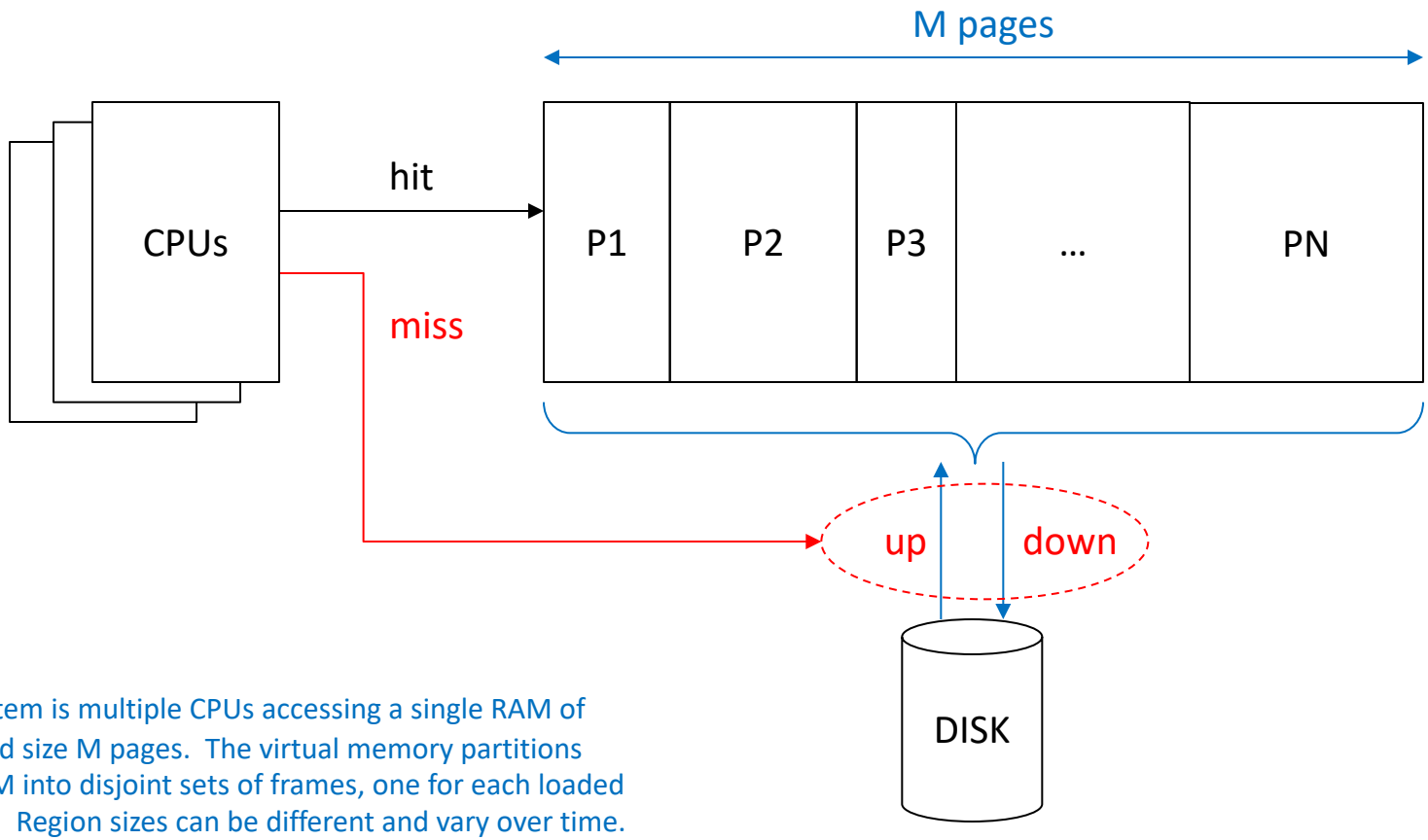
Working Set Management

Peter J. Denning

Multiprogramming

- The “singleton memory” perspective is easy to think about but does not represent multiprogramming, the common configuration
  - Multiple jobs (processes) loaded into non-overlapping regions of RAM at the same time
  - Regions may differ in size
  - Regions may vary in size over time
- Multiprogramming enables a CPU assigned to a stopped process to switch rapidly to another, keeping it busy during the long wait times of other jobs

Multiprogrammed Memory Principle



System is multiple CPUs accessing a single RAM of fixed size M pages. The virtual memory partitions RAM into disjoint sets of frames, one for each loaded job. Region sizes can be different and vary over time. Total number of jobs N (the multiprogramming level) can vary over time.

Partitioning

- Multiprogramming: Partition the main memory among N jobs
  - N is the multiprogramming level (MPL)
- How to partition?
  - Equal size regions = 1/N of the RAM?
  - Unequal but fixed size regions?
  - Variable size regions?

Can fixed-space replacement policies be generalized for multiprogramming?

Yes: with fixed partition or global policy

Fixed Partition

- When a job is loaded into RAM, it gets a fixed size region that does not change size during its execution
- The chosen replacement policy is applied individually in each region
  - Each job operates under the singleton memory principle in its own m-page region
- How to choose region size?
  - Very hard: choose m that minimizes space-time Y of job
  - Space-time law says that throughput X = m/Y
  - See Module 4.5, last five slides

Global Variable Partition

- Lump all pages of all jobs together in one pool (usually the M pages of RAM) and apply policy to the pool
  - FIFO: All pages in a single FIFO list
  - LRU: All pages in a single LRU stack
  - CLOCK: All pages around the single clock perimeter

