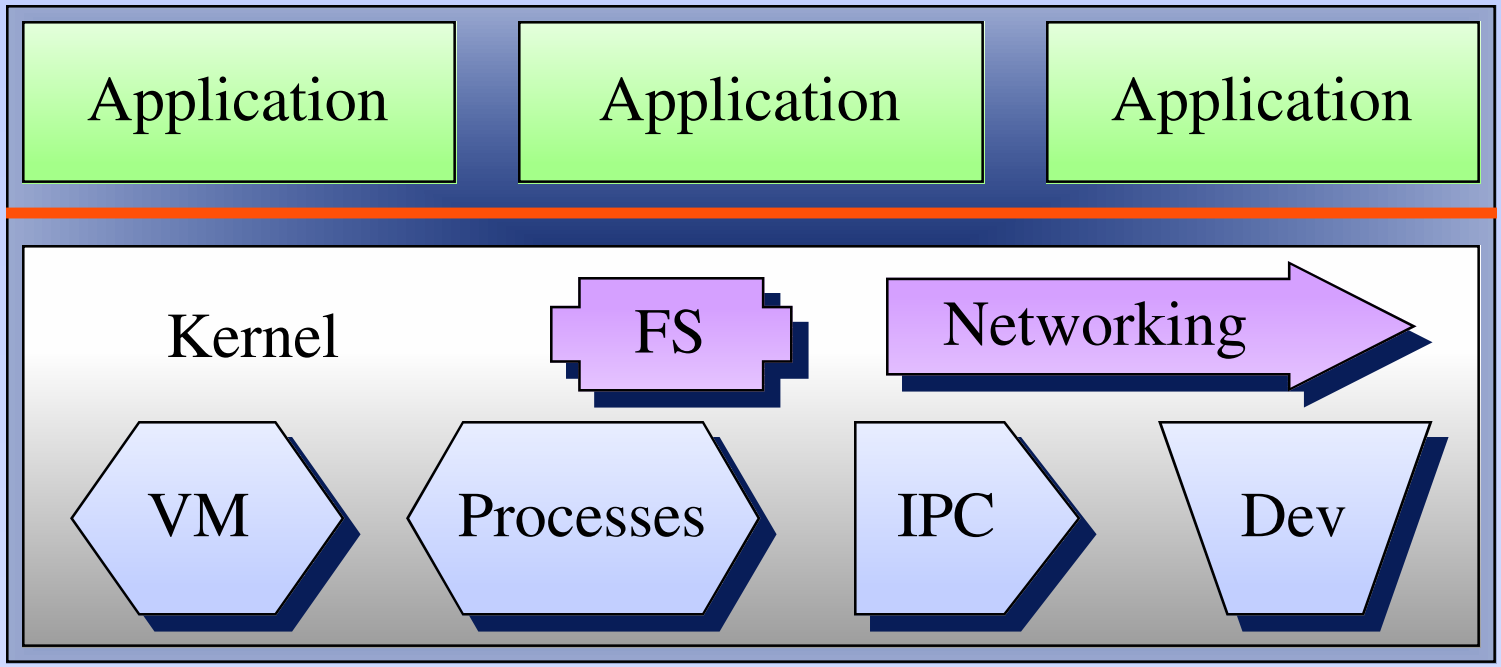
Conventional Kernels: E.g. UNIX



- Large, complicated supervisor-mode code
  - *e.g.*, 220K lines of code for SunOS 4.1.2
- Fixed amount of supervisor data, i.e.descriptors

Is Kernel Size Important?

All modules have to work for applications to work so why chop up the OS kernel?

- Supervisor-mode failure is unrecoverable
- Server modules isolate failures
  - tolerate by duplicating or re-exec'ing server.
- Small module size for efficient replication and faster development and maintenance

