Code Instrumentation, Dynamic Tracing

http://d3s.mff.cuni.cz/aosy

CHARLES UNIVERSITY
Faculty of Mathematics and Physics

Martin Děcký
deky@d3s.mff.cuni.cz
Observability

- **What is the system doing?**
  - Beyond the obvious (externally visible state changes)
  - Interactive debugging
  - Profiling
  - Tracing
  - Post-mortem analysis

- **Many flavors**
  - Development / testing / production
  - Static / dynamic
  - Intrusive / non-intrusive
Instrumentation

- Application of measuring instruments
  - Recording of states and values
  - Safety
    - How the measurements affect the state or the values
  - Overhead
    - How the measurements affect the performance (active / inactive)
  - Scope
    - Global vs. local phenomena
Interactive Debugging

**Mechanisms**

- **Breakpoints**
  - Software breakpoints (BREAK, INT3)
  - Hardware breakpoints
    - DR0 to DR7 debug registers on x86 (4 linear addresses, trigger conditions [read, write, execute, I/O, area size], status)

- **Single-stepping**
  - Trap flag in FLAGS, interrupt vector 1 on x86

- **Watchpoints**
  - Hardware memory access breakpoints (can be emulated via the paging mechanism)
  - WatchLo, WatchHi on MIPS (1 physical address, trigger [read, write])
Interactive Debugging (2)

**Debugger**
- User interface
- Exception handling code (privileged)
  - Kernel stub
    - Communicating with the debugger UI process
      - `ptrace(2)`, `SIGTRAP`, break-in thread (`DebugActiveProcess()`)  
    - Remote debugging
      - Serial line, FireWire, USB, virtualization extension
  - Full-fledged in-kernel debugger (`kmdb` in Solaris)
  - 3rd party debugger (SoftICE, Rasta Ring 0)
  - Firmware debugger, hypervisor debugger stub
    - Non-maskable Interrupt, SysRq

**Debugging countermeasures**
Interative Debugging (3)

```
pid_t pid = fork();
if (pid == 0) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);  // trace all signals
    execve(...);
    deliver SIGTRAP after successful exec
}

int wstatus;
waitpid(pid, &wstatus, 0);
// examine wstatus

ptrace(PTRACE_SETOPTIONS, pid, NULL,
       PTRACE_O_EXITKILL | PTRACE_O_TRACECLONE |
       PTRACE_O_TRACEEXEC | PTRACE_O_TRACEEXIT |
       PTRACE_O_TRACEFORK | ...);

ptrace(PTRACE_GETSIGINFO, pid, NULL, siginfo);
ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS, iovec);
ptrace(PTRACE_PEEKTEXT, pid, remote_addr, local_addr);
ptrace(PTRACE_POKETEXT, pid, remote_addr, local_addr);
ptrace(PTRACE_CONT, pid, NULL, NULL);
```
Profiling

- **Run-time performance instrumentation**
  - **Exact profiling**
    - Triggered when an interesting event happens, sampling relevant information (timestamp, CPU performance counters, stack trace, etc.)
    - GNU Profiler
      - Using gcc -pg or gcc -pg -mrecord-mcount -mnop-mcount
      - Call to mcount() in every function prologue (optionally epilogue)
        - Collects information into gmon.out
        - Postprocessed by gprof
  - **Statistical profiling**
    - Sampling information in regular intervals
      - PC mapped to symbol name
    - OProfile
      - System-wide profiling
Performance Metrics

**Resource accounting**

- Memory usage (resident/virtual/shared memory, buffers, caches)
- Time
  - User time, system time, idle time (measured precisely)
  - \( \%\text{user} + \%\text{system} + \%\text{idle} = 100 \% \)
  - Utilization: \( \%\text{user} + \%\text{system} \)
  - Saturation: How much more work is there than the machine can handle (sampled in regular intervals)
    - E.g. number of non-idle CPUs + length of scheduler ready queues
    - Usually exponential moving average: \( \text{cur} = \text{prev} \times \text{decay} + n \times (1 - \text{decay}) \)

