Fine-Grained Dynamic Kernel Instrumentation for OS Optimization

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

Evolving Operating Systems

– Code changes in response to runtime behavior

Fine-grained dynamic kernel instrumentation for:

– Performance measurement

– Performance assertions

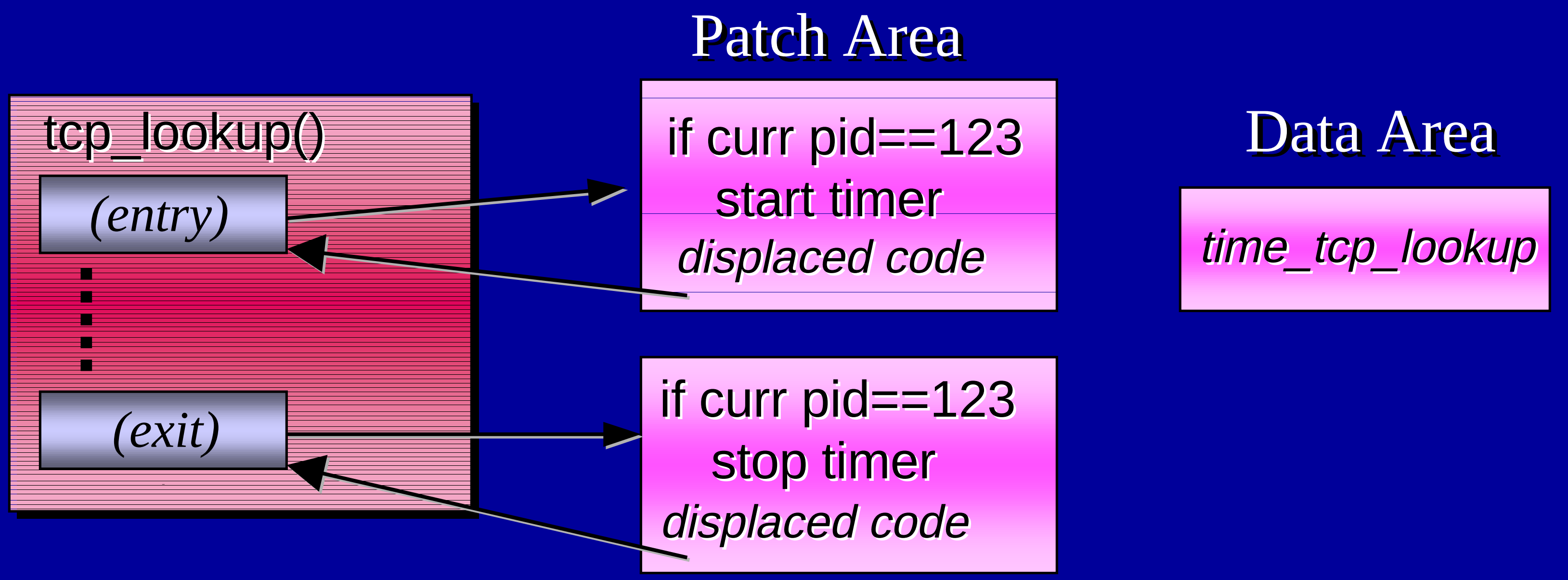
– Optimizations

  - Custom policies
  - Code rewriting

Measurement

- Primitives
  - Counts, elapsed cycles
  - On-chip counters (cache miss cycles, etc.)
- Predicates
  - 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

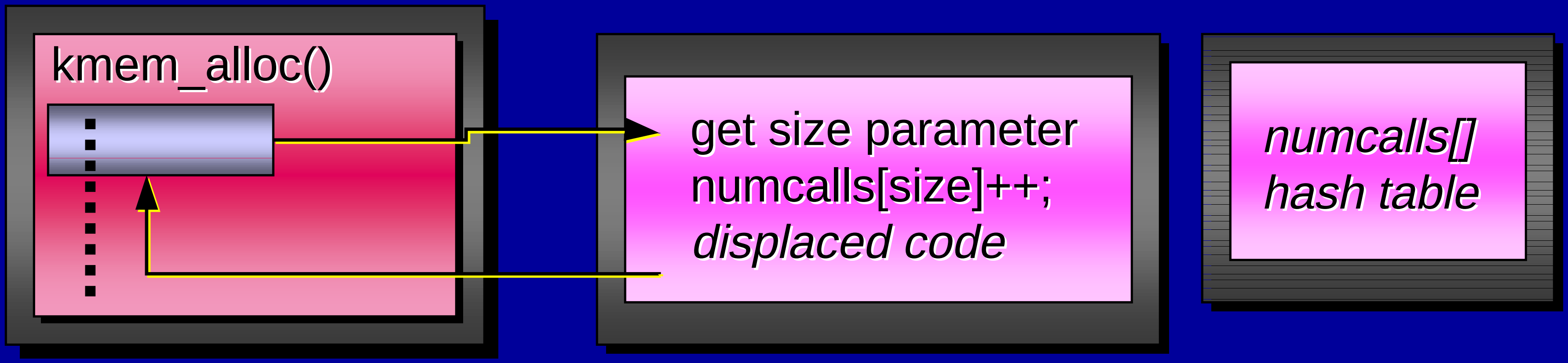
Time De-muxing TCP Packets



- Replace timer primitive with on-chip counter
  - Number of icache miss cycles
  - Branch mispredict stall cycles