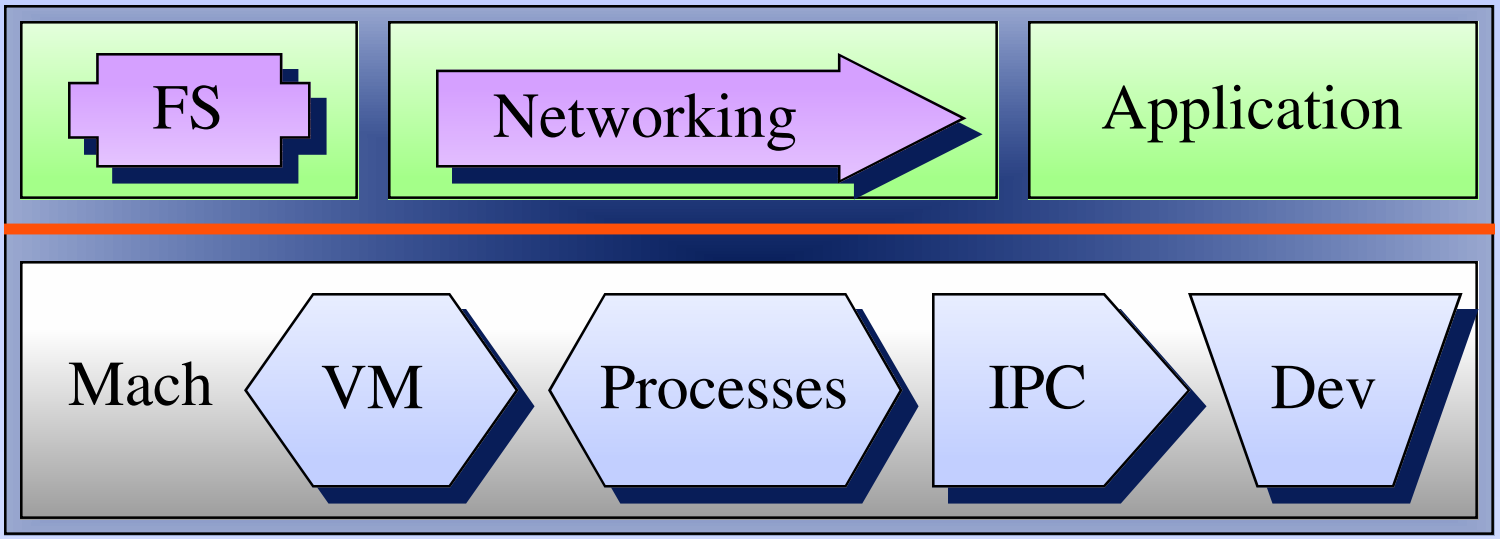
Size is our best measure of complexity and complexity is the biggest reliability problem

Fixed Numbers of Kernel Objects

- Conventional OS has a fixed maximum number of kernel objects
  - e.g. process descriptors, mbufs, sockets, etc.
  - descriptors stored in “pinned” kernel memory
- Paging the descriptors complicates supervisor code and not feasible in many cases
- Application cannot recover from OS running out of descriptors.

Micro-kernels

Move some OS objects to user-level servers.



