Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels

Ariel Tamches **Barton P. Miller**

{tamches,bart}@cs.wisc.edu

Computer Sciences Department

University of Wisconsin

1210 W. Dayton Street

Madison, WI 53706-1685

USA

The Vision

A unified infrastructure for dynamic OS's

Fine-grained runtime code instrumentation for:

- Performance measurement
- Tracing
- Testing (e.g., code coverage)
- Debugging: conditional breaks, access checks
- Optimizations: specialization, code reorganization
- Extensibility

Motivation: Measurement

- Measurement primitives
  - Counts, cycle timers, cache misses, branch misprediction time (on-chip counters)
    - Instrument kernel to self-measure as it runs
- Predicates
  - A specific code path; when a process is running, etc.
- Many interesting routines in the kernel:
  - Scheduling: preempt, disp, swtch
  - VM management: hat_chgprot, hat_swapin
  - Network: tcp_lookup, tcp_wput, ip_csum_hdr, hmeintr

Motivation: Optimization

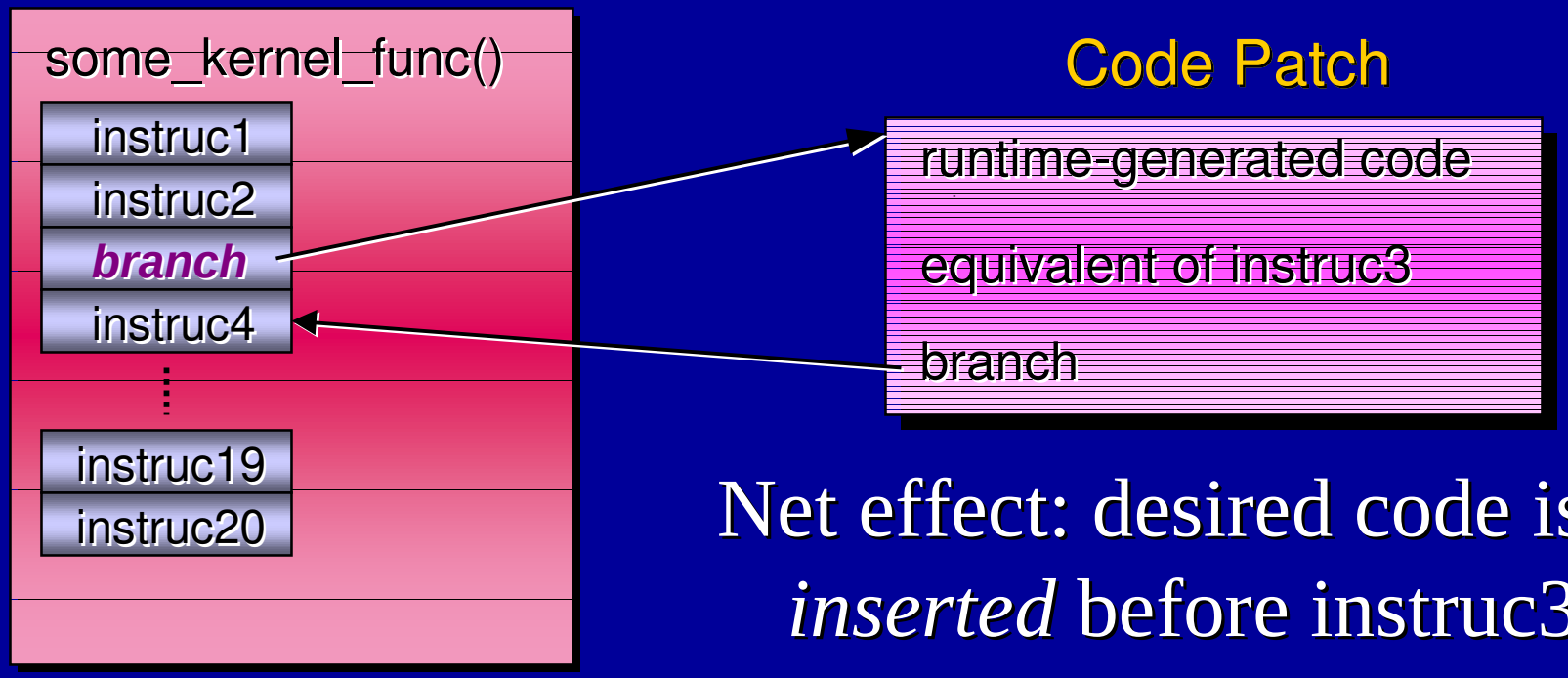
- Performance measurement shows slow code? Pick from a cookbook of on-line optimizations
  - Specialization
    - Instrument function to find common params
    - Generate specialized function
    - Install (old version jumps to new if condition met)
    - Can predicate specialization (e.g. a specific process)
  - Reorganize code to improve i-cache
    - Instrument function to measure icache miss cycles
    - Then instrument to find cold basic blocks
    - Generate “outlined” function & install

