Adaptive Operating Systems: An Architecture for Evolving Systems

Barton P. Miller

Alex Mirgorodskii **Ariel Tamches**

Michael Brim **Igor Grobman**

bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706-1685
USA

The Vision

The OS as a dynamically evolving entity:

- Annotations (orthogonal changes):
  - Performance profiling
  - Debugging, tracing
  - Testing (e.g., code coverage)
  - Security audits
- Adaptations:
  - Install patches
  - Optimizations: specialization, outlining, custom algs.
- Safety: Validating dynamic code

Enabling Technology

Fine-grained dynamic kernel instrumentation

- Inserts runtime-generated code into kernel
- Dynamic: everything at runtime
  - no recompile, reboot, or pause
- Fine-grained: insert (almost) anywhere
- Commodity kernel w/no modifications (Solaris)
  - Code does not need to be in a special form
- General: insert anything
    - Safety & security delegated to higher-level tool

Motivation: Code Annotations

Performance Monitoring

- Increment counter
- Start a timer on entry; stop the timer on exit
- Number of icache misses, FLOPS, etc.

On-line debugging

- Breakpoint if argument equals NULL
- Dynamically inserted assertions
  - e.g. Purify-type memory access checks

Code Annotations (2)

Tracing / auditing

Testing: fine-grained code coverage

- Instrument each basic block: set flag to true
- Remove coverage instrumentation once reached

Motivation: Code Adaptations

Optimizations (kernel code *evolves*)

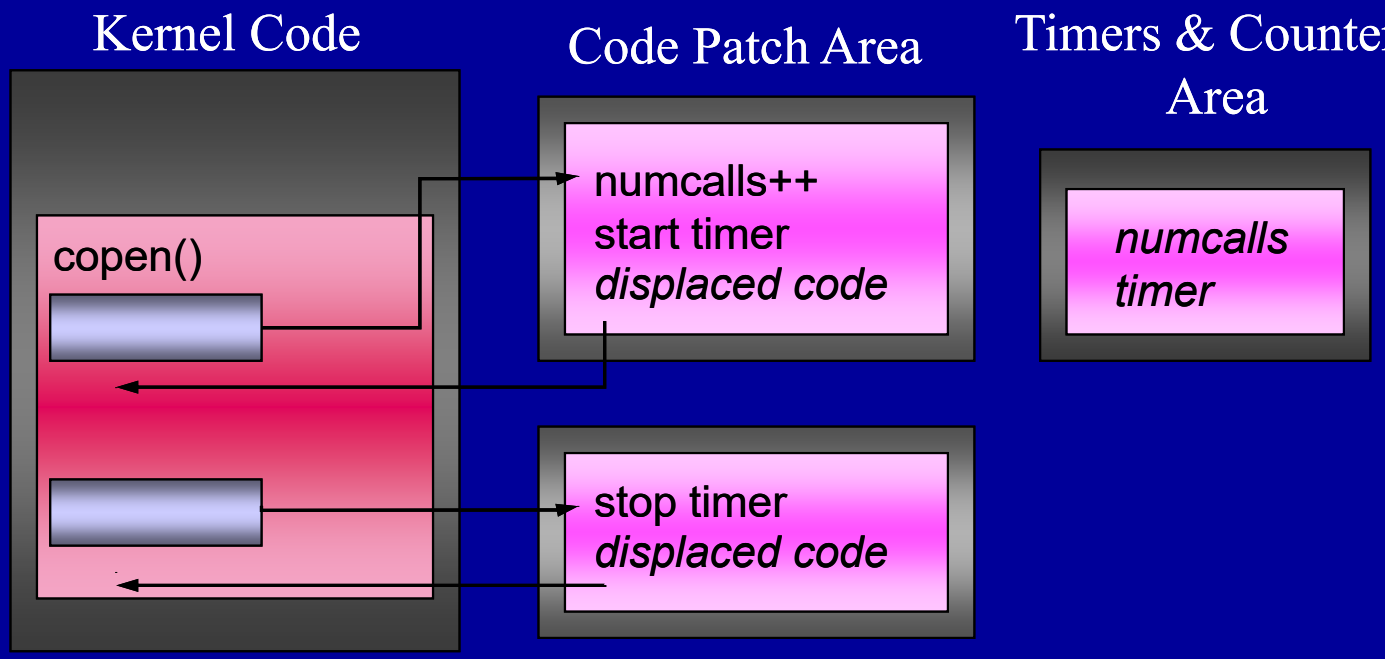
- Specialization (partial evaluation)
  - First annotate desired function to find common parameters
  - Analyze results (remove annotation)
  - Generate specialized version & install
- Outlining (move seldom-referenced code away)
  - Annotate function: is I-Cache performance bad?
  - Annotate function: find cold basic blocks
  - Generate outlined version of function & install

Code Adaptations (2)

Extensibility (process-customized policies)

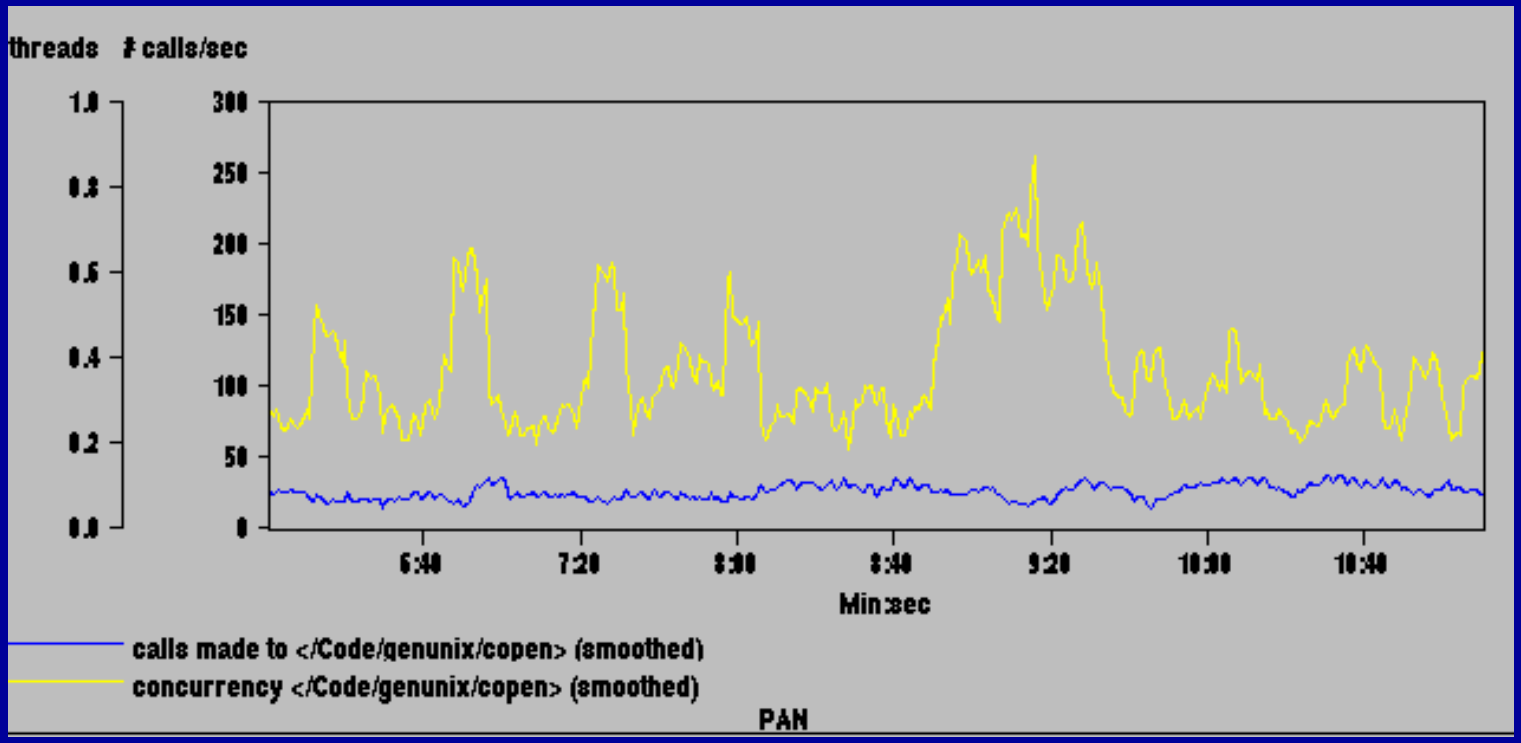
- Generate customized function, put in kernel
- Insert code at policy function: "if pid is *pid* then jump to custom version"
  - e.g. customize disk block prefetch strategy
- Like extensible kernels but:
  - Unmodified commodity kernel
  - Any kernel function can be customized
  - Fine-grained: can customize parts of functions