Basic OS services are complex, such as VM

– *e.g.*, Mach 3.0 kernel is 172k lines of code

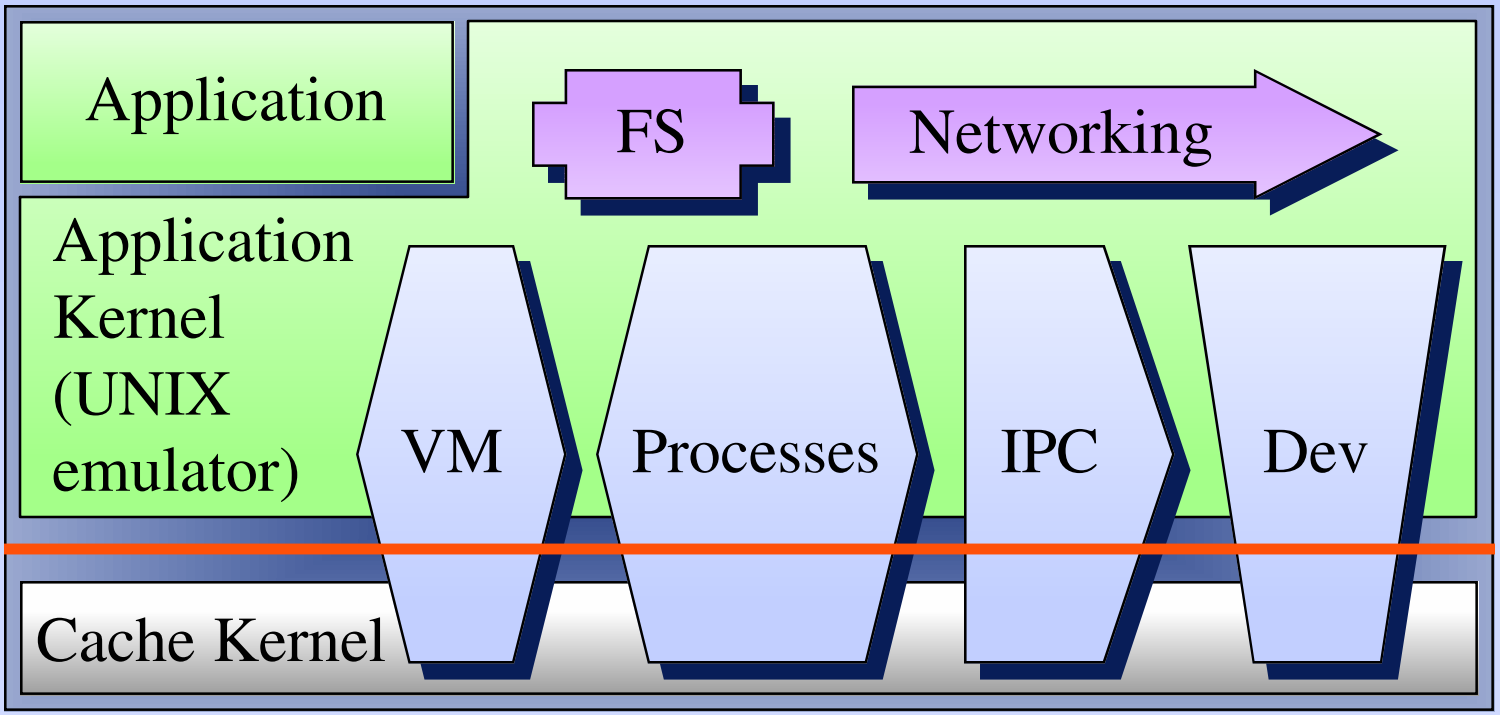
Still too big and complicated, and the same descriptor limitations

More “Micro”-Kernel Complexity

- Server modules moved back into kernel for performance reasons
  - e.g. Mach network server
- Extensions for application-specific resource management optimizations
  - e.g. page replacement control
- Complex exception handling with servers and networked operation.