Technology to Make it Happen

KernInst: fine-grained dynamic kernel instrumentation

- Inserts runtime-generated code into kernel
- Dynamic: everything at runtime
  - no recompile, reboot, or even pause
- Fine-grained: insert at instruction granularity
- Runs on unmodified commodity kernel
  - Solaris on UltraSparc

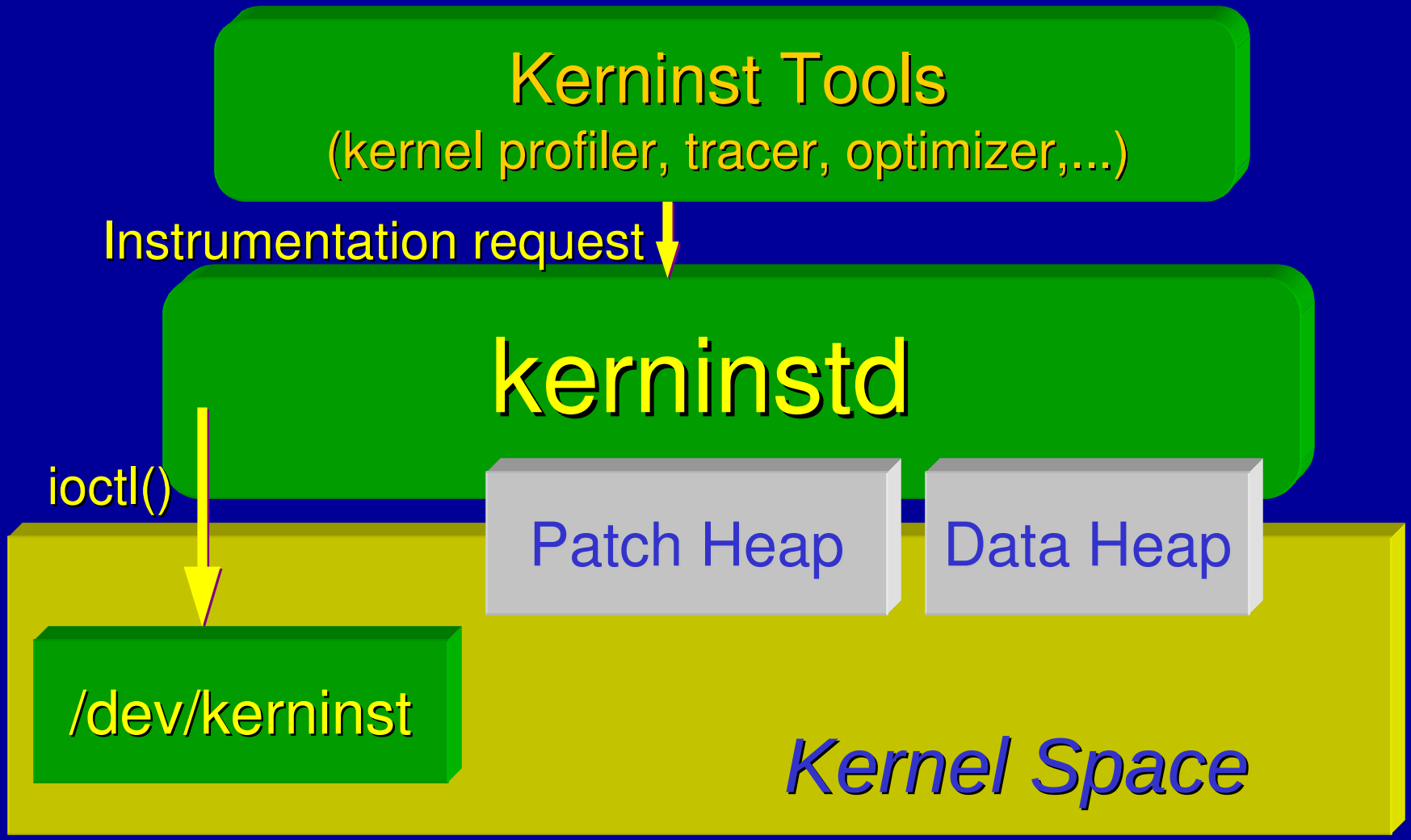
Dynamic Instrumentation



Net effect: desired code is *inserted* before instruc3

- Insert any code, almost anywhere (fine-grained), entirely at runtime (dynamic)