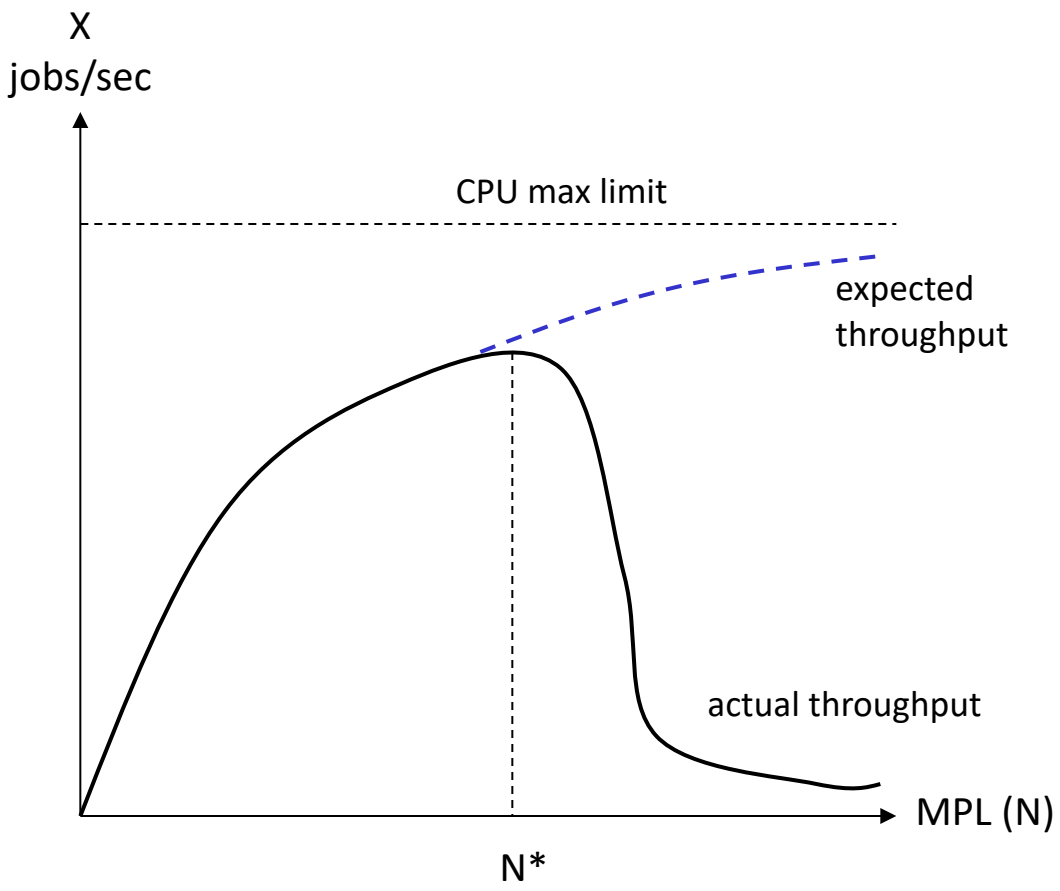
- Global replacement tends to perform much worse than when the policy is applied separately to jobs:
  - Scheduling order of jobs distorts page use statistics.
  - If the load N is too high, the process at front of RL has lost some or all its page since last time slice.

    For example, the LRU page is more likely to belong to the process at the front of the ready list, due to the round-robin scheduling of the N processes loaded.
  - This forces "page stealing" – satisfying a page fault by taking a page from another process
  - Page stealing is likely to cascade and induce thrashing.

What is thrashing?

Thrashing

- Sudden collapse of system throughput when MPL is too high
- X: symbol for system throughput (jobs/sec)
- CPU is engine driving all processes forward
- X = (max CPU speed (job/sec)) x (CPU utilization)



Thrashing is unexpected, sudden drop in system throughput with increased multiprogramming level.

Our intuition expects X to asymptotically approach its max, the raw CPU speed, as increasing load pushes CPU utilization toward 100%.

Thrashing occurs when MPL so large that global paging algorithm steals pages from other jobs to satisfy a page fault. When a job get its next time slice, some of its key pages are missing, causing an instant page fault and queueing it at the DISK.

When this happens to all the jobs, almost all are queued at the DISK. The system slows to DISK speed and the CPU is mostly idle.

The onset (N*) is unpredictable. Loading a single new job to RAM can trigger it the sudden collapse. If N is close to N*, a job that suddenly generates a burst of page faults can trigger it.

How do we prevent thrashing?

- It’s complicated because multiprogramming raises many new questions
  - Fixed or variable partition?
  - What replacement policy?
  - Control MPL to prevent overload and avoid thrashing?
- What to do?

Mindshift Needed

- Singleton memory view sees the available memory (m pages) as an **external** constraint and tries to maintain the memory contents for least paging. Globalizing fixed-memory policies is not productive.
- Experimental studies show
  - LRU tends to give fewest faults but has high overhead
  - FIFO has low overhead but high faults
  - Hybrid CLOCK close to LRU in faults, FIFO in overhead
  - But all are significantly worse than MIN
- Near optimal performance seems unattainable for this view

Supply-Demand Thinking

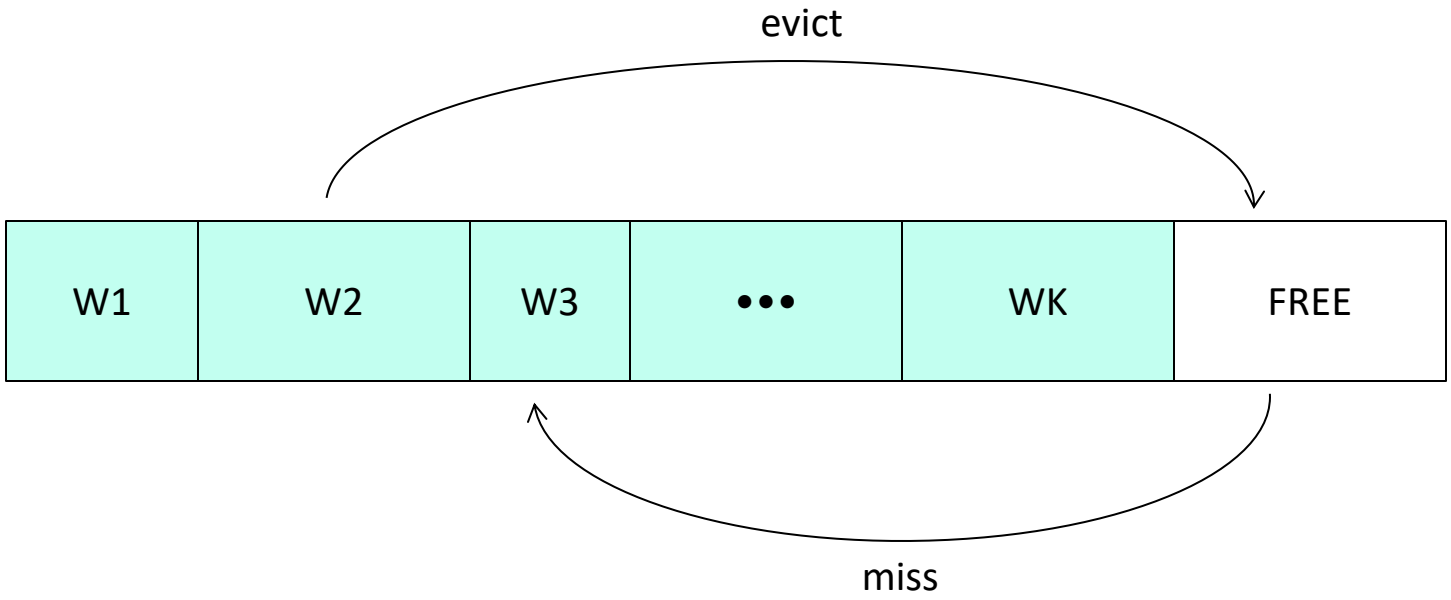
- What if, instead, we knew the **intrinsic** memory demand of a process? The pages it needs to operate efficiently?
- Let's call that its **working set**.
- Jobs **demand** memory for their working sets. The OS virtual memory **supplies** memory to contain the working sets.