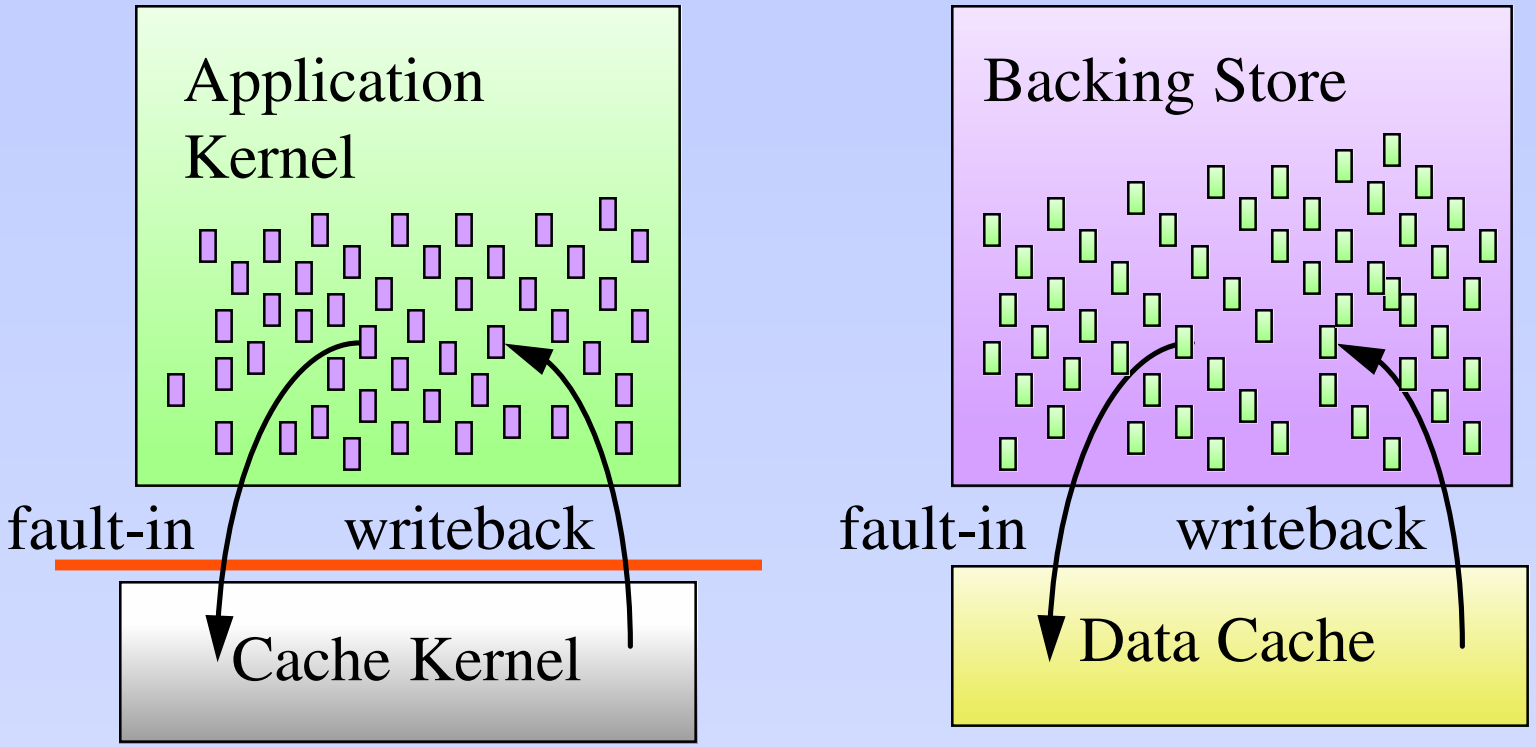
More future complexity unless a new approach.

New Approach: OS Object Caching



- Only caches and manages “active” objects
- V++ Cache Kernel: 15K lines of C++
- No kernel limits on number of objects

