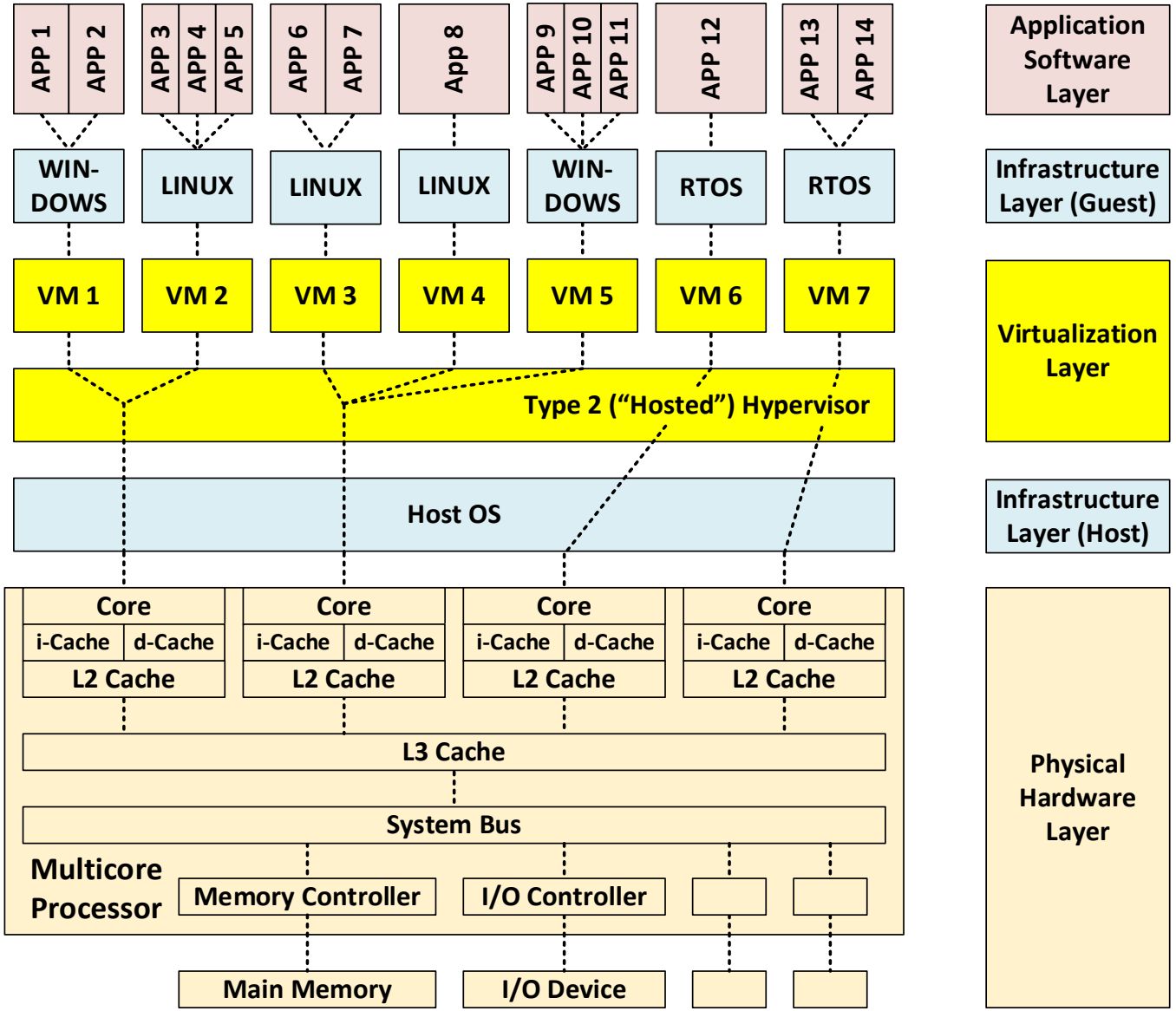
Type 1 (“Bare Metal”) Hypervisor on MCP

Notional Diagram



Type 2 (“Hosted”) Hypervisor on MCP

Notional Diagram



Current Trends – 1

Virtualization is reaching saturation at the server level for:

- IT applications
- Data centers
- Cloud computing

Virtualization is increasingly being used for:

- Storage virtualization (mass storage)
- Network virtualization
- Mobile devices (especially testing on virtual mobile devices)

Virtualization is only just beginning to be used for real-time, safety-critical, and security-critical systems such as:

- Automotive software
- Internet of Things (IoT)
- Military software

Current Trends – 2

Virtualization is being combined with Containerization.