Optimization: Specialization

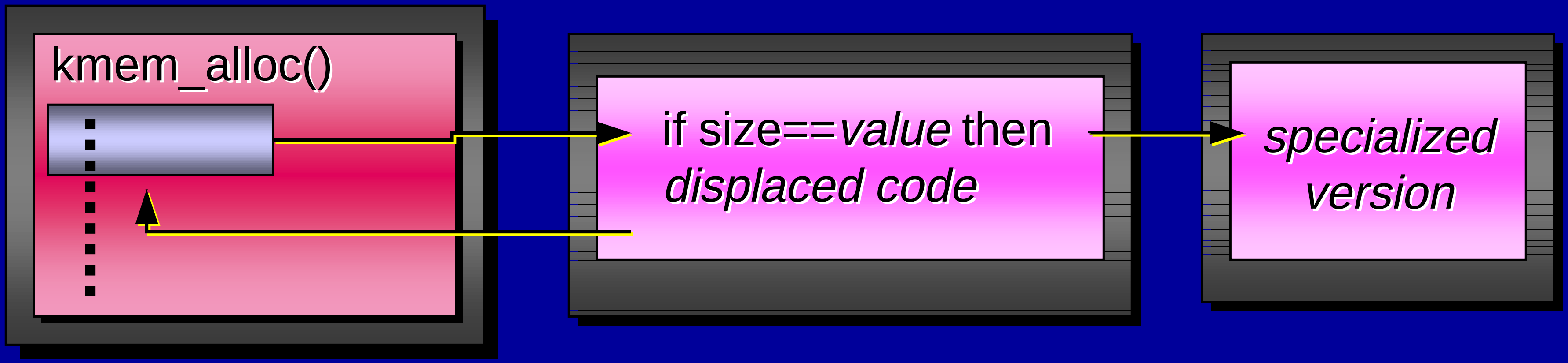
- Profile:



- Decision: examine hash table
- Generate specialized version:
  - choose fixed value & run constant propagation
  - expect unconditional branches & dead code

Motivation: Specialization

- Splice in the specialized version:



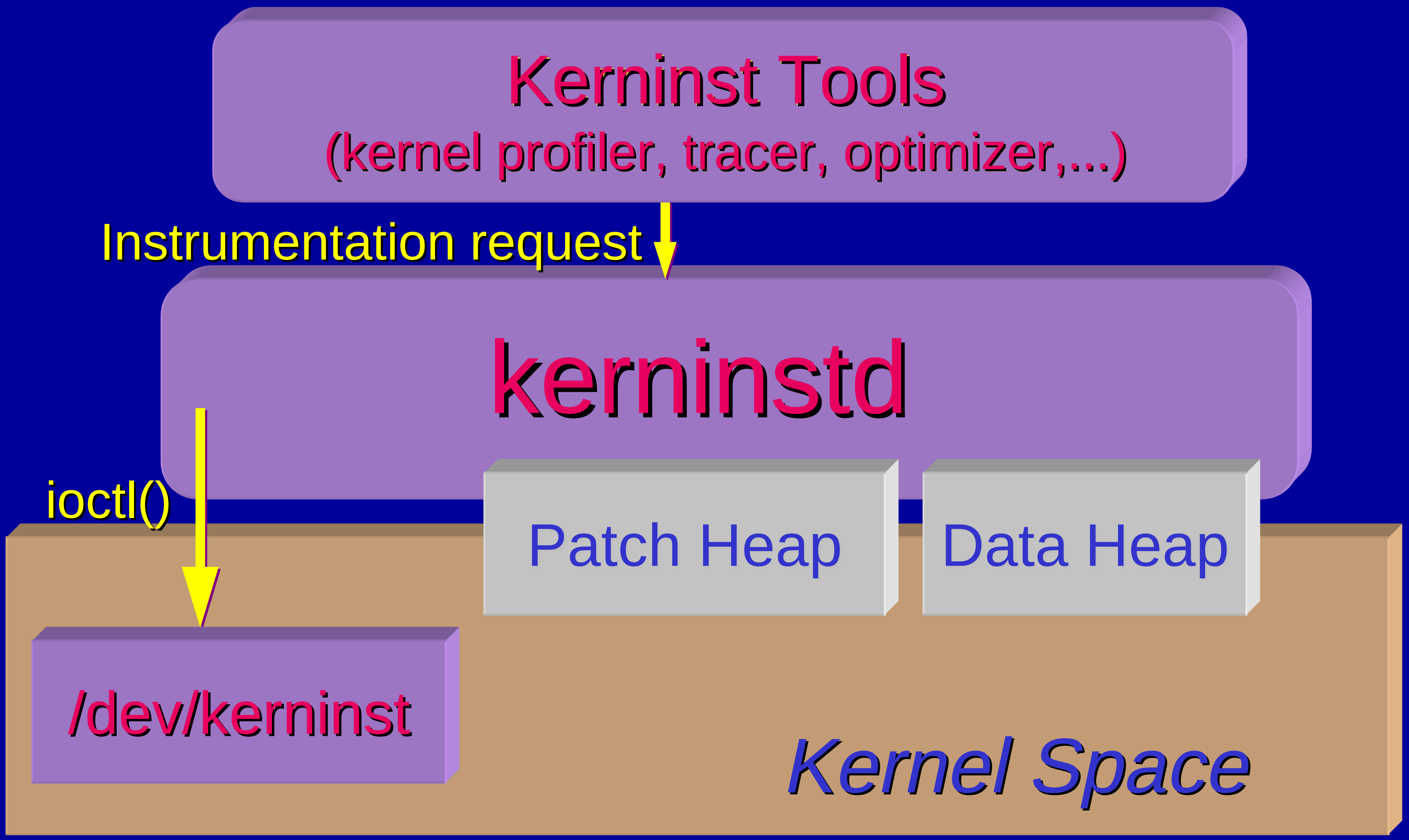
- Patch calls to kmem_alloc
  - – Detect constant values for size, where possible
  - – If specialized version appropriate, patch call
    - No overhead in this case