Example of Instrumentation

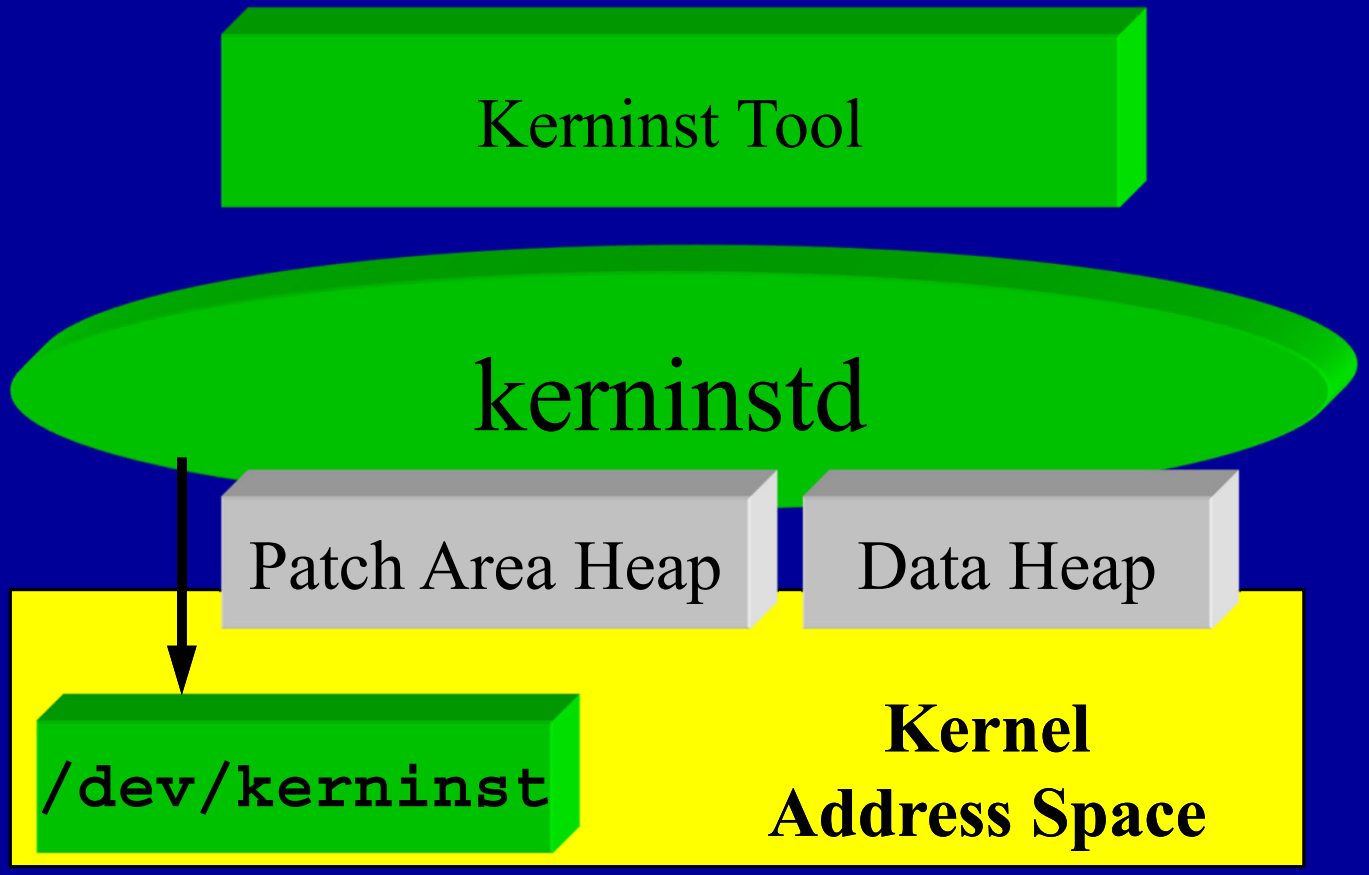


- Two annotations: count # of calls to copen; accumulate latency
- Can insert multiple instrumentation at one point

Web Proxy Server Measurement



Mechanisms: How KernInst Works



Bootstrapping

Kerninstd (a user-level process) must:

- Allocate patch area heap
- Get the kernel's symbol table
- Get permission to write to all kernel code (/dev/kmem only a partial solution)

By using a driver (/dev/kerninst):

- Installed on-the-fly (common OS feature)
- Communicate with kerninstd via file operations
- Special functions (segkmem_mapin) to write to nucleus

Structural Analysis

Create control-flow graph

- Get function addresses from kernel symbol table
- Parse machine code into basic blocks
- Fast: 20 seconds

Use free registers at instrumentation points

- For jumping to dynamically inserted code.
- For executing dynamically inserted code.
- Run interprocedural live register analysis algorithm

Splicing in Code…*Safely*

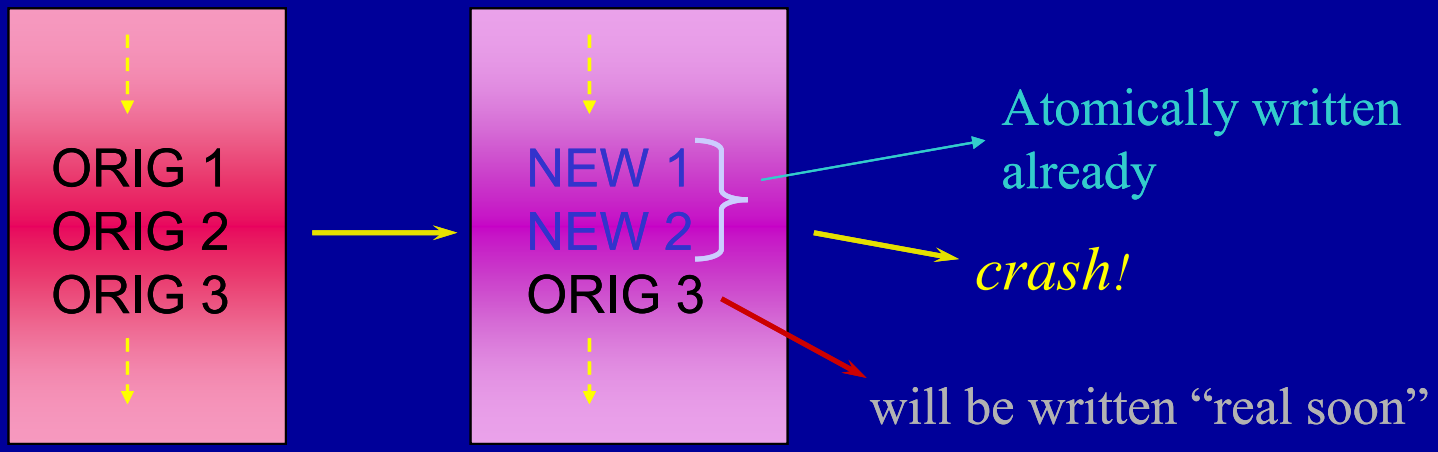
To splice in code at a point:

- Allocate & write code and data space in kernel.
- Modify kernel code at instrumentation site with a jump to allocated code.

But there is a potential safety hazard...

- Cannot pause kernel during splicing.
- A kernel thread might be executing in or around code while it is being changed.