Where appropriate, VMs are being replaced by lighter-weight containers.

Security is increasingly important as vulnerabilities (VM escapes) in virtual machines and hypervisors are discovered.

Pros – Increased Hardware Isolation

Increased hardware isolation:

- Supports reuse of software written for different, potentially older operating systems and hardware
- Enables upgrade of obsolete hardware infrastructure software
- Improves portability to multiple hardware and OS platforms
- Enables virtualized test beds

Pros – Decreased Hardware Costs

Decreases hardware costs by enabling consolidation (i.e., the allocation of multiple applications to the same hardware).

- Take advantage of multicore hardware architecture
- Replace several lightly-loaded machines with fewer, more heavily-loaded machines to:
  - Minimize SWAP-C (size, weight, and power, and cooling)
  - Free up hardware for new functionality
  - Support load balancing
- Support cloud computing, server farms, and mobile computing

Pros – Performance, and Availability

Optimized for general purpose computing and maximizing throughput (valuable for IT and cloud computing):

- Maximizes throughput and *average* case response time

May improve operational availability by:

- Supporting failover and recovery
- Enabling dynamic resource management

Pros – Isolation

Hypervisor should improve (but does not guarantee) spatial and temporal isolation of VMs, whereby:

- **Physical isolation** means that different *VMs* are prevented from accessing the same physical memory locations (e.g., caches and RAM).
- **Temporal isolation** means that the execution of software on one *VM* does not impact the temporal behavior of software running on another *VM*).

Spatial and temporal isolation improves:

- Reliability and robustness by:
  - Localizing the impact of defects to a single VM
  - Enabling software failover and recovery
- Safety by localizing impact of faults and failures to a single VM
- Security by localizing impact of malware to a single VM

Pros – Security

Spatial isolation largely limits impact of malware to a single VM.

- However, sophisticated exploits can escape from one VM to another via the hypervisor.

A VM that is compromised can be terminated and replaced with a new VM that is booted from a known clean image.

- Enables a rapid system restore or software reload following a cybersecurity compromise

A bare-metal type 1 hypervisor has a relatively small attack surface and is less subject to common OS exploits and malware.

Security software and rules implemented at the hypervisor level can apply to all of its VMs.

Cons – Increased HW Resources Needed

Virtualization needs increased hardware resources:

- VMs and hypervisor require more CPU
- VMs and hypervisor require increased RAM
- Images (software state and data) require increased mass storage