Technology to Make it Happen

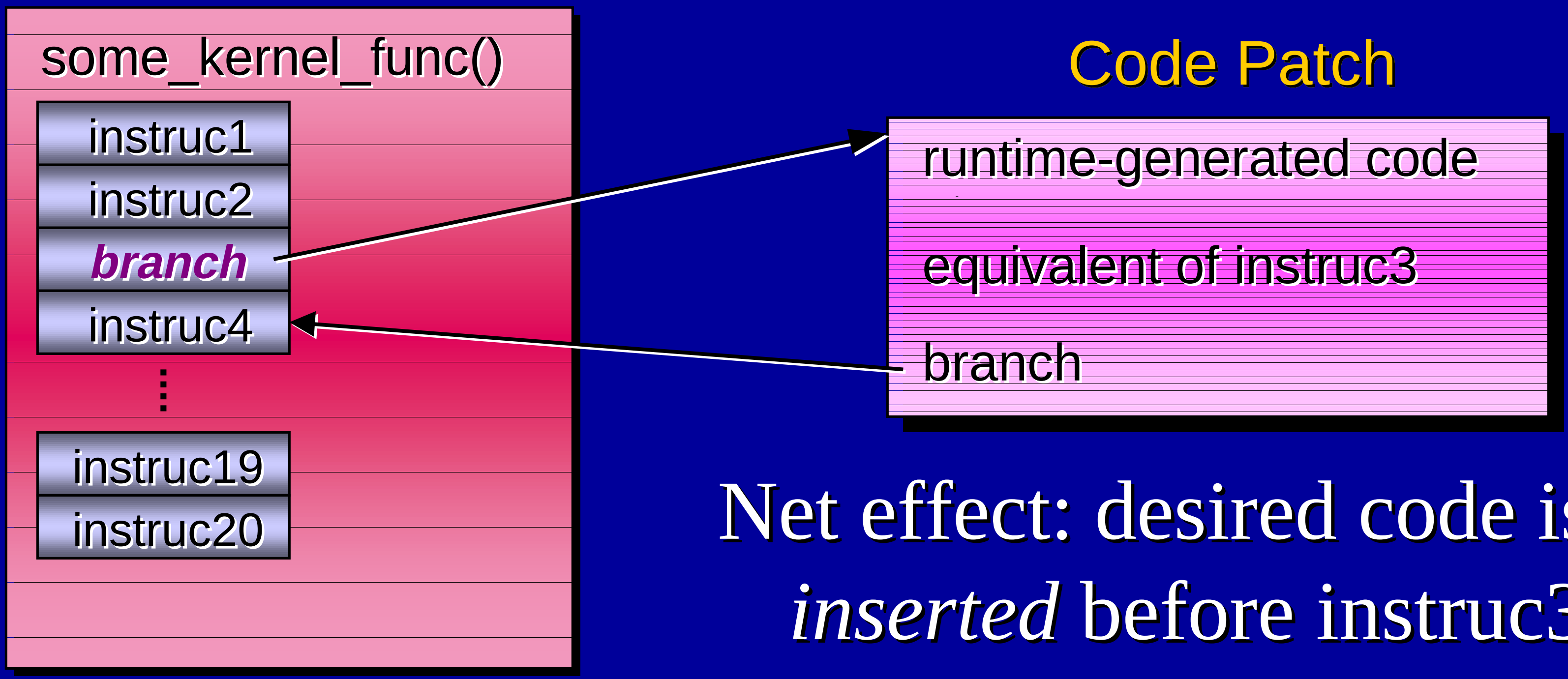
KernInst: fine-grained dynamic kernel instrumentation

- 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 7 on UltraSparc

Our System: *KernInst*



KernInst Splicing



Net effect: desired code is *inserted* before instruc3

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

kerninstd: Startup

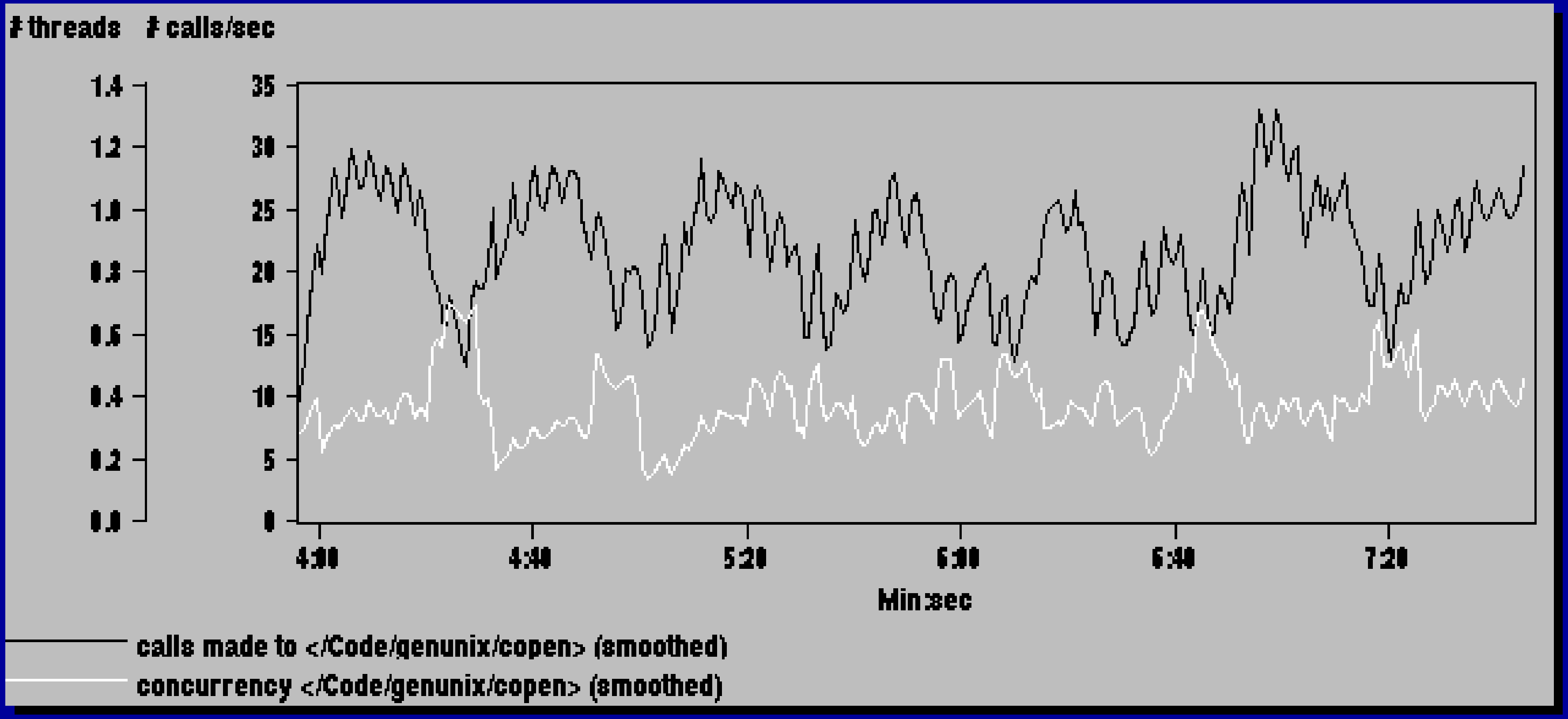
- Create heaps
- Read kernel symbol table
  - With assistance from /dev/kerninst
- Parses kernel code into CFG
- Finds unused registers
  - Inserted code will use these registers (avoid spills)
- Fast: about 20 seconds

