Dynamic Kernel Instrumentation

Ariel Tamches
Computer Sciences Department
University of Wisconsin-Madison

tamches@cs.wisc.edu

The Big Picture

Currently, OS kernels are a performance black box...let's open it!

Extend Paradyn's dynamic instrumentation to an OS kernel

- A commodity OS (Solaris)
- Attach to an already-running kernel
- Instrument on-the-fly

Enables:

- Measurement of OS performance
- Customizable kernels
- Kernel debugging / assertion checks
- Data breakpoints / watchpoints
- Performance assertion checks

Making Kernels Customizable

Want to avoid requiring

- taking a checkpoint,
- implementing changes to kernel by hand,
- recompiling kernel,
- rebooting system,
- restoring checkpoint.

Use dynamic kernel instrumentation to implement the changes

- The same technology that allows us to instrument and measure the kernel lets us customize it on-the-fly

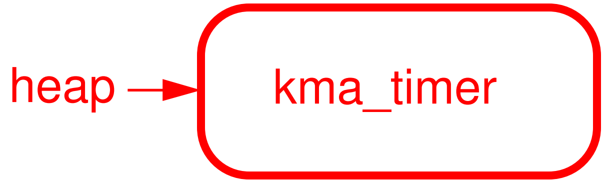
Measuring Kernel Routines

Unlock the black box of kernel code

Measure kernel calls just as Paradyn currently measures application routines

- Parse kernel symbol table to obtain name & location of kernel routines
- Write to /dev/kmem to dynamically instrument kernel routines

```
kmem_alloc() {
    startTimer(kma_timer);
    ...
    stopTimer(kma_timer);
}
```



Measure kernel background activity

- Kernel has many global (inter-application) algorithms
- Page daemon, scheduling, TLB miss handler, network daemon

Kernel-Specific Metrics

From kernel global variables come new Paradyn metrics

```
mmap() {
    startAccum(mmap_nallocs, &num_kmem_alloc);
    ...
    stopAccum(mmap_nallocs);
}
```

heap → mmap_nallocs

Must virtualize new metrics to obtain correct values for different foci

- Above code would yield num_kmem_allocs done in mmap() system-wide.
- Must instrument context-switch handler to isolate “when invoked by process P”.
- Per-CPU global values (e.g. hardware counters) must account for migration.

Implementation Techniques

Instrumentation

- Still one Paradyn daemon per machine.
- It accesses kernel memory through /dev/kmem
- Use TRAPs for mutual exclusion when instrumenting OS code.

Heap space for counters and timers

- Use kernel's kmem_alloc() routine
- Access the counters and timers through /dev/kmem

Bootstrapping

- Loadable (on-the-fly) device driver
- Driver installs TRAP handler, allocates space for trampolines, counters, timers.

Future Ideas

Dynamic kernel instrumentation opens up some new domains

- Data breakpoints (watchpoints) could trigger new instrumentation code.
- Assertion checks to debug kernel.

Steering the kernel

- Customized / optimized kernel routines inserted via dynamic instrumentation.
- Triggered by predicate code also inserted via dynamic instrumentation.