Our System: *KernInst*



How KernInst Works

kerninstd startup:

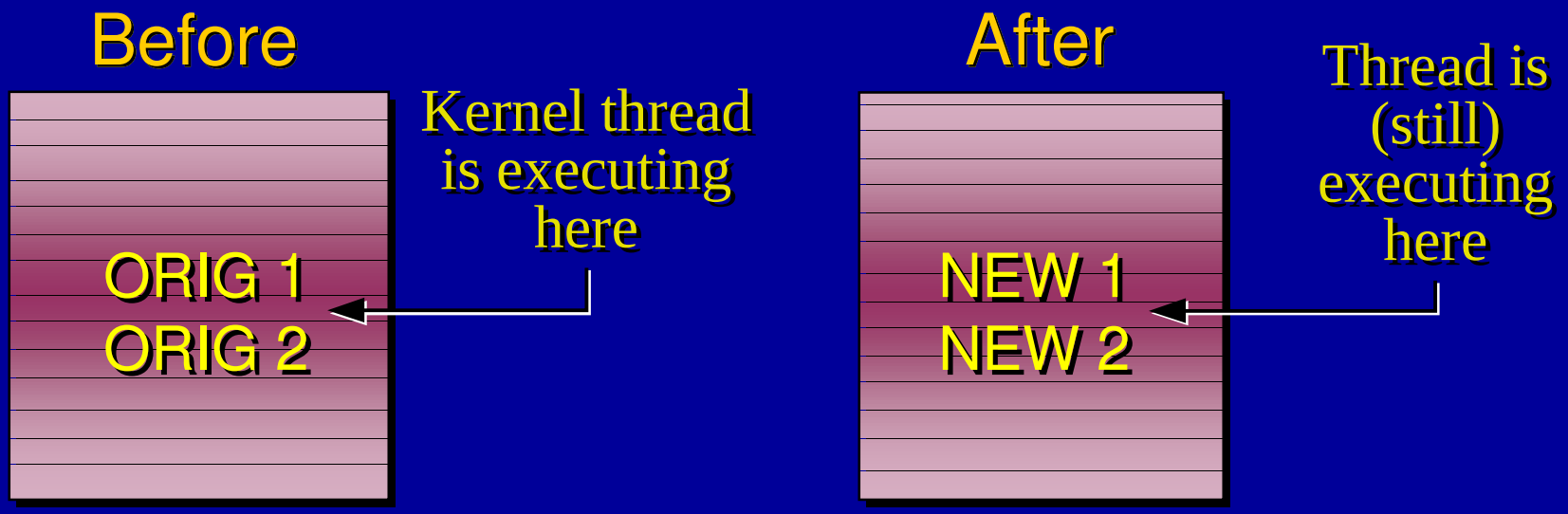
- Installs the KernInst driver, /dev/kerninst
- Allocates patch and data heaps, and reads kernel symbol table (with assistance from /dev/kerninst)
- Parses kernel code into CFG
  - Tools may want to identify basic blocks
- Finds unused registers
  - Inserted code will use these registers (avoid spills)
  - From an interprocedural data-flow analysis on the CFG
- 15 seconds

