Linux Observability and Tuning using bpftrace

Instructor: Chandrashekar Babu <training@chandrashekar.info>

A practical guide to performance analysis and troubleshooting with bpftrace

Module 3: Performance Tuning Deep Dive

In this module, we'll explore advanced techniques for diagnosing and resolving complex performance issues in Linux systems using bpftrace.

1

CPU Profiling

On-CPU vs. Off-CPU analysis for complete CPU utilisation understanding

2

Memory Analysis

Tracking allocations, page faults, and memory pressure events

3

Scheduler Latency

Identifying and resolving issues with task scheduling and runtime

4

Block I/O Analysis

Measuring and optimising storage performance with detailed I/O tracking

CPU Profiling: Fundamentals

On-CPU vs. Off-CPU Analysis

Complete CPU performance analysis requires both:

- **On-CPU Analysis:** Where is CPU time being spent? Which functions consume most cycles?
- **Off-CPU Analysis:** Why are threads not running? What are they waiting for? (I/O, locks, scheduling)

bpftrace excels at both, giving you visibility into the entire execution lifecycle of your processes.

CPU Profiling: On-CPU Analysis

Example: Sampling Stack Traces with profile

```
# Sample user+kernel stacks at 99Hz for process 1234
bpftrace -e '
profile:hz:99 /pid == 1234/ {
    @[kstack, ustack] = count();
}
interval:s:10 {
    exit();
}
'
```

This script:

- Samples the process at 99Hz (approximately 99 times per second)
- Collects both kernel (kstack) and user (ustack) stack traces
- Aggregates identical stacks and counts occurrences
- Runs for 10 seconds before exiting

Higher counts indicate where more CPU time is being spent, helping identify hotspots.

CPU Profiling: One-liners

These powerful one-liners can quickly identify CPU usage patterns:

```
# Show top kernel functions consuming CPU
bpftrace -e 'profile:hz:99 { @[kstack] = count(); }'

# Sample process 1234 and print top user functions
bpftrace -e 'profile:hz:99 /pid == 1234/ { @[func] = count(); }'

# Identify CPU time by process name
bpftrace -e 'profile:hz:99 { @[comm] = count(); }'
```

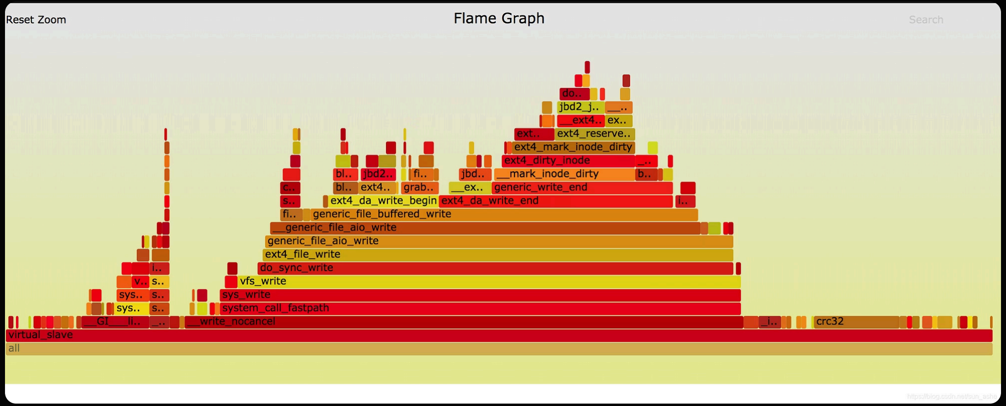
Use these as starting points before diving deeper with more targeted scripts. The 99Hz frequency helps avoid lockstep sampling with periodic system activities.

CPU Profiling: Flame Graphs

Flame graphs visualize stack trace data for intuitive performance analysis:

```
# Capture stacks for flame graph
bpftrace -e '
profile:hz:99 /pid == 1234/ {
  @[ustack] = count();
}
' > stacks.out

# Generate flame graph
cat stacks.out |
  flamegraph.pl > profile.svg
```



The x-axis shows stack population, the y-axis shows stack depth. Wide blocks represent hot code paths consuming CPU time.