Cons – Shared Resources

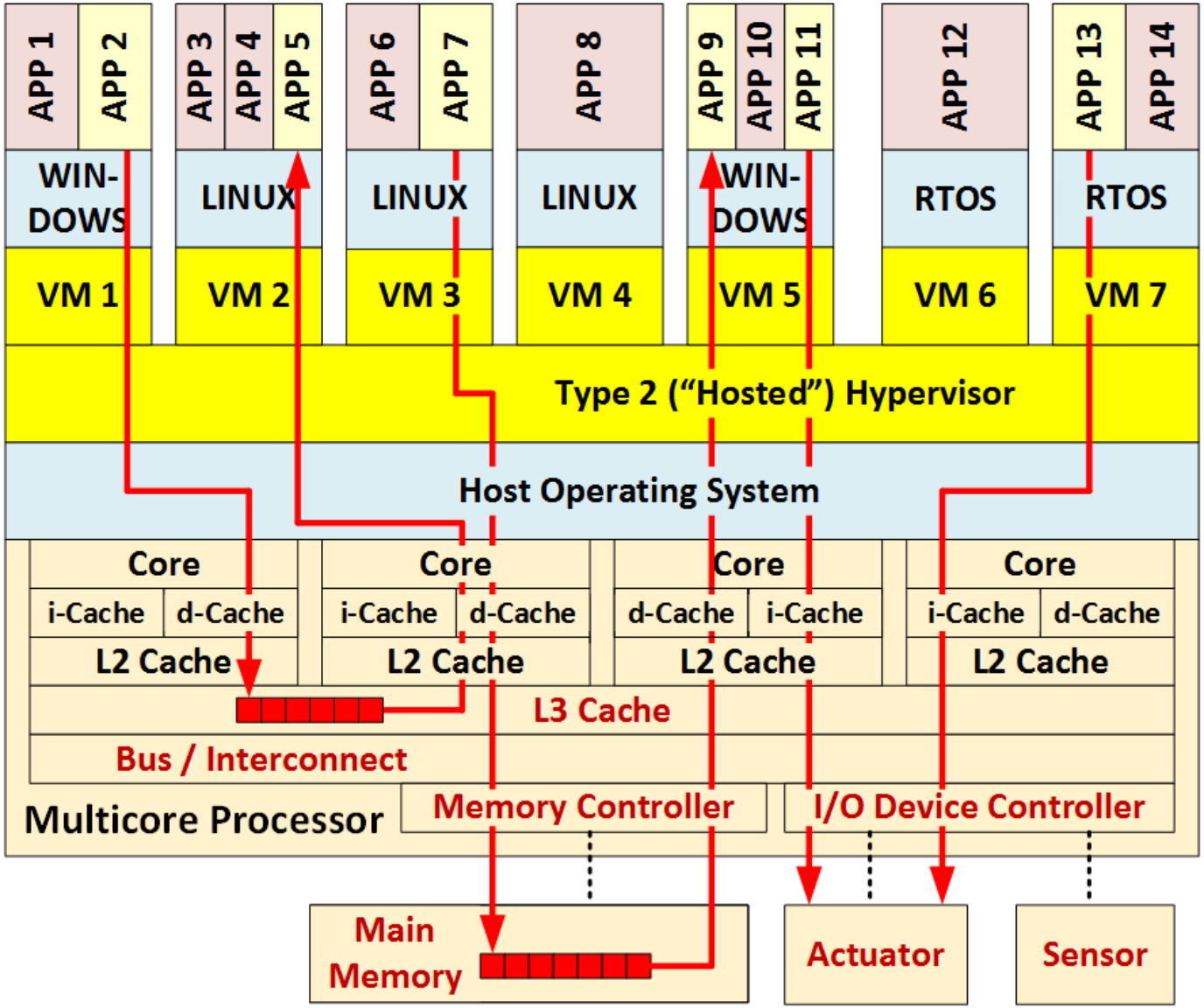
VMs share:

- *Hypervisor*
- Host OS
- Same shared resources as with multicore processors:
  - *Processor-internal* resources (L3 cache, system bus, memory controller, I/O controllers, and interconnects)
  - *Processor-external* resources (main memory, I/O devices, and networks)

Shared resources imply:

- Single points of failure
- Two applications running on *same VM* can interfere with each other.
- Software running on one *VM* can impact software running on *another VM* (i.e., interference can violate spatial and temporal isolation).

Cons – Example Interference Paths



Cons – Analysis

Analysis of temporal interference (e.g., meeting timing deadlines) is difficult and overly conservative.

Interference analysis becomes more complex as:

- The number of VMs increases
- Virtualization is combined with multicore processing

The number of interference paths increase very rapidly with number of VMs.

- Exhaustive analysis of all interference paths is typically impossible.
- Representative selection of paths is necessary.

Cons – Safety Ramifications

Moving to a virtualized architecture based on hypervisors and virtual machines will probably require safety recertification.

Interference between VMs can cause missed deadlines and excessive jitter:

- Can cause faults (hazards) and failures (accidents)
- Requires proper real-time scheduling and timing analysis

Safety policy guidelines are based on obsolete assumptions.

Safety policy guidelines need to be updated based on the following recommendations.

Safety-critical applications can be run on multiple VMs (redundancy with voting).

Cons – Security Ramifications

Moving to a virtualized architecture based on hypervisors and virtual machines will probably require security recertification.

Security vulnerabilities can violate isolation. Sophisticated exploits can escape from one VM to another via the hypervisor.

Cons – Miscellaneous

Real-time lack of predictability causes:

- Jitter
- Failure to meet hard real-time deadlines (response time)

Increased cold start and restart times

Virtualization is a relatively new technology

- Hypervisors and VMs are more buggy than operating systems

Increased system integration and test:

- Increased number of test cases

Increased licensing costs (unless using FOSS)

MCP, Virtualization, and Containerization

Containerization (C)

Definitions

A **container** is a virtual runtime environment that runs on top of a single OS kernel without emulating the underlying hardware.

A "**pod**" is a cohesive collection of containers that are collocated and share resources.