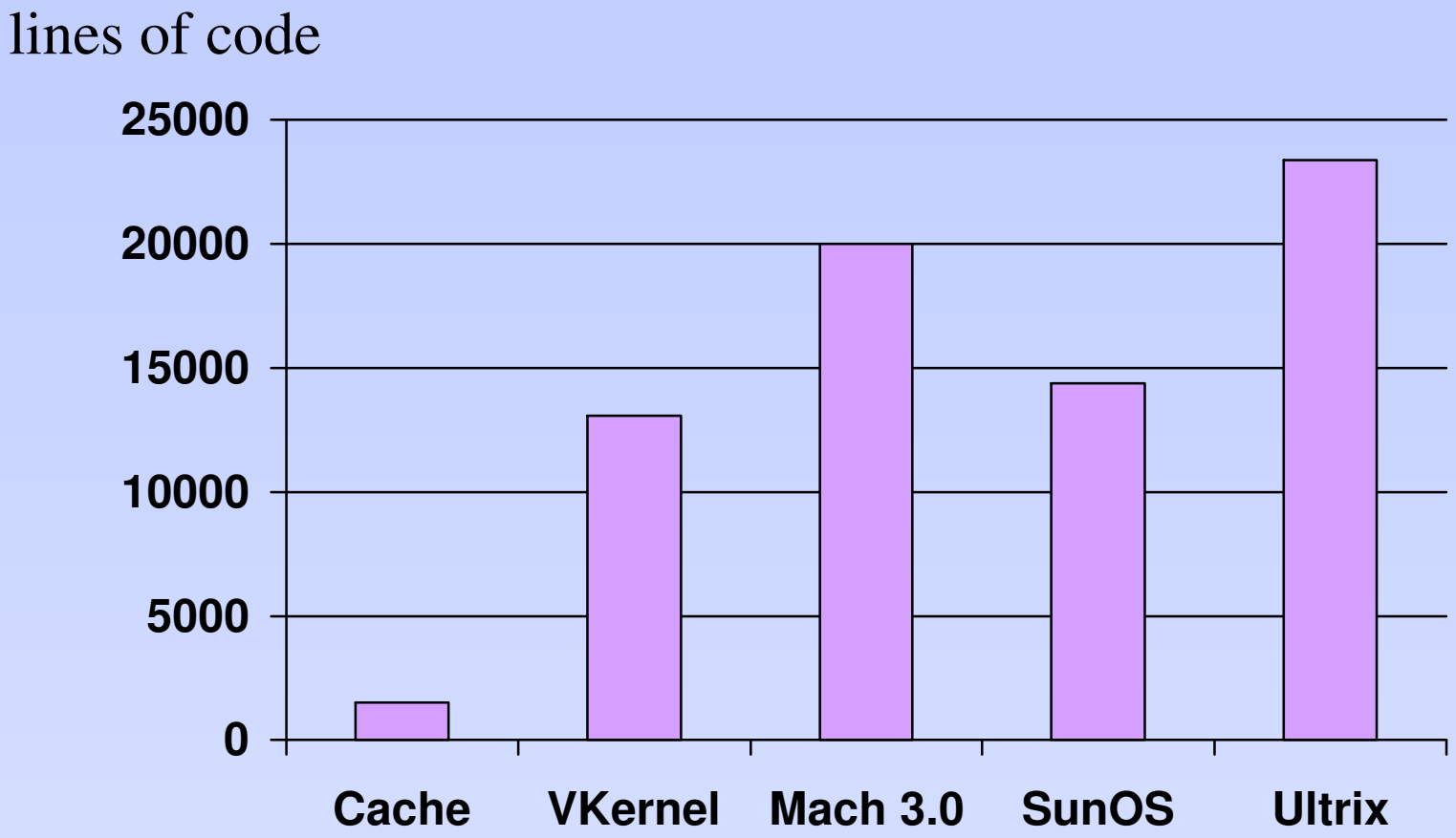
OS Caching vs. Data Caching



- Fault-in and writeback behavior
- Large backing store and small fixed cache space
- Most objects in backing store only, active in cache

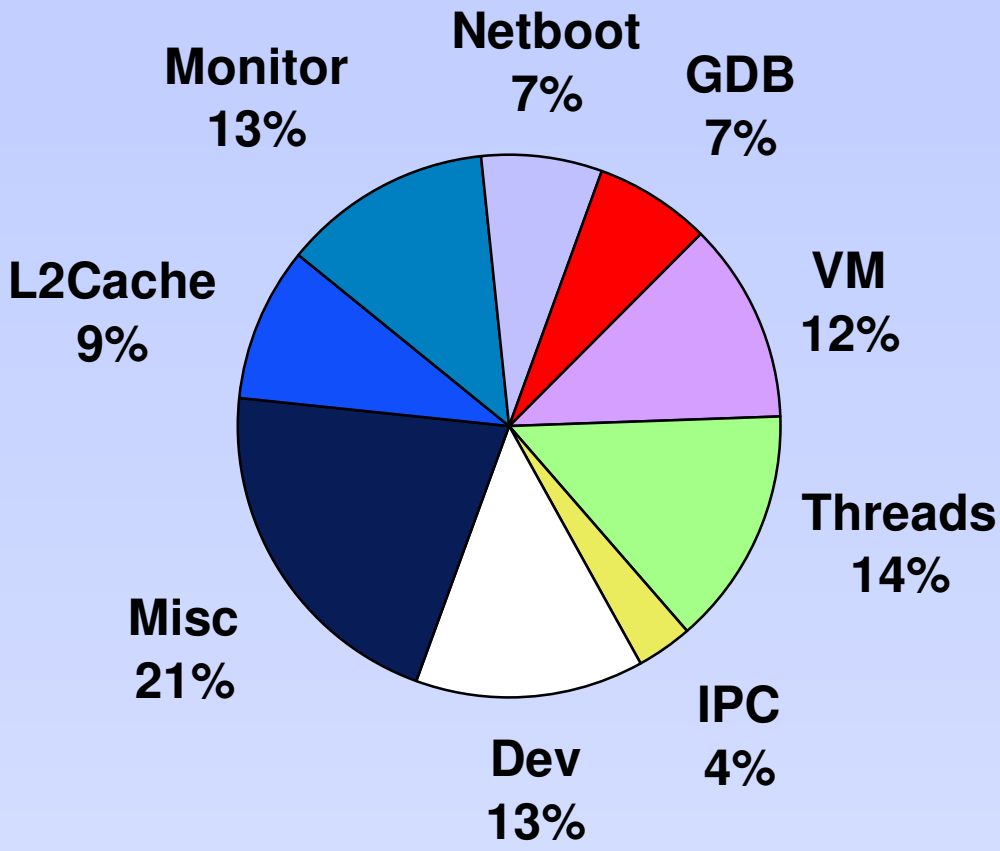
VM Source Code Size Comparison

(supervisor code only)