How KernInst Works (2)

- To splice in instrumentation code, kerninstd:
  - Allocates code patch
  - Fills code patch with instrumentation code, overwritten instruction, and a jump back
  - Overwrites instruction at instrumentation point with a branch to the code patch
- Writing to kernel memory
  - /dev/kmem works for *most* of the kernel
  - Have /dev/kerninst map into D-TLB for nucleus

Code Splicing Hazard

Jumping to the patch using two instructions:

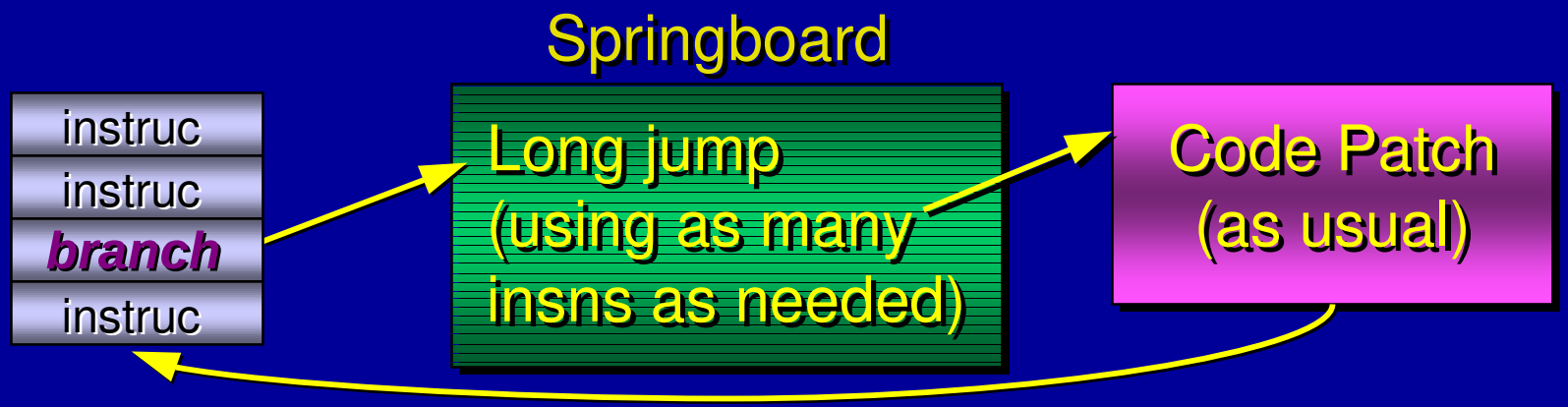


Execution sequence: (ORIG1, NEW2) ⟶ *crash!*

- Cannot pause kernel to check for hazard
- Splicing must replace *only one* instruction!

Code Splicing: Reach Problem

- Tough to reach patch with just 1 instruction!
  - Usually too far from the instrumentation point.
  - SPARC branch instruction has only +/- 8MB offset (ba,a <offset>)
- General solution: *springboards*