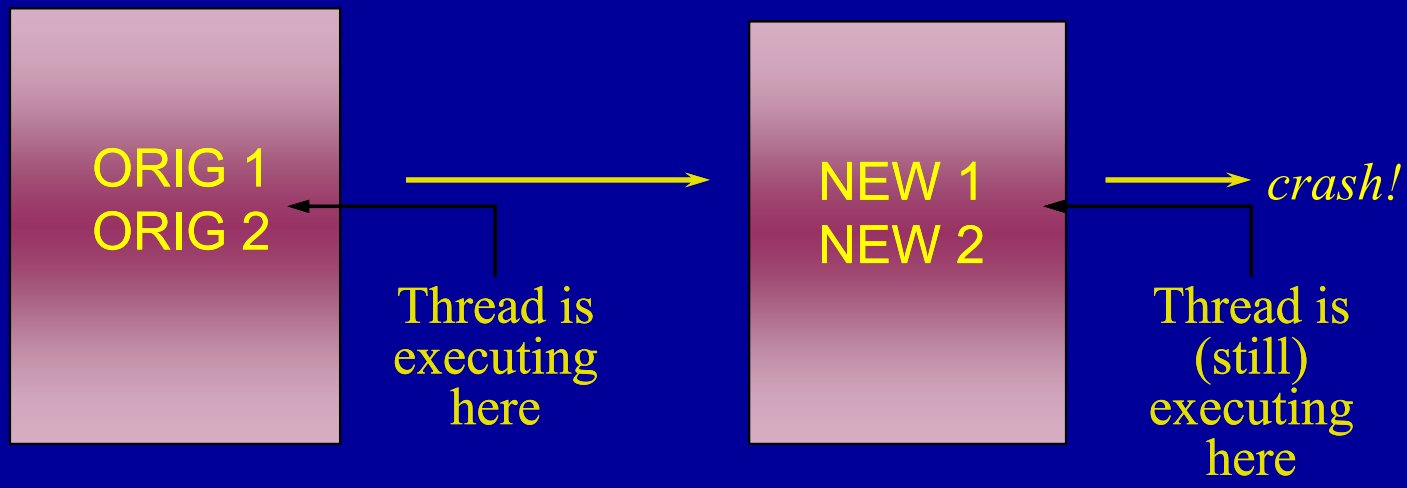
Code Splicing Hazard #1



Moral: splicing must be atomic

– On SPARC, this limits splices to (at most) two instructions.

Code Splicing Hazard #2



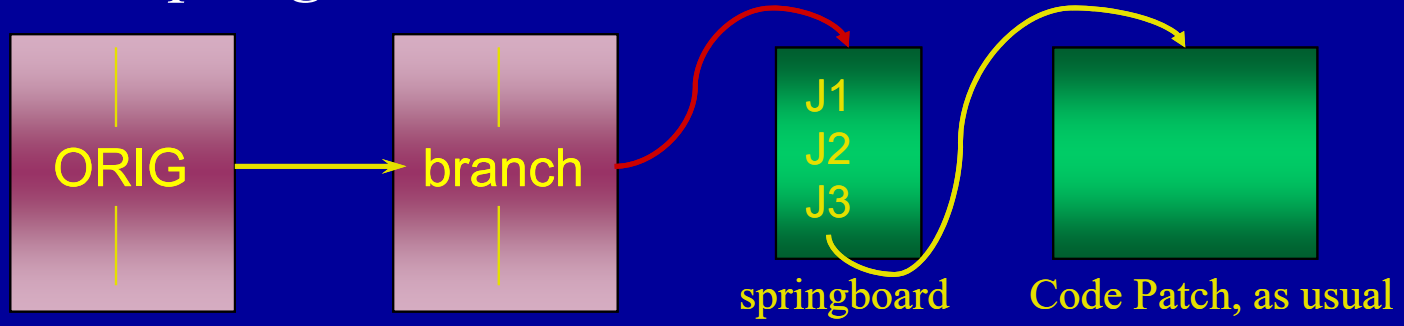
Moral: splicing must replace only one instruction.

Code Splicing: Reach Problem

Tough to reach patch with just 1 instruction!

- The patch space is usually too far from code.
- On SPARC, e.g., can jump only +/- 8MB in a single instruction (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 splice.
- SVR4 modules have initialization and termination routines that can be overwritten (_init and _fini).
- Other places are available (_start and main).
- At boot-time, we cause large patch areas to be allocated.

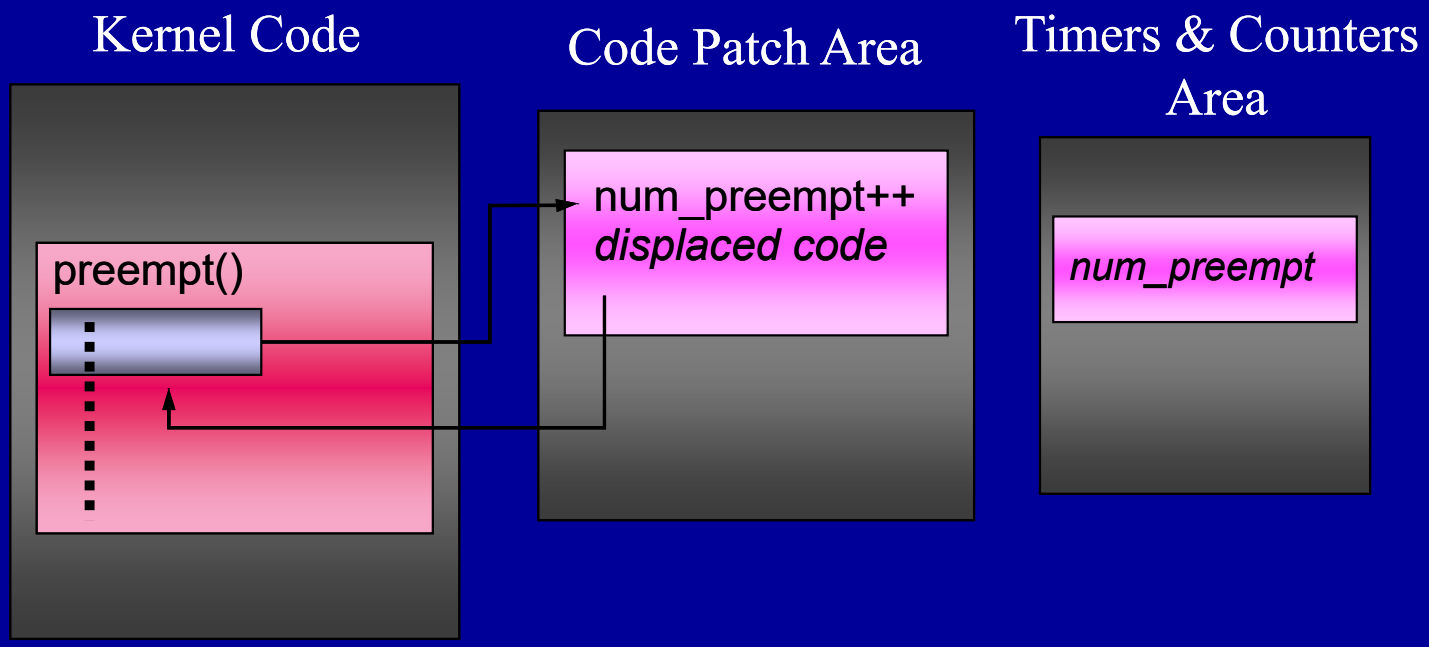
Performance Measurement

Hundreds of important kernel routines make great metrics

- Scheduling: preempt, disp, swtch
- Process creation: fork
- VM management: hat_chgprot, hat_swapin
- Network activity: tcp_lookup, tcp_wput, ip_csum_hdr, ip_rput, hmeintr

