Caching Strategies for High Performance

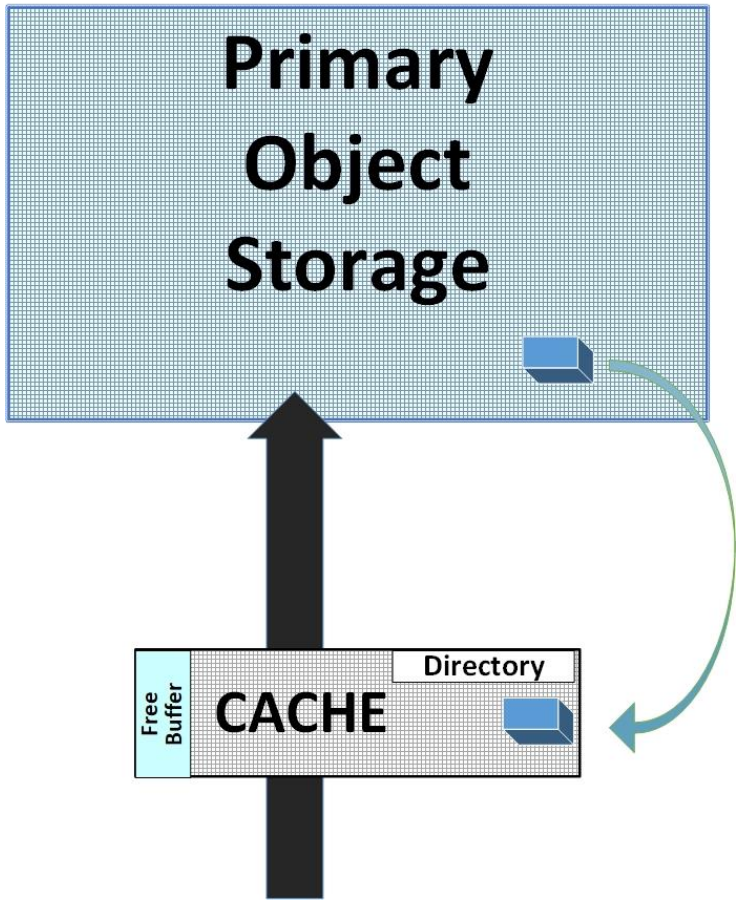
Presented by
Mark B. Friedman

Seminar Outline

- What is a cache?
  - Why does caching work?
  - Measuring cache effectiveness.
  - Cache management and cache coherence.
- Cache implementations
  - CPU caching
  - Virtual memory
  - Disk caching
  - Database caching
- Caching and web-based applications

What is a cache?

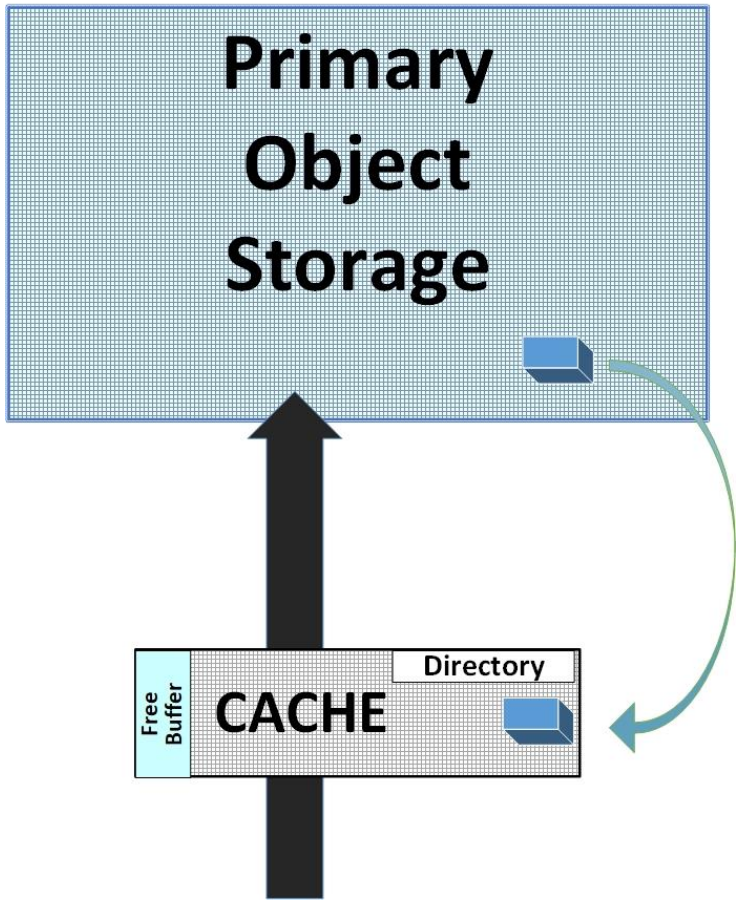
- The design *Goal* of adding cache to a subsystem is to achieve
  - *performance* near the speed of the smaller, faster, more expensive cache unit
  - at a *cost* closer to cost of the larger, slower, less expensive backing store
- **Hierarchical memory management**



What is a cache?

- Caches store a ***subset*** of the objects in Primary Storage in a compact storage medium, with significantly ***reduced latency***
  - ∃ a mapping from Primary ⇔ Cache
- The cache is "hidden" from the application
  - i.e., *not* directly addressable
  - access to data in the cache is *transparent* to the application
  - In general, an n-tiered ***memory hierarchy***

- Access to data in Cache (a cache "hit") is *faster* than accessing Primary storage



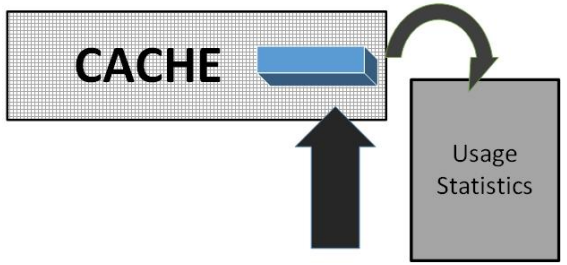
How caches work

Cache Read Hit

Primary Object Storage

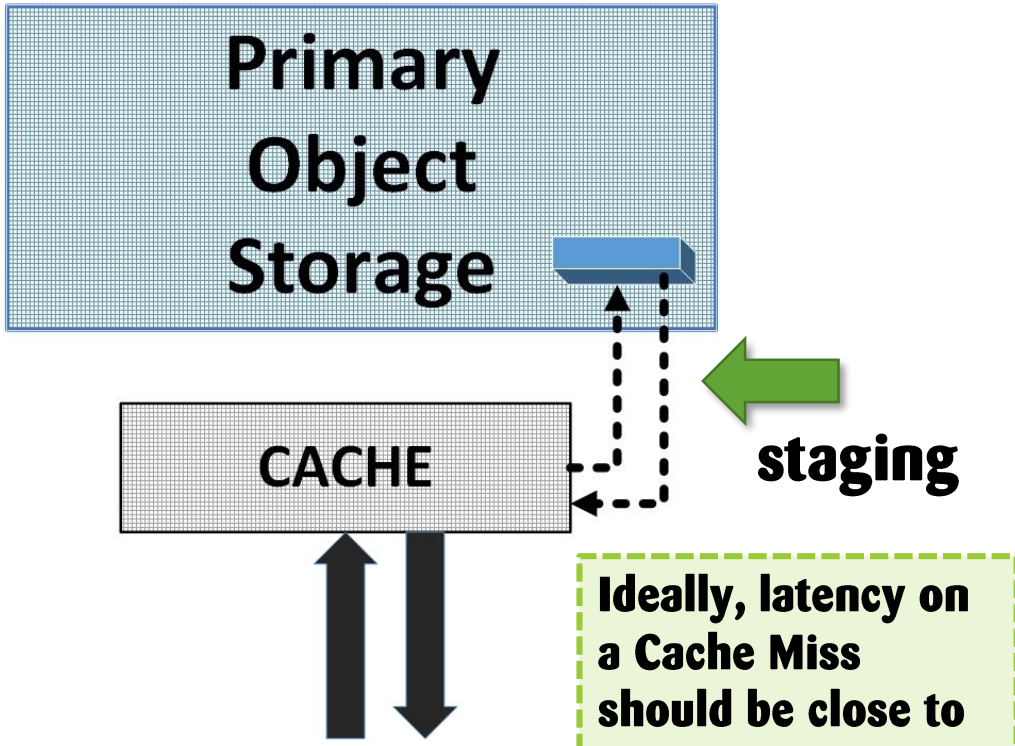
The Cache contains a subset of the items in Primary object storage. Requests that can satisfied from cache are "hits"; requests that cannot are "misses."

Low Latency Cache Hit



Reduced latency on Cache Hits is the main performance benefit of a cache

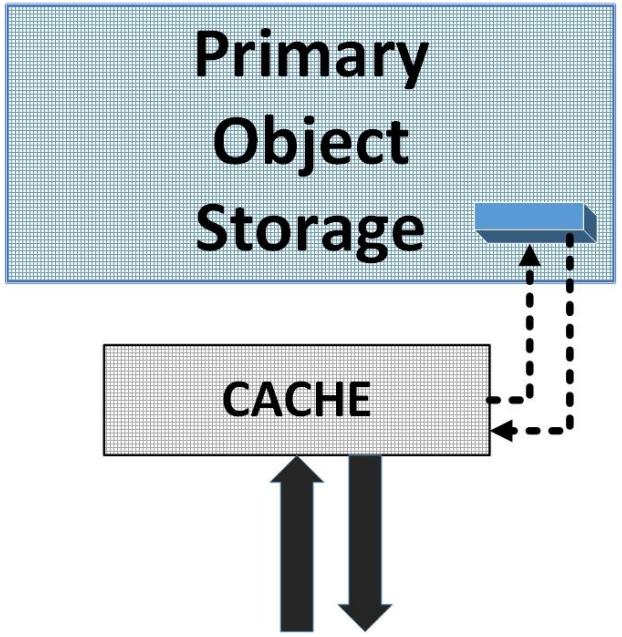
Cache Read Miss



Ideally, latency on a Cache Miss should be close to the performance of native access

Cache Read hits

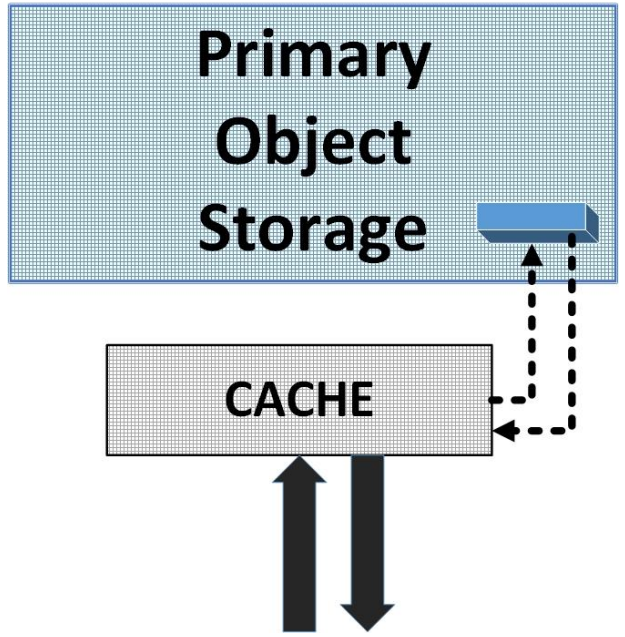
- **Read hits are serviced at the speed of cache**



Type	Cache latency	Backing store latency
Processor memory	CPU clock speed	
Virtual memory	CPU clock speed	1-20 ms
Disk controller	500 μsecs	1-20 ms
Relational Database	CPU clock speed	1-20 ms
Web browser	Direct file system access	Network access

Cache Read misses

- **Data requested from the Primary is usually *staged* on demand into the cache first (forking also possible)**
- **The amount of data staged (a chunk) can be > than the request size**

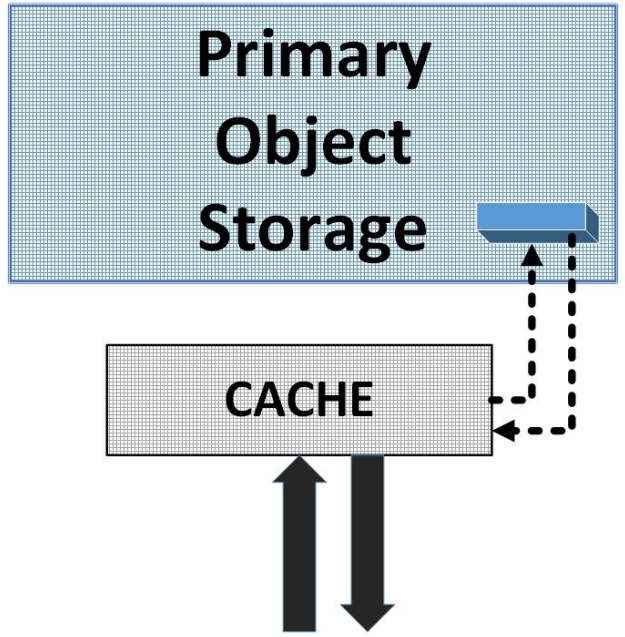


Type	Request	Chunk size
Processor memory	Address	Cache line (e.g., 256 bytes)
Virtual memory	Address	Page (e.g., 4KB, 1 MB)
Disk controller	Record	Track
Relational Database	Block	Block
Web browser	File	File

Cache Read misses

- When the amount of data staged = the request size and the size of the Request is non-uniform:
  - First Fit vs. Best Fit
  - allocation failures due to fragmentation
  - de-fragmentation or compaction
  - e.g., LOH in .NET Memory Mgmt

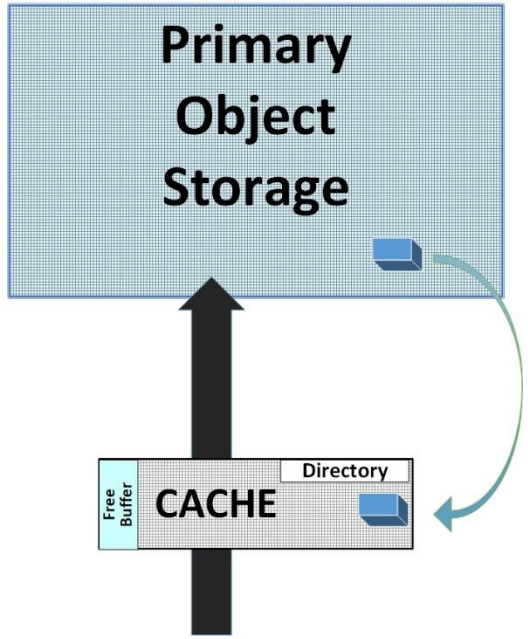
- Object size matters!
  - front-end vs. back-end bandwidth
  - pre-fetching
  - “superfluidity”



Cache Read misses

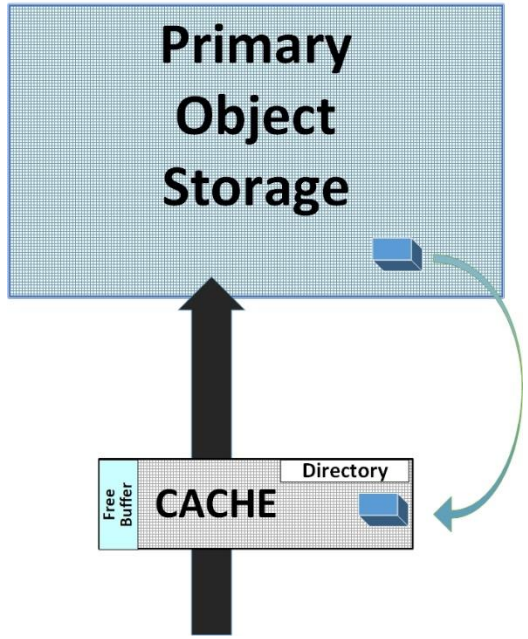
- Cache "cold start" vs. a "warm start"
- Resident set vs. an ideal Working set
  - if *sizeof* (cache) << working set

- Context switch
  - e.g.,
  - a *different* thread starts execution
  - a different thread from a *different process* starts execution
  - a new tab is opened in the browser



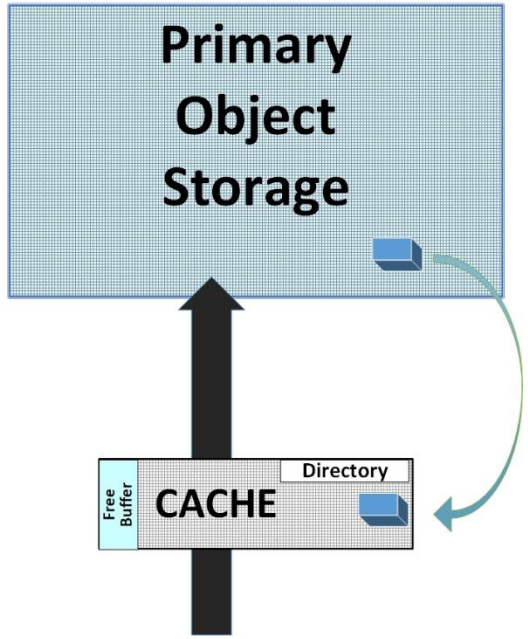
Cache Read misses

- Since Primary Storage >> Cache, the Cache will (eventually) fill up
- So, the cache requires a *replacement policy*
  - most studied: LRU
    - make room for the new (Most Recently Accessed) object by removing an older (Least Recently Used) object
    - order items in cache by Last Access timestamp
    - "Stack" algorithm



Cache Read misses

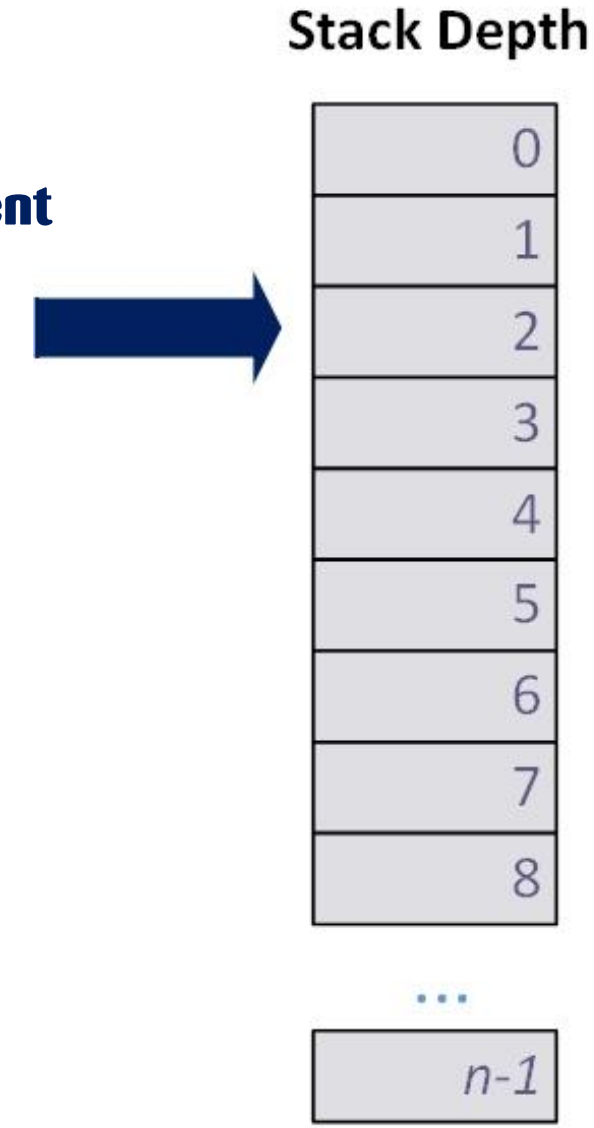
- Replacement policy invoked when cache is full
  - optimal: Most Recently Referenced
  - near optimal: LRU + application hints
  - modified LRU
    - partial ordering by Last Memory Management event
  - sequential limiting
    - read ahead : delete behind
  - FIFO: approximates LRU
  - random replacement
    - if cache is large enough, often performs better than you might otherwise expect