**Microstate accounting**
Observing events

- Similar to debugging, but usually high-level events
- Similar to logging, but activated on-demand
  - System calls, kernel functions, library functions, logical events (e.g. context switches, sending/receiving packets, etc.), even user space logical events
  - Usually asynchronous (avoiding serialization)
  - truss(1), strace(2), ltrace(1)
  - DTrace, SystemTap
DTrace

- D script
- dtrace(1M)
- lockstat(1M)
- plockstat(1M)
- intrstat(1M)
- libdtrace(3LIB)
- dtrace(3D)
- consumer

D compiler

user-space provider

user space

kernel

D virtual machine

pid

sysinfo

sdt

fasttrap

syscall

fbt

usdt

providers

communication device
Properties

- Probe specification language (D script)
- Active / inactive probes
  - Zero / small overhead
- Safety for production system
  - Virtual machine, no branching, no loops, safety checks, no state change (unless specified)
- No debug builds
  - Compact Type Information
- Correlation of events, aggregate statistics
**Properties**

- Probe specification language (SystemTap script)
  - Preprocessed into a kernel module source → kernel module
- Active / inactive probes
  - Zero / small overhead
- Safety for production system not guaranteed
- Requires debugging kernel build
- Correlation of events, aggregate statistics
- Uses ftrace and kprobes as kernel backends
DTrace sample

#! /usr/sbin/dtrace -s

syscall:::entry {
    @count[probefunc] = count();
    self->ts = timestamp;
    self->tag = 1;
}

syscall:::return /self->tag == 1/ {
    self->ts_diff = timestamp - self->ts;
    @total[probefunc] = sum(self->ts_diff);
    @avg[probefunc] = avg(self->ts_diff);
}

END {
    printa("%a @%d @%d @%d\n", @count, @total, @avg);
}
DTrace sample (2)

```dtrace
#!/usr/sbin/dtrace -s
#pragma D option quiet

hotspot$target:::method-entry {
    self->indent++;
    self->trace = 1;
    printf("%*s -> %s.%s\n", self->indent, "", stringof(copyin(arg1, arg2)), stringof(copyin(arg3, arg4)));
}

hotspot$target:::method-return /self->trace == 1/ {
    printf("%*s <- %s.%s\n", self->indent, "", stringof(copyin(arg1, arg2)), stringof(copyin(arg3, arg4)));
    self->indent--;
    self->trace = (self->indent == 0) ? 0 : self->trace;
}

pid$target:libc::entry /self->trace == 1/ {
    self->indent++;
    printf("%*s => %s\n", self->indent, "", probefunc);
}

pid$target:libc::return /self->trace == 1/ {
    printf("%*s <= %s\n", self->indent, "", probefunc);
    self->indent--;
}

syscall:::entry /self->trace == 1/ {
    self->indent++;
    printf("%*s :> %s\n", self->indent, "", probefunc);
}

syscall:::return /self->trace == 1/ {
    printf("%*s <: %s\n", self->indent, "", probefunc);
    self->indent--;
}
```
DTrace Code Instrumentation