Springboard Heap

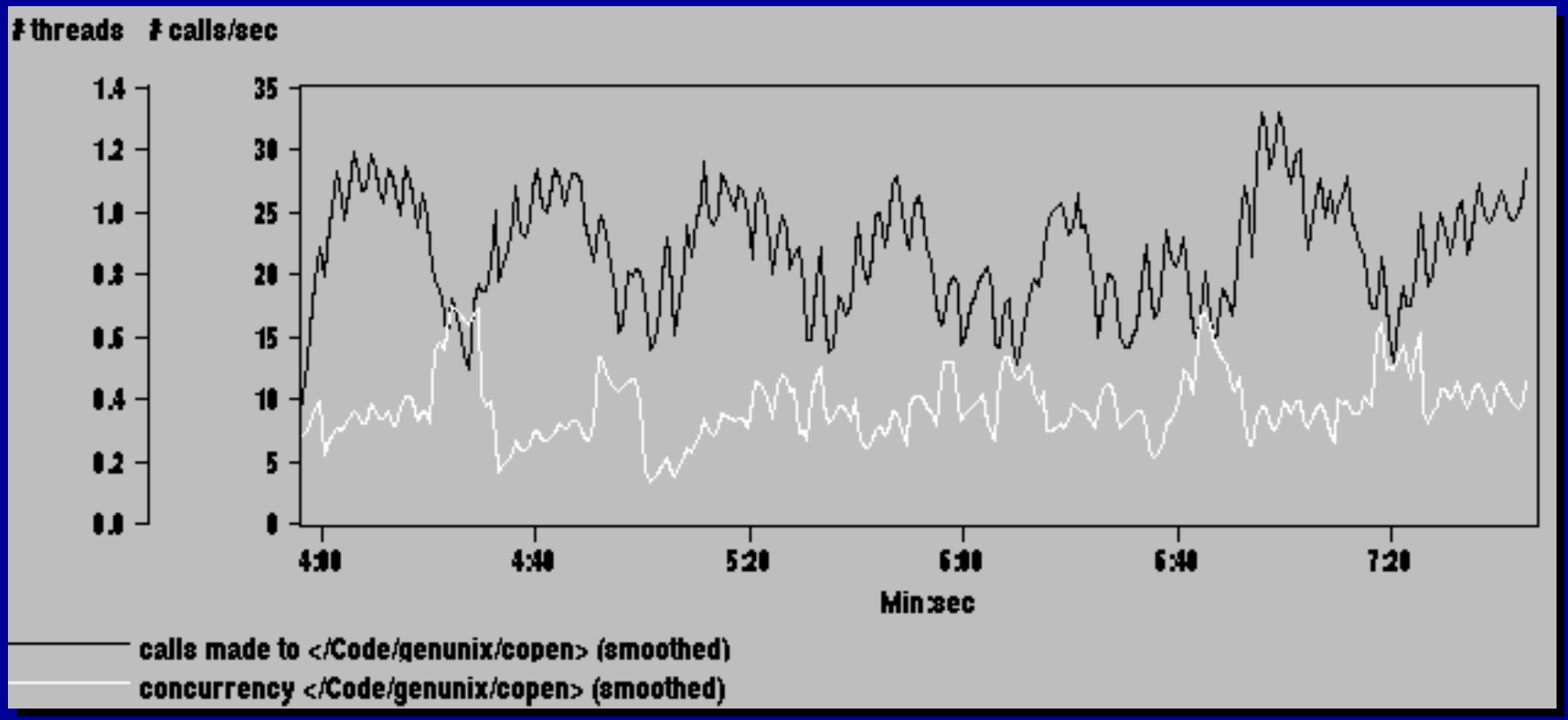
- Any scratch space located close to the splice point is suitable for a springboard
  - Must be reachable by the 1-instruction branch
  - Kernel modules have initialization and termination routines that can be overwritten
    - _init and _fini on SVR4
    - Ideal because located *throughout* the kernel
    - Lock modules into memory for safety
  - Other unused space in the kernel:
    - _start and main are only used when booting