LRU stack algorithm

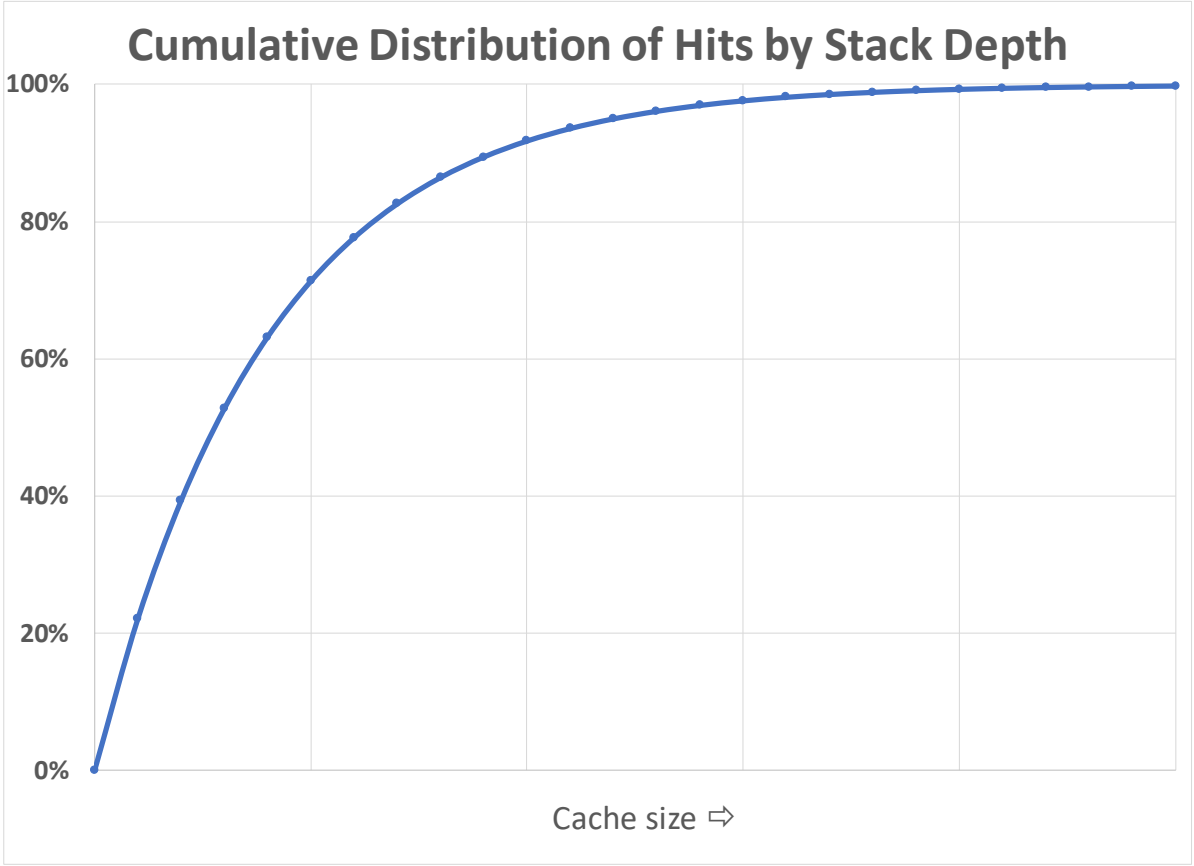
- The Top of Stack is the Most Recently Referenced object,
- The Bottom of the Stack is the Least Recently Used object in the current resident set, and
- Maintain a ***Total Order*** by time of Last Reference

- For a new Request:
  - Search the Stack in sequence from Top to Bottom
  - if there is a Hit at depth *i*,
    - Push down elements 0 through *i-1* and
    - Move the object at depth$_i$ to TOS
  - on a miss,
    - Remove the object at depth$_{n-1}$
    - Push down elements 0 through *n-2*, and
    - Insert the new object at the TOS



LRU stack algorithm

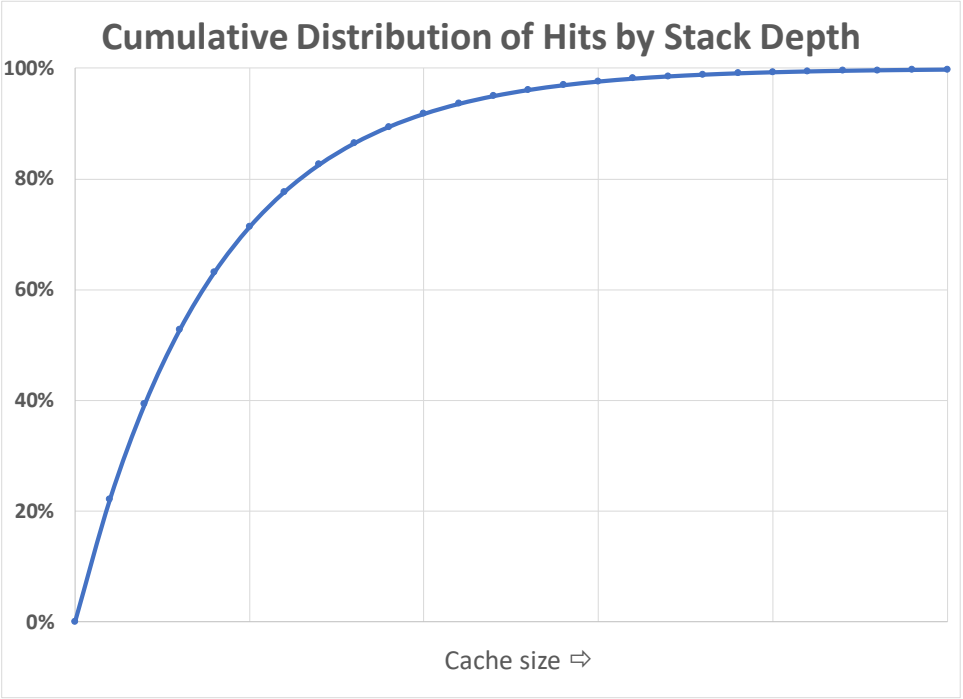
- Typical distribution of cache hits by stack depth (cache size)
- Most hits occur at or near the TOS
- Diminishing returns from adding more cache memory
- ***Why*** do caches work so well?



• *Why* do caches work so well?

- **Locality of Reference**
  - Tendency for objects that were recently referenced to be referenced again soon
  - Tautological explanation, but there might be something to it

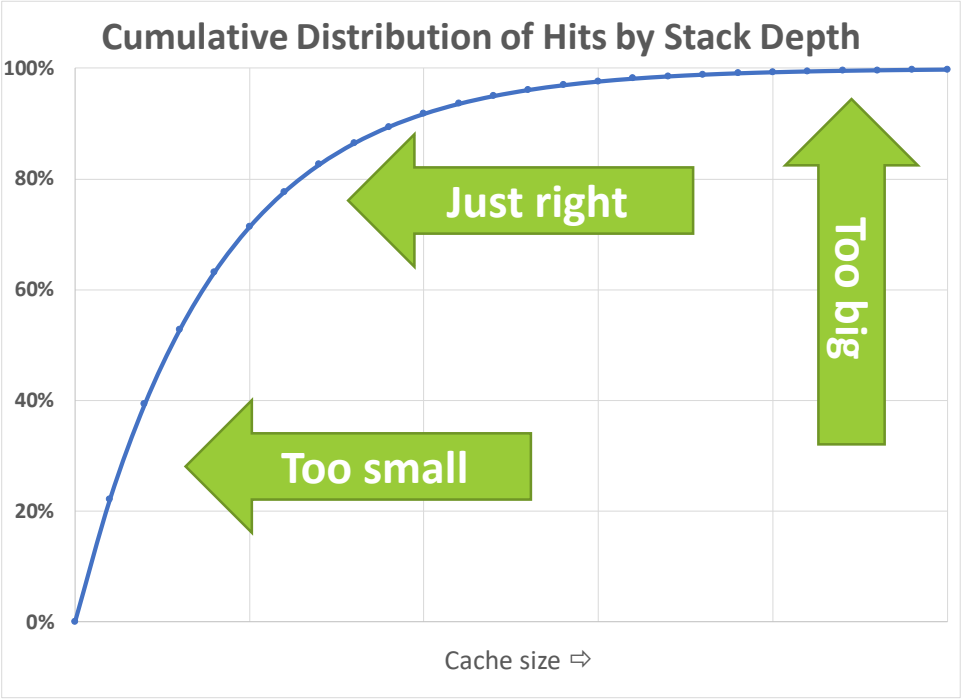
  - What are some of the reasons Locality of Reference occurs in the following?
    - CPU instruction execution streams
    - Processor memory management
    - Disk and Database Access
    - Web access



Cache sizing

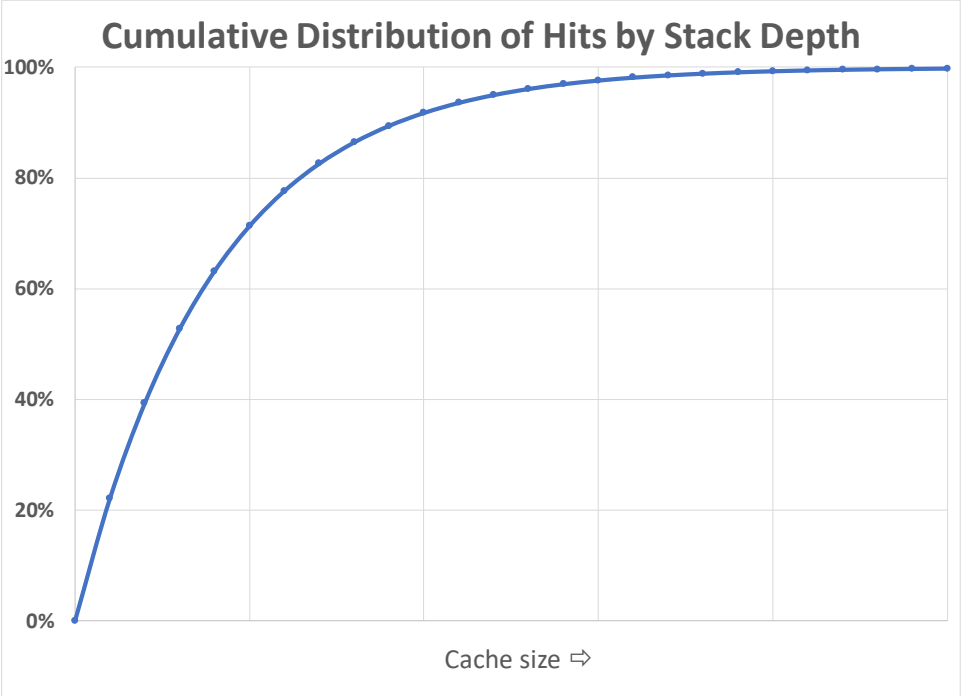
- Diminishing returns from adding more memory
- Increasing memory management "overhead" from larger memories
  - Coherence (Write Back caches)
- Three operating regions:
  - Too small
  - Too large
  - Just right

- What about multiple Caches arranged in a hierarchy



Cache sizing

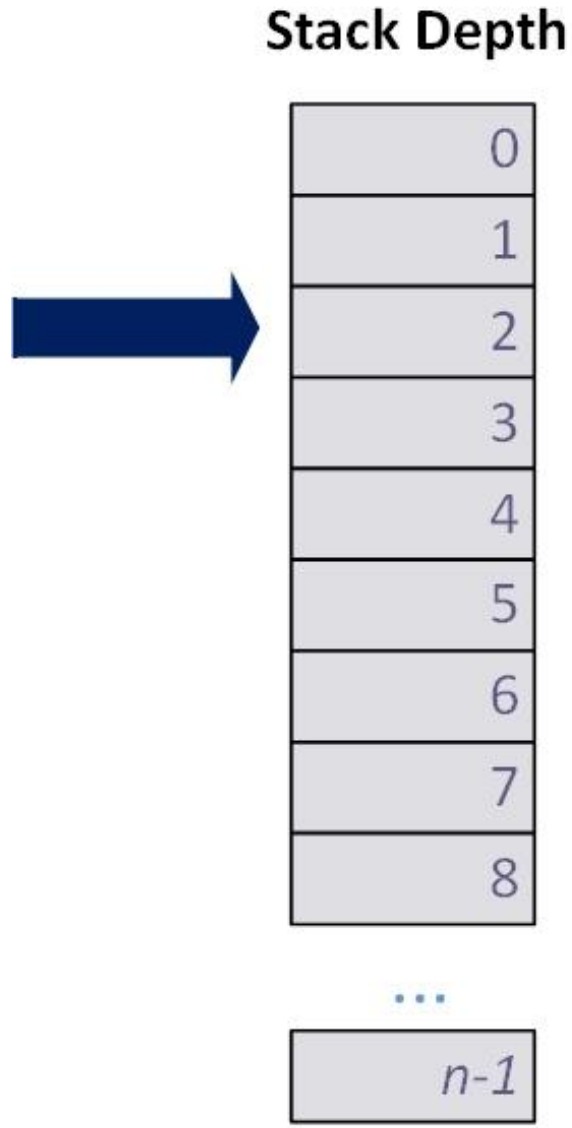
- Multiple Caches arranged in a hierarchy
  - Example: multiple levels of cache in Intel processors: L1, L2, L3
  - Logically,
    - L1 ⊇ L2 ⊇ L3
  - So, the L3 cache contains duplicate entries for everything stored in L2



LRU stack algorithm

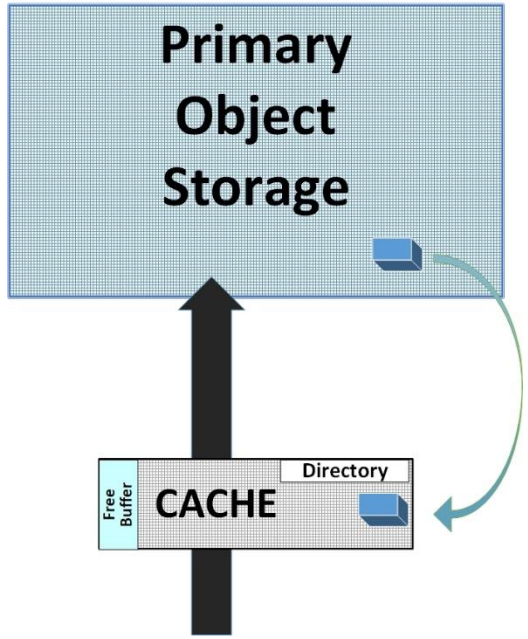
- Consider the performance of this algorithm:
  - *minimum* Search time occurs for a Hit near the TOS
  - *maximum* Search time occurs for a cache miss and increases linearly with the size of the cache

  - For your cache implementation, you may prefer a modified form of LRU such that the service time to search the cache has less variance
    - e.g., use a Hash function to place items in the Cache



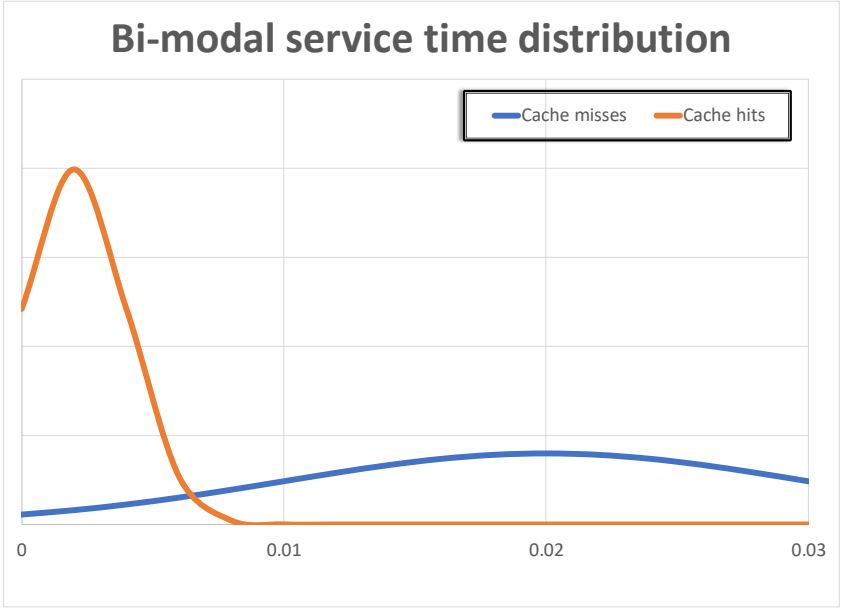
Cache Replacement policy

- Because maintaining a sorted list in the cache directory LRU can be expensive…
  - modified LRU: *partial order* based on some recurring (and relevant) management interval
    - Clock → partial sort by residence time
    - Low memory event (in lieu of using the reference bit): VAX VMS and Windows OS
    - the Garbage Collection interval in .NET memory mgmt.



How does cache impact the overall service time distribution?