Web Proxy Server Measurement

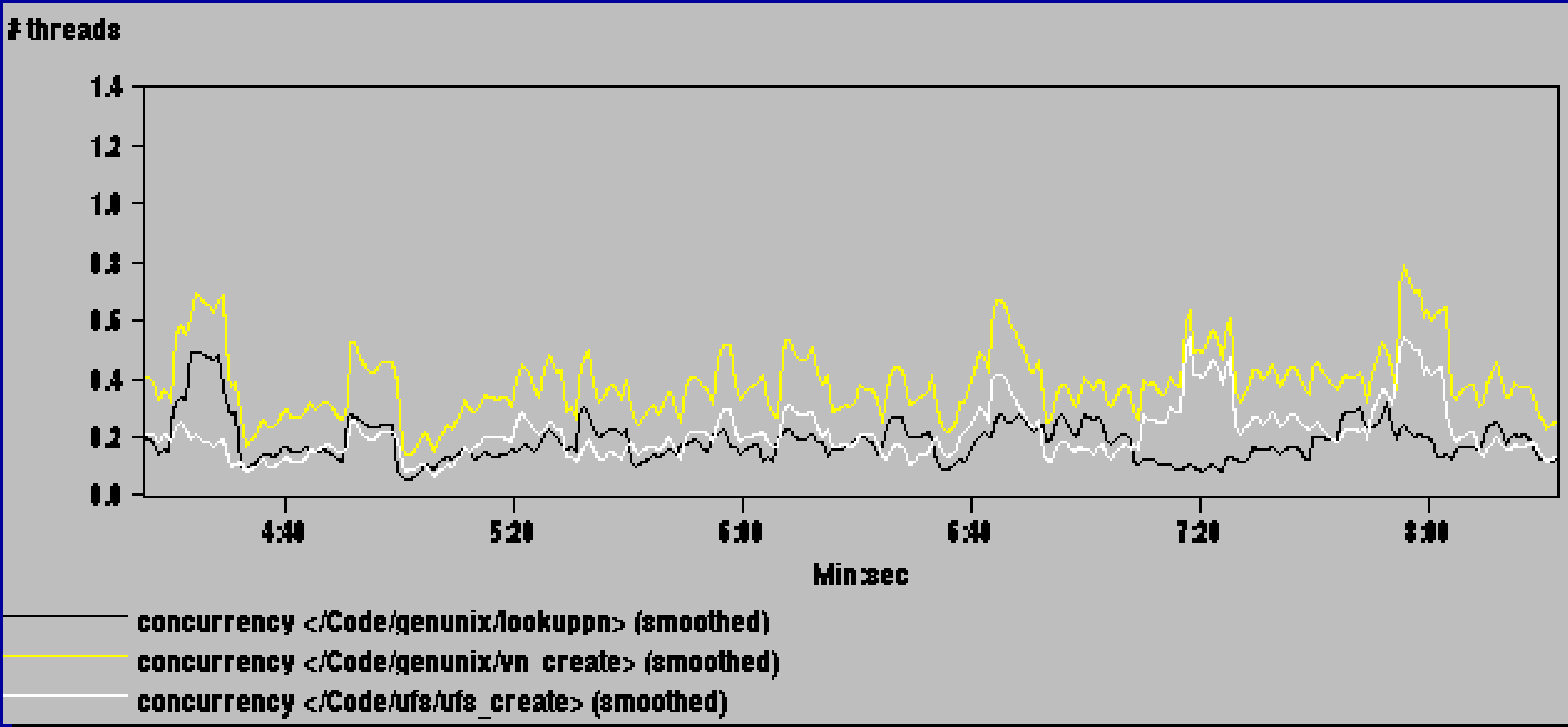
- Using kperfmon GUI
  - Number of calls made to a kernel function
  - Number of kernel threads executing within a kernel function (“concurrency”)
- Squid v1.1.22 http proxy server
  - Caches HTTP objects in memory and on disk
  - We used KernInst to understand the cause of two Squid disk I/O bottlenecks.