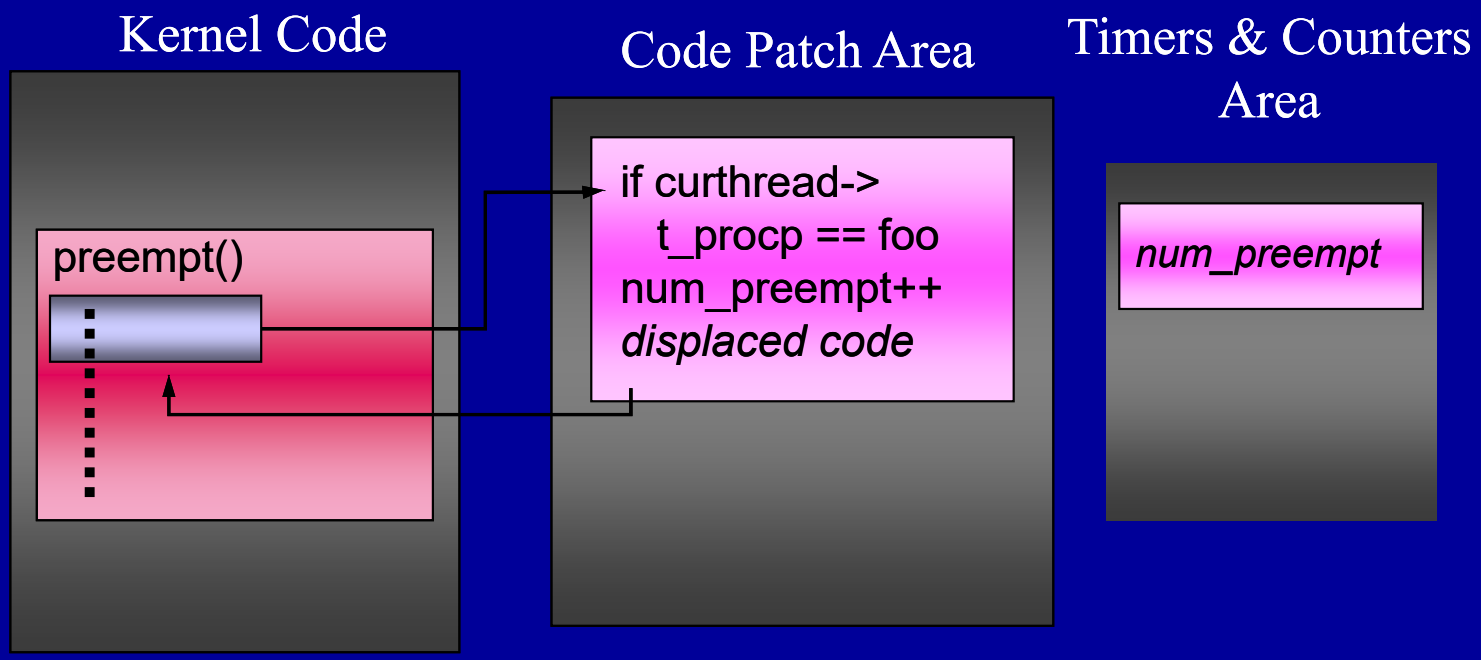
Kernel Metrics

Number of preemptions



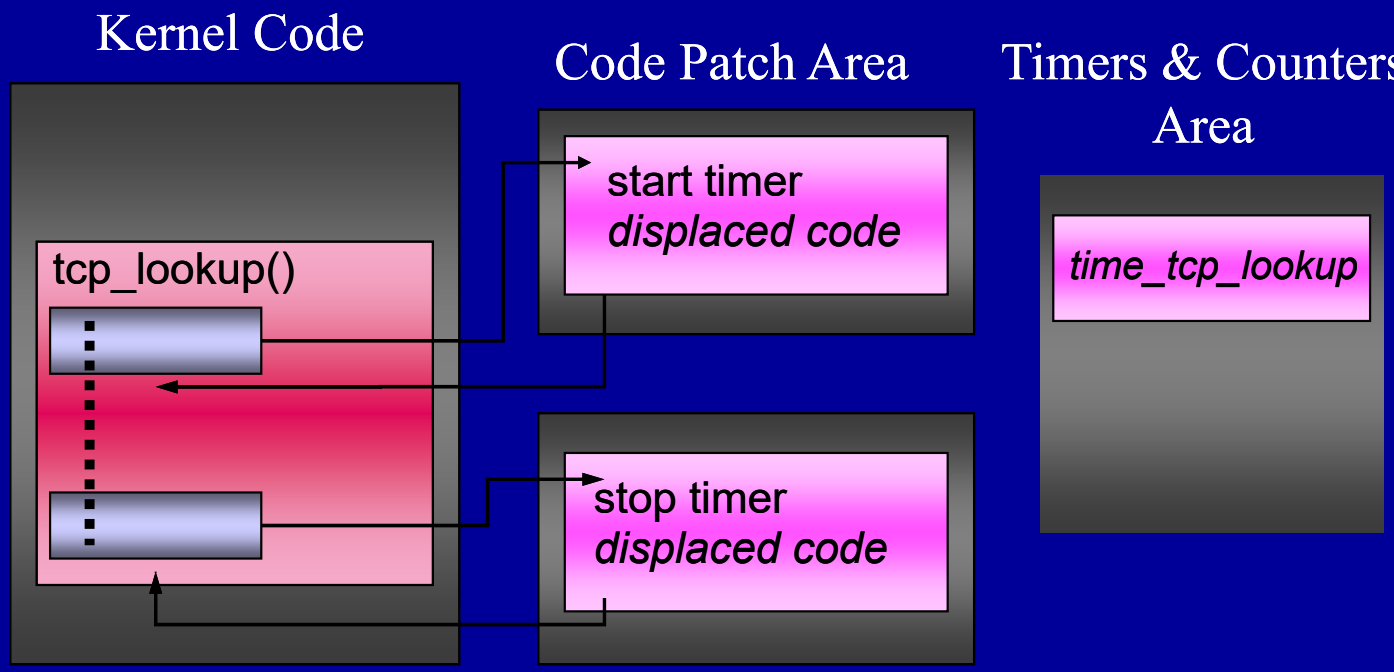
Kernel Metrics

Number of preemptions of process foo



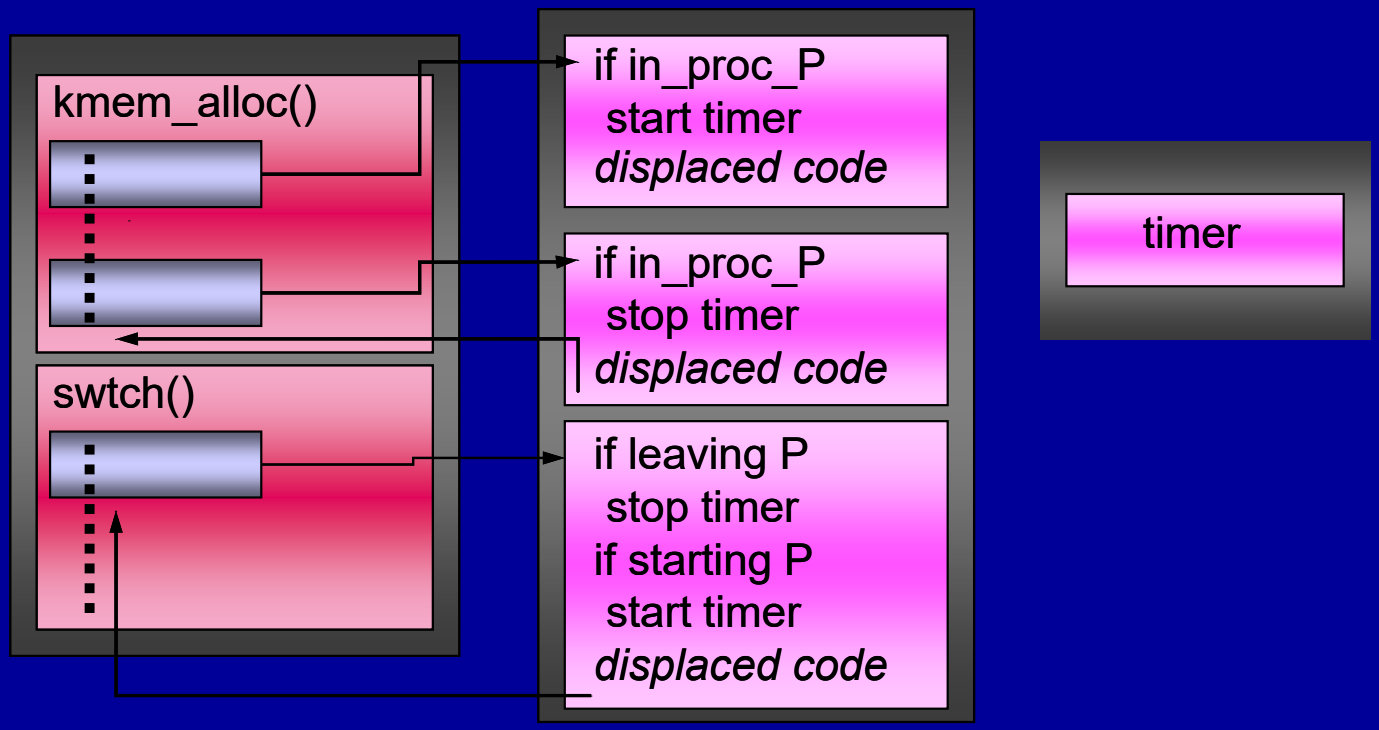
Kernel Metrics

Time spent filtering incoming TCP packets



Kernel Metrics

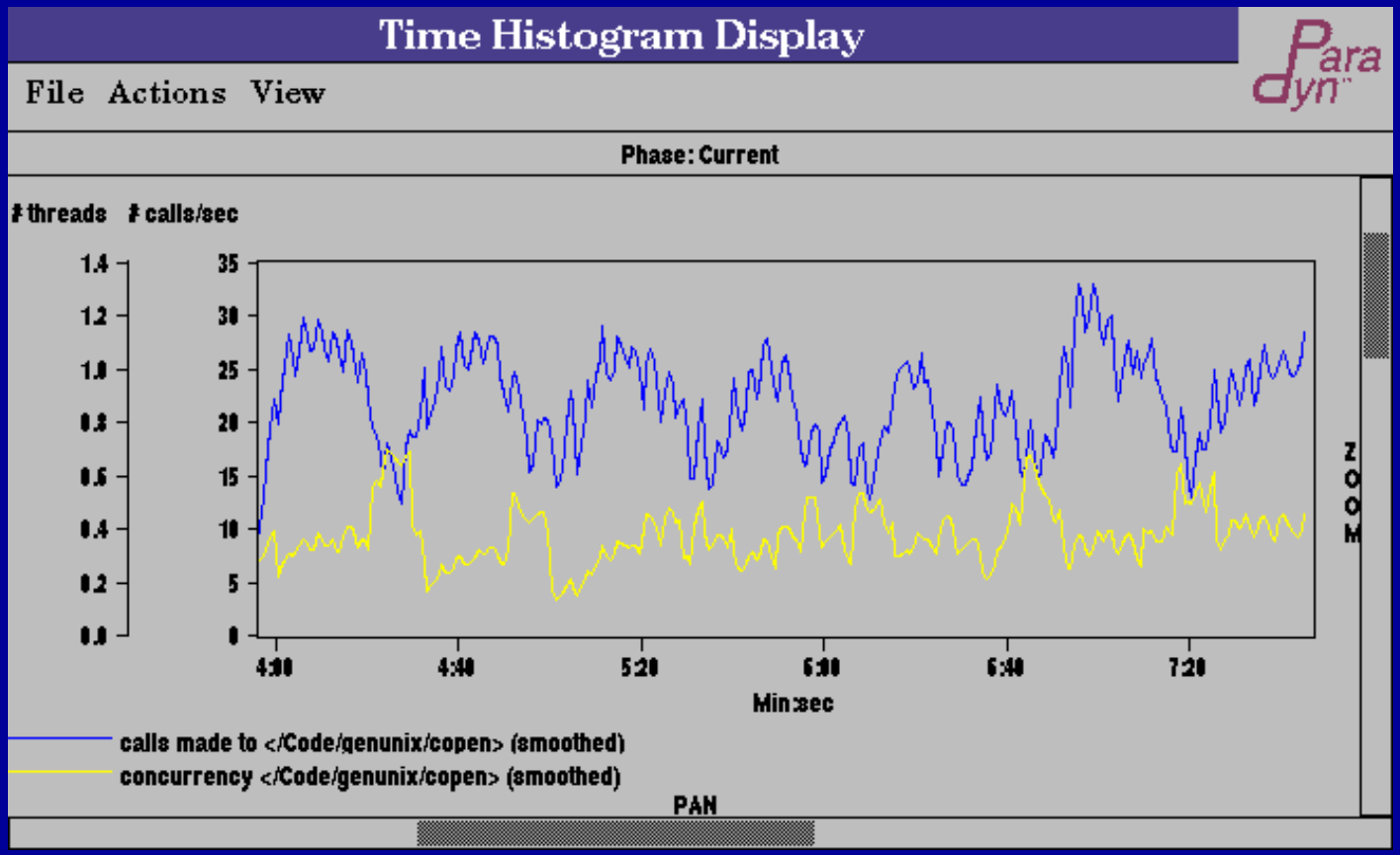
Virtual time spent allocating kernel memory



Web Proxy Server Measurement

- Performance profiling tool using KernInst