Web Proxy Server Measurement

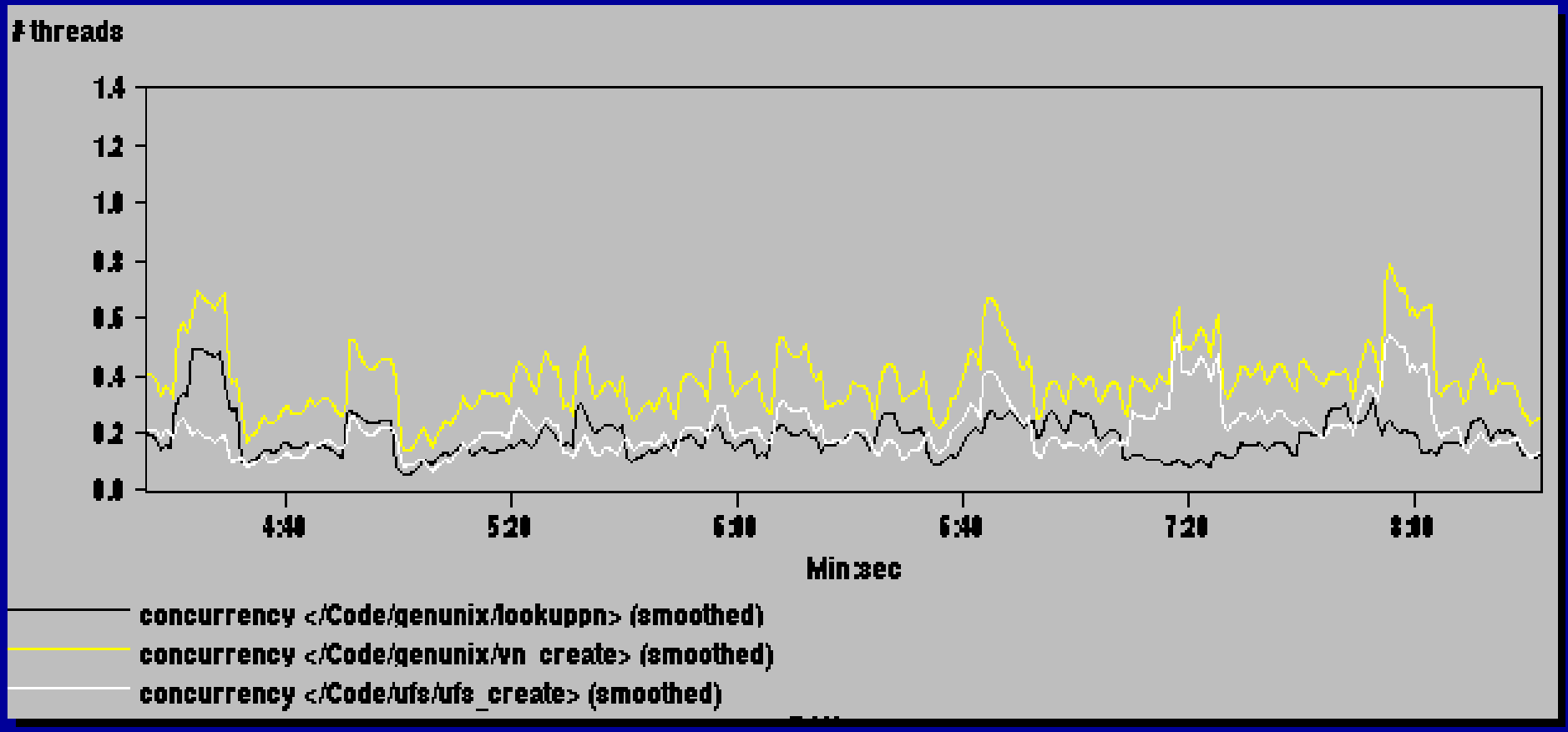
- Simple kernel measurement tool
  - Number of calls made to a kernel function
  - Number of kernel threads executing within a kernel function (“concurrency”)
- Squid v1.1.22 http proxy server
  - Caches HTTP objects in memory and on disk
  - We used KernInst to understand the cause of two Squid disk I/O bottlenecks.

Web Proxy Server Measurement

- Profile of the kernel open() routine



- Called 20-25 times/sec; taking 40% of time!