- **Service time distribution is bi-modal**



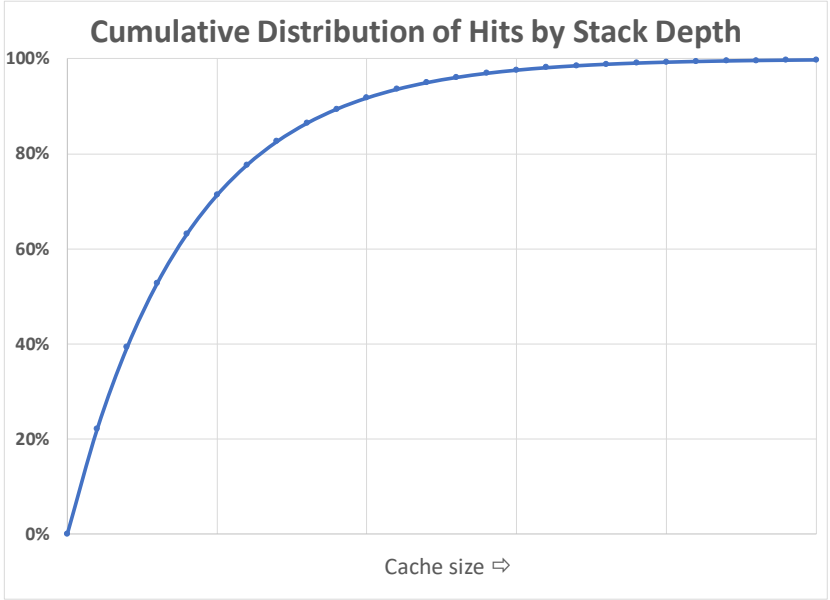
- **Calculate a weighted average service time:**

$$\bar{w} = hit\,\% * \bar{w}_{hits} + ((1 - hit\,\%) * \bar{w}_{misses})$$

- **Cache benefit depends on *both***
  - **the *hit %*, and**
  - **how much *faster* requests are serviced on a cache hit.**

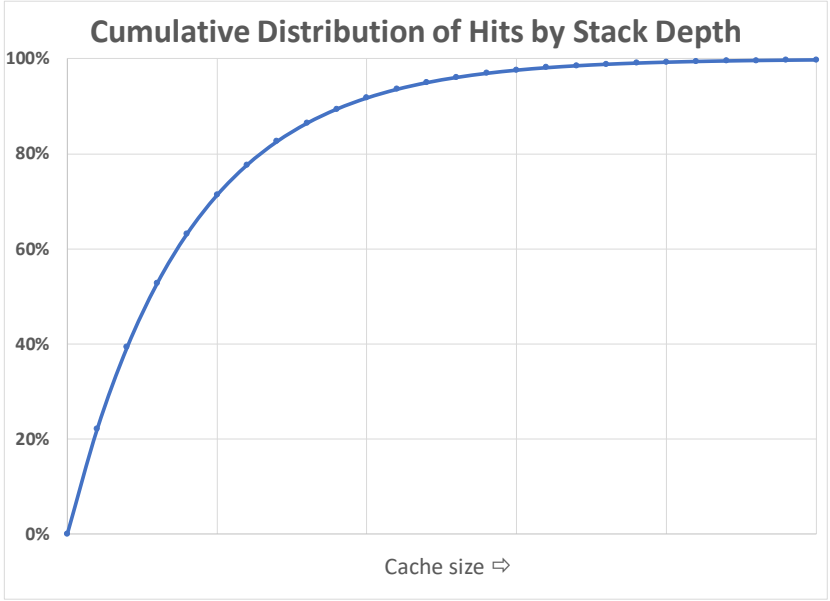
Cache sizing

- Despite the prevalence of cache-friendly access patterns (principle of Locality), there are also well known cache *anti-patterns*
  - extended, rapid sequential access
  - exploit superfluidity
  - truly random access
- Caches can only be modeled as finite-state machines where the sequence of events matters
  - Load-dependent servers



Cache sizing

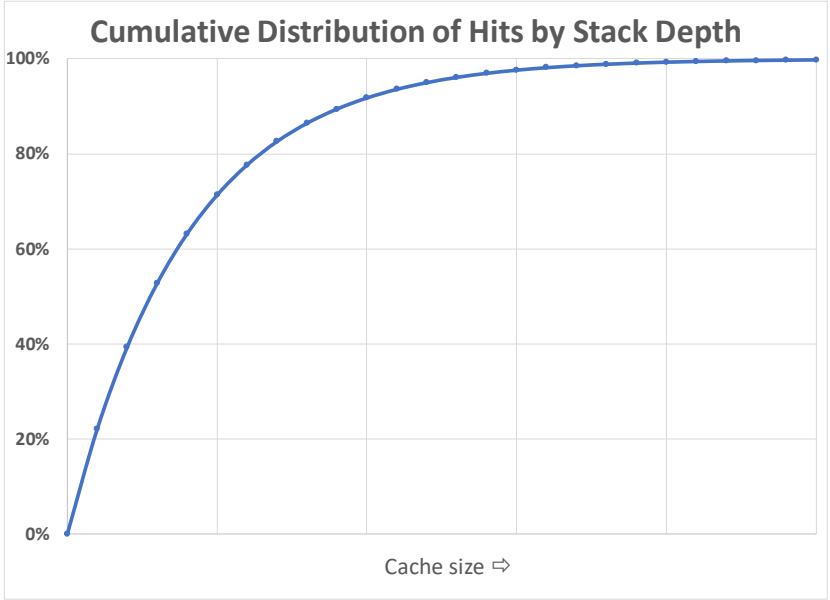
- Virtual Memory “Thrashing”
  - When cache size is too small, the overhead of the page replacement policy may be excessive



Cache sizing

- Experience predicting cache hits as a function of cache size from address reference traces

- Issues:
  - generic stack algorithm varies from the actual replacement policy
  - Cost of acquiring traces
  - Representativeness of trace samples



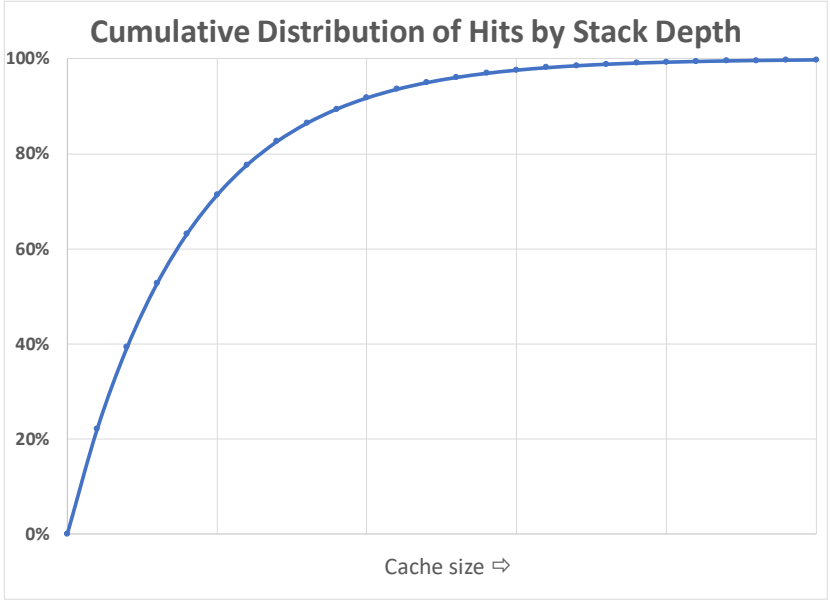
Cache sizing

- **Experience predicting paging as a function of the ratio of**

  real memory : virtual memory

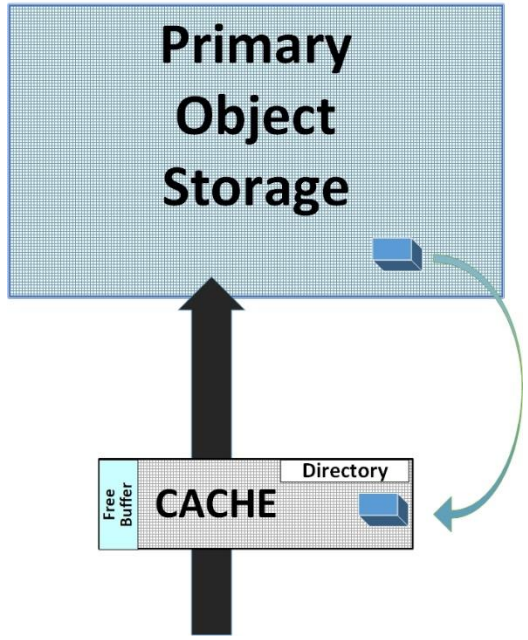
- **V/R ratio**

- **To Do: show relationship of Committed Bytes (V) to "Hard" Paging rates on a Windows Server**



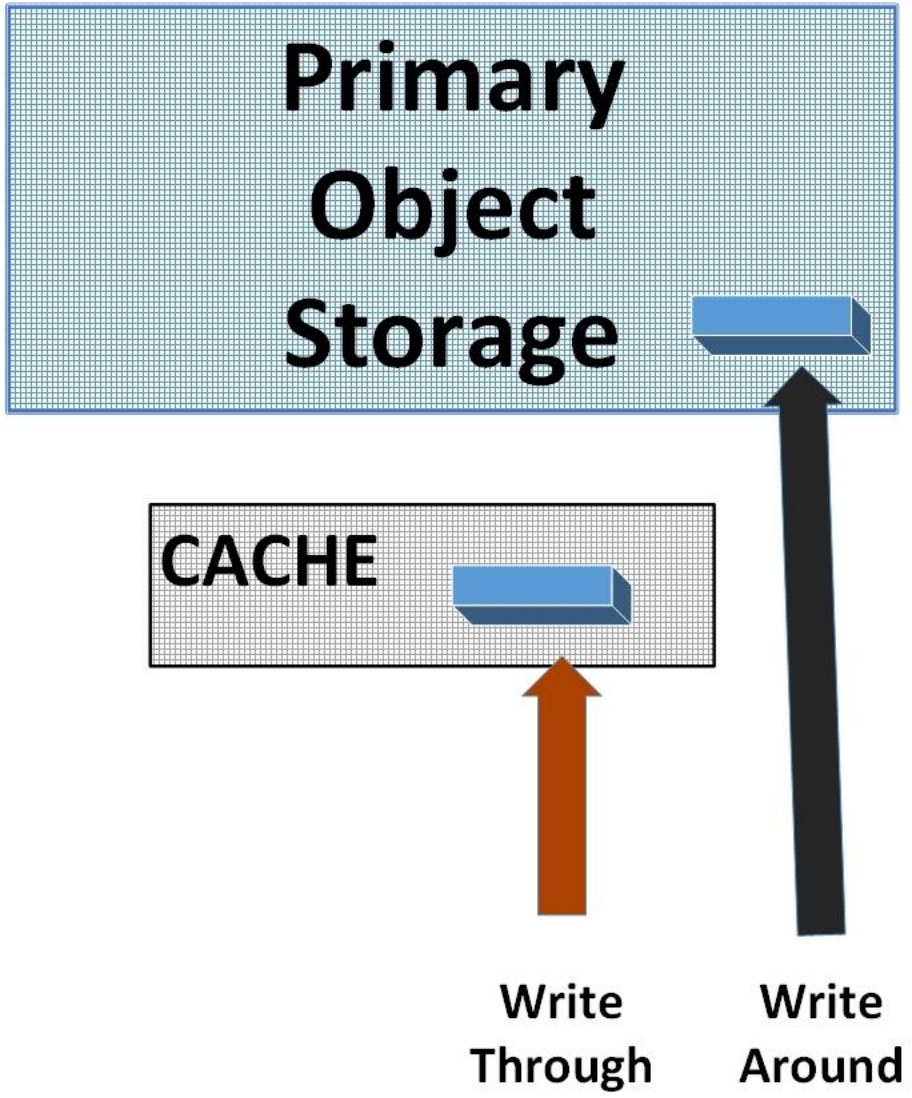
Cache Writes

- **Replacement Policy**
  - maintain a Dirty bit associated with each object that resides in the Cache
    - if the data in the Primary object storage is current, than the cache block can be safely discarded
    - if the object in Primary Storage is not current, it must be updated before the cache block can be discarded
  - Maintain an elastic free space buffer so that there is seldom any additional latency associated with



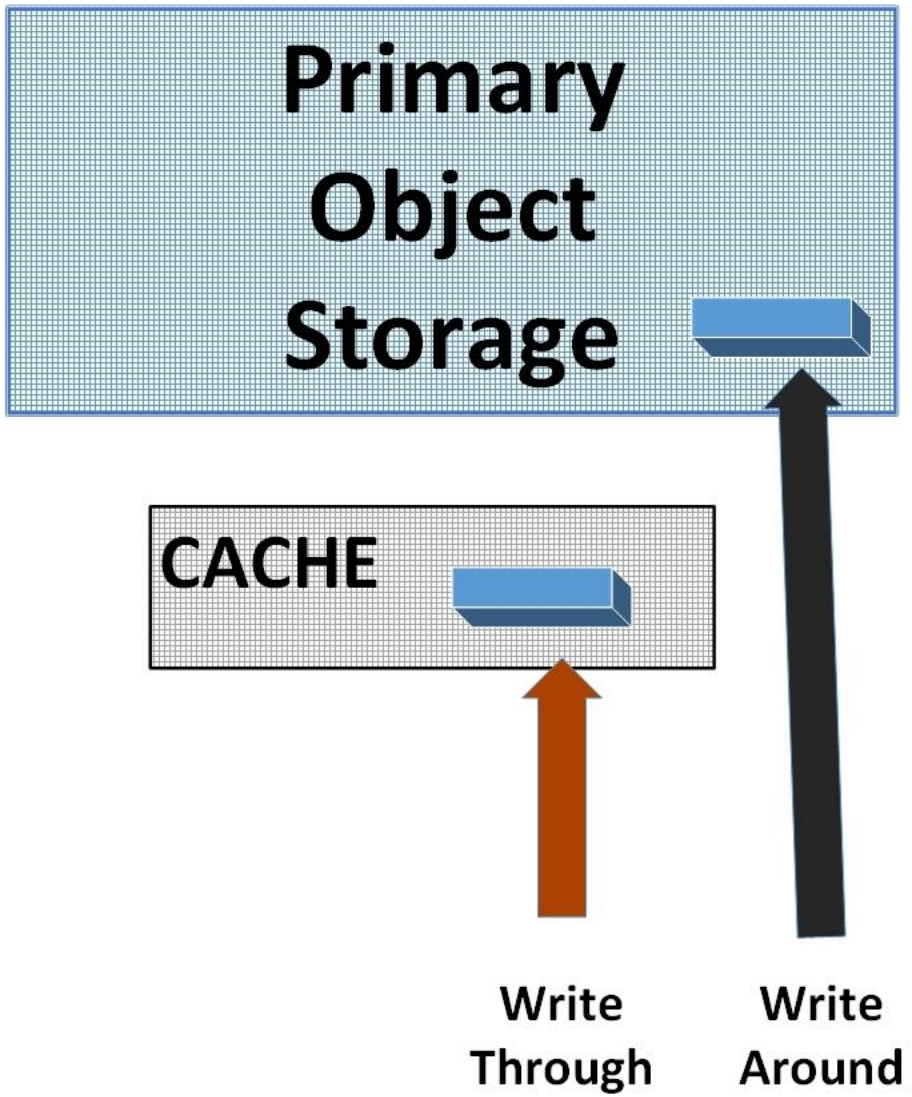
Cache writes

- **Write Through**
  - Update primary and cache copies *synchronously*
  - Write operation executes at the speed of the slow memory component
- **Write Around**
  - when Read after Write is unlikely
- **Write Back**
  - Update cache immediately; update primary storage eventually
  - Lazy write benefits many workloads



Write Back caching

- Write Back
  - Always Write to cache
    - Write service time ≅ Read Hit service time
  - Update cache immediately; update primary storage eventually
    - Asynchronous writes to the backing store requires that the cache is non-volatile storage
    - *Lazy write* benefits many workloads

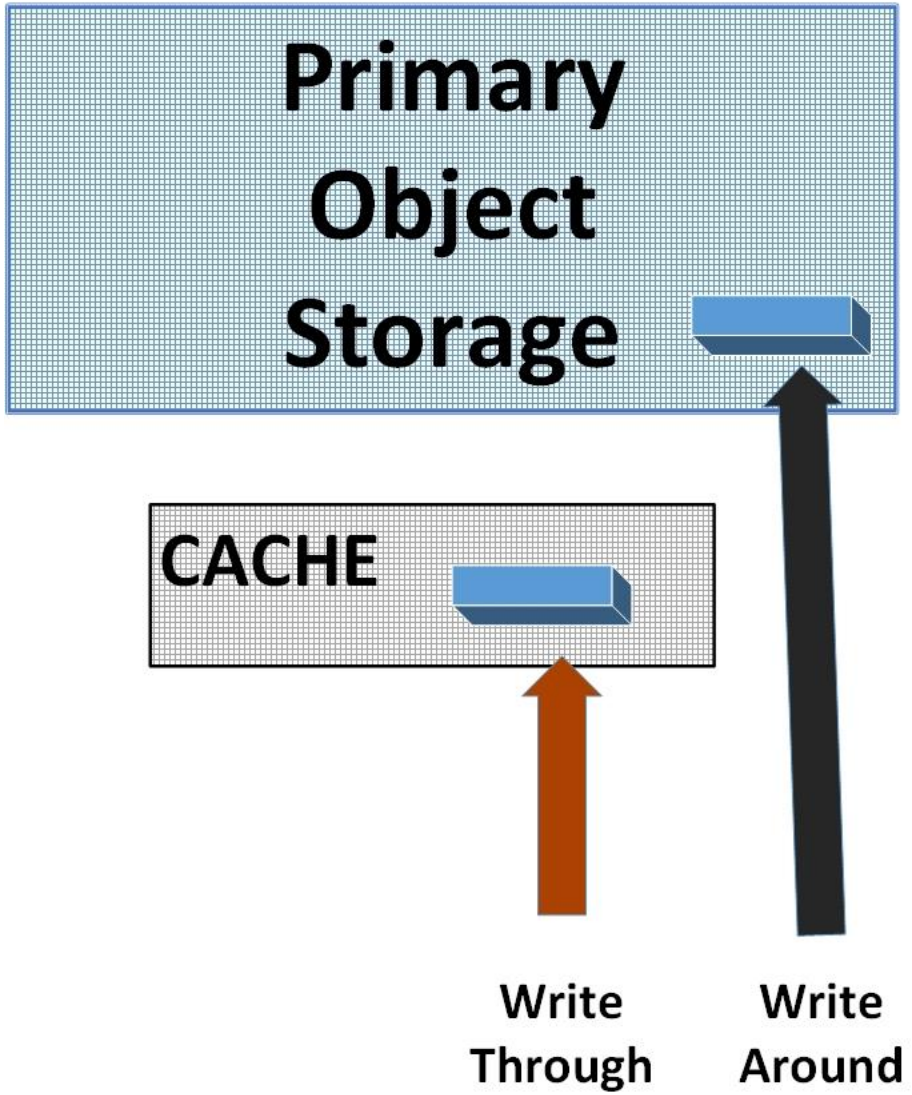


Instrumenting the cache

- rate of Reads : Writes
- rate of Hits : Misses

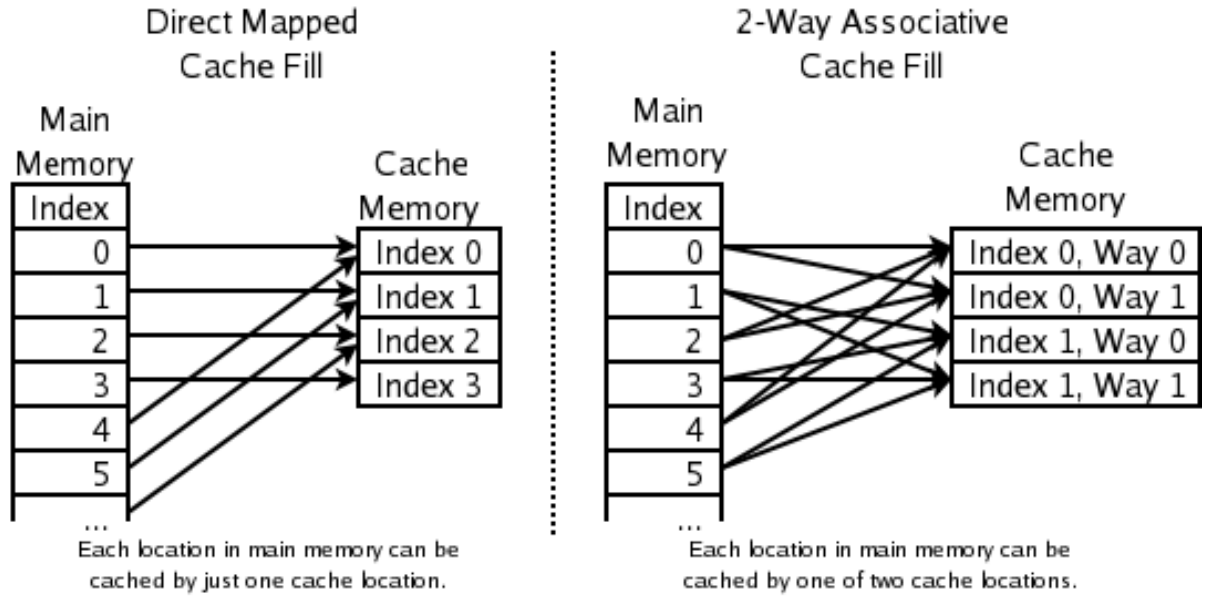
- Latency (Reads, Writes, Hits, Misses) to calculate the weighted average service time