- Metrics used
  - Number of calls made to
  - Concurrency (average number of kernel threads executing in code at a given time)
- Squid http proxy server
- Wisconsin Proxy Benchmark
- Identified both kernel and application fixes!

copen handles file opens & creates



Web Proxy Server Measurement

copen major calls: falloc, vn_open

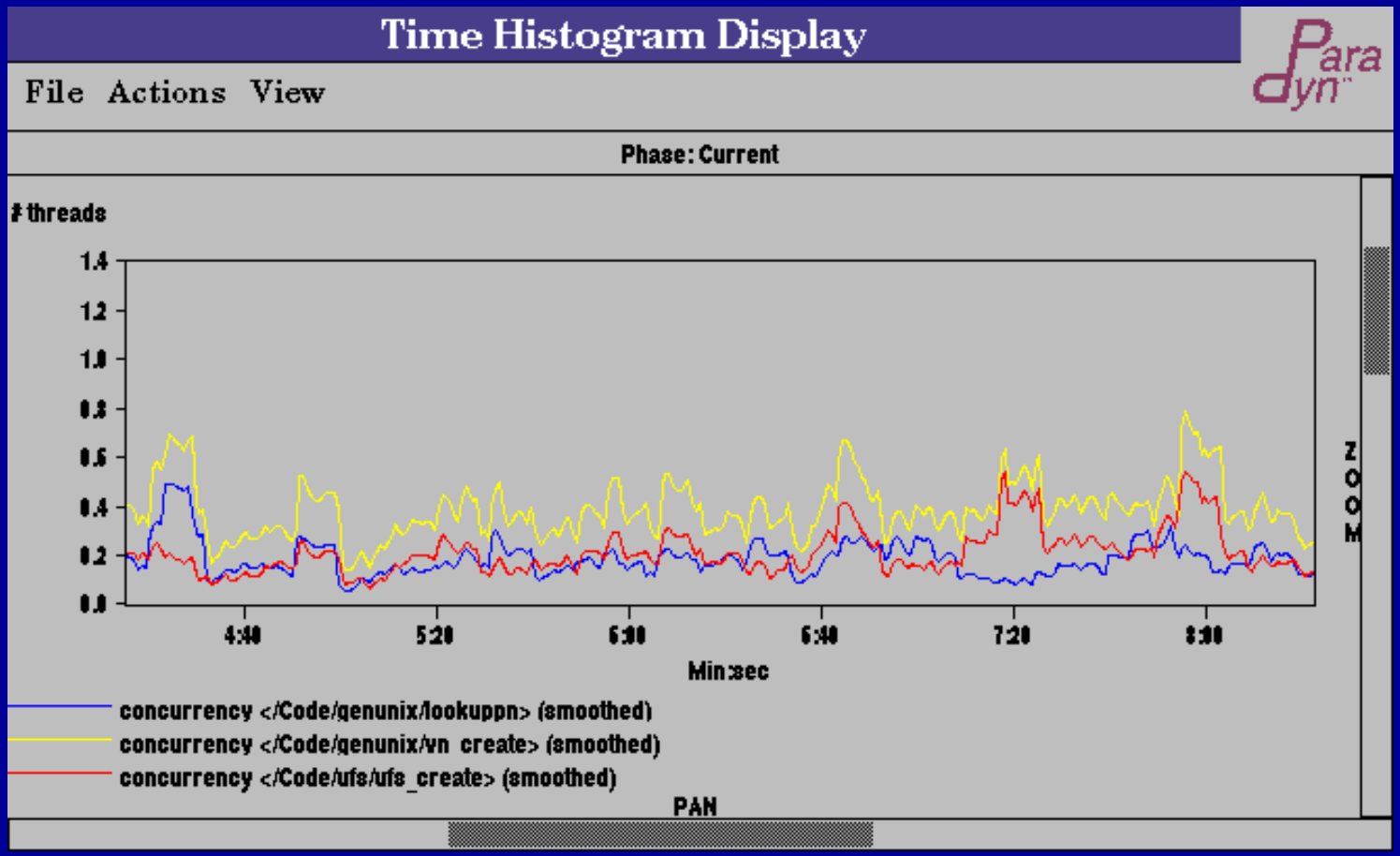
- falloc (surprisingly) not a bottleneck (0.2%) despite linear search