Web Proxy Server Measurement

- Profile of the kernel open() routine



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



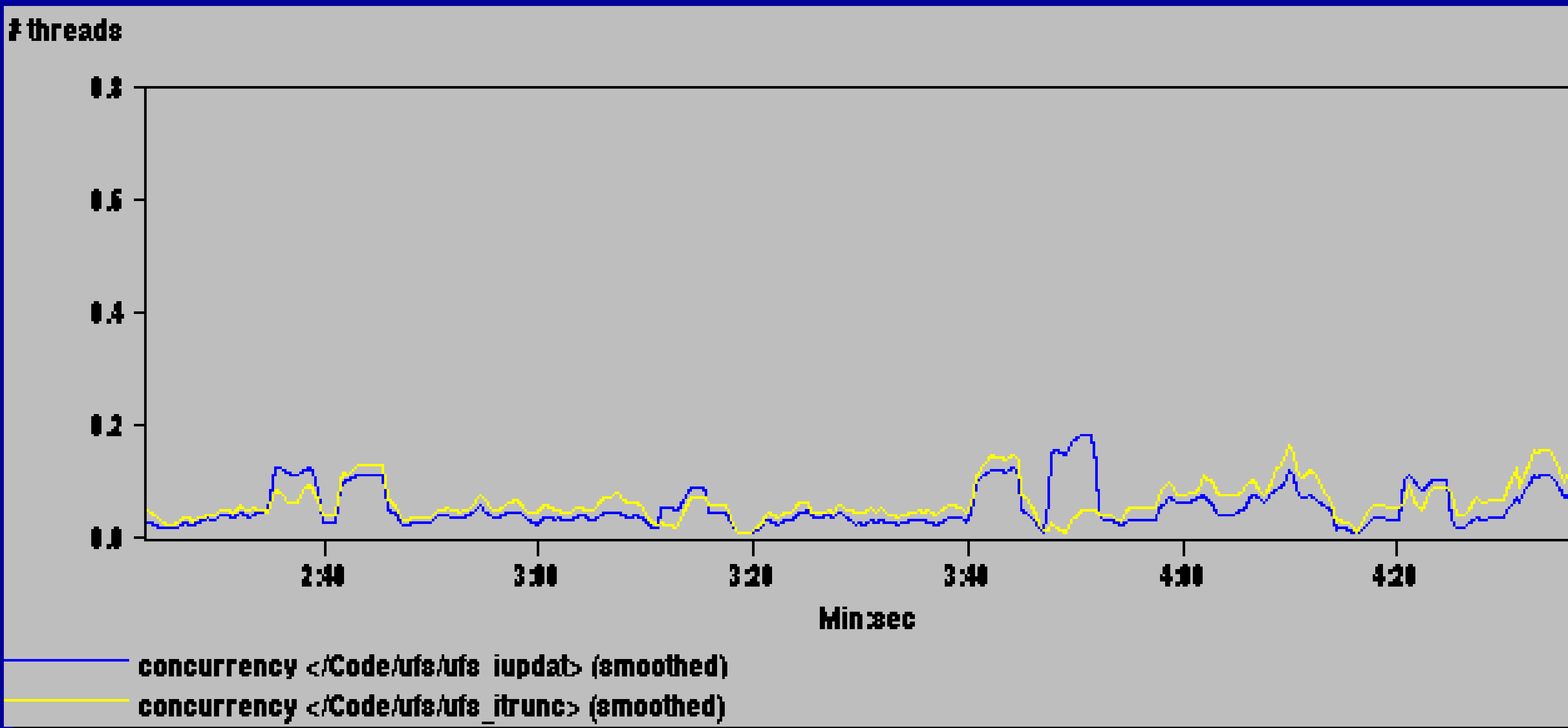
- open() calling vn_create; has 2 sub-bottlenecks:
  - lookuppn (a.k.a. namei): path name translation (20%)
  - ufs_create: file create on local disk (20%)

File Creation Bottleneck

- How Squid manages its on-disk cache:
  - 1 file per cached HTTP object
  - A fixed-size hierarchy of cache files
  - Stale cache files overwritten
- lookuppn bottleneck (dnlc_lookup)
  - Too many files overwhelms DNLC
- File creation bottleneck (ufs_itrunc)
  - When overwriting a stale cache file: truncates first
  - UFS semantics: meta-data changed synchronously