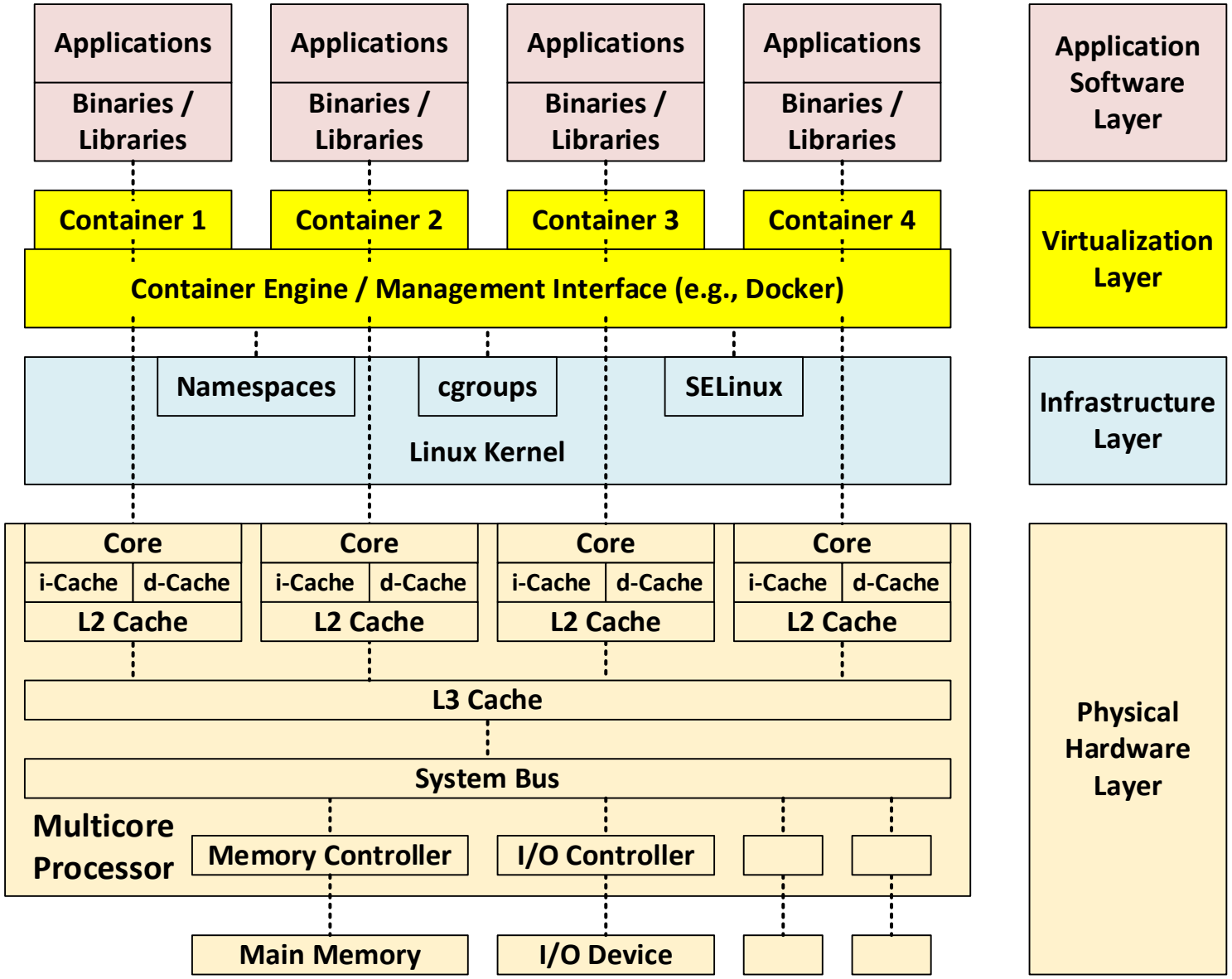
Containerization is sometimes called *virtualization via containers*:

- Virtual machines: multiple *virtual* hardware platforms
- Containers: multiple *virtual* operating systems

Containerization is the process of engineering a software architecture using multiple containers.