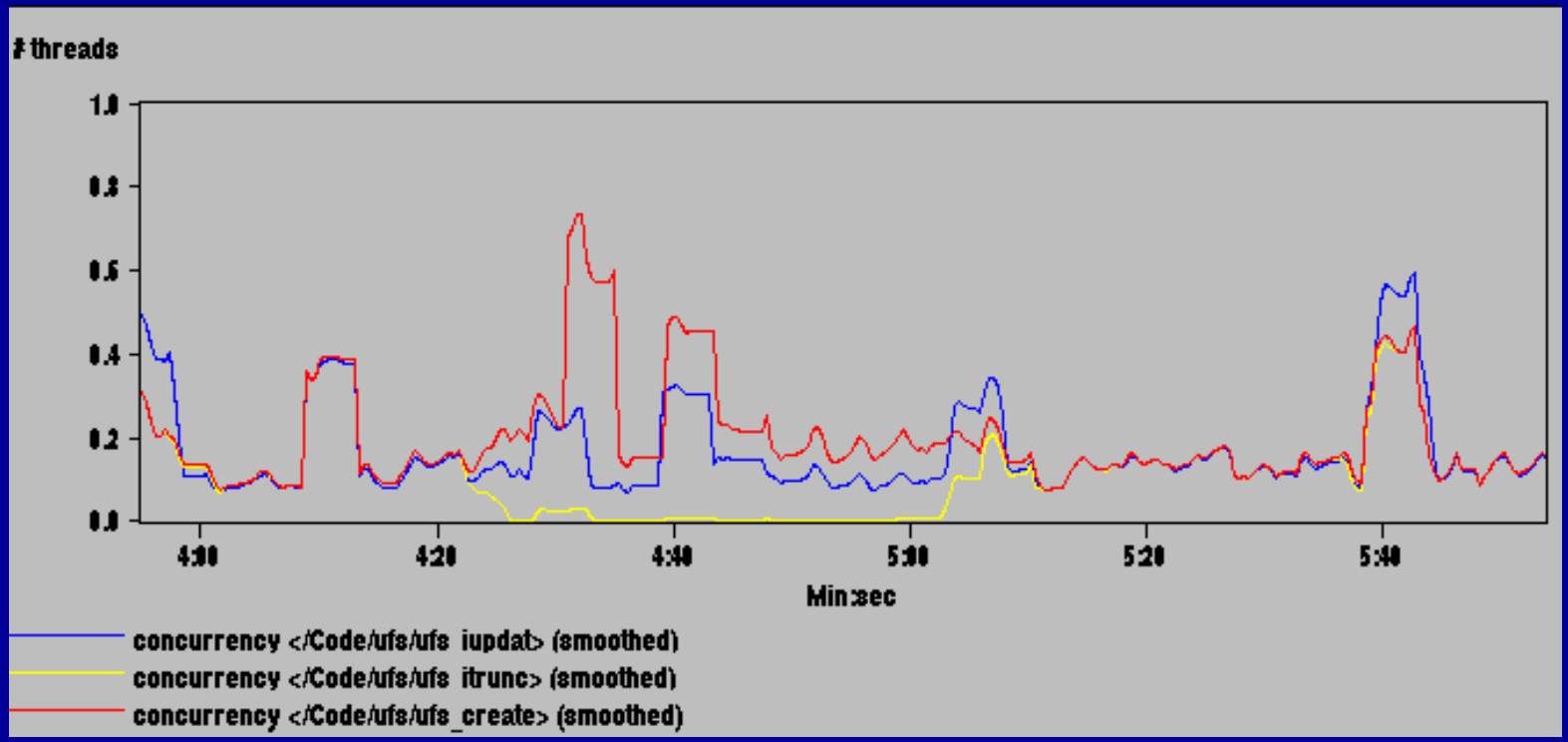
vn_open spending its time in vn_create

So proxy is creating files, not reading them

- calling open with O_TRUNC flag

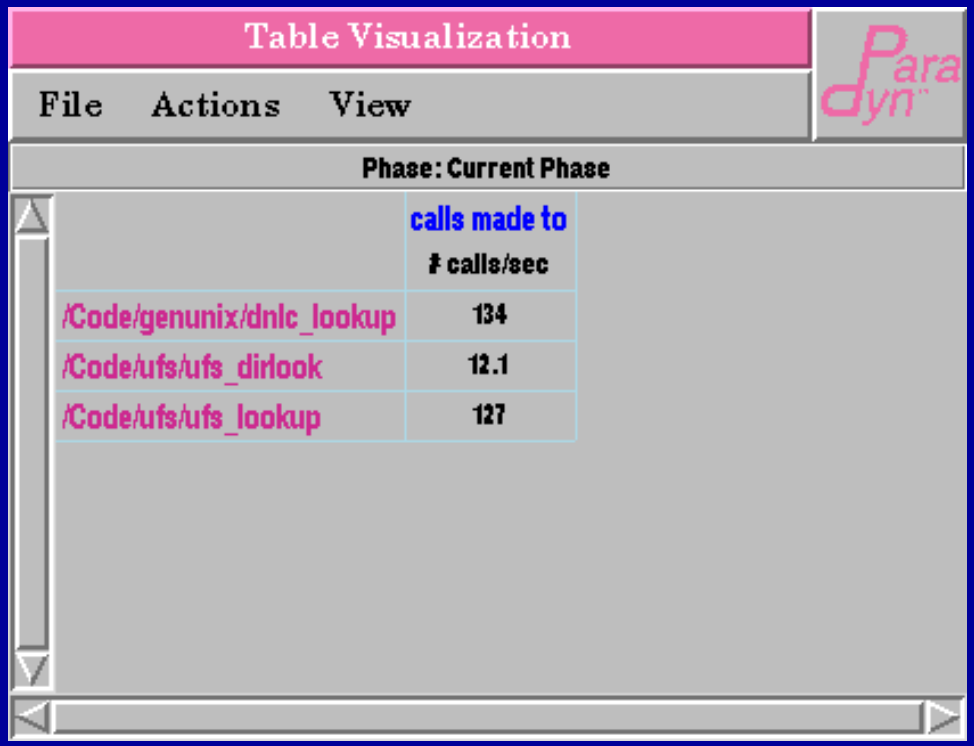
vn_create calls lookuppn and ufs_create





ufs_create: mostly ufs_itrunc

- Bottleneck: Truncating cache file to 0 size
- Fix: overwrite file and only trunc if new size less



Directory Name Lookup Cache (DNLC):

- Bottleneck: Not all name → vnode mappings fit
- Fix: Make the cache bigger!