File Creation Optimization

- Overwrite cache file; truncate only if needed



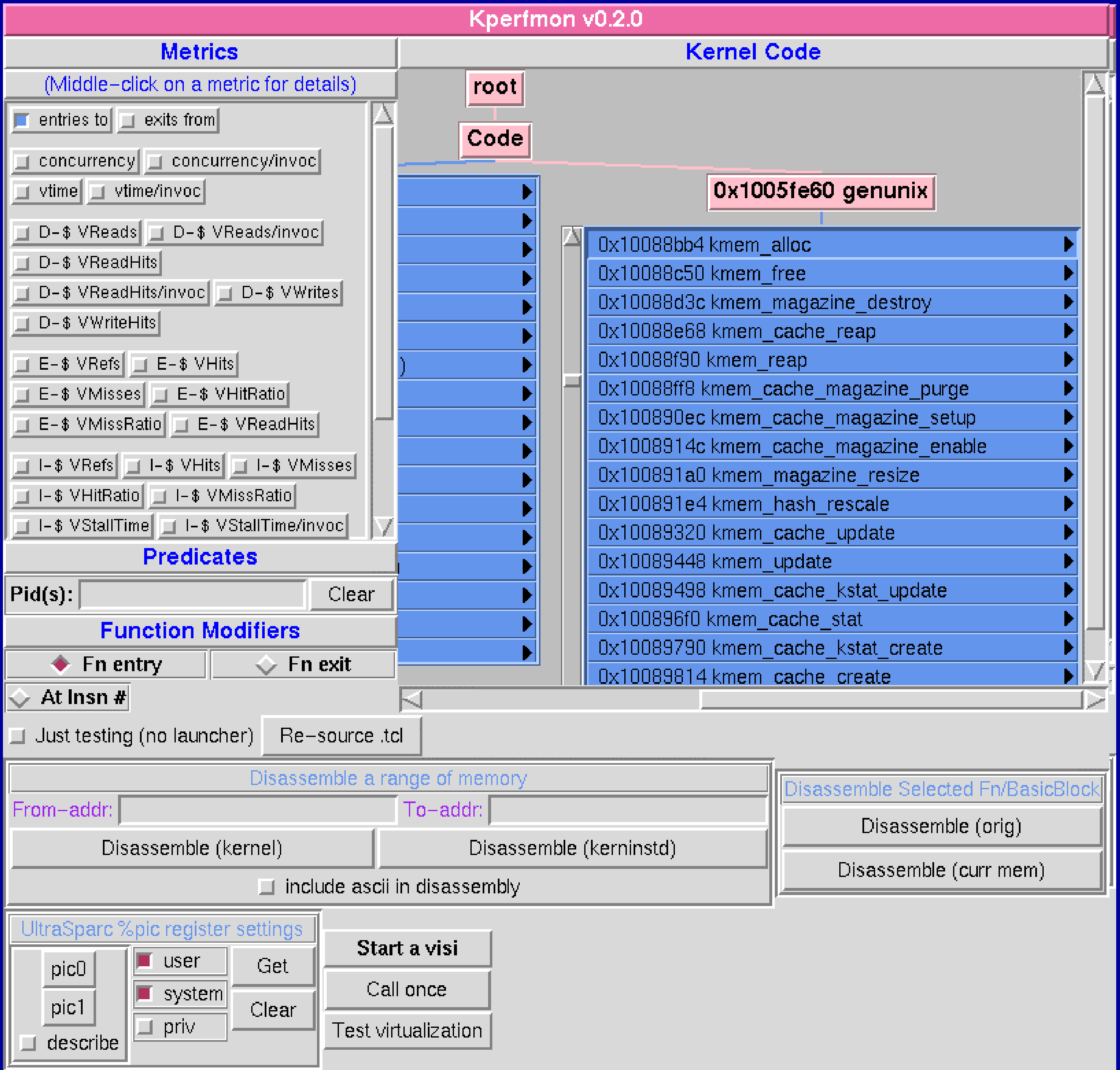
- What took 20% now takes 6%

Kperfmon

Single-click on function(s) or basic block(s) to select.

Single-click on metric(s) to select.

Then pull down the “Start a visi” menu to start a visualization process.



Kperfmon: Metrics

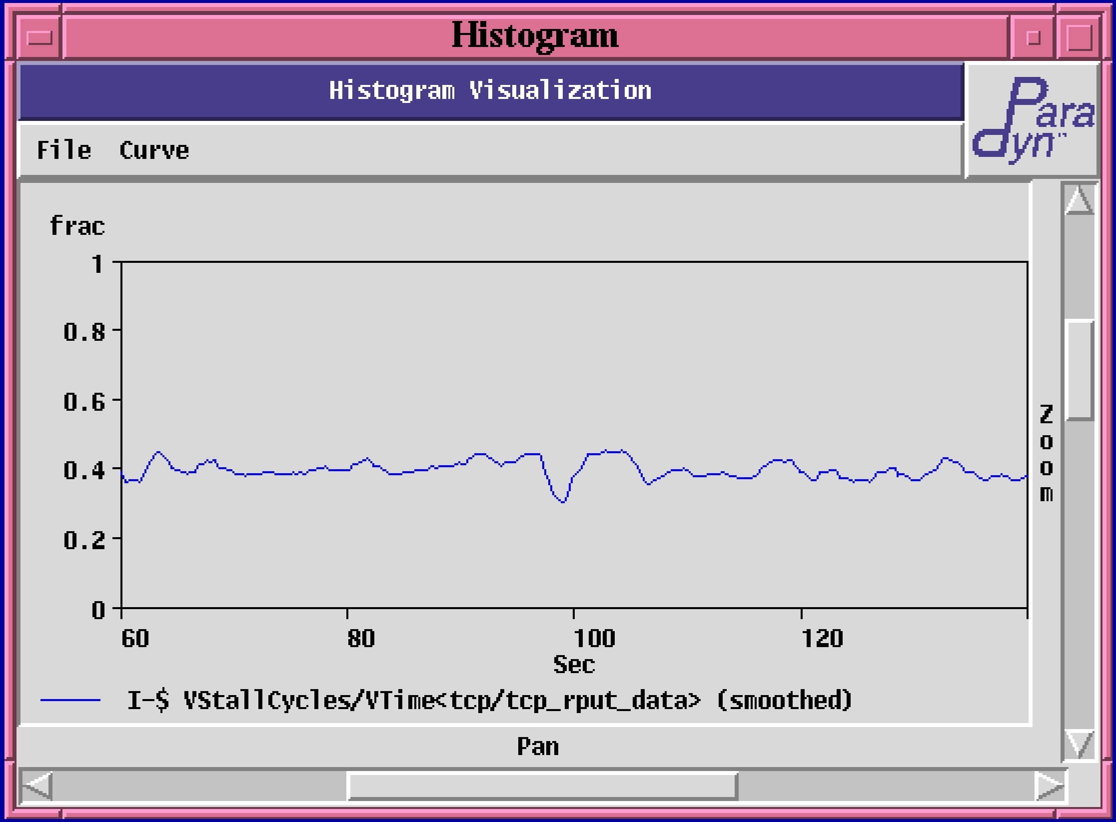
- Counts
  - Functions, basic blocks, or individual instructions
- Concurrency (# kthreads executing)
  - Start timer on entry, stop on exit(s)
  - Thread-seconds (wall time seconds) in a routine
  - Per-invocation available (concurrency/invoc)
- Virtualized metrics (vtime, cache reads, etc.)
  - Start with usual “wall” measurements (start on entry, stop on exit)
  - How to exclude time spent context switched out?

Metrics: Virtualization

- On kthread switch-out:
  - Stop all active vtimers
    - They must have been started by this kthread
    - Use per-cpu timers to handle multiprocessors
  - Make a note of the vtimers that were stopped
- On kthread switch-in:
  - Get vtimers stopped at last switch-out of this thread
  - Restart those vtimers