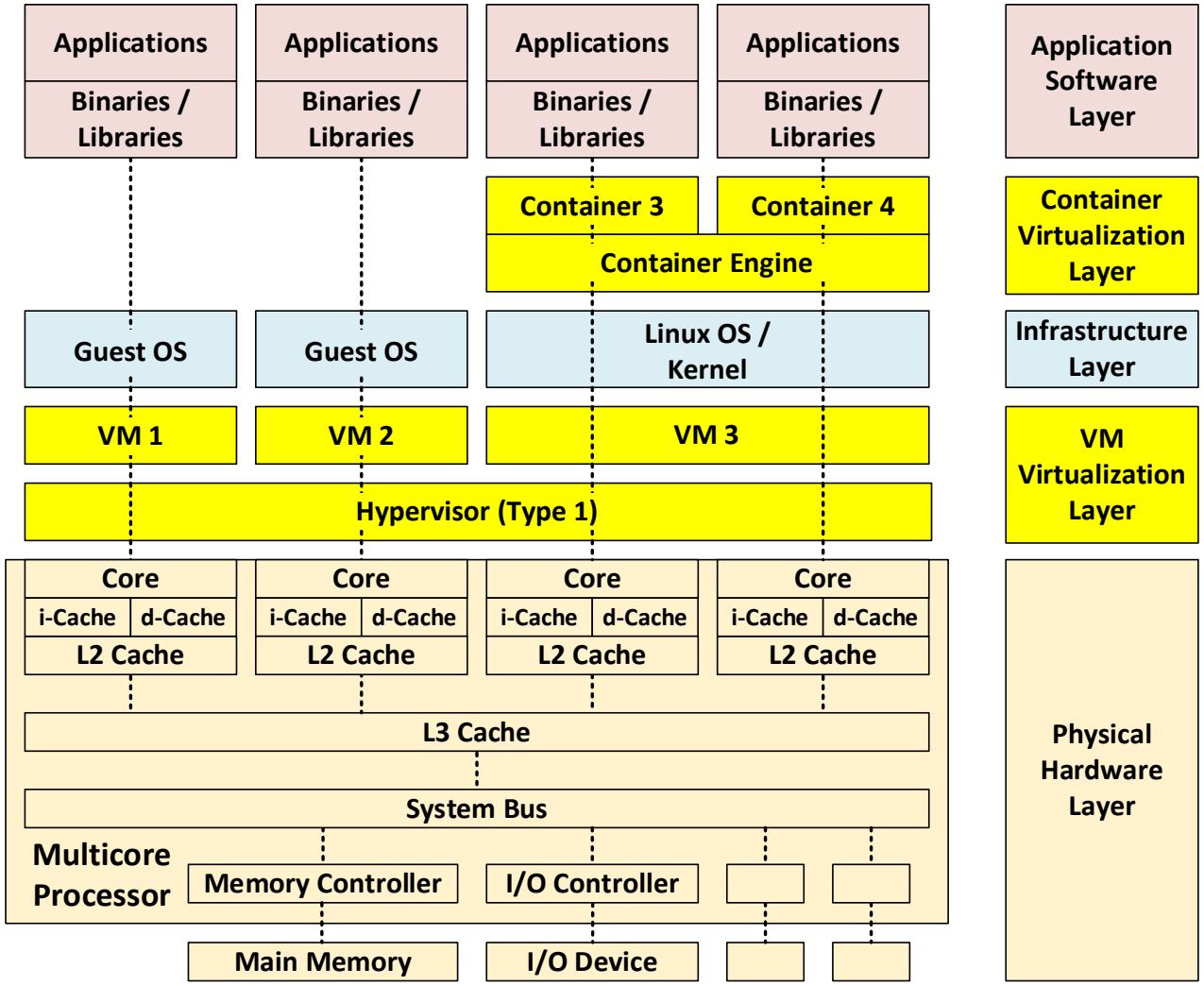
Container orchestration is the process of managing (e.g., creating, deploying, securing, and monitoring) *multiple* containers, possibly spread across multiple VMs, cores, processors, and clusters.

Definition – Pure Containerization



Definition – Hybrid Virtualization

Virtualization combined with containerization



Current Trends

Containers are becoming more common because they provide many of the isolation benefits of VMs without as much overhead.

Although containers are typically hosted on some version of Linux, they are beginning to also be hosted on other OSs such as Windows.

Pros

Supports lightweight spatial and temporal isolation:

- Provides each container with its own resources (e.g., CPU and memory)
- Uses container-specific namespaces

Requires less overhead than VMs, which must emulate underlying hardware.