- Memory management overhead
  - replacement policy trigger rate
  - object Residency time: age of the last removed object



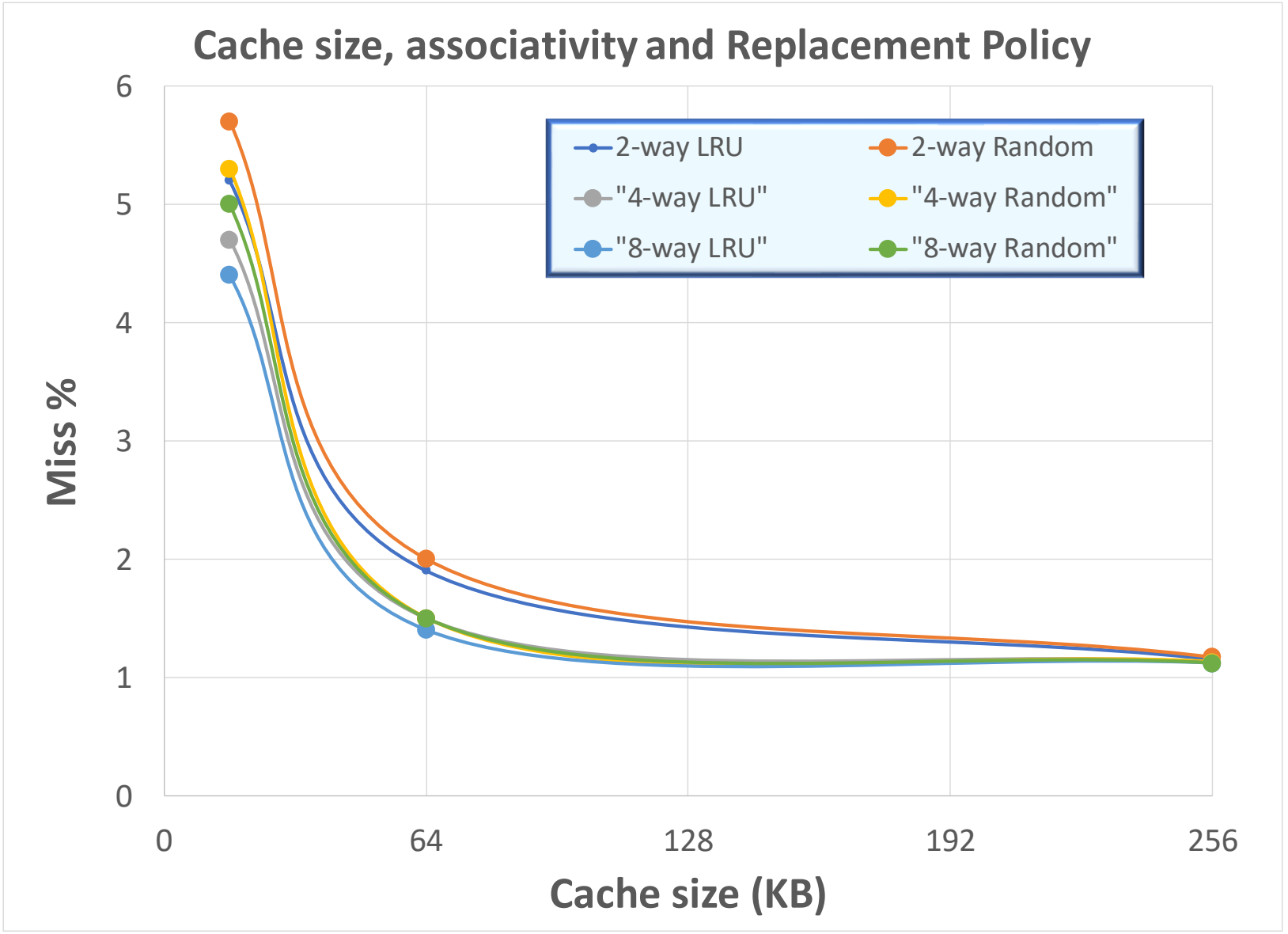
CPU caching

- Direct mapped
- n-way set associative Cache
  - Replacement policy
    - Random
    - (approximate) LRU
    - FIFO
- Types of misses:
  - Compulsory (on first access; a cold start following a context switch)
  - Capacity
  - Collisions



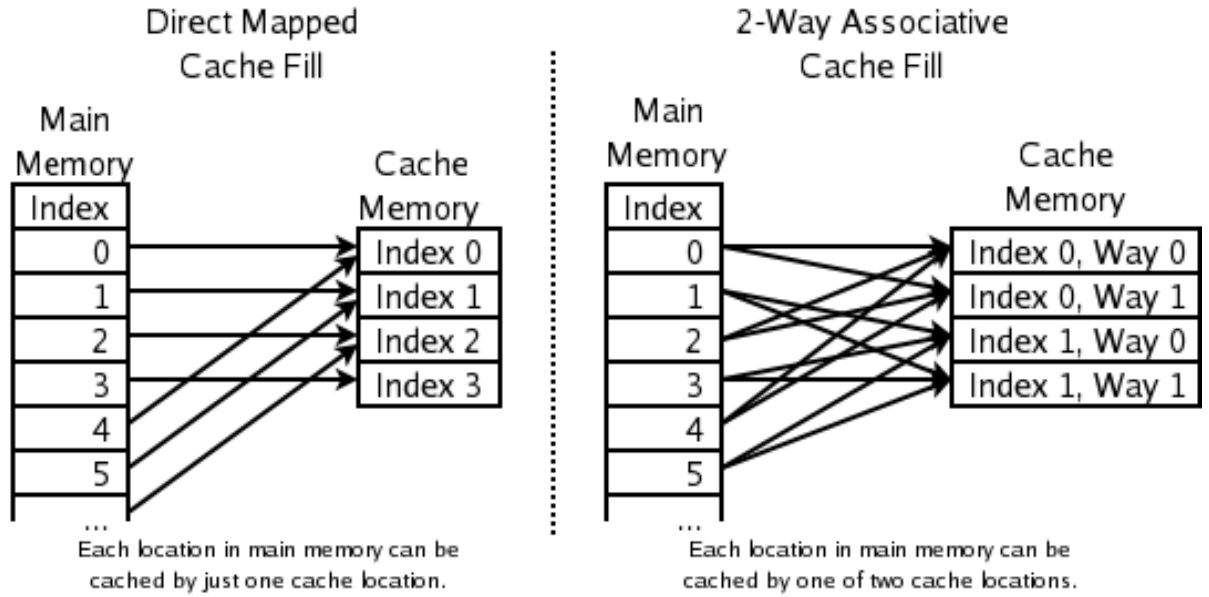
CPU caching

- n-way set associative Caches don't experience quite as many collisions



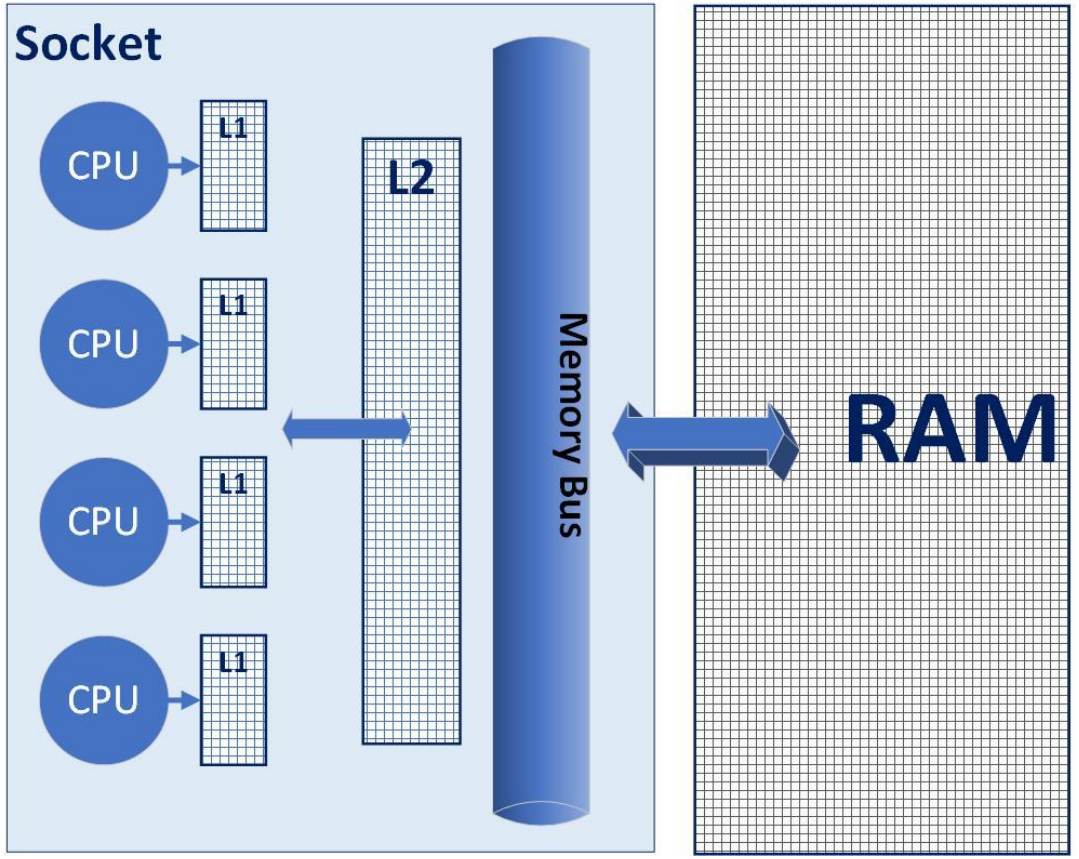
CPU caching

- Significant stall of CPU instruction pipeline on a miss
  - Memory Wall
  - motivation for o-o-o execution and simultaneous multithreading
- Unified or Separate Instruction and Data caches



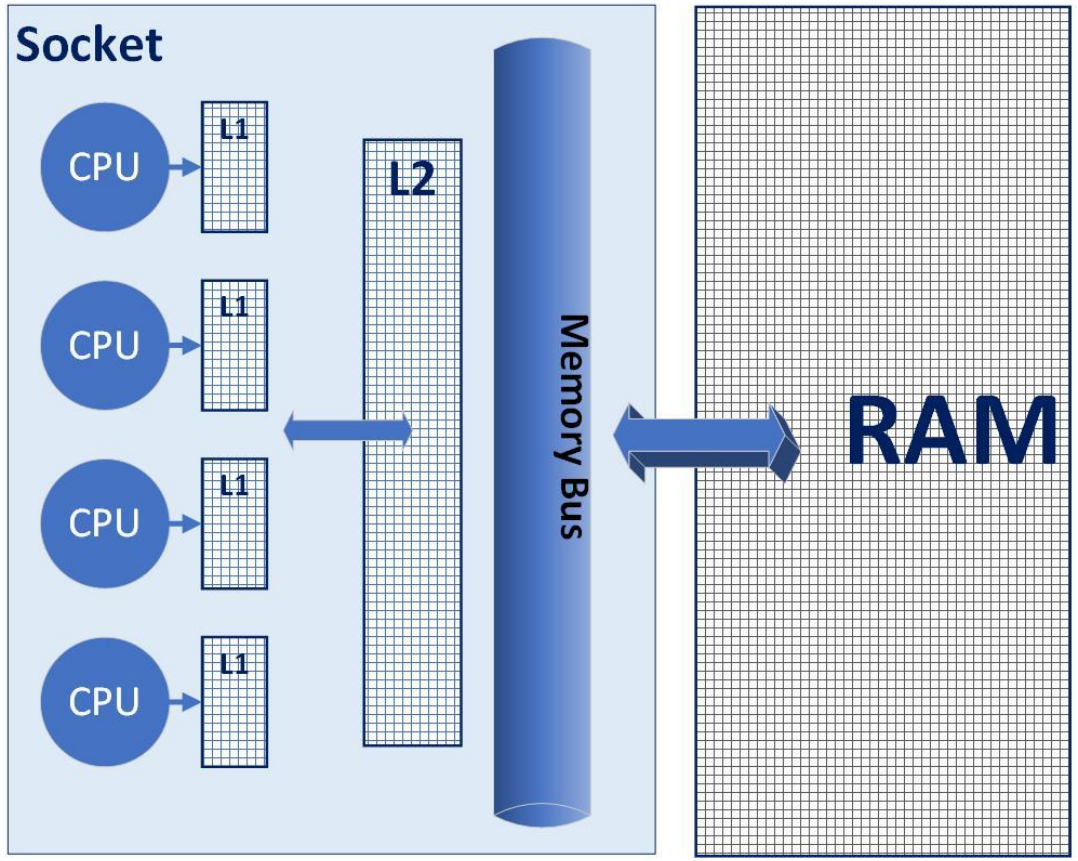
CPU caching

- Main memory is a resource shared across CPU cores, while caches are dedicated to processor cores and/or sockets
- Write-back caching with coherence
- False Sharing



CPU caching

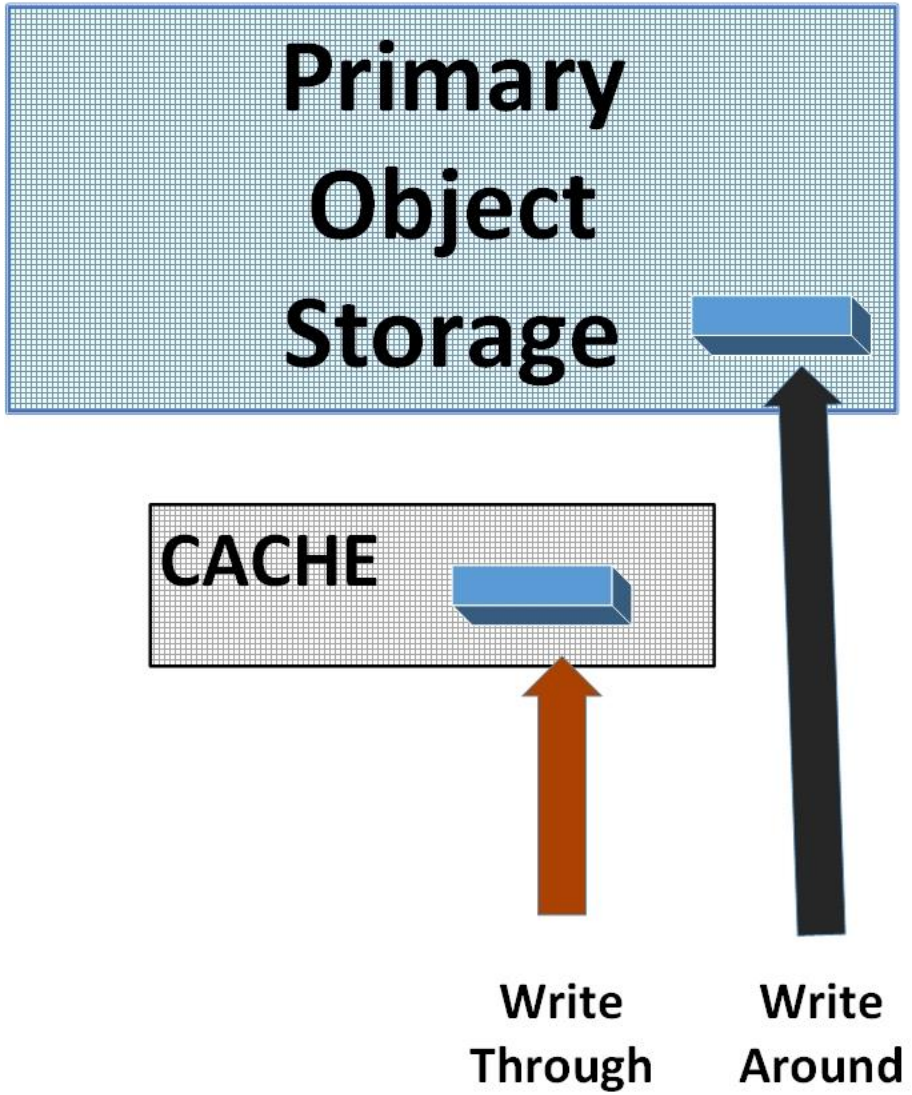
- Multiple levels of Cache in a hierarchy
  - Latency is a function of distance (signal propagation delay)
  - Fewer misses, but increases complexity
  - LRU or approximate LRU
- Context switches and Processor node affinity
  - (cold vs. warm starts)
- Additional caches
  - e.g., Translation Lookaside buffer
  - e.g., CISC ⇨ μinstruction cache