- open() calling vn_create; has 2 sub-bottlenecks:
  - lookuppn (a.k.a. namei): path name translation (20%)
  - ufs_create: file create on local disk (20%)

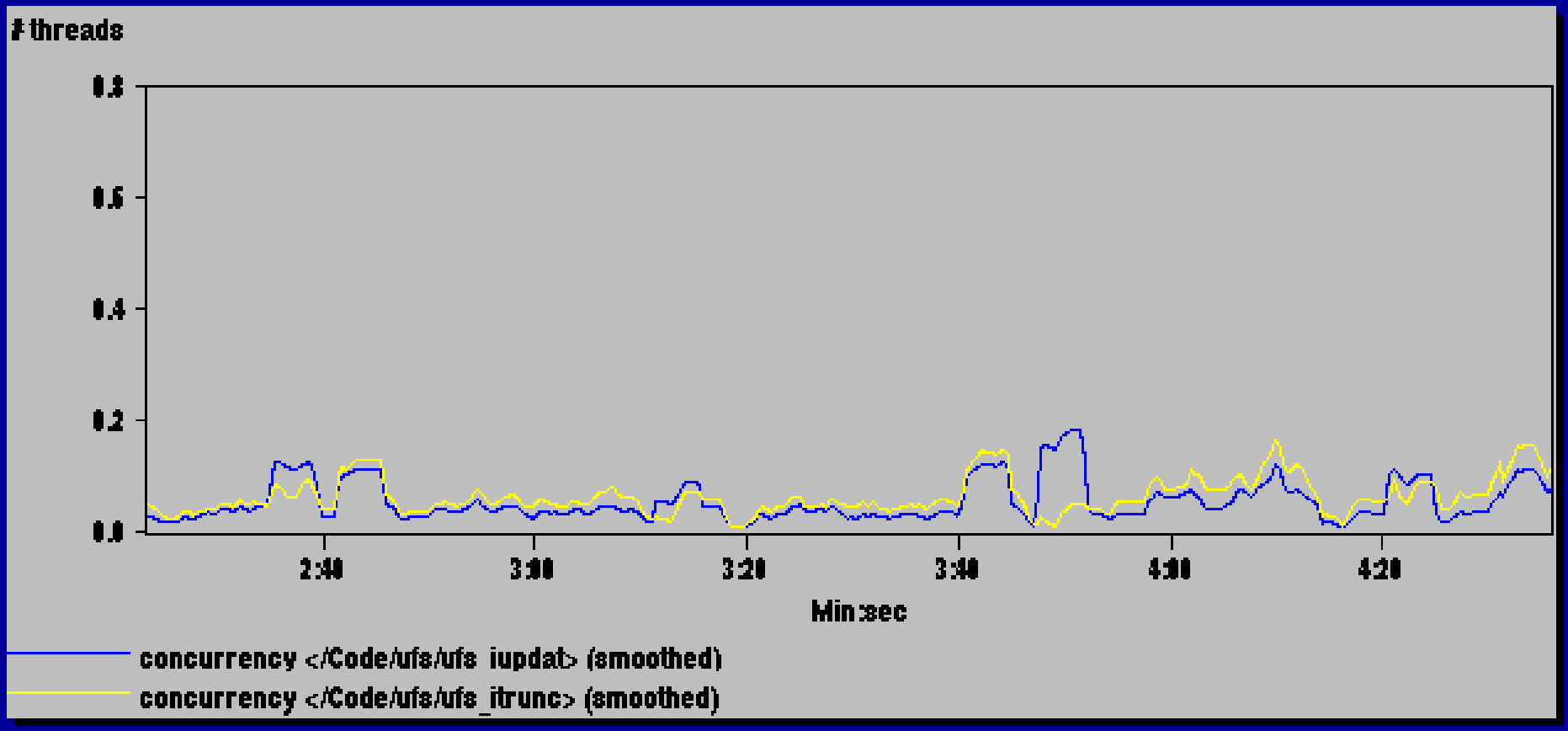
File Creation Bottleneck

- How Squid manages its on-disk cache:
  - 1 file per cached HTTP object
  - A fixed hierarchy of cache files
  - Stale cache files overwritten

- File creation bottleneck
  - Overwriting existing files: truncates first
  - UFS semantics: meta-data changed synchronously

File Creation Optimization

- Overwrite cache file; truncate only if needed



- What took 20% now takes 6%

Conclusion

Fine-grained dynamic kernel instrumentation is feasible on an *unmodified* commodity OS