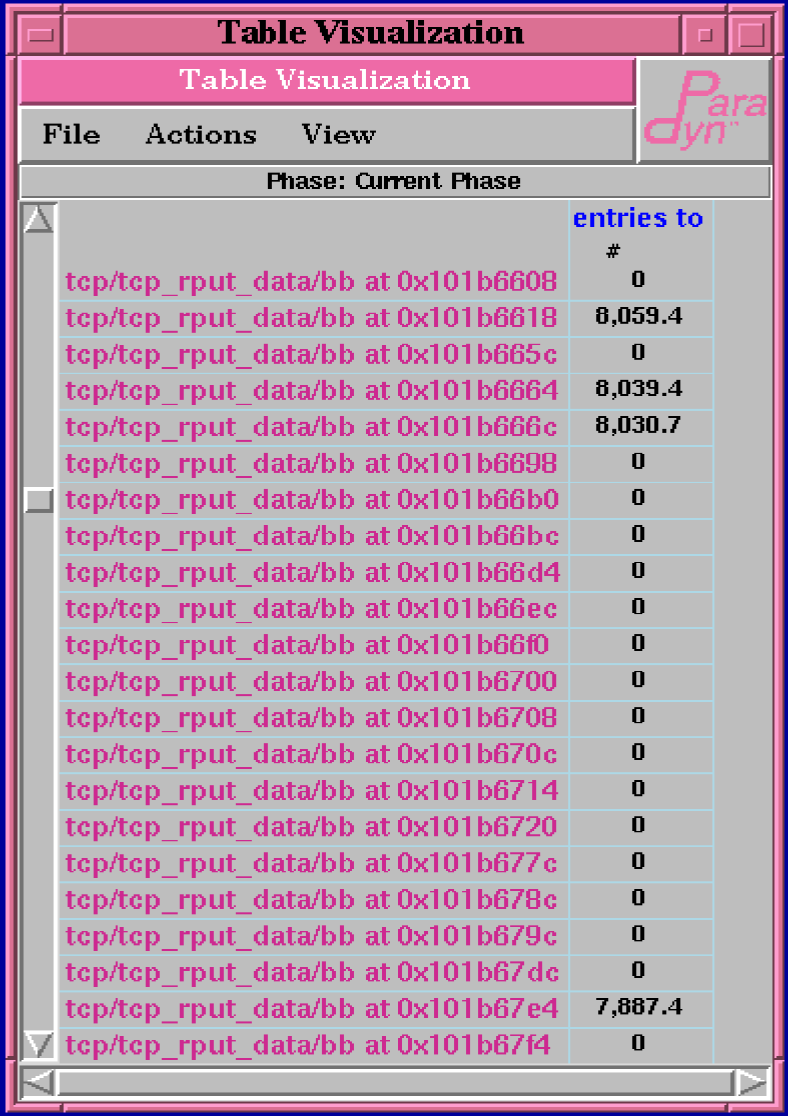
Outlining

- Profile based dynamic optimization
- Spending a high fraction of time stalled on I-cache miss handling?
- Measure with dynamic instrumentation



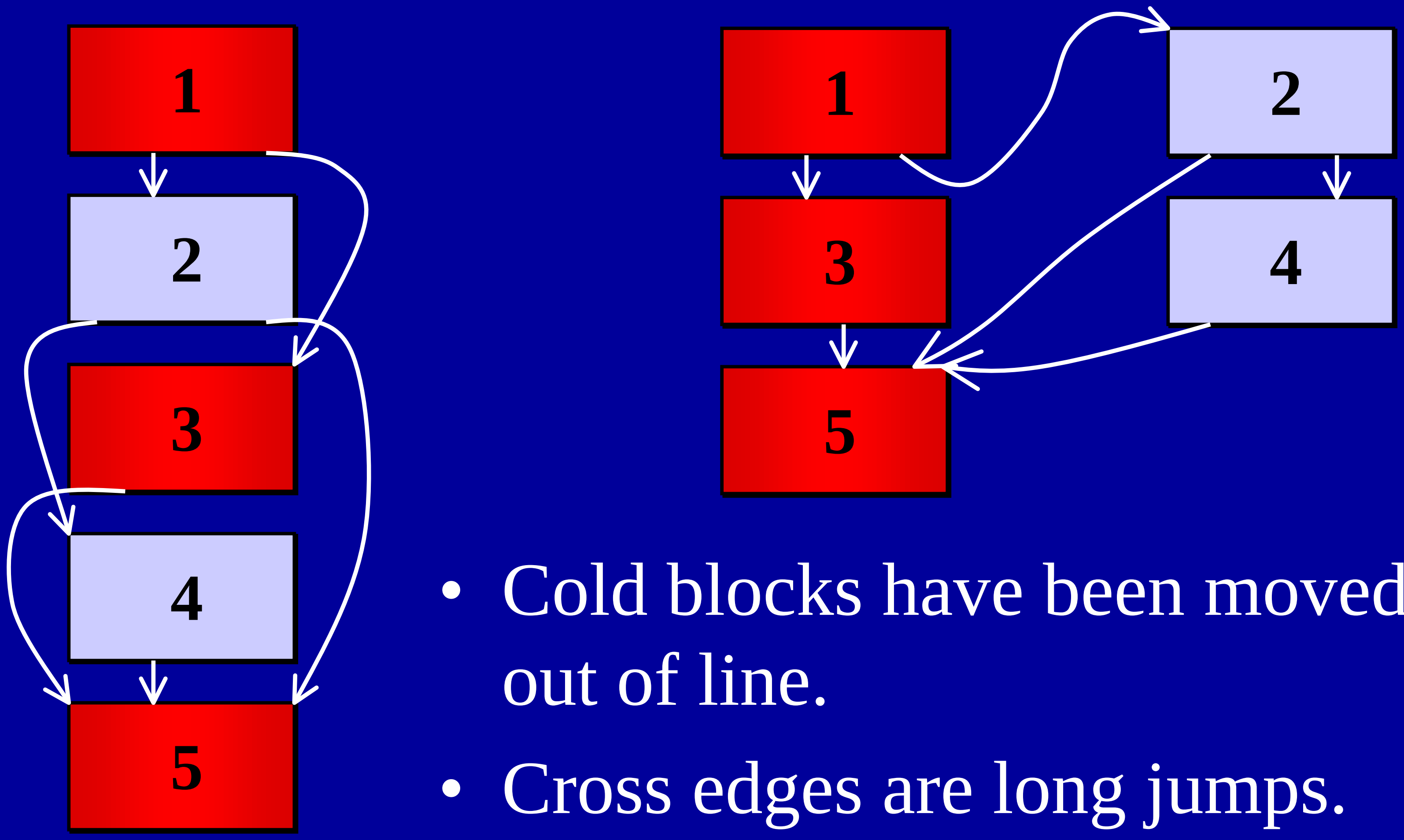
Outlining: Estimate Benefit

- Many cold basic blocks?
- Measure dynamically
- tcp_rput_data():
  - – 32% of blocks are hot
  - – 68% of blocks are cold
    - Typical of kernel (extensive error checking, calls to panic, etc.)