Cache coherence

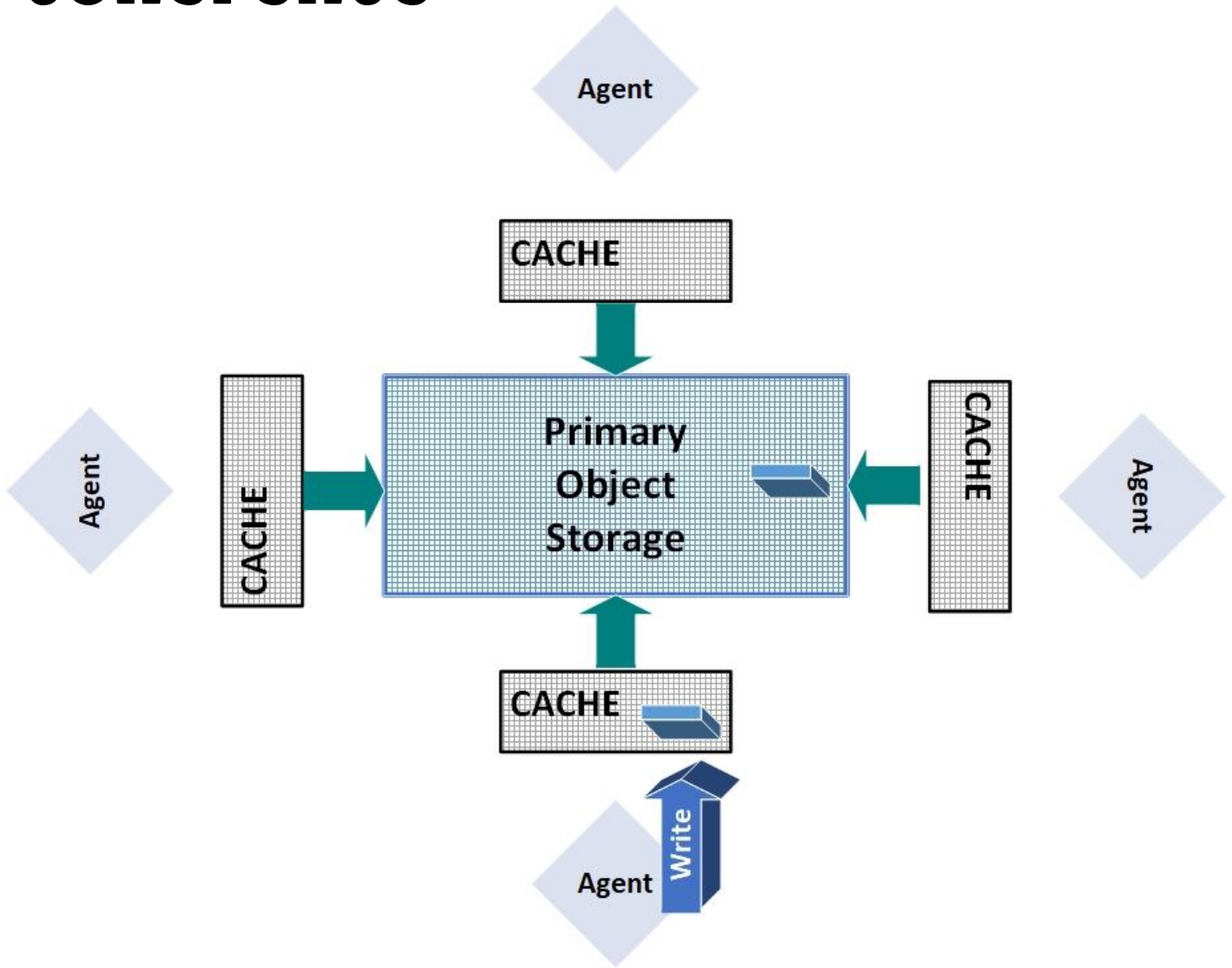
- When multiple threads have concurrent access to primary storage
- e.g., primary storage is shared and each active thread has its own private cache

- Ensuring that each concurrent execution thread has consistent Read access to the most current data is known as the *cache coherence* problem



Cache coherence

- **Shared Write Back cache**
  - **Writes to memory must be propagated to the other caches**
  - **Preserve the order of Write operations across all agents**
    - **atomic**
    - **serializable**
- **Consistent**



Cache coherence

Intel MESI protocol

- **Each cache line is tagged as belonging to one of four mutually exclusive states:**

M	Modified	Dirty Data is stored in this cache
E	Exclusive	Clean Data is stored in one cache
S	Shared	Clean Data is stored in more than one cache
I	Invalid	Data in the cache is invalid; upon access, the contents must be refreshed from the Primary store

Cache coherence

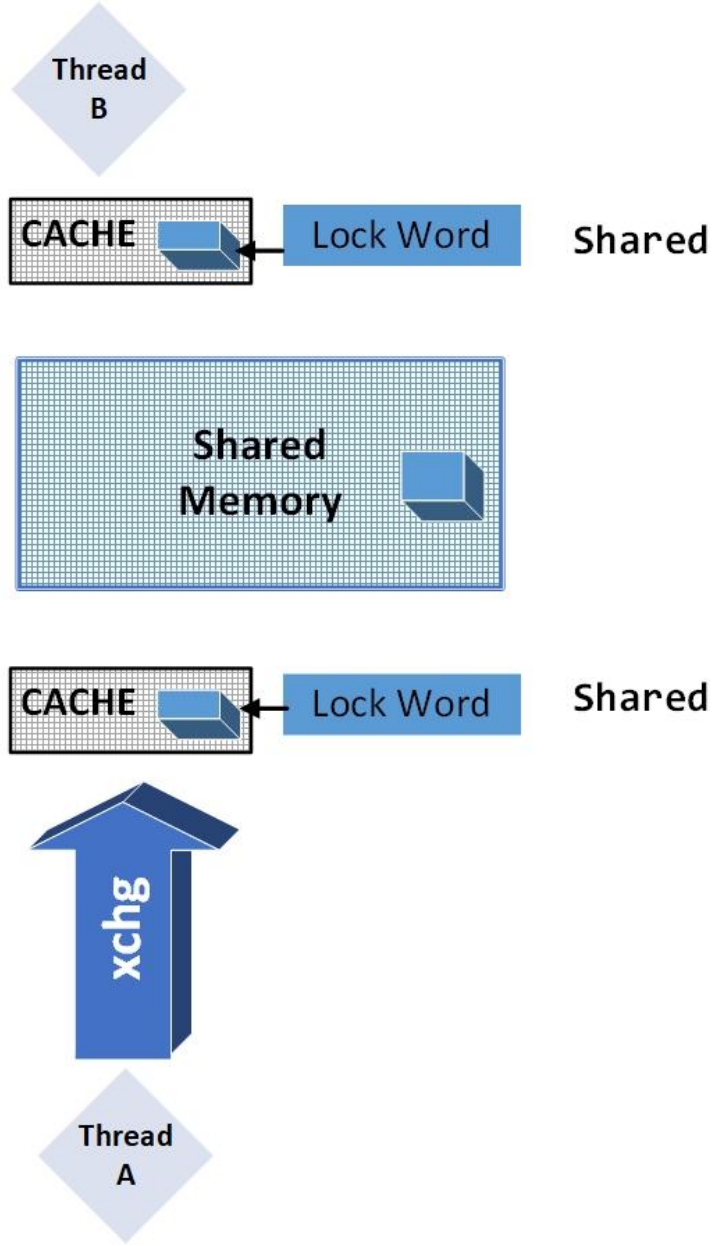
- **Spinlock example**
  - **e.g., the NTFS Lock**

```
spin_lock:
        mov         eax, 1
        xchg        eax, [lockword]
        test        eax, eax
        jnz         spin_lock
        ret

spin_unlock:
        xor         eax, eax
        xchg        eax, [lockword]
        ret
```

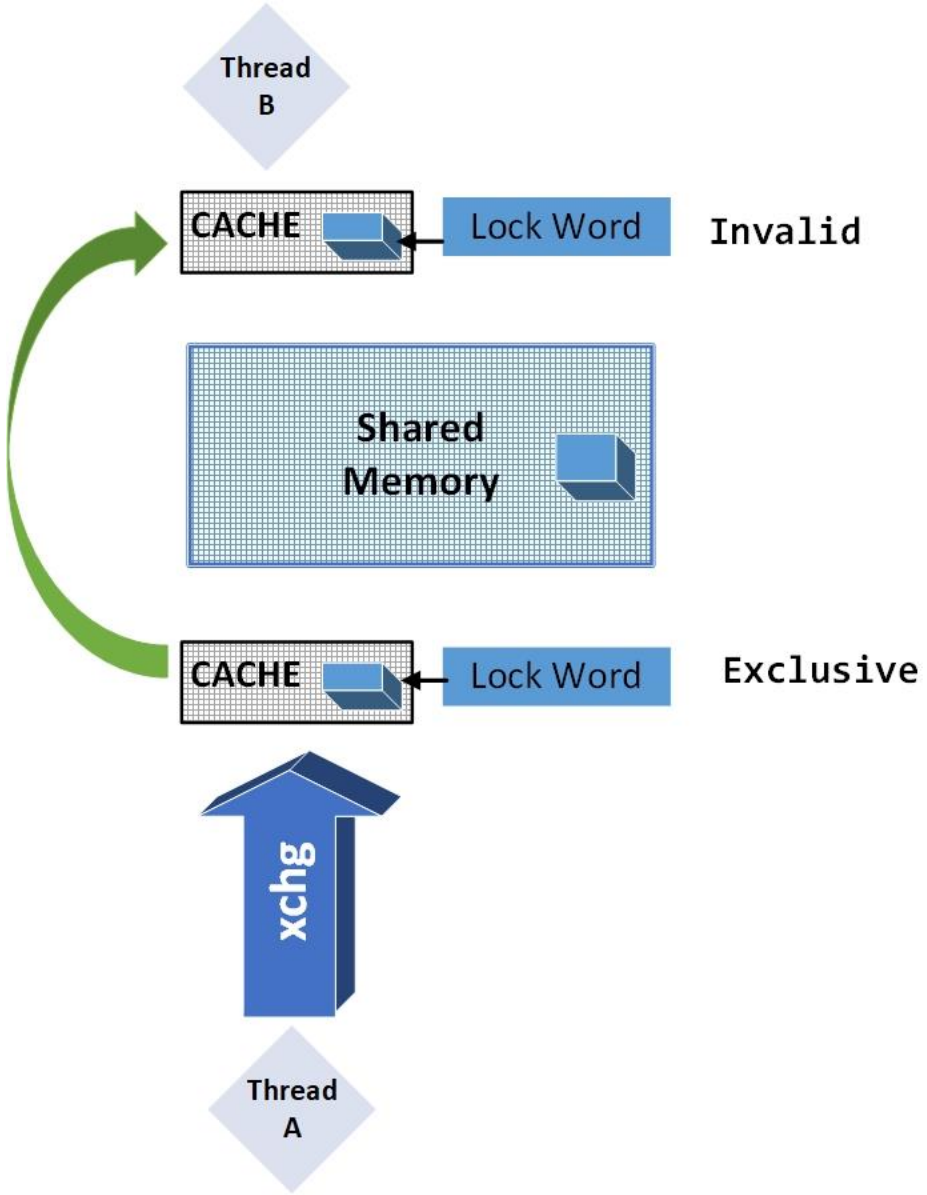
Cache coherence

- **Spinlock example**
  - **e.g., the NTFS Lock**
  - **Initially, the Lock Word, which contains a "1" indicating the LOCK is currently held, resides in two caches in a SHARED state**
  - **Thread B is in a Busy Wait, executing its `spin_lock` code until the Lock is released**
  - **Thread A holds the lock and issues a Compare and Swap instruction (`xchg`) in its `spin_unlock` routine to Release it.**



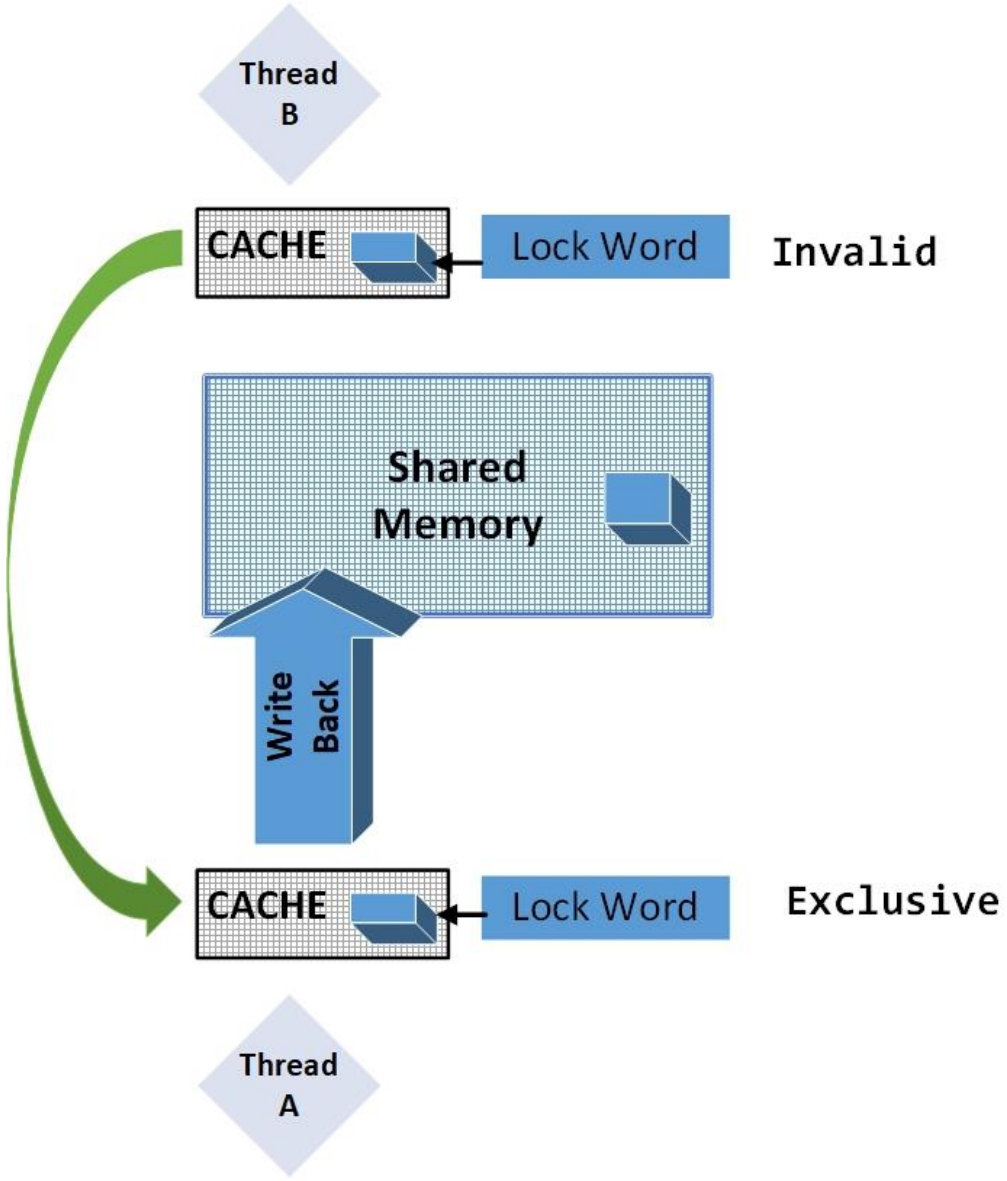
Cache coherence

- **Spinlock example**
  - **The cache line containing the Lock Word in Thread A's cache changes to an `EXCLUSIVE` state**
  - **Thread B, listening to the Shared Memory bus, transitions its corresponding Cache line to the `INVALID` state**
  - **Forcing Thread B to refresh the Cache line from shared Memory the next time the `xchg` instruction executes**



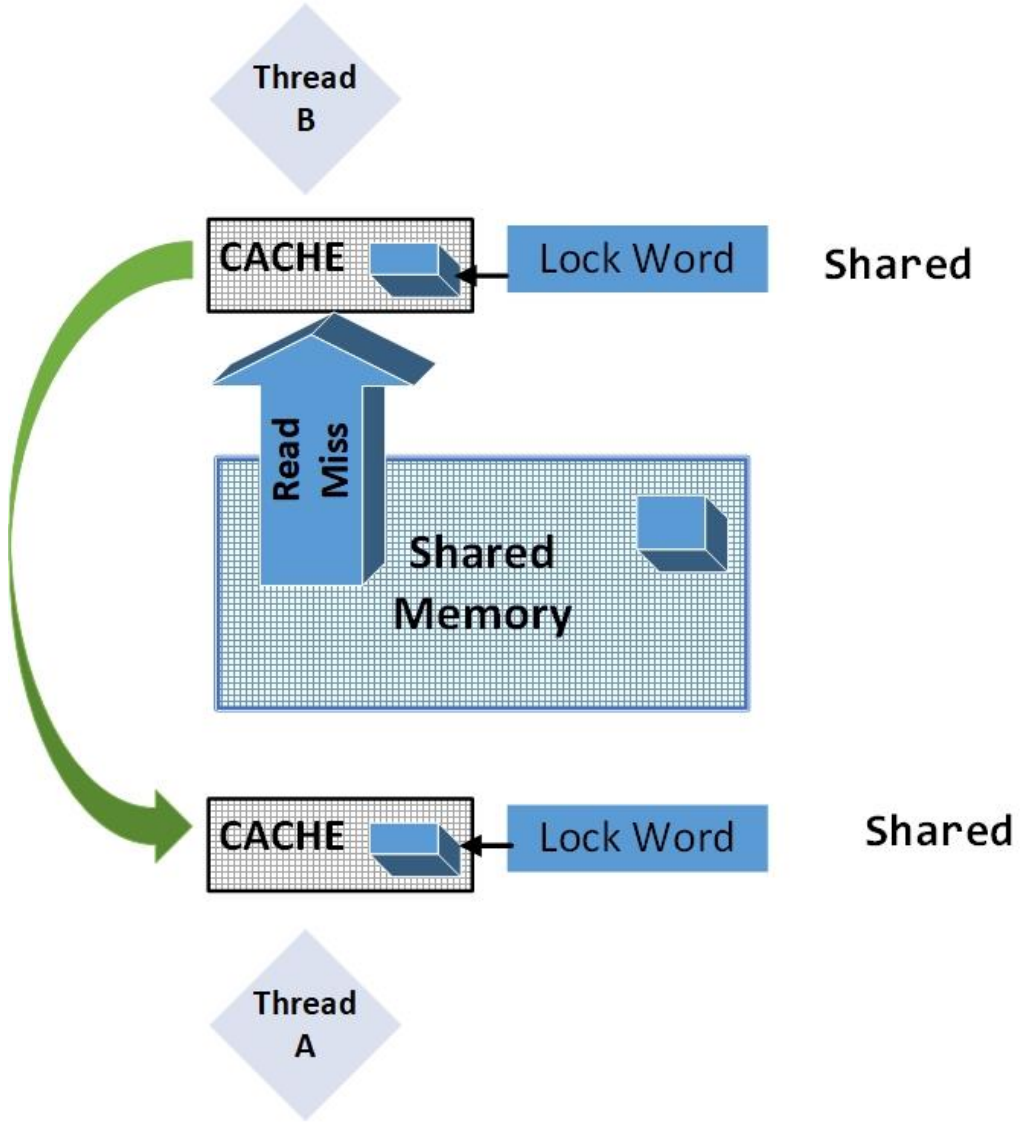
Cache coherence

- **Spinlock example**
  - **Thread A, listening to the Shared Memory bus, pushes the contents of its Cache line to shared Memory**
  - **Thread B's fetch from shared Memory is then allowed to execute**