Optimization

Use dynamic instrumentation to optimize kernel functions on-the-fly.

- Specialization
  - generate partially evaluated function for a fixed value of one or more parameters
- Outlining
  - can move infrequently accessed code blocks out of line (to patch space), improving icache performance.

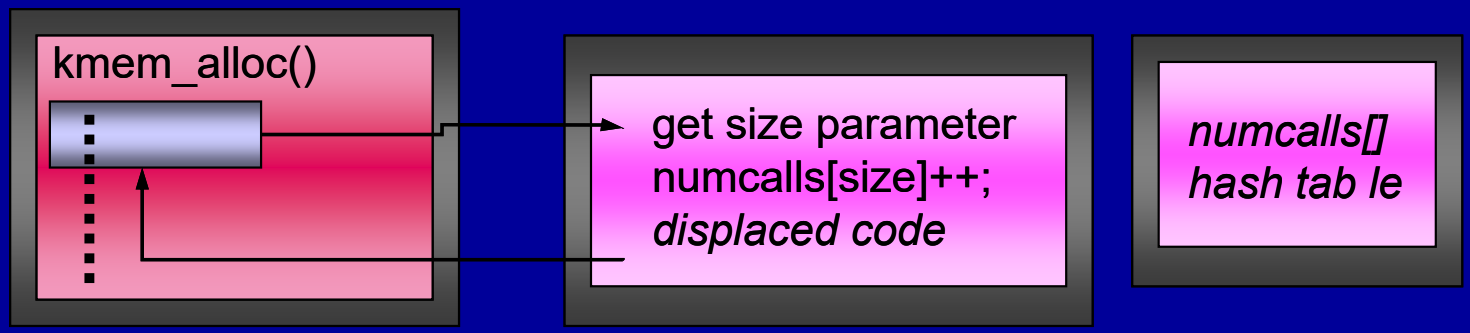
Works best combined with performance-measurement annotations.

Measurement-Based Optimization: Steps

- Dynamically insert profile annotations
  - for specialization: histogram of input values
  - for outlining: basic block profile
- Analyze annotation, decide on optimization
- Generate optimized version of function
- Splice in optimized version of function

Example: Specialization

Profile:



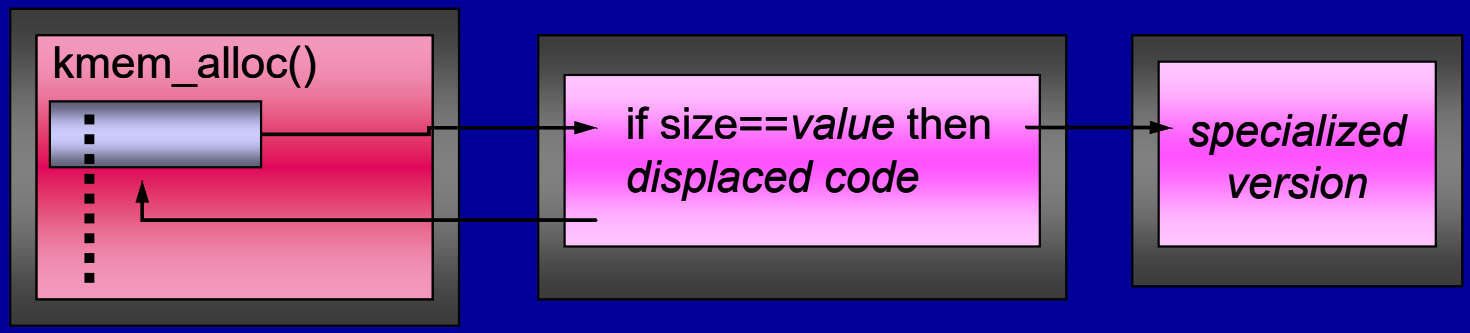
Decision: examine hash table

Generate specialized version:

- choose fixed value & run constant propagation
- expect unconditional branches & dead code

Example: 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

Run-time I-Cache Optimization

Code positioning [Pettis & Hansen 90]

- An example of an evolving kernel algorithm
- Reorder basic blocks to improve I-$ performance
  - What's new: at run-time, and on a kernel
- Procedure splitting
  - Segregate hot vs. cold basic blocks
- Basic block positioning
  - Reorder blocks to facilitate straight-lined execution
  - Try to make hottest branches untaken
  - Requires edge counts

Code Positioning Steps

Measure root function

- Is there poor I-cache performance?

Measure block counts

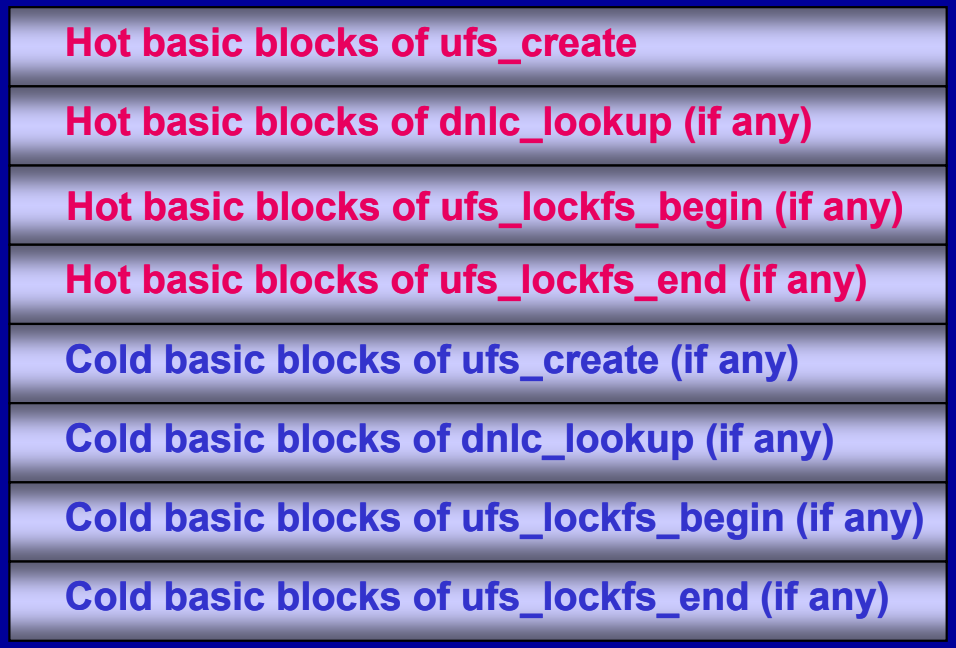
- Of the root function and its descendants
  - A traversal of the call graph
- Weed out descendants with no "hot" basic blocks
  - Hot: executed > 5% as often as root function is invoked

Emit optimized group of functions

The Optimized Function Group

Root function is ufs_create

Other functions are the hot subset of ufs_create's call graph descendants



Use code replacement on root function to install

Case Study of Code Positioning

Benchmark: mirror Paradyn papers Web page

- 10 simultaneous connections
- Perform code positioning on `tcp_rput_data`
  - Forwards data from IP to socket (on the hot path)
  - A large function (12K+ excluding callees)
    - So a good candidate for code positioning

Macro results

- Before optimization: 36.0 seconds
- After optimization: 33.6 seconds
- About 7% end-to-end speedup

Results (cont'd)