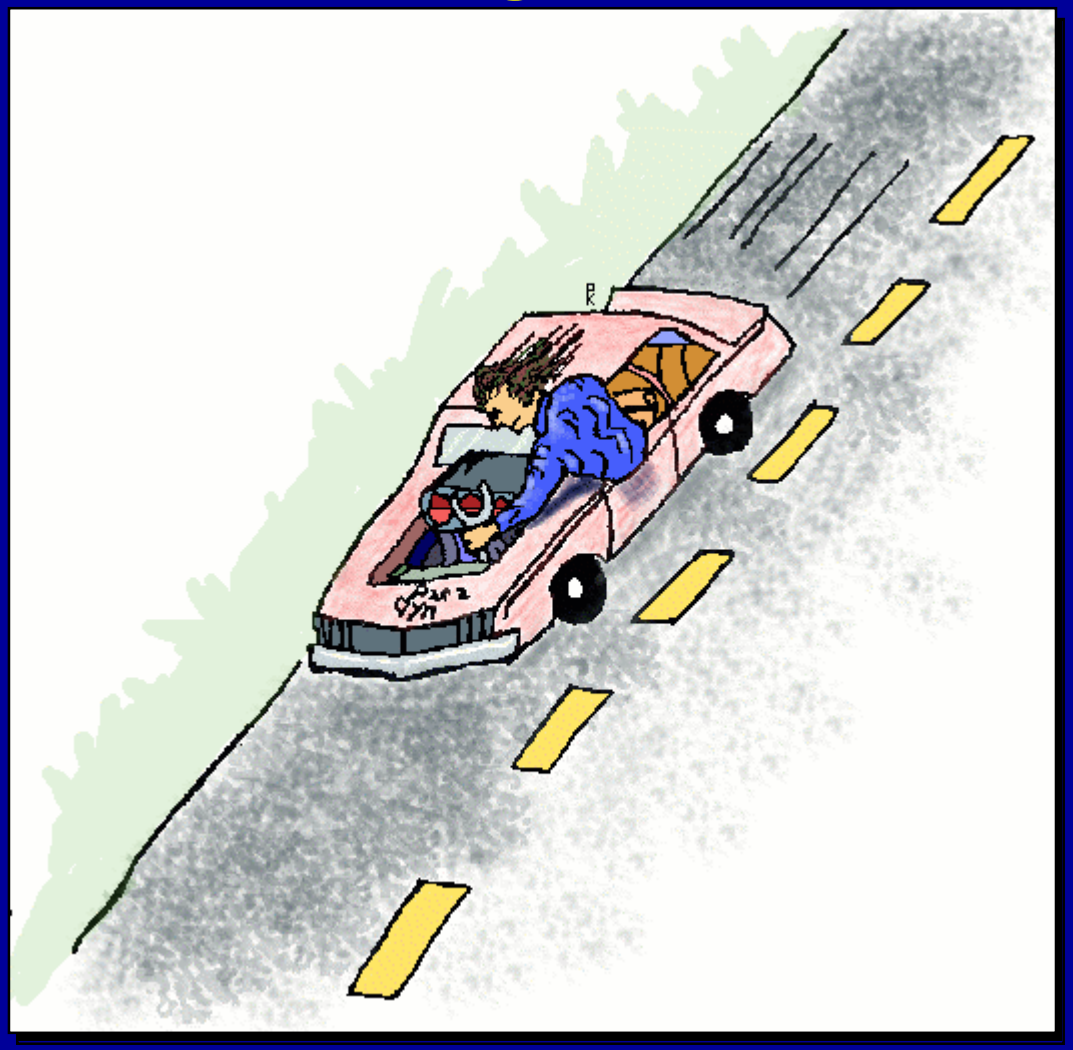
A single infrastructure for

- Profiling, debugging, code coverage
- Optimizations
- Extensibility

The foundation for an evolving OS

Measures and constantly adapts itself to runtime usage patterns

The Big Picture



http://www.cs.wisc.edu/paradyn