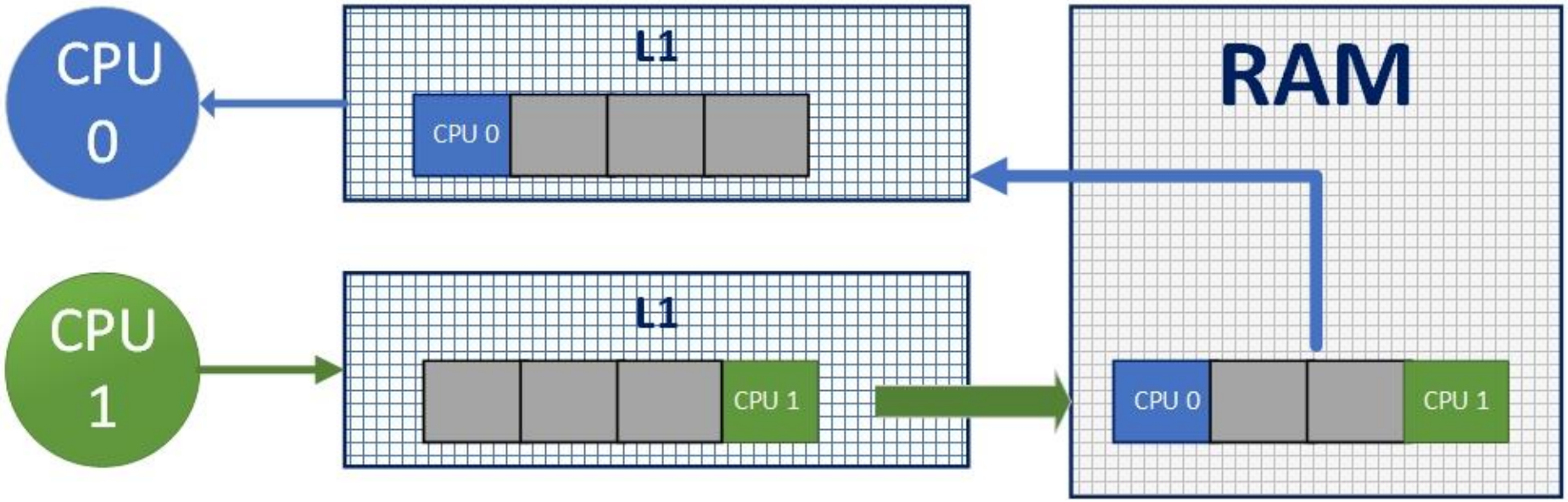
Cache coherence

- **Spinlock example**
  - **Following Thread B's fetch from shared Memory, its Cache line is tagged SHARED**
  - **Thread A, listening to the Shared Memory bus, changes its Cache line state to SHARED**



False sharing

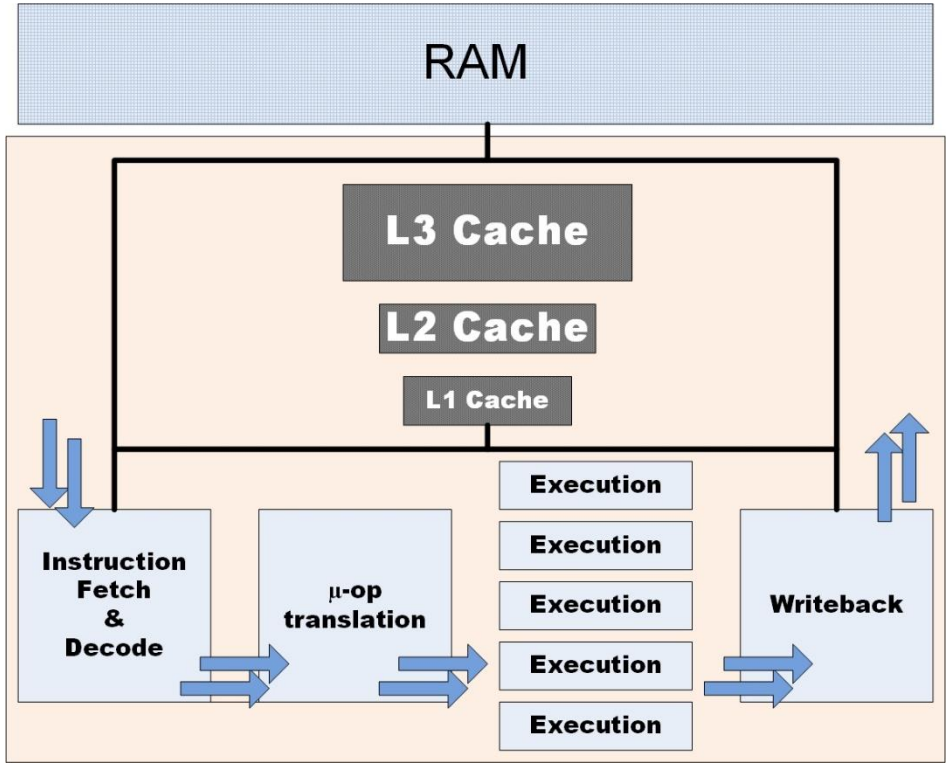
- On an SMP, false sharing occurs when threads on *different* processors modify variables that reside on the *same* cache line.
- Example: Two different Lock Words allocated on the same cache line



See https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

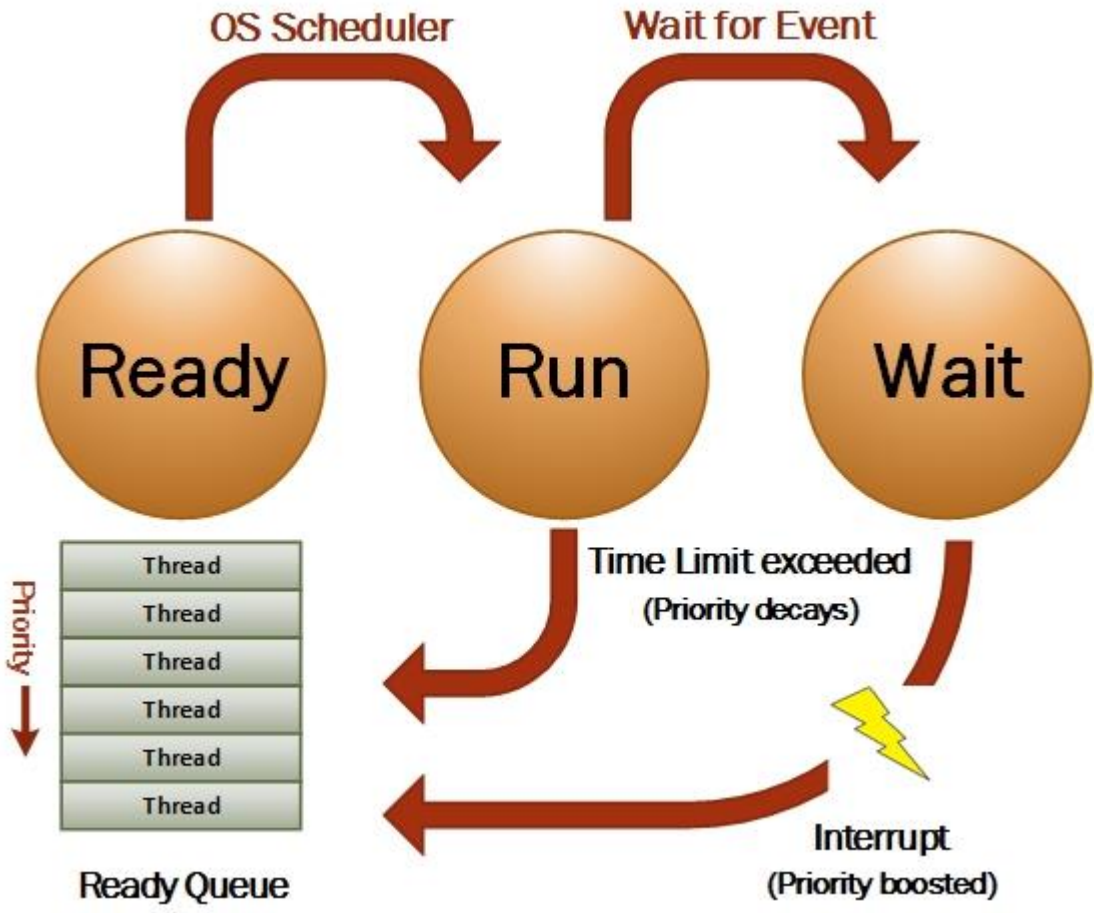
CPU cache and pipelining

- Additional caches are utilized at different pipeline stages to increase instruction execution rates
  - e.g., instruction : μ-op translation
  - Translation Look-aside Buffer (TLB)
    - caches recent virtual address : physical address mappings
    - TLBs reduce the cost of virtual memory translation to a minimum
    - A context switch that dispatches a new thread from a different process loads a pointer to a new set of Page Tables, effectively clearing the TLB



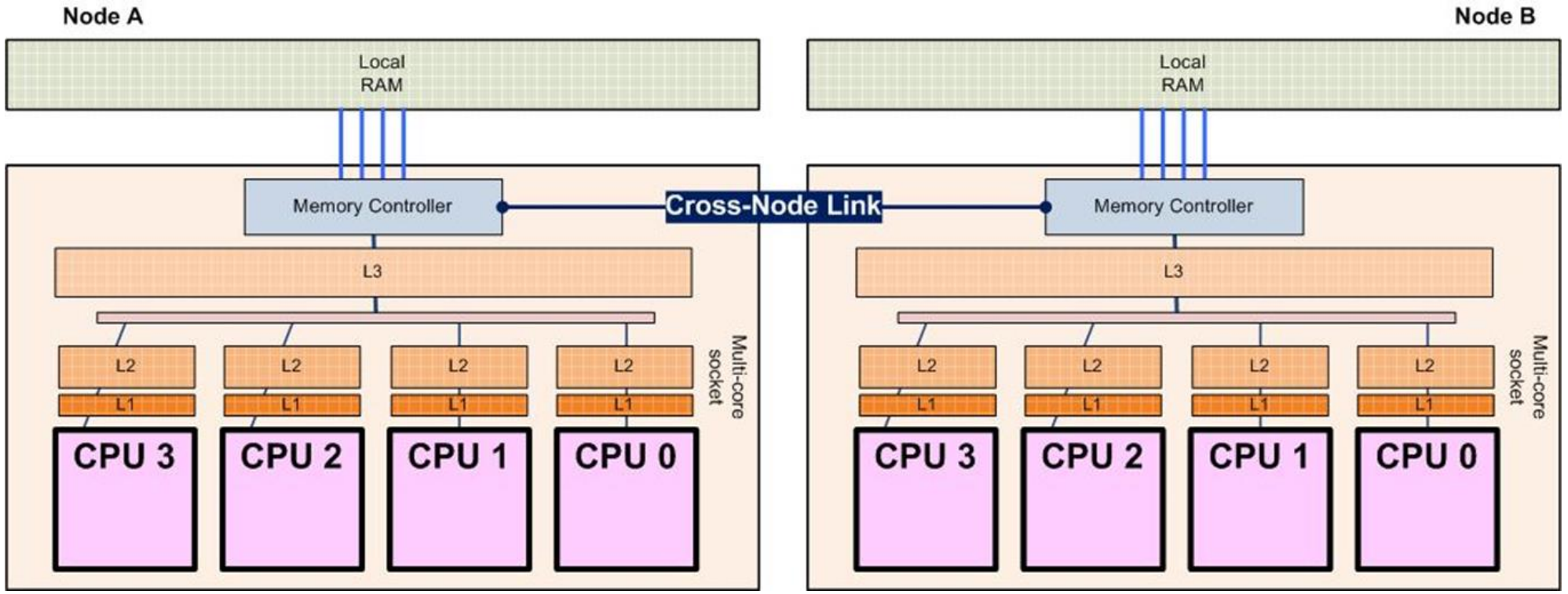
CPU cache: Thread scheduling

- Multiprogramming
  - Dispatch a waiting Thread from the Ready Queue when the current thread enters an IO Wait
  - time-slicing
- Processor affinity
  - remember the processor ID from the previous dispatch
  - “Ideal processor”
  - increase the probability of a cache warm start if the thread’s ideal processor is idle



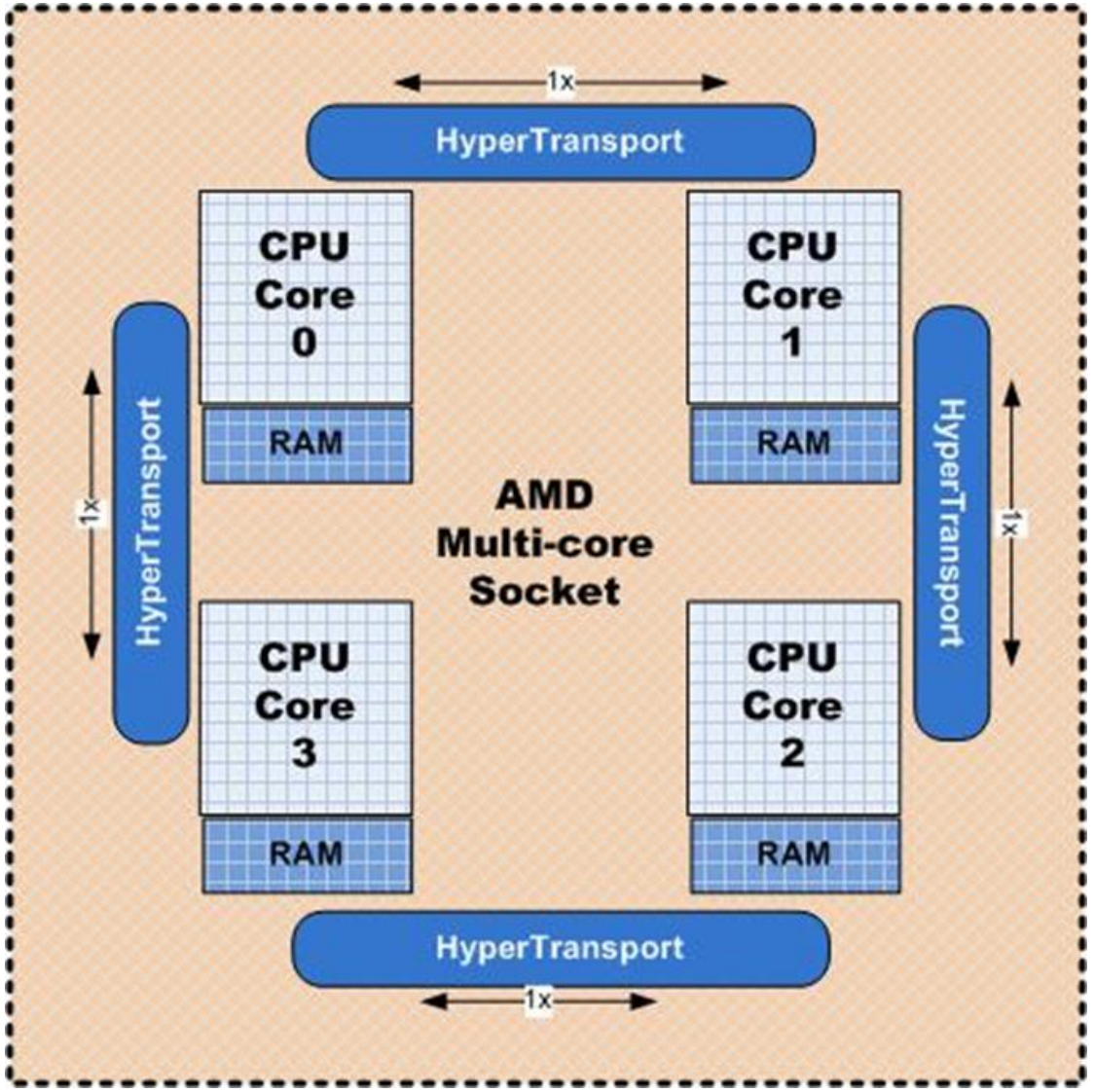
CPU Cache and NUMA

- Multi-socket servers have ccNUMA performance characteristics
  - cache-coherent Non-uniform Memory Access



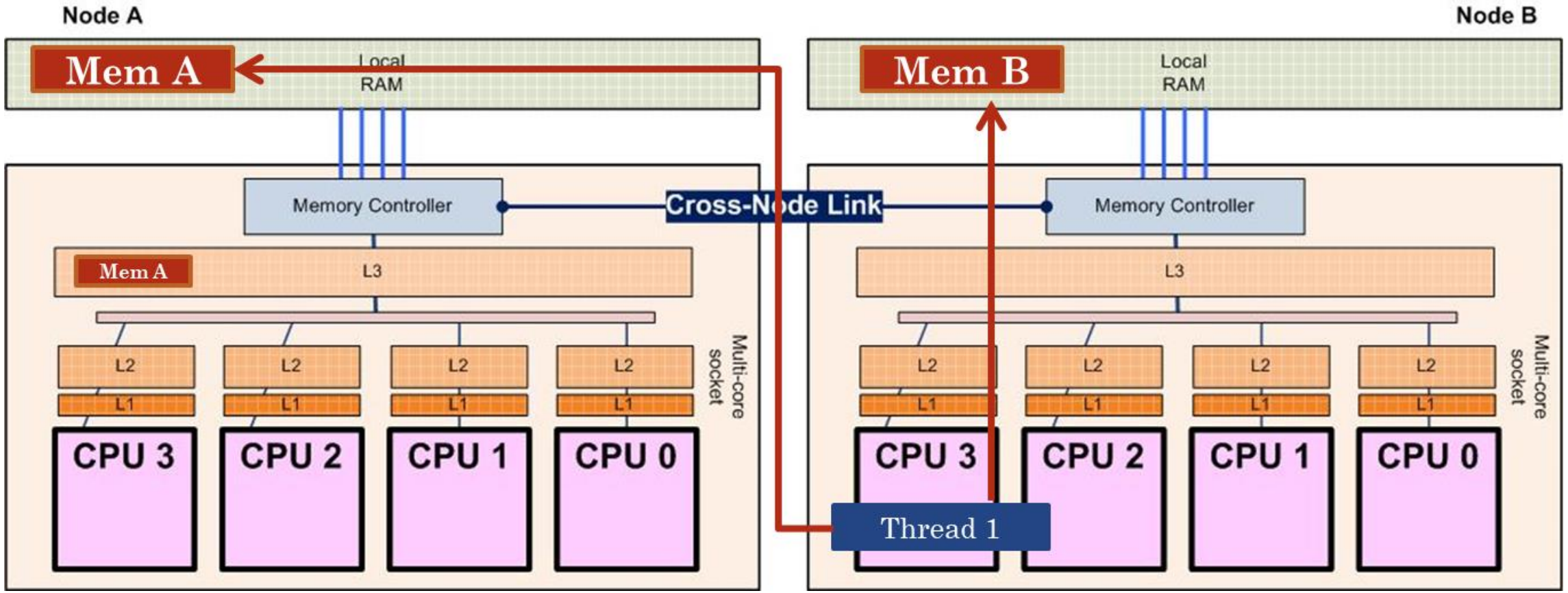
CPU Cache and NUMA

- Accesses to **remote** memory take longer than **local** memory
- Developers encounter NUMA characteristics on a single socket in some hardware architectures