Relatively easy multiple instantiation of individual containers, which supports

- Scalability
- Availability and reliability via redundancy and failover
- Load balancing

Supports DevOps and continuous integration/deployment (CI/CD)

Supports consistency between development, test, and operational environments

More consistent timing than VMs (supports real-time and safety)

Both Pros and Cons

Applications within a container may share binaries and libraries.

- Decreased code size (Pro)
- Shared code can lead to interference (Con)

Cons – Shared Resources

Containers share:

- *Container engine* and *OS kernel*
- VM, hypervisor, and host OS (if container runs on a VM using a type 2 hypervisor)
- Same shared resources as with multicore processors:
  - *Processor-internal* resources (L3 cache, system bus, memory controller, I/O controllers, and interconnects)
  - *Processor-external* resources (main memory, I/O devices, and networks)

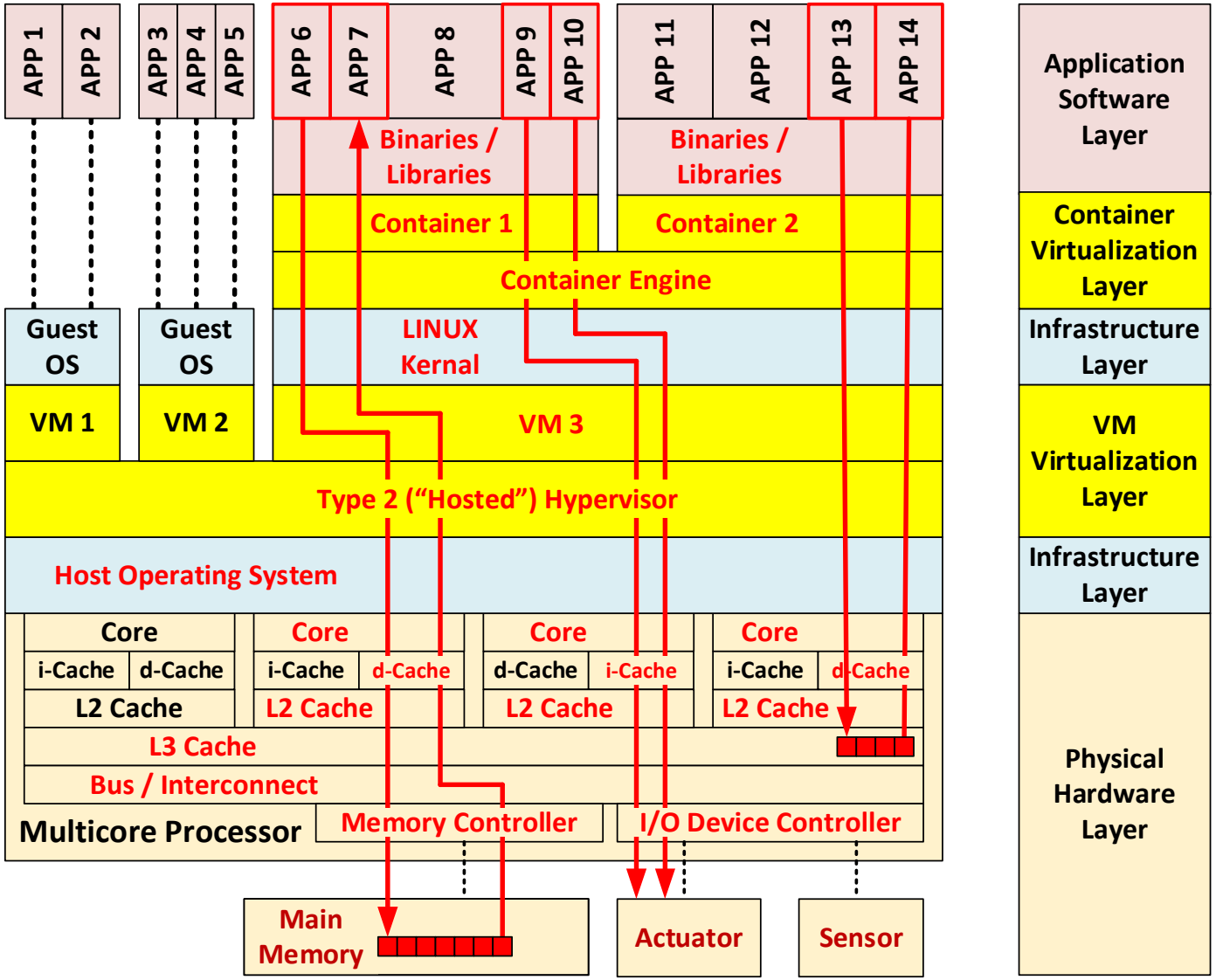
Shared resources imply:

- Single points of failure
- Two applications running in the *same container* can interfere with each other.
- Software running in *one container* can impact software running in *another container* (i.e., interference can violate spatial and temporal isolation).