DTrace Code Instrumentation

<table>
<thead>
<tr>
<th>uninstrumented</th>
<th>instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td>squeue_enter_chain+0x1af:</td>
<td>xorl %eax,%eax</td>
</tr>
<tr>
<td>squeue_enter_chain+0x1b1:</td>
<td>nop</td>
</tr>
<tr>
<td>squeue_enter_chain+0x1b2:</td>
<td>nop</td>
</tr>
<tr>
<td>squeue_enter_chain+0x1b3:</td>
<td>nop</td>
</tr>
<tr>
<td>squeue_enter_chain+0x1b4:</td>
<td>nop</td>
</tr>
<tr>
<td>squeue_enter_chain+0x1b5:</td>
<td>nop</td>
</tr>
<tr>
<td>squeue_enter_chain+0x1b6:</td>
<td>movb %bl,%bh</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ufs_mount:</td>
<td>pushq %rbp</td>
</tr>
<tr>
<td>ufs_mount+1:</td>
<td>movq %rsp,%rbp</td>
</tr>
<tr>
<td>ufs_mount+4:</td>
<td>subq $0x88,%rsp</td>
</tr>
<tr>
<td>ufs_mount+0xb:</td>
<td>pushq %rbx</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>ufs_mount+0x3f3:</td>
<td>popq %rbx</td>
</tr>
<tr>
<td>ufs_mount+0x3f4:</td>
<td>movq %rbp,%rsp</td>
</tr>
<tr>
<td>ufs_mount+0x3f7:</td>
<td>popq %rbp</td>
</tr>
<tr>
<td>ufs_mount+0x3f8:</td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td>int $0x3</td>
</tr>
<tr>
<td></td>
<td>movq %rsp,%rbp</td>
</tr>
<tr>
<td></td>
<td>subq $0x88,%rsp</td>
</tr>
<tr>
<td></td>
<td>pushq %rbx</td>
</tr>
<tr>
<td></td>
<td>popq %rbx</td>
</tr>
<tr>
<td></td>
<td>movq %rbp,%rsp</td>
</tr>
<tr>
<td></td>
<td>popq %rbp</td>
</tr>
<tr>
<td></td>
<td>int $0x3</td>
</tr>
</tbody>
</table>

Later replaced by a call

Later replaced by a call

Later replaced by a call
Code Instrumentation

- **Static instrumentation of executables**
  - Non-intrusive
    - Avoiding shifting instruction locations (complicated unless the code is PIC)
    - Replacing a byte of an instruction by a trap instruction
      - At run-time, replacing the trap with a call
      - The call emulates the original instruction(s) that are replaced and jumps back to the next instruction
  - Intrusive
    - Techniques similar to binary translation (later)
    - Internal representation of code basic blocks
      - Instrumentation: Inserting new basic block, updating jump/call locations
      - Tricky with self-modifying code, dynamic dispatch, etc.
  - Valgrind, DynInst, Pin, Vulcan, BIRD, PEBIL
Post-mortem Analysis

- Analyzing a root cause of a crash
  - Core dump
    - Snapshot of a single process
    - On-disk format similar to executable format (plus register context and other metadata)
      - Can be opened in interactive debugger
  - Crash dump
    - Snapshot of an entire system
      - Sometimes without user pages and other sensitive data
      - Created by a failing system, rescue system, firmware or out-of-band management
      - NMI to help
    - Special analysis tools
Core Dump
Core/ Crash Dump Analysis

- **Identifying the immediate cause**
  - Examining the crash IP location, register context, stack trace, log buffer
  - Maybe instrumentation values (if available)

- **Identifying the root cause**
  - Art, science and craftsmanship
  - Heuristic tools to analyze typical crashes (mdb::findlocks, crash)
  - Gradually reconstructing the events prior to the crash
    - Distrusting information encountered, formulating hypotheses
      - Code optimization gets into the way
  - Analyzing data structures, threads, locks, etc.
    - Looking for interesting literals (0xdeadbeef, 0xbaddcafe, 0xfeedface)
Linux Performance Observability Tools

Applications

System Libraries

Operating System

Hardware

Various:

- perf
- mpstat
- turbostat
- rdmsr
- tiptop
- perf

CPU

Memory Bus

DRAM

Device Drivers

I/O Bus

I/O Controller

Disk

Disk

Swap

Interface Transports

Network Controller

Port

Port

I/O Bridge

expansion interconnect

iptraf
tcpdump

nicstat
netstat
netstat
netstat
netstat

lldptool

ethtool

snmpget

swapon

Q&A
References