If we know each job's working set

- Partition memory to give each job its working set
- Allow partition and MPL to vary as working sets vary
- Never steal a page from another working set
- System would be efficient and could not thrash

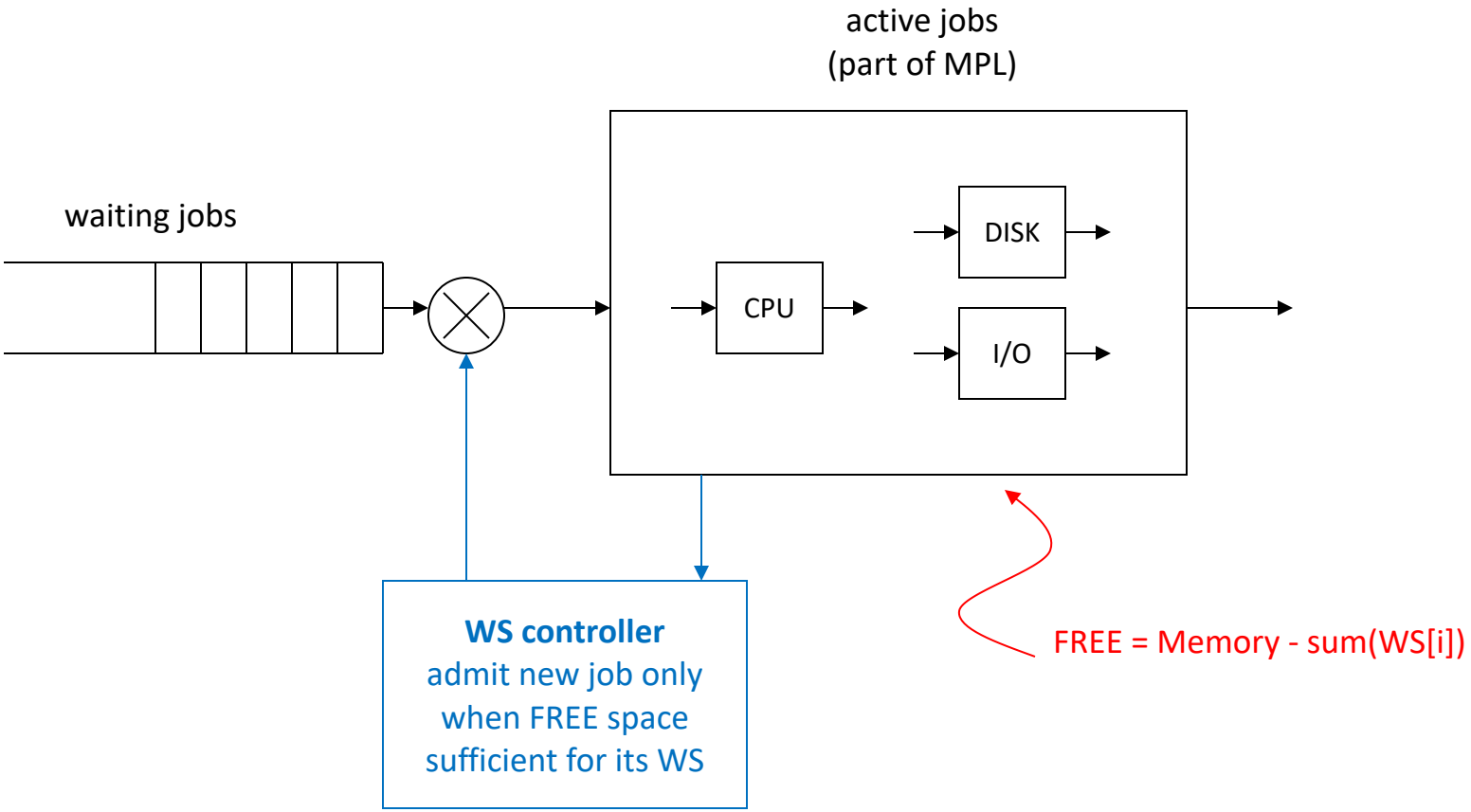
Working set policy

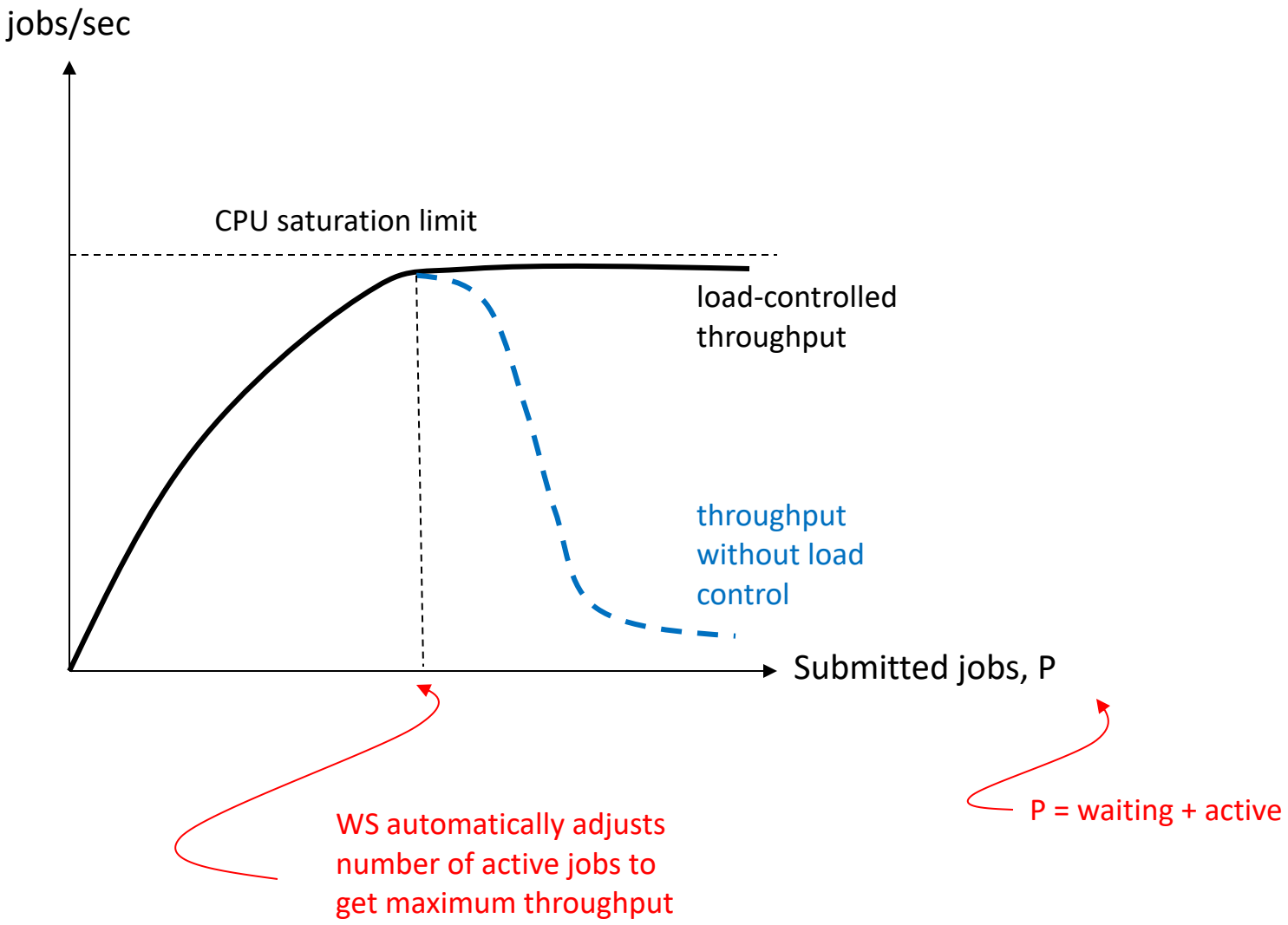
- A WS memory policy is ideal for variable partition multiprogramming.
- The policy grants each job its WS, and allows the MPL to rise only until memory is full of WSs.
- This implies some of memory is FREE – not assigned to any WS