Interference Paths In Hybrid Architecture

Components with red labels are shared resources.

Red arrows are corresponding interference paths



Cons – Analysis

Analysis of temporal interference (e.g., meeting timing deadlines) is difficult and overly conservative.

Interference analysis becomes more complex as:

- The number of containers increases
- Containerization is combined with:
  - Virtualization
  - Multicore processing

The number of interference paths increase very rapidly with the number of containers.

- Exhaustive analysis of all interference paths is typically impossible.
- Representative selection of paths is necessary.

Cons – Security Ramifications

Containers by default are typically not secure and require significant work to make secure:

- No data stored inside containers pattern
- Force container processes to write to container-specific file systems
- Set up container's network namespace to connect to specific private intranet
- Minimize container services' privileges (e.g., non-root if possible)

Moving to a containerized architecture might require recertification.

Cons – Miscellaneous

Largely restricted to Linux-based operating systems.

Container sprawl (excessive containerization) increases management needs.

MCP, Virtualization, and Containerization

Recommendations

When to Use

Use **Multicore Processors** when:

- Hardware technology refresh makes sense (i.e., regularly)
- Software can be decomposed to benefit from true concurrency

Use **Virtualization** when:

- Isolation is important
- Configurability (flexible deployment) is important
- Multiple hardware types is important

Use **Containerization** when:

- Isolation is important
- Configurability (flexible deployment) is important
- Minimizing size is important

Comparison of VMs vs. Containers

Criteria	VMs	Containers
Portability - Number of operating systems	One or more	One
Portability - Number of OS versions	One or more	One
Portability - Number of OS types	One or more	Primarily Linux
Size of Applications	Medium or large	Small or medium
Security (see notes page)	More isolation	Less isolation
Number of applications per server	Lower	Higher
Number of copies of single application	One	Many
Performance (throughput, not response time)	Lower	Higher
Overhead - Administration	Higher	Lower
Overhead - resource usage	Much higher	Much lower
Readily share resources (devices, services)	No	Yes
Robustness via failover and restart	Not supported	Supported
Scalability and load balancing via dynamic deployment	Slower and harder	Faster and easier
Application runs on bare metal	Not supported	May be supported

Allocation

Keep allocation of VMs/containers to hardware static:

- Simplifies architecture and reduces number of test cases

Only use hybrid virtualization architectures (allocating containers to VMs) where appropriate due to complexity, overhead, and interference.

Certification

Tailor safety and security certification process for MCP, virtualization, and containerization architectures.

- New analysis and testing guidelines
- New documentation guidelines

Document the categorization of software in terms of:

- Real-time deadlines
- Associated safety (hazard) analyses

Recommendations for – Interference Analysis Process

Require the performance of MCP and interference analysis:

1. **Identify relevant software** (e.g., hard real-time & safety-critical)
2. **Determine deployment** of relevant software to cores and data paths
3. **Adequately identify** processor's *important* interference **paths** based on behavior of deployed software, whereby exhaustive identification and analysis is likely infeasible.
4. **Determine potential consequences** based on the interference penalties of the important interference paths.
5. **Categorize interference paths** as acceptable or unacceptable (having either bound or unknown interference penalty).
6. **Use interference elimination and mitigation** (bounding and reducing) **techniques** for unacceptable interference paths.
7. **Repeat steps 2 through 6** until all interface paths are acceptable.
8. **Document analysis results and limitations**
9. **Review analysis results and limitations**

Interference Analysis - Challenges

Challenges of interference analysis include determining:

- Interference paths due to:
  - Processor complexity (number of paths grows exponentially with number of components)
  - Black-box processor components
  - Lack of documentation involving proprietary data
- Multiple “types” of interferences along same path:
  - Operations involved (e.g., read vs. write)
  - Sequences of operations
  - CPU usage (stress)
  - Memory locations accessed
- Sufficient coverage in terms of interference paths
- Interference penalties

Use equivalence classes of paths and path redundancy to limit cases.