CPU Cache and NUMA

- Accesses to **remote** memory take longer than **local** memory

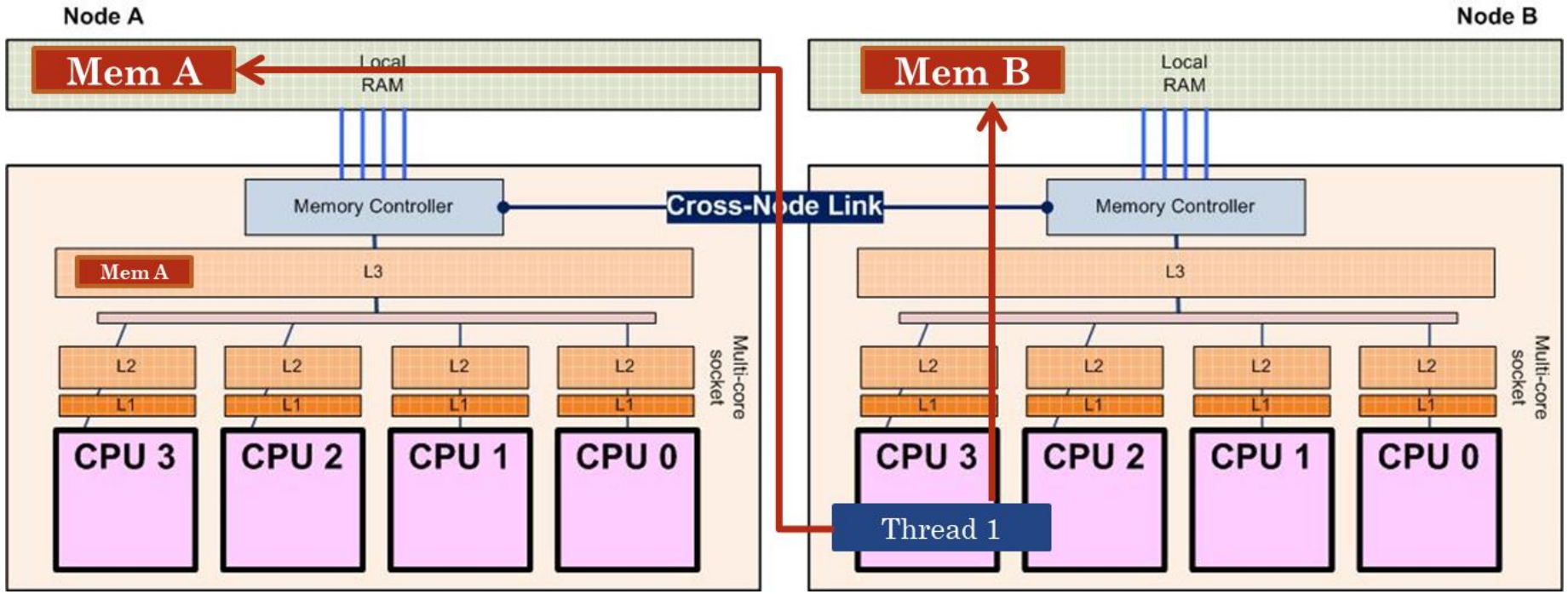


CPU cache: NUMA thread scheduling

- Processor “soft” affinity
- Processor node “soft” affinity
  - remember the node from the previous dispatch
  - “Ideal node”
  - reduce the probability of remote memory accesses
  - augmented by per Node memory management
  - Node affinity is a very important consideration in hypervisor virtual machine scheduling

CPU Cache and NUMA

- Accesses to **remote** memory take longer than **local** memory
- overheads vary greatly based on the cache coherence scenario
  - e.g., Remote memory cache hit vs. Remote memory cache miss



CPU Cache instrumentation

- Intel processors support an extensive set of internal performance counters
  - Documented measurement interface
  - Only a small number of the available counters can be active at any one time
  - Some are general purpose:
    - Instruction execution rate (Instructions retired)
    - Cache hit rates
  - Many are processor-specific and change from release to release

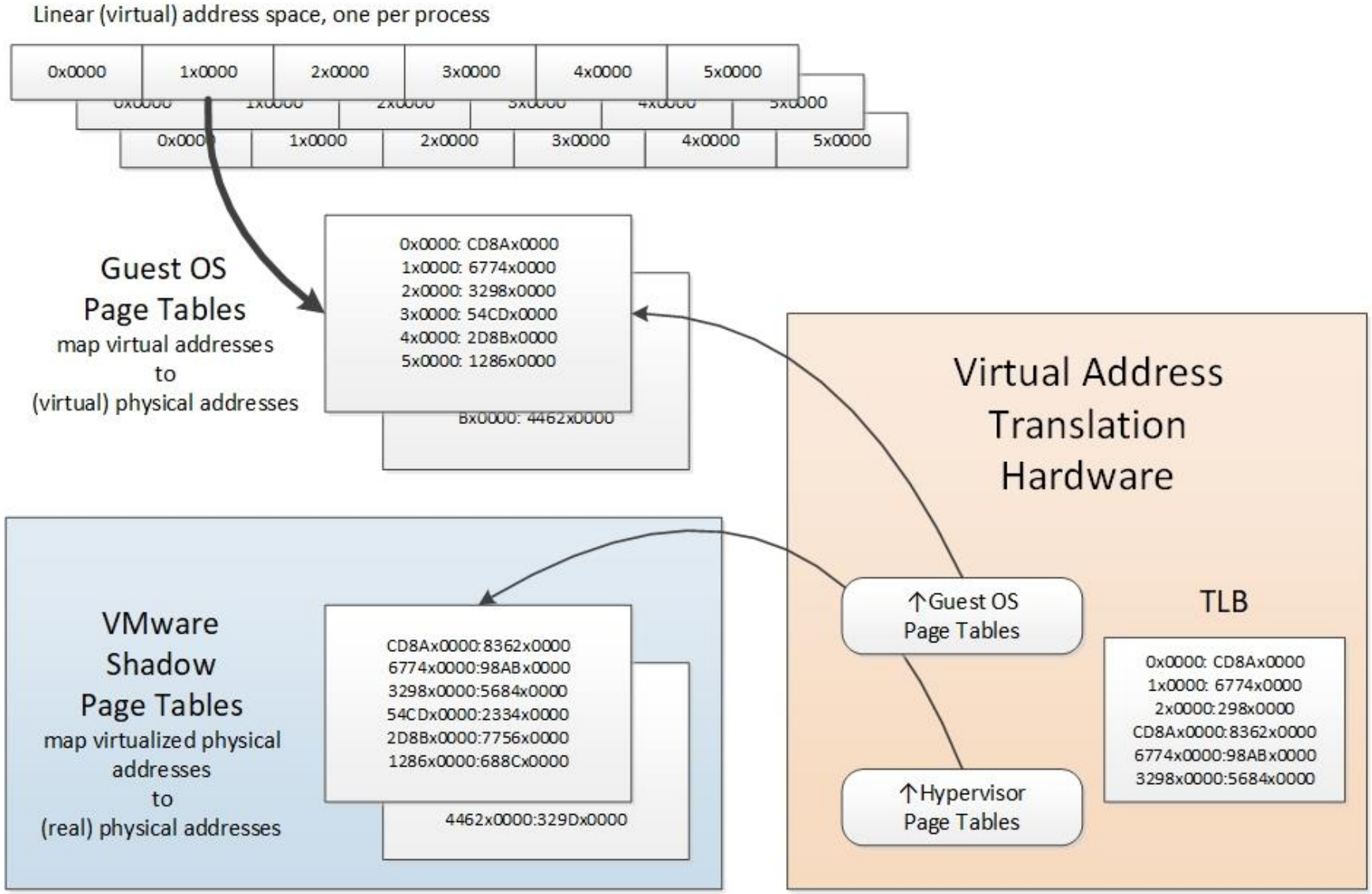
- Intel vTune performance tool

Hypervisor Memory Management

- Terminology
  - VM Host machine memory:
    - physical memory installed on the VM Host machine
  - Guest machine physical memory:
    - virtualized physical memory granted to the guest machine; looks like physical memory to the guest OS
    - tracked by a set of shadow Page Tables
  - Guest machine virtual memory:
    - per process virtual memory created by the guest OS and mapped to guest machine physical memory using Page Tables

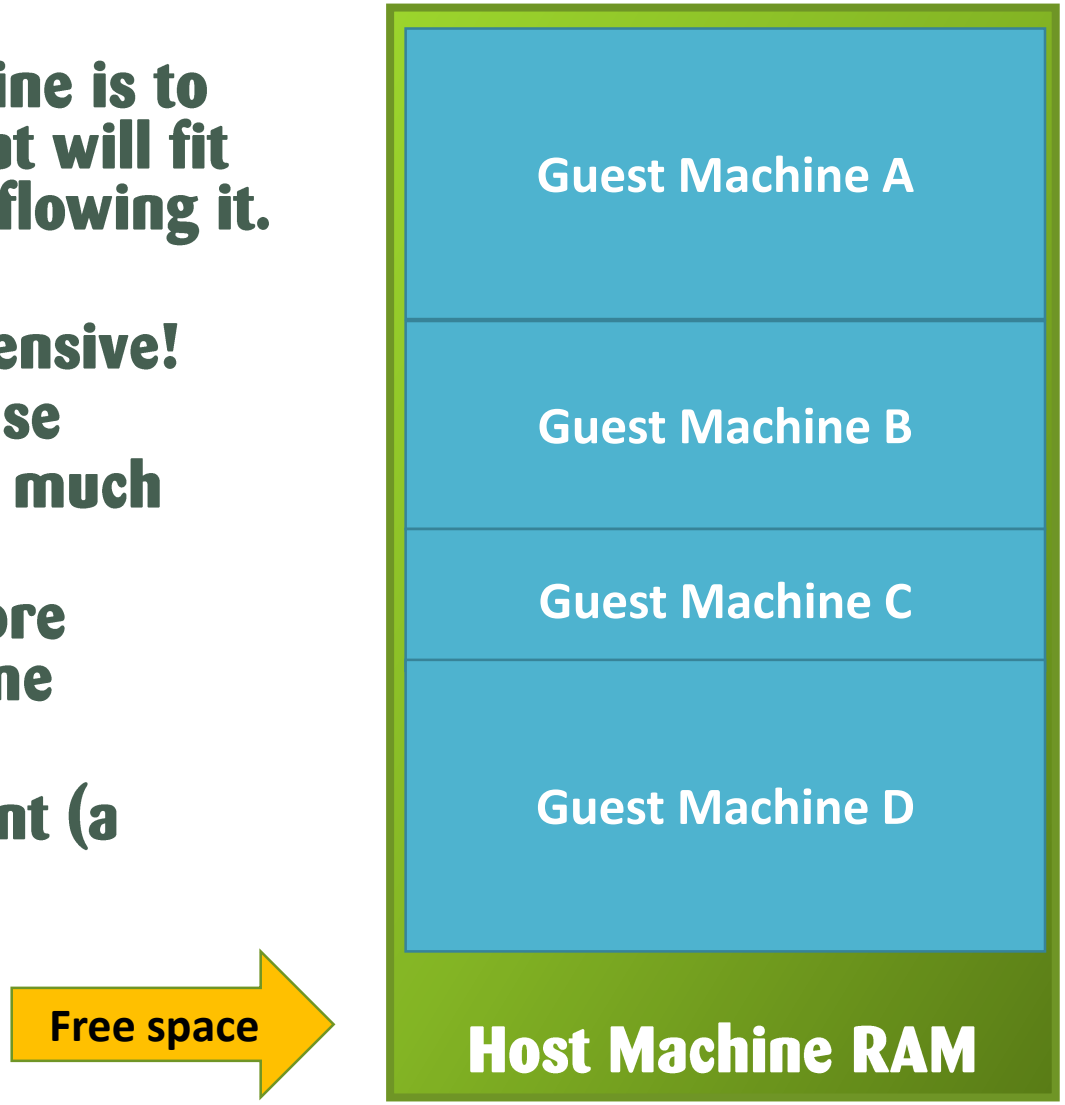
Hypervisor Memory Management

- Hardware virtual address translation uses both sets of Page Tables



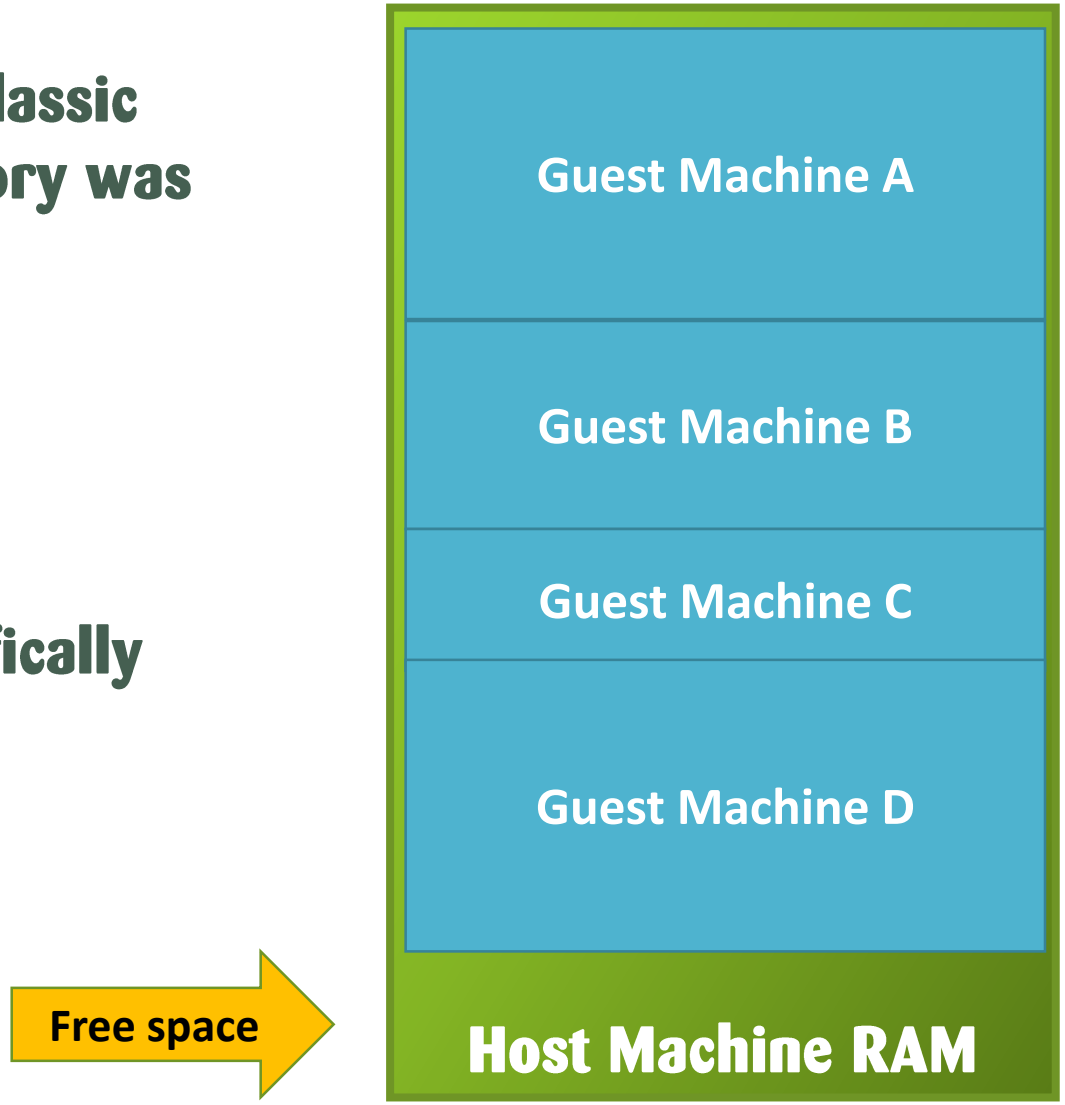
Hypervisor Memory Management

- The safe way to configure a VM Host machine is to run only the number of guest machines that will fit inside Host machine memory without overflowing it.
- Static memory allocation schemes are expensive!
  - machine memory is under-utilized because
    - Guest machines don't understand how much memory they need
    - Guest machines are often allocated more physical memory that they can consume
- Guest machine virtual memory management (a caching approach) is very dynamic!



Hypervisor Memory Management

- In the history of computing, this was the classic problem with early machines where memory was partitioned statically
  - e.g.,
    - OS/MFT
    - OS/MVT
- that virtual memory technology was specifically designed to address
  - MULTICS



Hypervisor Memory Management

- Early versions of VMware allowed customers (often Web site and Host machine vendors like GoDaddy) to run more VMs than would fit neatly into machine memory
  - Grant physical memory to a VM on a provisional basis
  - Provide paging/swapping mechanisms to relieve the pressure on a Host machine whose memory is "over-committed"

Hypervisor Memory Management

- Grant physical memory to a VM provisionally
- Provide paging/swapping mechanisms to relieve the pressure on a Host machine whose memory is “over-committed”

without a Paging file &
without page usage data

Hypervisor Memory Management

- Memory Ballooning
  - VMware does not duplicate the very dynamic memory management facilities of the guest OS due to overhead considerations
    - On initial access by a guest OS to a Page granted to that guest Machine, the VMware Memory Manager turns on the valid bit in the shadow Page Table entry (PTE)
    - VMware estimatese guest machine memory usage using sampling, periodically flipping the valid bit in a small number of shadow PTEs and then seeing if they are accessed in the next interval
      - Designed to identify idle Guest machines that can swapped out of machine memory completely: idle machine tax
      - There is an optional swap file, but no paging file(s)!