- Minimizes page faults by tracking working sets
- No page stealing (fault in one job never steals page from another job)
- Cannot thrash



- Miss when r(t) not in WS; add FREE page to WS
  - If FREE empty, use LRU page in WS
- Evict when page not in WS, return to FREE
- Misses and evictions need not coincide
- If job quits, all its WS pages go to FREE
- Scheduler only loads new job if its WS<FREE

WS Policy: Load-Controlled Scheduling





How do we know this works?

How do we know jobs have working sets?

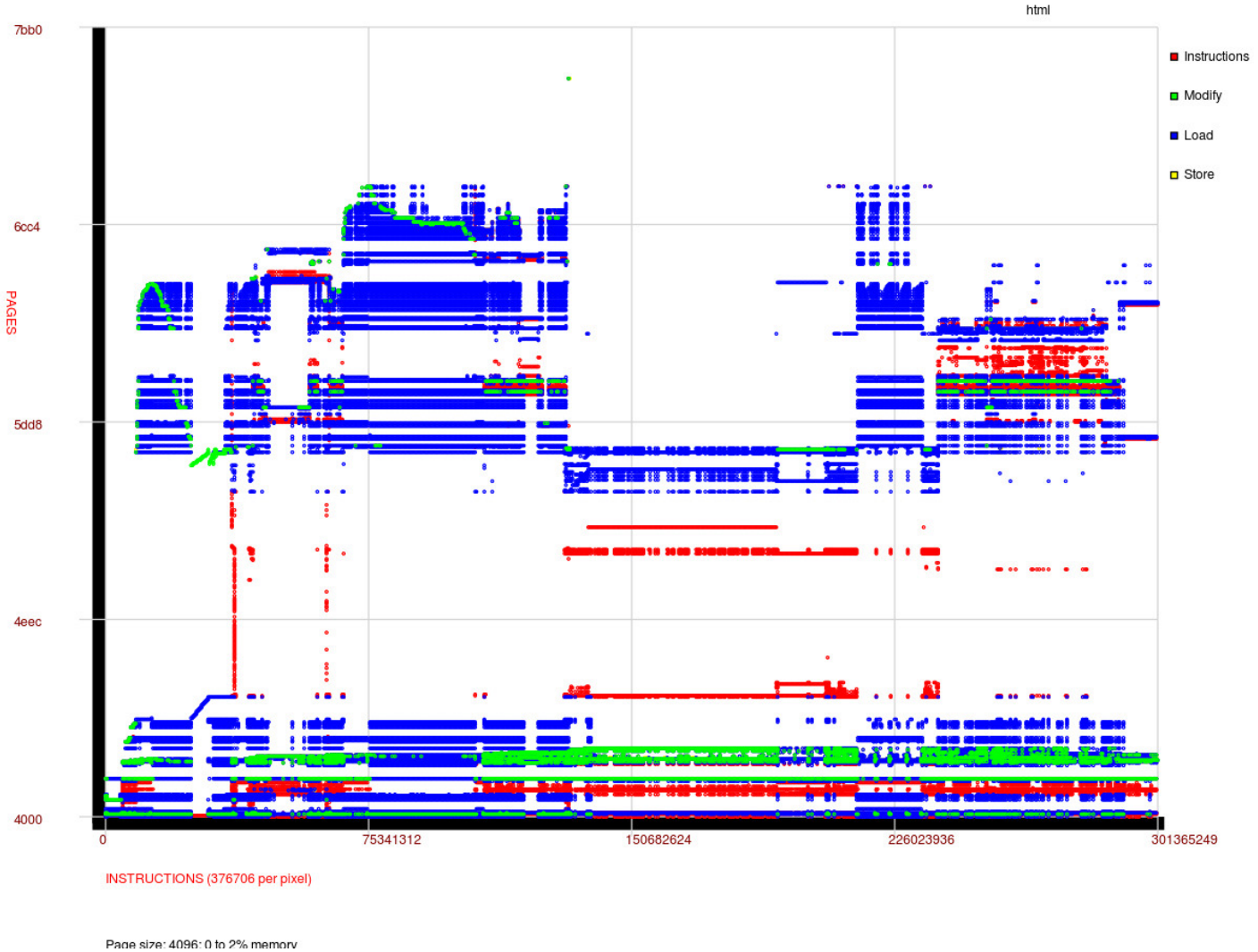
How do we measure them?