	entries to #
tcp/tcp_rput_data/bb at 0x101b6608	0
tcp/tcp_rput_data/bb at 0x101b6618	8,059.4
tcp/tcp_rput_data/bb at 0x101b665c	0
tcp/tcp_rput_data/bb at 0x101b6664	8,039.4
tcp/tcp_rput_data/bb at 0x101b666c	8,030.7
tcp/tcp_rput_data/bb at 0x101b6698	0
tcp/tcp_rput_data/bb at 0x101b66b0	0
tcp/tcp_rput_data/bb at 0x101b66bc	0
tcp/tcp_rput_data/bb at 0x101b66d4	0
tcp/tcp_rput_data/bb at 0x101b66ec	0
tcp/tcp_rput_data/bb at 0x101b66f0	0
tcp/tcp_rput_data/bb at 0x101b6700	0
tcp/tcp_rput_data/bb at 0x101b6708	0
tcp/tcp_rput_data/bb at 0x101b670c	0
tcp/tcp_rput_data/bb at 0x101b6714	0
tcp/tcp_rput_data/bb at 0x101b6720	0
tcp/tcp_rput_data/bb at 0x101b677c	0
tcp/tcp_rput_data/bb at 0x101b678c	0
tcp/tcp_rput_data/bb at 0x101b679c	0
tcp/tcp_rput_data/bb at 0x101b67dc	0
tcp/tcp_rput_data/bb at 0x101b67e4	7,887.4
tcp/tcp_rput_data/bb at 0x101b67f4	0

Outlining: Generate New Version



- Cold blocks have been moved out of line.
- Cross edges are long jumps.

Outlining: Installing

- Known call sites changed to new address
  - Leave behind a jump in original function to handle indirect calls
- Note that measurement and installation uses the same underlying technology
- Each step of outlining can be automated!
  - A self-evolving kernel, optimizing in response to actual run-time behavior.

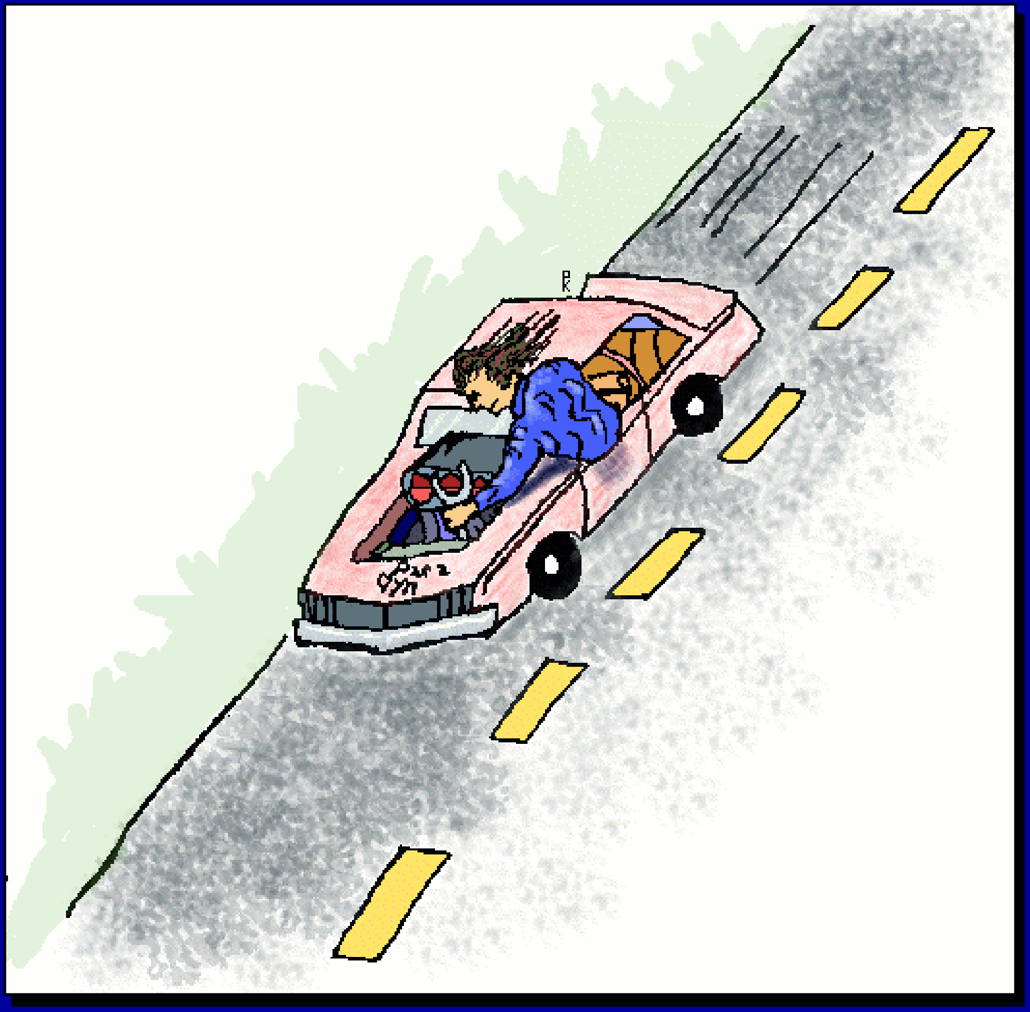
KernInst: Current Work

- Runtime optimizations (Ari)
- Safety and security (Zhichen Xu)
  - Now: must trust code that kperfmon inserts
  - Allow untrusted instrumentation code
- x86/Linux port (Vic Zandy)
  - As before, overwrite just 1 instruction
    - The catch: tough given variable-length instructions

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