Hypervisor Memory Management

- **Memory State***

State	Value	Free Memory Threshold	Reclamation Action
High	0	≥ 6%	None
Soft	1	< 6%	Ballooning
Hard	2	< 4%	Swapping to Disk or Pages compressed
Low	3	<2%	Blocks execution of active VMs > target allocations

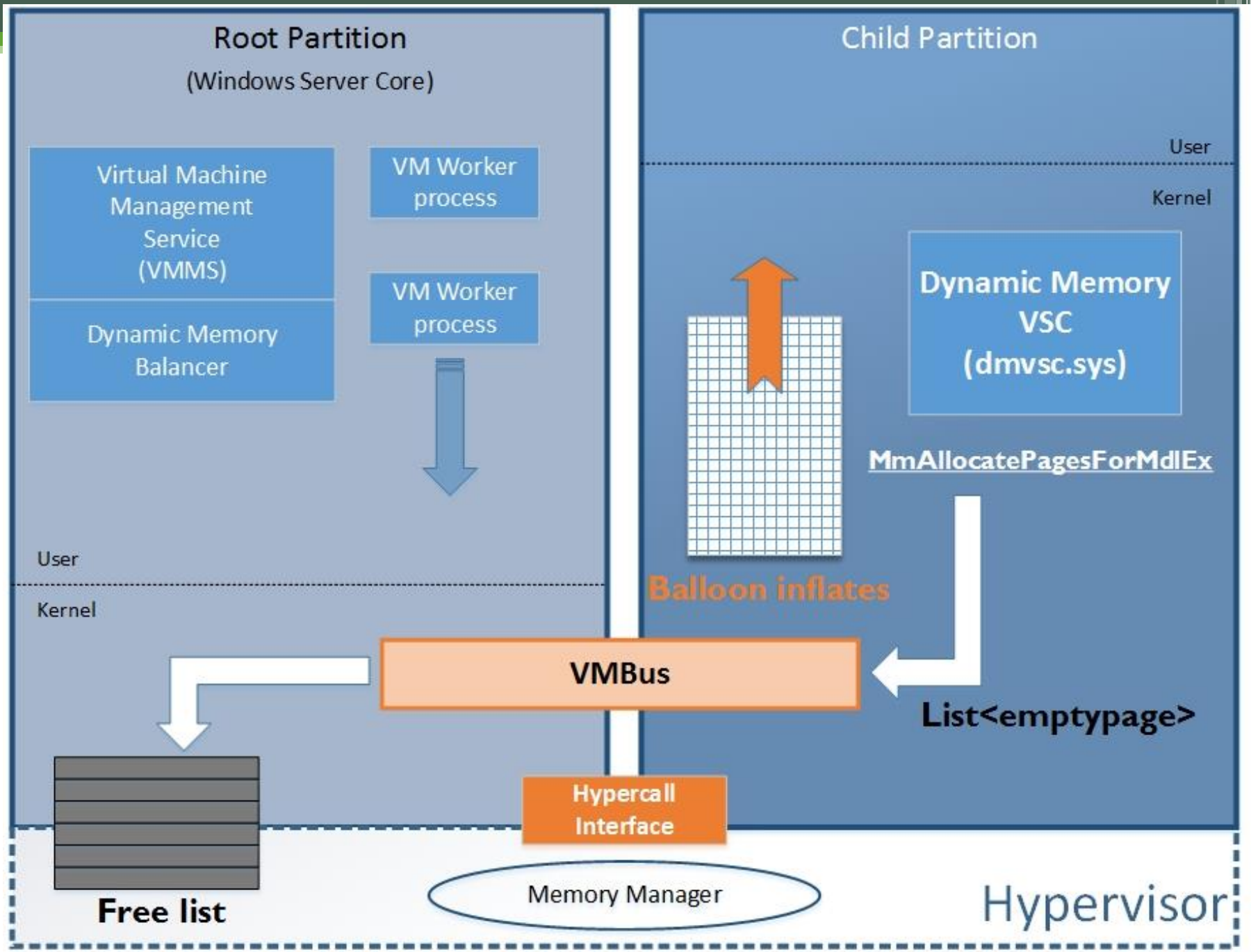
- see "Understanding Memory Resource Management in VMware® ESX™ Server" white paper

Memory Ballooning

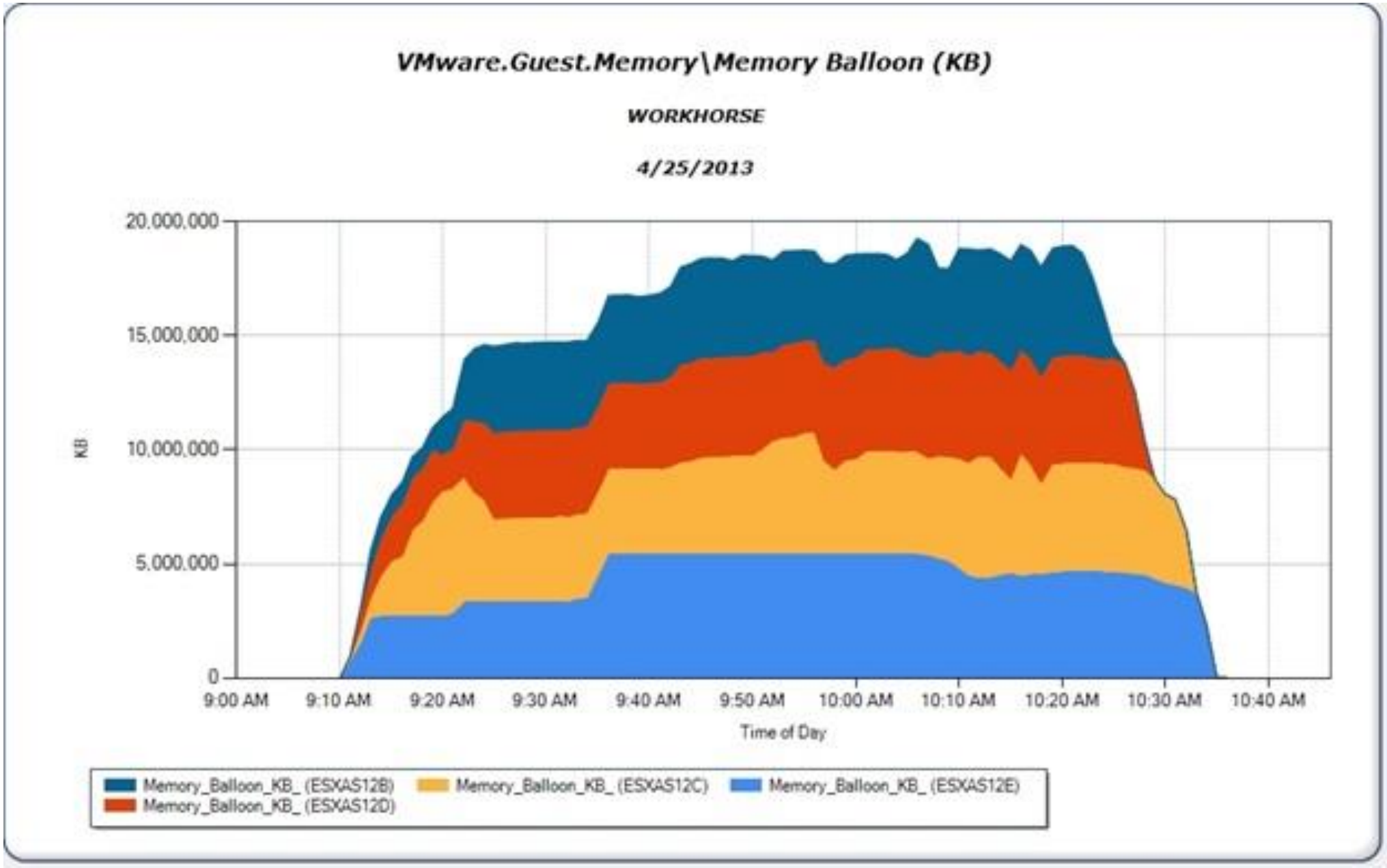
- When there is a shortage of available machine memory, the Hypervisor uses “ballooning” to push the decision about which allocated virtual pages can be replaced down to the guest OS, which maintains page usage data
- a management thread inside the guest OS “inflates” its memory balloon, which are physical pages the thread acquires that are empty, idle, and exempt from guest OS page replacement
- if ballooning results in a shortage of pages inside the guest OS, it has recourse to its page replacement algorithm

Memory Ballooning

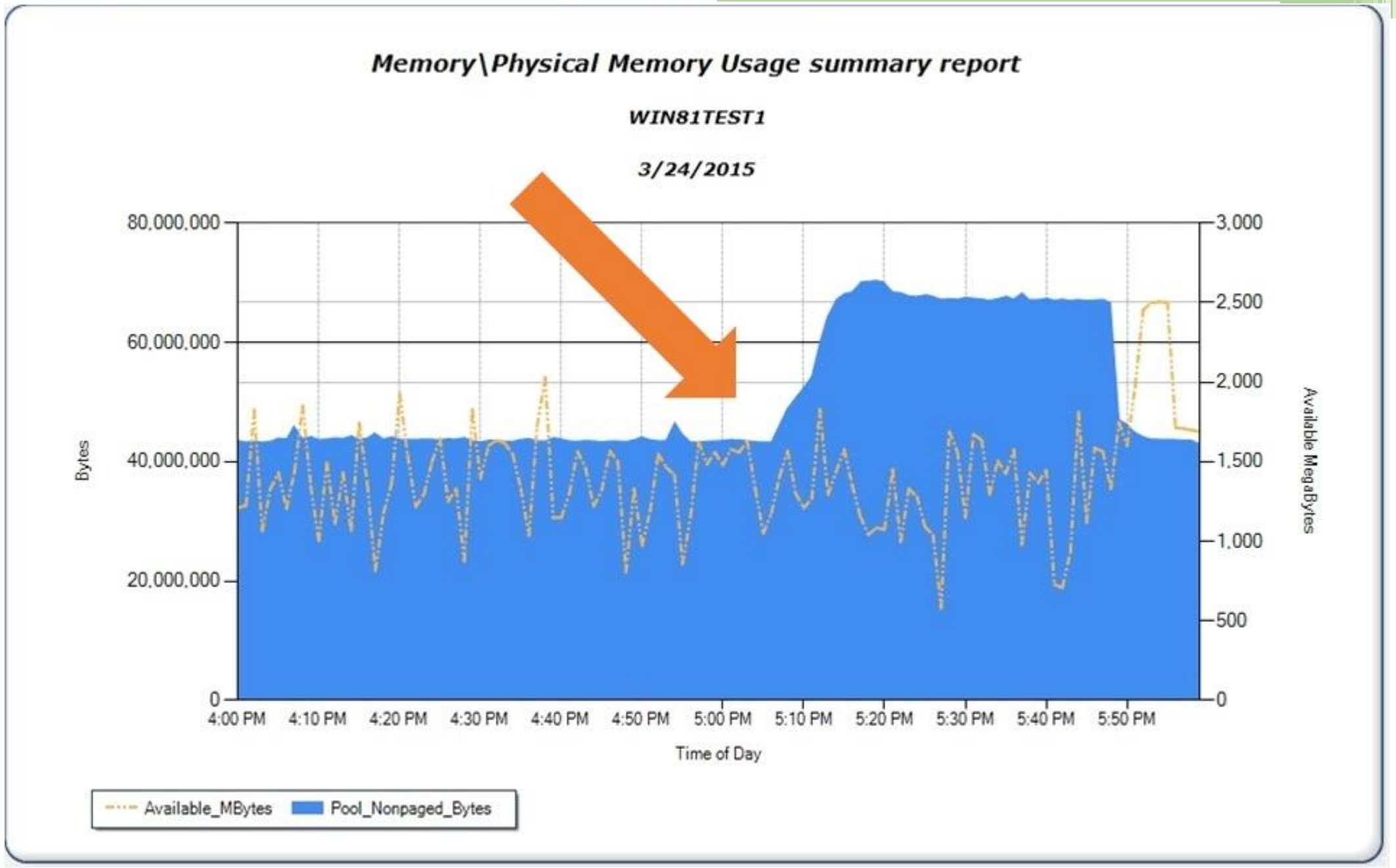
- Pages pinned in memory by the balloon driver are reported to the Hypervisor, which then makes them available to grant to other guest Machines on demand



Memory Ballooning



Memory Ballooning



Virtual Machine memory balancing in Hyper-V

- on Over-committed VM Host machines
  - Hypervisor does not have direct access to guest machine internal performance indicators
  - With one notable exception of a proprietary “enlightenment” used by the Hyper-V Memory Manager
  - Instead, manual tuning knobs are provided
    - Scheduling priority settings
    - QoS reservations and limits
  - Crude controls that are difficult to implement (trial & error)
  - Given the size and complexity of the configurations SysAdmins must manage, these tuning options are poor alternatives to goal-oriented control systems that have access to guest machine feedback

Virtual Machine memory balancing in Hyper-V

- when memory is over-committed on Hyper-V Host machines
  - Hyper-V attempts to equalize **Memory Pressure** across all Windows VMs with the same dynamic memory allocation priority
    - an "enlightenment" used by the Hyper-V Memory Manager
  - Memory Pressure is a **Memory contention index** ($^{V}/_{R}$):

$$\frac{guest\ machine\ \boldsymbol{Committed\ Bytes}\ *100}{current\ machine\ memory\ allocation}$$

  - because the likelihood of guest machine **paging** increases as **Memory Pressure** >> 100

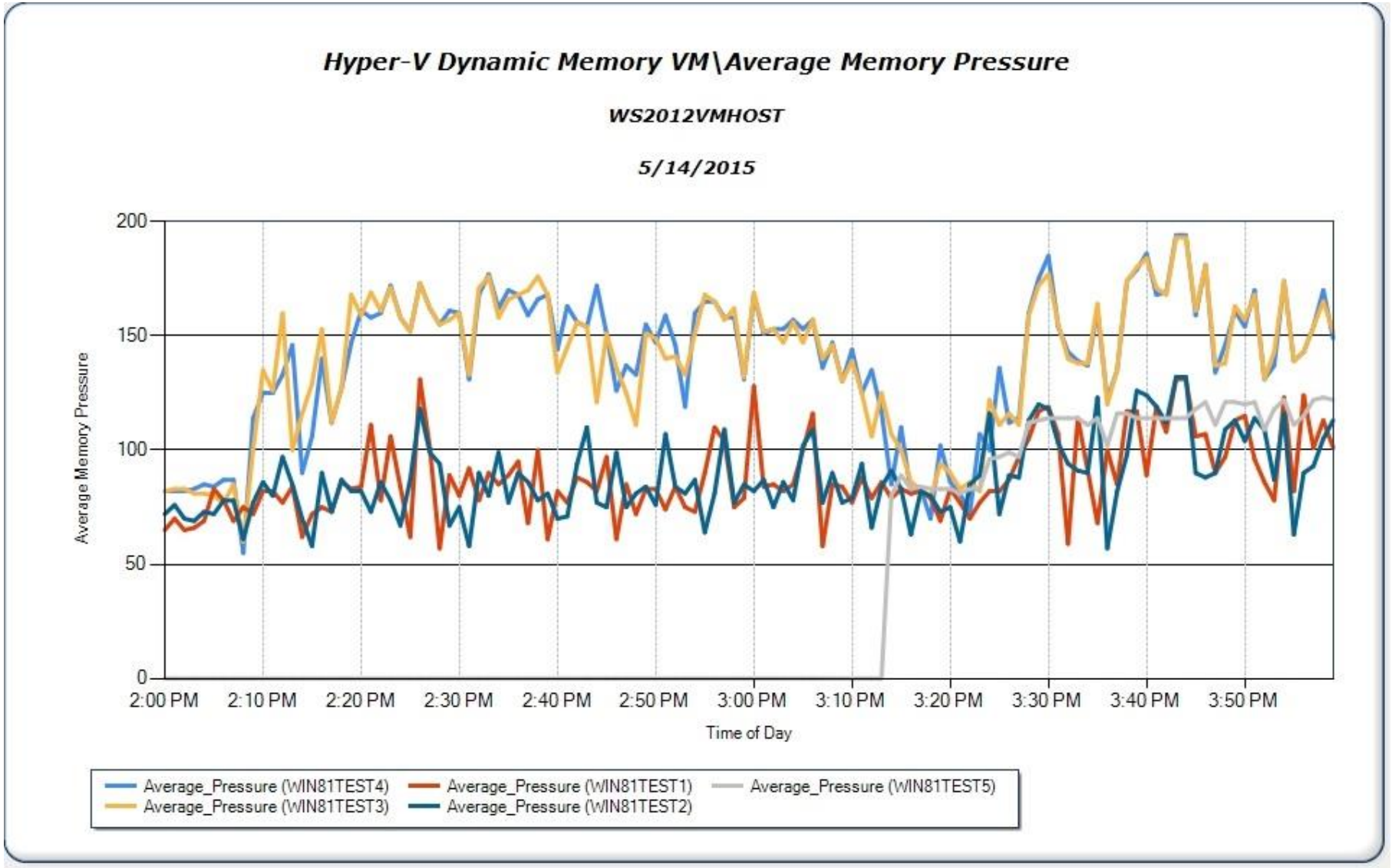
Virtual Machine memory balancing in Hyper-V

- when memory is over-committed on Hyper-V Host machines
  - Hyper-V attempts to balance **Memory Pressure** across all Windows VMs running at the same dynamic memory allocation priority
  - Memory Pressure = $\frac{guest\ machine\ Committed\ Bytes * 100}{current\ machine\ memory\ allocation}$
  - since the likelihood of guest machine **paging** increases when **Memory Pressure** >> 100
  - Control engineering approach
    - interfaces with the "hardware" hot memory Add/Remove facility
    - memory priority creates "bands" of machines based on Memory Pressure

Virtual Machine memory balancing in Hyper-V

- However, Committed Bytes is not always a reliable indicator of actual memory requirements on a Windows machine
  - SQL Server immediately allocates all available RAM
    - Uses a manual setting to override the default policy
  - well-behaved Windows apps that respond to Lo/Hi memory notifications issued by the OS
    - e.g., Lo/Hi memory notification trigger garbage collection by the .NET Framework Common Language Runtime (CLR)
- Access to additional guest machine metrics would likely improve the Hyper-V dynamic memory management routines
  - access to Lo/Hi memory notifications
  - balance physical memory to minimize demand paging

Virtual Machine memory balancing in Hyper-V



Questions

References

- Alan Jay Smith, "Cache memories", *ACM Computing Surveys*, Sept. 1982.
- Alan Jay Smith, "Disk cache miss ratio analysis and design considerations", *ACM Transactions on Computer Systems (TOCS)*, Aug. 1985.
- Waldspurger, Memory Resource Management in VMware ESX Server, *Usenix*, Dec. 2002"
- Friedman, "Virtual memory management in VMware: memory ballooning."