Principle of Locality

- Discovered at same time WS invented (1966) as a way to explain why WS should work
- It says that jobs tend to use the same subsets of its address space over extended periods of time
  - locality set: the subset in use
  - phase: the time interval for use of the same locality set
  - transitions: job switches to new locality sets
- Ideally, WS = locality set
  - little or no paging during phase
  - most paging generated by transitions

The page reference map is a tool for visualizing the locality behavior of a job

Page reference map of Firefox browser (from Adrian McMenamin)



- Page reference map us a bitmap showing which pages of address space are used at different times during a job's execution
  - Vertical (y-axis) enumerates pages
  - Horizontal (x-axis) is time measured in sample intervals
- The vertical column of pixels at each time t indicates which pages are used in sample interval t, shown in colors
- In the Firefox browser on a Linux system
  - Page size 4096 bytes
  - Sample window size 376,706 memory accesses (approx. 3.8×10^5)

What we see in page reference maps

- **Locality sets**: pages used repeatedly in successive sample intervals
  - Locality sets are small subsets of address space
  - 5 sets in the example, each 15% to 30% of address space
- **Phases**: intervals where locality set does not change
  - 5 phases in the example
  - 50 to 220 sample intervals each
- **Transitions**: abrupt changes of locality set
  - 5 transitions in the example
- We NEVER see random page reference maps

Notations for locality

- Executing processes reference their memory objects with **phase-transition behavior**:

$$(L_1, H_1), (L_2, H_2), \ldots, (L_i, H_i), \ldots$$

locality set

holding time (of phase)

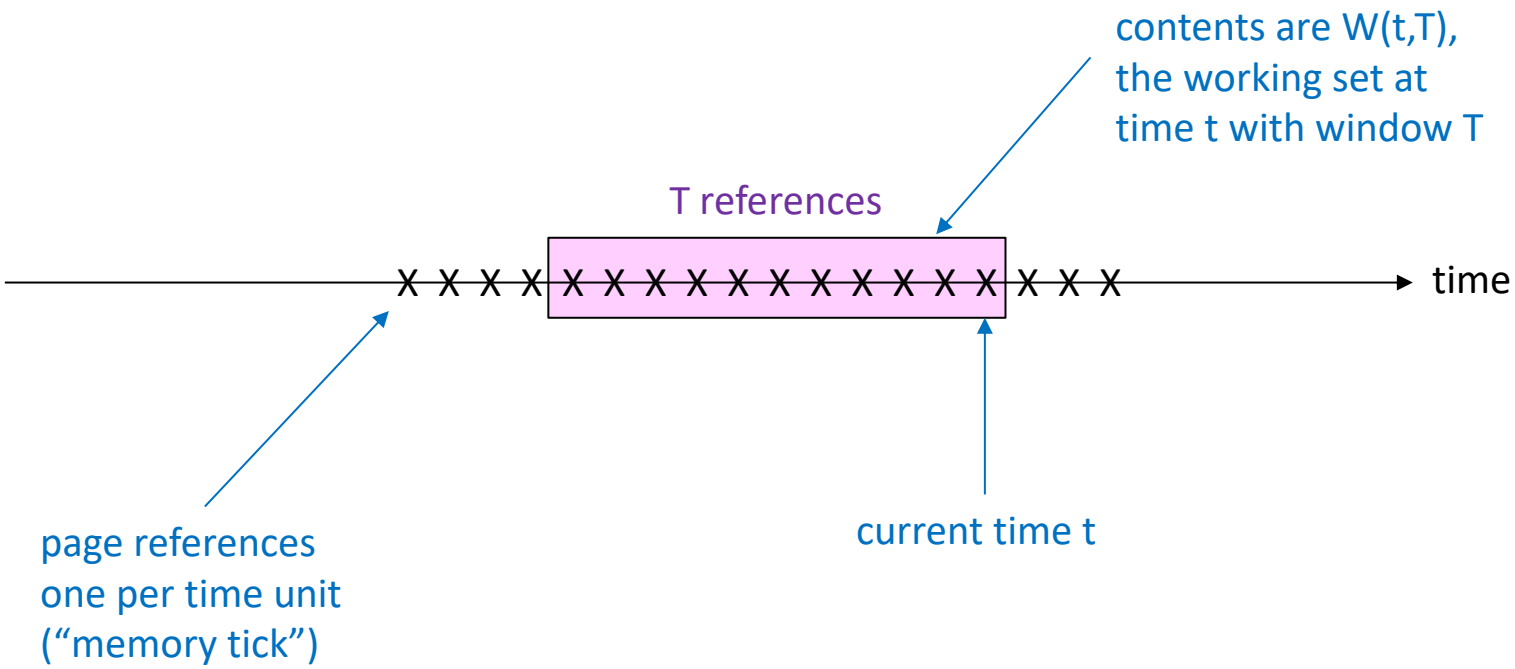
- Minor overlaps (abrupt changes) between successive locality sets

What locality is not

- Unequal overall page reference frequencies
- Statistically independent references
- Slow drifts
  - As in textbook “temporal” and “spatial” locality