Interference Management Techniques – 1

Techniques for eliminating, bounding, and reducing interference include:

- Allocation of software to cores
- Interference-related fault/failure/health monitoring (e.g., performance, missed deadlines) with exception handling
- Configuration (hardware and operating system)
- Cache coloring or partitioning to reduce cache conflicts
- Interference-free scheduling (only one critical task per timeslot)
- Deterministic execution scheduling (tasks accessing the same shared resource execute in different timeslots)

No single silver bullet

Multiple techniques required

Interference Management Techniques – 2

Some interference-management techniques are:

- Completely up to the software architect
- Configurable by the software architect
- Selected and implemented by the processor vendor

Augment Analysis

Combine interference analysis with:

- Relevant testing
- Expert feedback on the processor (or similar processors)
- Information shared with the manufacturer

Testing – 1

Because multicore defects are rare, generate *large* suites of test cases:

- Use soak testing
- Ensure size of test suite is adequate to uncover rare faults and failures.
- Use very large numbers of simulation runs to detect rare events. Google simulates 3 million miles of autonomous driving per day.
- Use statistical analysis when desired behavior is stochastic.
- Use combinatorial testing to achieve adequate coverage of combinations of conditions and of edge and corner cases.
- Use M&S to control non-deterministic hardware and environmental inputs to ensure coverage of corner cases.
- Do ***not*** rely on simple demonstrations of functional requirements (i.e., one test case per requirement).

Testing – 2

Because multicore-, virtualization-, and containerization-related defects are difficult and expensive to uncover, concentrate your effort testing for them on mission- and safety-critical software.

Incorporate Built-In-Test (BIT), especially Continuous BIT (CBIT), Interrupt-Driven BIT (IBIT), and Periodic BIT (PBIT)

Instrument software so that logs can be scrutinized for rare timing and other anomalies.

Program non-deterministic systems (especially autonomous learning systems) to be able to answer questions regarding *why they did what they did*.

Documentation – MCP

Document the following multicore processor information in the system/software architecture models/documentation:

- Vendor and model number
- Processor type:
  - Number of cores
  - Type (Symmetric vs. Asymmetric) including core types and speeds
  - Levels and sizes of caches
- Processor configuration information
- Instruction set architecture
- Memory consistency model
- Successful usage on real-time, safety-critical systems

Documentation – Virtualization

Document the following hypervisor information in the system/software architecture models/documentation:

- Vendor and model number
- Hypervisor type
- Hypervisor configuration information
- Successful usage on real-time, safety-critical systems

Documentation – Containerization

Document the following container information in the system/software architecture models/documentation:

- Vendor and model number
- Container engine
- Container engine configuration information
- Successful usage on real-time, safety-critical systems

Documentation – Deployment

Document the following in the system/software architecture models/documentation:

- Software to container to VM to MCP deployment:
  - Software deployment to guest operating systems
  - Guest OSs to virtual machines or containers
  - VMs to hypervisor (or containers to container engine)
  - Hypervisor to host operating system (if any)
  - Containers/VMs, hypervisor/container engine, and host OS to cores
  - Cores to multicore processors
  - Multicore processors to computers (e.g., blades in racks) and processor-external shared resources
  - Containers/VMs to data partitions in memory

Documentation – Techniques

Document actual deployment using tables or simple relational database rather than deployment diagrams, which likely will be too complex to be more than notional.

Document the techniques used to eliminate, reduce, and mitigate:

- *Important* container-, VM-, and core-interference penalties
- Non-deterministic behavior

Documentation – Analysis and Test

Document the analysis and testing performed to verify performance deadlines are met:

- Analysis and test method(s) used
- Interference paths analyzed and path selection criteria
- Analysis results
- Known limitations of analysis/test results

MCP, Virtualization, and Containerization

Conclusion

Conclusion

Multicore processing, virtualization, and containerization are quite similar, causing similar problems that can be addressed in similar ways:

- MCP – allocate software to multiple *physical* cores on a processor
- Virtualization – allocate software to multiple *virtual* hardware platforms
- Containerization – allocate software to multiple *virtual* software platforms (OS and middleware)

They require changes to safety policy guidance.

They may improve – but do not guarantee – spatial and temporal isolation.

They must overcome interference.

Analysis can help but cannot be exhaustive, requiring augmentation with testing, expert opinion, and past experience.

Contact Information

Presenter / Point of Contact

Donald Firesmith

Principal Engineer

Telephone: +1 412.268.6874

Email: dgf@sei.cmu.edu