Order of magnitude reduction in code size (and complexity)

Cache Kernel Module Sizes



27% is boot/development code (monitor,boot,gdb)

Misc. is primarily caching support (base classes)

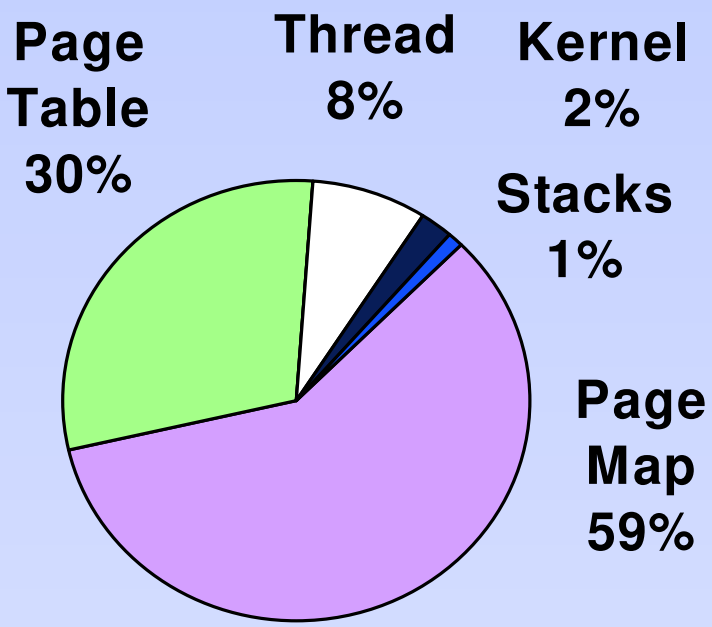
ParaDiGM and V++ Cache Kernel

- ParaDiGM:
  - 4 25 MHz M68040 sharing an 8 megabyte L2 cache per multiprocessor module (MPM)
  - MPM interconnect by VME bus and 266 FCS
- V++ Cache Kernel
  - 139 kilobyte binary code, burned into PROM
  - 2 megabytes of local memory for per-processor stacks and object caching

Caching Space Allocation

Cache Kernel Desc.	Size
Page Mapping	16
Thread	532
AddrSpace	60
Kernel	2160

Cache Space Breakdown



Cache covers 60K pages

< 1 percent space overhead

Cache space dominated by VM mapping structures.

Cache Loading/Unloading Costs
(microseconds)

Object Types	Load	Load Writeback	Unload
Mapping	45	145	160
Mapping-and-return	67	167	
Thread	113	489	206
AddrSpace	101	229	152
Kernel	244	291	80

Page map loading is most common, and fast

Page Fault, Trap, Communication Times

- Basic page fault: 99 usecs

  32 usecs. from fault to application kernel

  67 usecs. for optimized mapping load-and-return

  – Page zeroing, disk/network I/O dominate cache kernel costs.

- Simple trap: 37 usecs vs. 25 on Unix

  – extra cost for trap forwarding to appl. kernel

  12 usec. overhead insignificant for most calls.

Cache Kernel does not degrade performance

Related Work

- Mach Pmap interface - unprotected
- Multics “virtual processor” redesign idea
- Memory-based messaging in revised version of V
- Application-controlled Page Cache Management in V
- SPIN - “safe” customizing of the kernel
- Aegis - minimal kernel calling “safe” user code

Conclusions

Caching Model for an OS Kernel:

- Reduces supervisor-level code size and complexity
- Allows an unlimited number of objects within a fixed supervisor data space
- Increases application control of physical resources

Caching lowered the red line without lowering the performance or the protection