Working Set Precise Definition

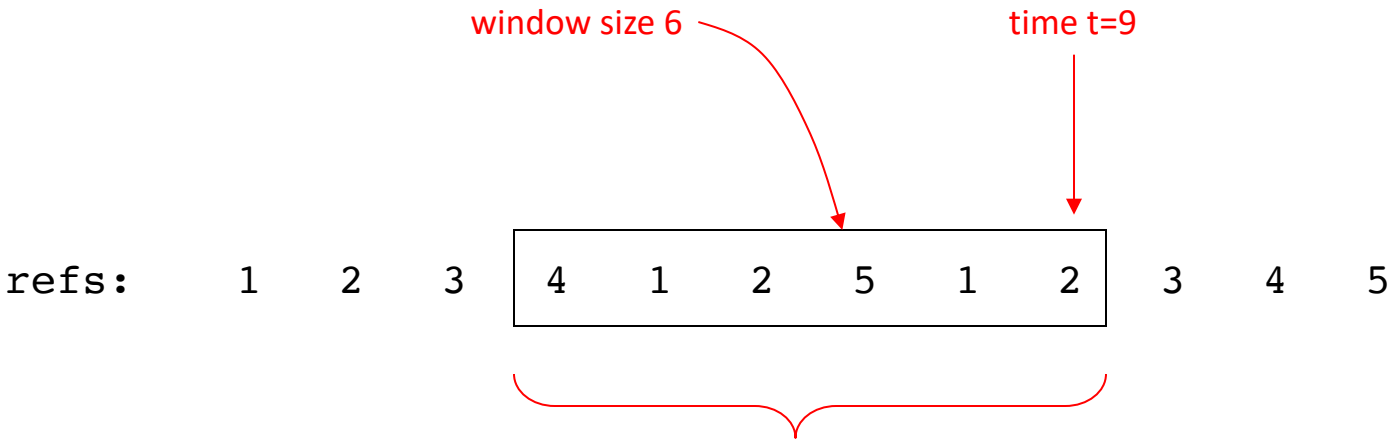
- A measure of current locality set of a job
- The working set at time t is the set of pages observed in a **backward looking window** of fixed length T
- Notation W(t,T)
- In McMenamin map, T is pixel size (= 376,706 memory references)
- W(t,T) is the column of colored pixels at time t, where t counts pixels



Working Set Lease

- The window size can be called a “lease”
  - Each page has its own lease
  - When page loaded, its lease is set to T
  - A lease ticks down to 0 with each passing memory cycle
  - When page accessed, its lease resets to T
  - When a lease expires, the page is removed

Example



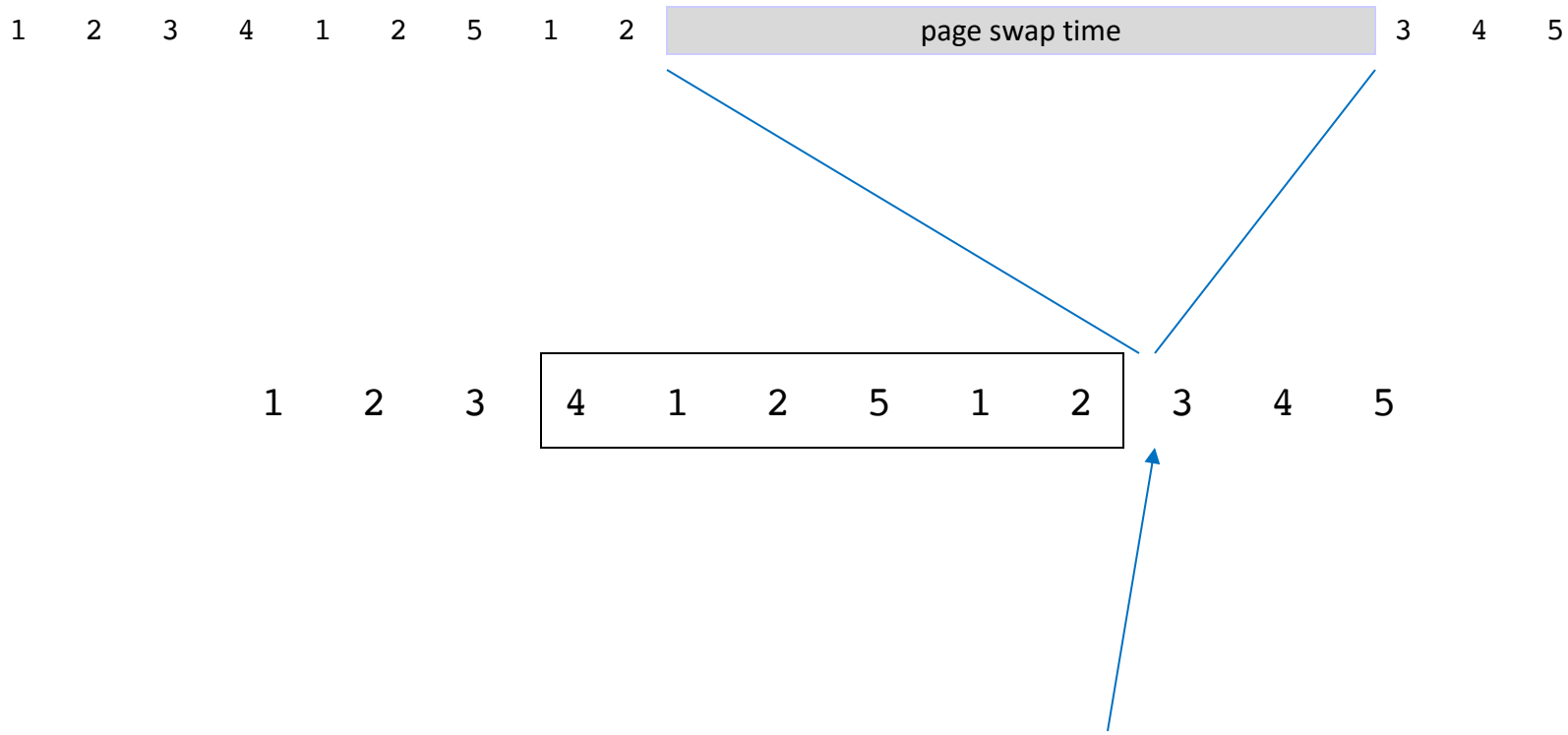
WS = pages 1, 2, 4, 5

W(t,T)

W(9,6) = {1,2,4,5}

WS "sees" a subset of the pages through its window and adjusts memory allocation to match.

A page fault will occur at t=10 because page 3 is not in the working set W(9,6).



In real time, a page swap delay is inserted between t=9 and t=10 because of the page fault.

WS analysis does not include page swap delays. Can estimate total swap delays as product of number of faults and average swap delay.

What window (lease) size?

- Let MSI be the size of the smallest sampling interval contained in a phase that sees the entire locality set
- Minimal sampling intervals likely to be short compared to phase times
- Ideally T = MSI. Then nearly 100% hit ratio when window contained in a phase; phase transitions are the main causes of page faults
- A wide range of window sizes gives the same WS contents
- Thus there is a single system-wide window size T that works for all jobs (note T > MSI of every job)
- Can find it by “tuning” – adjust its value until throughput maximized then leave it alone.