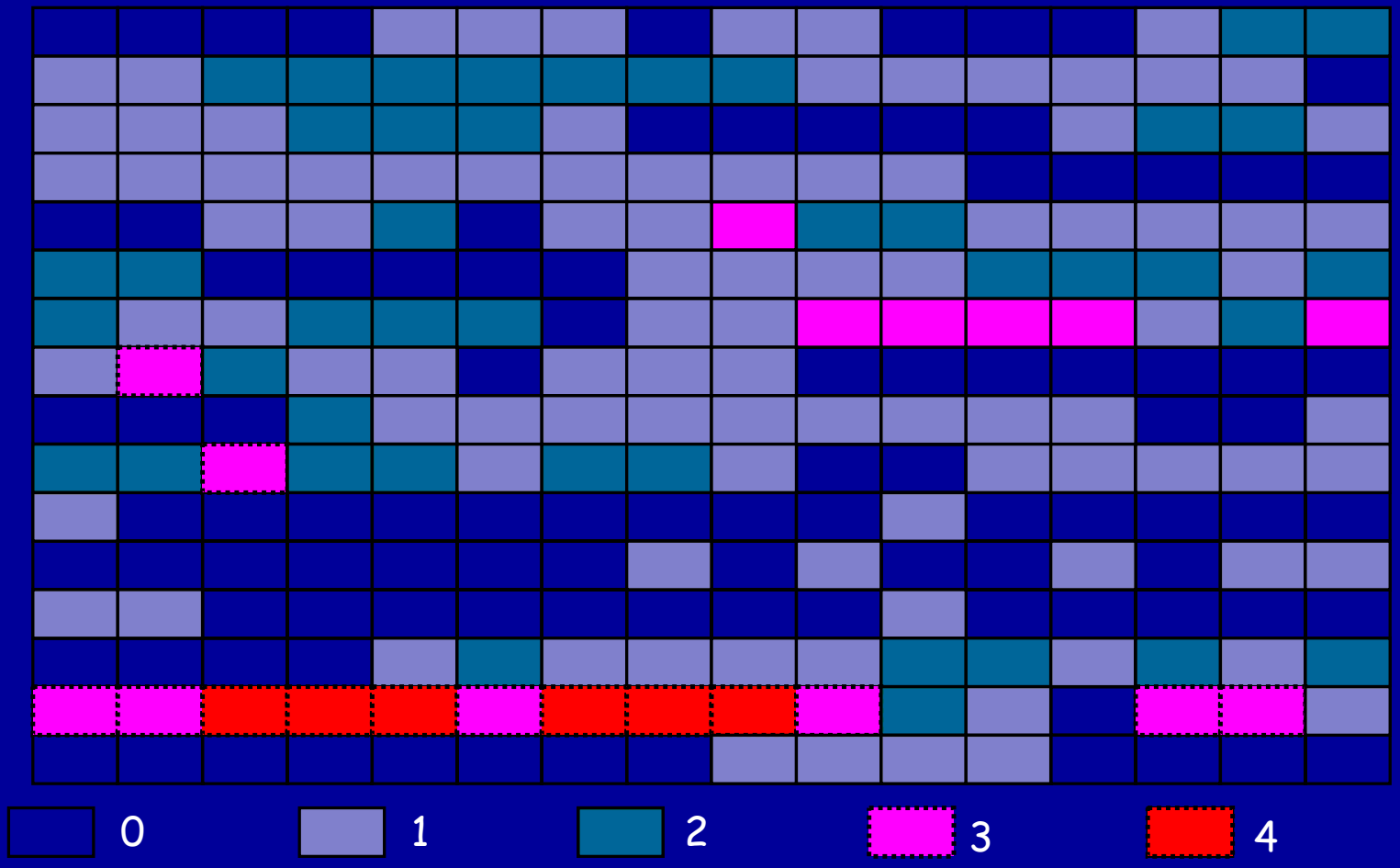
22 functions in the group

- 4260 bytes of hot code, 14624 bytes of cold code
- Most functions had all hot blocks covered in 1 path

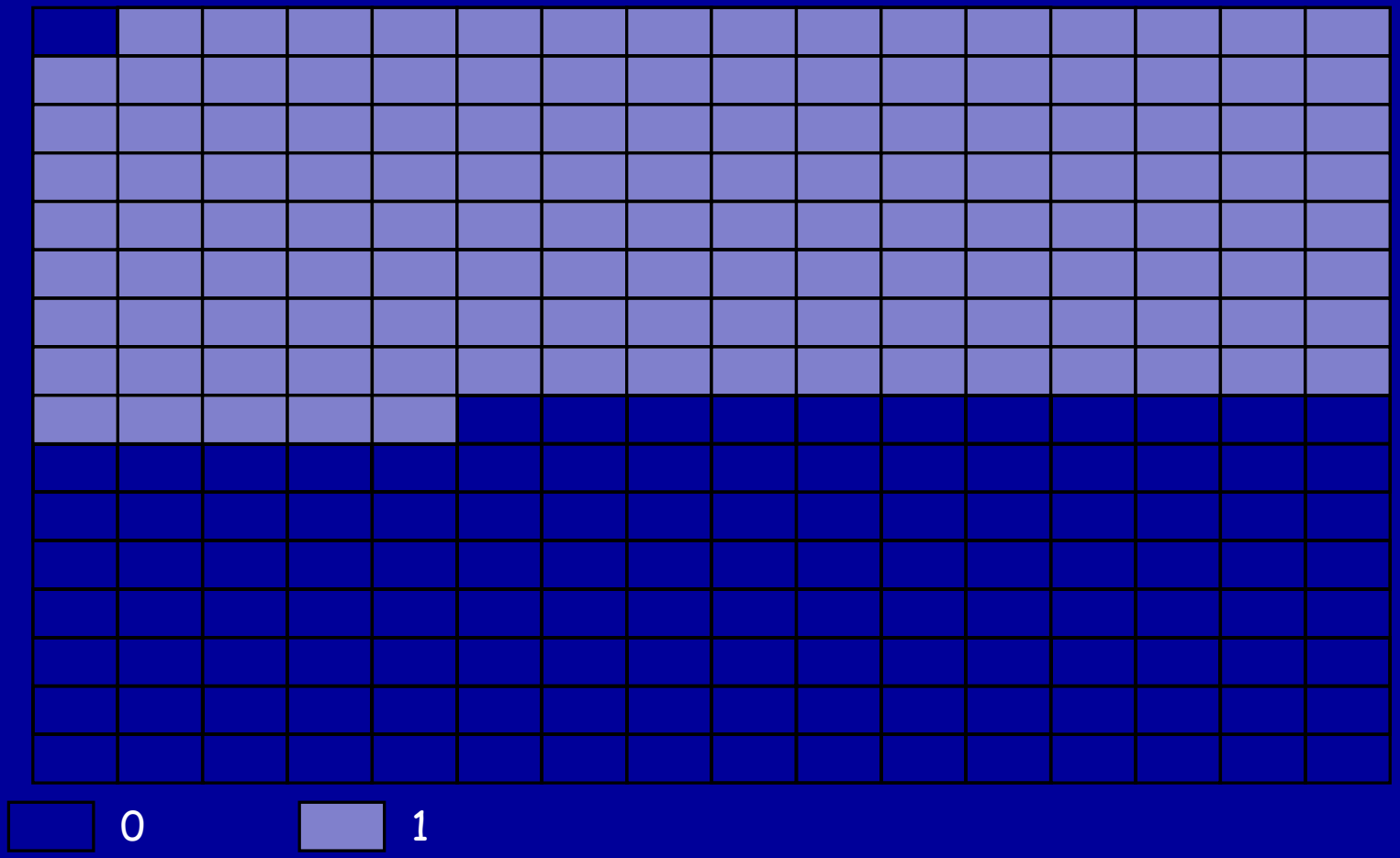
Micro-results (per invocation of `tcp_rput_data`)

- Virtual time per invocation
  - From 6.6 µs to 5.4 µs (reduction of 17.6%)
- I-cache stall time per invocation
  - From 2.4 µs to 1.55 µs (reduction of 35%)
- Branch mispredict stall time per invocation
  - From 0.38 µs to 0.20 µs (reduction of 47%)
- IPC: from 0.28 to 0.38

I-Cache Footprint: Before



I-Cache Footprint: After



Current Work

- x86 / Linux port
  - Variable-length and short instructions
- Power / Linux port

Conclusion

Operating Systems are not static entities anymore

Annotate for debug, profile, test, etc.

Adapt for customization of kernels.

Forms the foundation for an evolving OS...

...constantly changing in response to load and use.

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

Evolving Kernels: The Big Picture