Performance

- How does WS compare with other policies that do not steal pages?
  - For example, in a fixed partition page faults take from within the same block
- Can WS maximize throughput?

Performance of thrashing

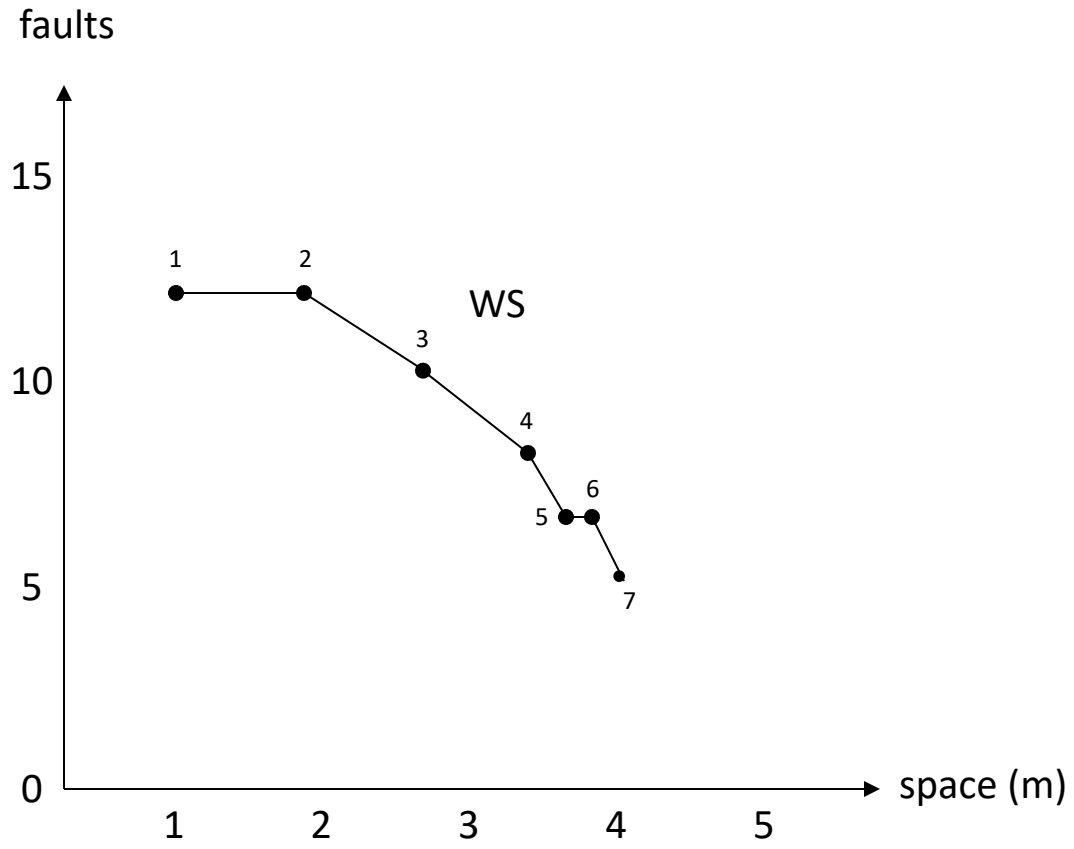
- Thrashing is caused by replacement taking pages from other jobs' locality sets.
- When its locality sets are loaded, a job does not page during a phase.
- When its memory space is constrained to be less than locality set, the job will page heavily.

- The effect on throughput is dramatic. To see this:
  - Define memory efficiency E to be ratio of RAM access time to effective access time in the presence of page faults.
  - Then E = 1/(1+fD), where f is the fault rate and D the DISK/RAM access ratio, typically 10^6. The expansion factor fD represents CPU downtime while it waits for the disk to service a page fault.
  - E is very sensitive to faults become of the large size of D.
  - In the McMenamin example, suppose the pixel size $3.8*10^5$ memory accesses is at the threshold of seeing the full locality set in every pixel of a phase. Suppose that shrinking the window slightly to $3.3*10^5$ causes 1 fault per pixel. Then E = T/(T+D), where T is pixel length. Filling in T and D, we get E=1/4.
  - The small reduction of window thus causes CPU utilization to drop from 1 to 1/4, a significant drop, resulting in throughput drop to 1/4 its previous value.

- This applies to any policy. For example, if LRU memory space not sufficient to hold the locality set, LRU will page frequently.
- Global LRU can steal, eating the memory allocations of other jobs and rendering them too small for their locality sets.
- WS detects the locality set correctly most of the time and thus avoids driving CPU efficiency down.
- So “no stealing” is a necessary condition to avoid eating other jobs’ locality sets, but it may not prevent thrashing if jobs’ locality sets are not loaded.
- To fully prevent thrashing, need also to detect and protect locality sets.

- In the next module “Working Set Analytics” we will show fast algorithms for measuring working set fault rate and mean size
- We will define VMIN, the optimal policy over all possible memory allocation policies
  - Same as WS but with forward window
  - Page not seen in forward window: evict immediately
  - WS=VMIN while T contained in phase
  - Most paging from transitions
- Correlating with page reference map: WS within 1-3% of VMIN

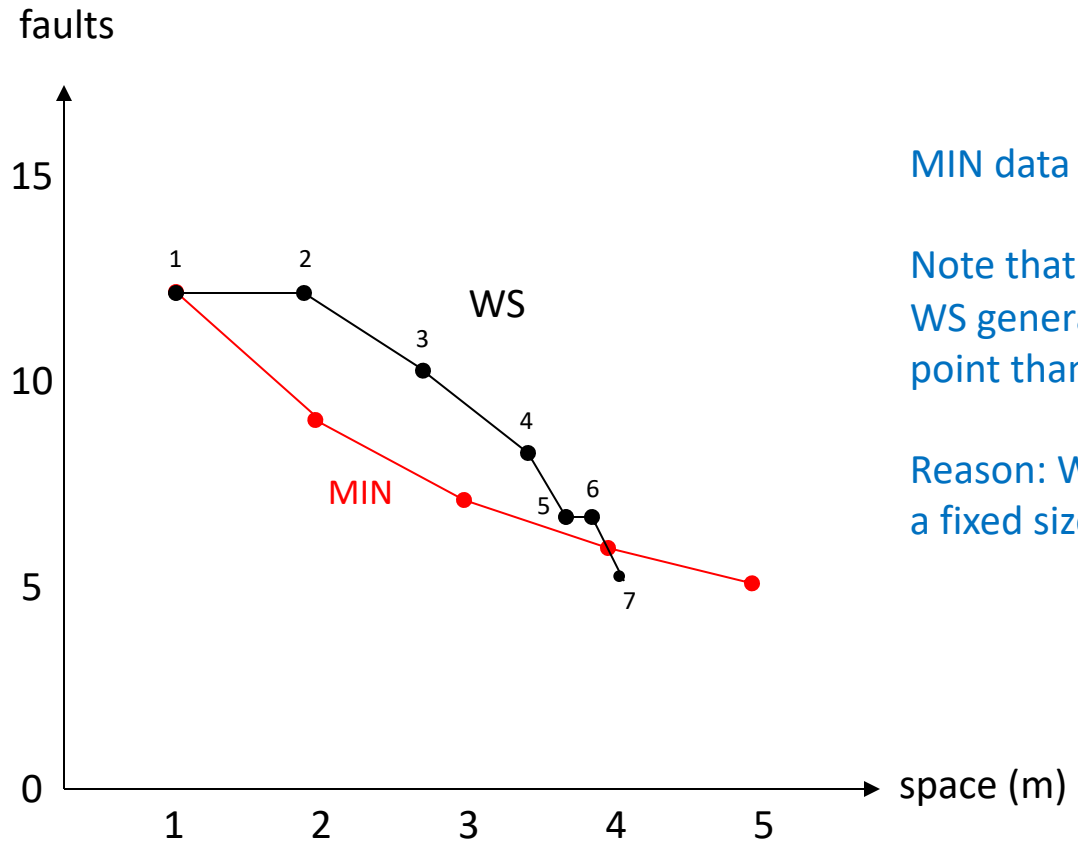
The next three graphs illustrate what the analytic methods produce for the example address trace used earlier.



The mean working set sizes are points on the space-axis. The corresponding miss counts are points on the faults-axis.

Above window size 7 there is no change in the WS size or fault count

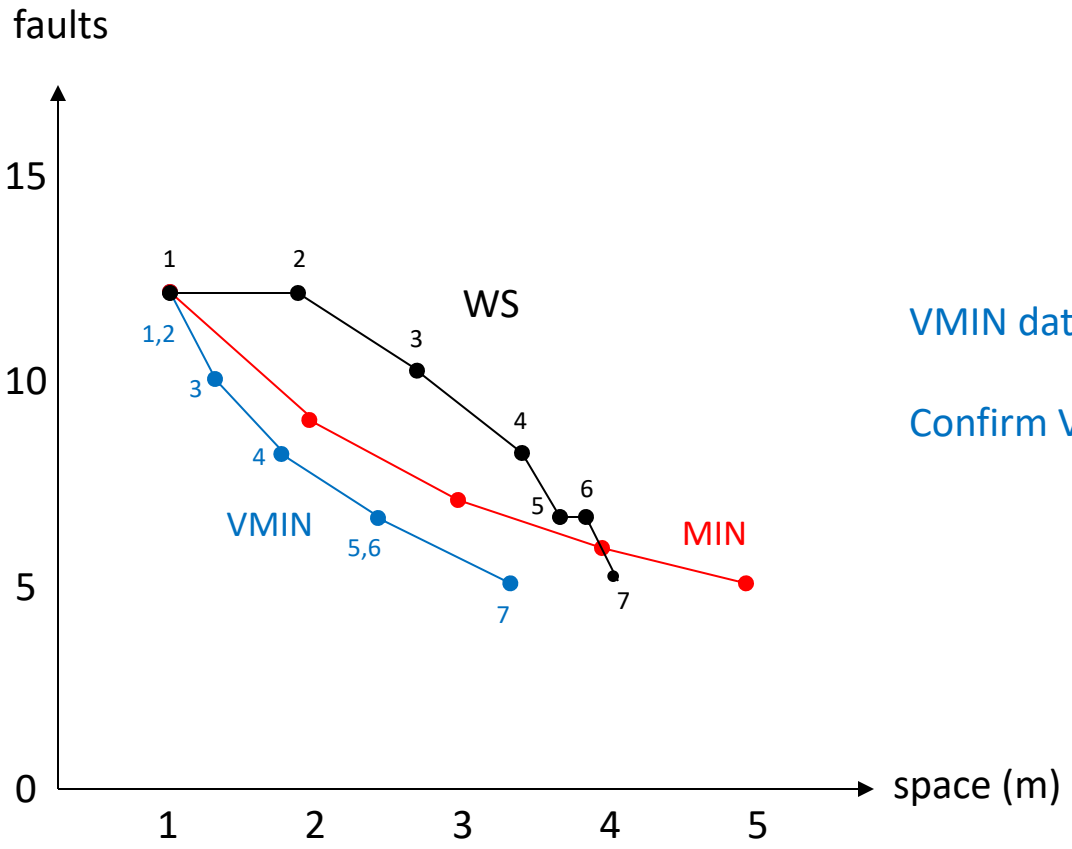
The numbers are the window sizes that generate the data points.



MIN data added to the graph.

Note that at one point (T=7) WS generates a better operating point than (fixed-space) MIN.

Reason: WS is not constrained to a fixed size.



VMIN data added to the graph.

Confirm VMIN is better than MIN.

The main goals of WS are achieved:

- Prevent thrashing
- Deliver near optimal system throughput

The fixed space policies, when generalized to multiprogramming, are likely to thrash and cannot deliver paging rates close to MIN

Can WS be implemented efficiently?

Yes.

WSClock

- Variation of CLOCK to include a WS window T.
- Each frame has a time stamp of last use.
- Scan the frames of a process in a cyclic order, repeating until select:
  - If use bit U =1, set U=0, skip; otherwise if U=0
    - if time since time stamp ≤ T, skip
    - If time since time stamp > T, select
- Marginally more overhead than FIFO and performs comparable to WS in multiprogram environment.
- Because WS so insensitive to T, can “tune” system by finding T that maximizes throughput and leave it there

WS can be used in shared caches by associating a lease-register with each cache frame

When the register reaches 0, the frame is marked as free

(This “hardware implementation” was part of the original WS proposal, but the chip hardware could not support it.)

finis