Advanced CPU Profiling Script

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Sampling CPU stacks for processes. Press Ctrl-C to end.\n");
}

// Sample at 99Hz across all CPUs
profile:hz:99 {
  // Collect process info
  $pid = pid;
  $comm = comm;
  $ts = nsecs;

  // Store in stack-aggregated map
  @stacks[$comm, $pid, kstack, ustack] = count();

  // Track total samples per process
  @proc_samples[$comm, $pid] = count();
}

// Print summary every 10 seconds
interval:s:10 {
  time("%H:%M:%S Sampling summary:\n");
  print(@proc_samples);
  clear(@proc_samples);
}

END {
  printf("Top stacks by process and count:\n");
  print(@stacks, 10);
  clear(@stacks);
}
```

This script provides periodic summaries and detailed stacks upon completion, making it ideal for longer profiling sessions.

CPU Profiling: Off-CPU Analysis

Off-CPU analysis reveals why threads aren't running:

```
# Track time spent off-CPU for process 1234
bpftrace -e '
tracepoint:sched:sched_switch / args->prev_pid == 1234 / {
  @start[args->prev_pid] = nsecs;
}

tracepoint:sched:sched_switch / args->next_pid == 1234 / {
  $pid = args->next_pid;
  $ts = nsecs;
  $start_ts = @start[$pid];
  if ($start_ts != 0) {
    @off_cpu_time[$pid, comm] = sum($ts - $start_ts);
    delete(@start[$pid]);
  }
}
'
```

Common Off-CPU Reasons:

- I/O operations (disk, network)
- Mutex/lock contention
- Sleep or wait conditions
- Scheduling delays
- Page faults

Investigating off-CPU time often reveals more optimization opportunities than on-CPU profiling alone.

Off-CPU Analysis: Full Stack Tracing

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Tracing off-CPU stacks. Press Ctrl-C to end.\n");
}

// Track when process goes off-CPU with stack
tracepoint:sched:sched_switch {
  // Only interested in our target process
  if (args->prev_pid == $1) {
    // Store timestamp and stack when going off-CPU
    @start[args->prev_pid] = nsecs;
    @last_stack[args->prev_pid] = kstack;
  }
}

// When process comes back on-CPU
tracepoint:sched:sched_switch {
  $pid = args->next_pid;
  if ($pid == $1 && @start[$pid] != 0) {
    // Calculate off-CPU duration
    $duration_ns = nsecs - @start[$pid];

    // Only record if off-CPU for >10ms
    if ($duration_ns > 10000000) {
      // Store stack and duration
      @off_cpu_stack[@last_stack[$pid]] = sum($duration_ns);
    }
    delete(@start[$pid]);
  }
}

END {
  printf("Off-CPU stack samples (>10ms):\n");
  print(@off_cpu_stack);
  clear(@off_cpu_stack);
}
```

Usage: `bpftrace offcpu.bt 1234` (where 1234 is the target PID)

Combined On/Off-CPU Analysis

For complete performance understanding, combine both analyses:

```
# Install and use bcc's profile tool
# This captures both on and off-CPU stacks
sudo /usr/share/bcc/tools/profile -df -p 1234 30
```

The `-f` option includes stack traces, `-d` includes deltas between samples (showing off-CPU time), and the final number (30) is the duration in seconds.

This comprehensive view helps identify both CPU bottlenecks and wait conditions slowing your application.

Memory Analysis: Fundamentals

Memory performance issues can manifest as:

- Excessive allocation/deallocation patterns
- Memory leaks
- Page faults (major and minor)
- Memory pressure affecting overall system performance
- Fragmentation

bpftrace provides visibility into kernel memory subsystems and user-space allocations that traditional tools cannot access.

Tracking Memory Allocations

Kernel Memory (kmalloc) Tracking

```
# Trace all kmalloc calls by size
bpftrace -e '
kprobe:kmalloc {
  @bytes[comm] = hist(arg1);
}
'
```

This displays a histogram of kernel memory allocation sizes by process.

User-space Memory Tracking

```
# Track malloc() calls by size
bpftrace -e '
uprobe:/lib/x86_64-linux-gnu/libc.so.6:malloc {
  @bytes[comm] = hist(arg0);
}
'
```

The exact libc path may vary on your system - adjust accordingly.

Memory Leak Detection

Track the balance of allocations and frees to identify potential leaks:

```
#!/usr/bin/env bpftrace

// Track mmap/munmap balance to find potential memory leaks
BEGIN {
  printf("Tracing mmap/munmap calls... Press Ctrl-C to end.\n");
}

tracepoint:syscalls:sys_enter_mmap {
  @mmap[pid, comm] = count();
}

tracepoint:syscalls:sys_enter_munmap {
  @munmap[pid, comm] = count();
}

END {
  printf("Memory mapping summary:\n");
  printf("mmap calls by process:\n");
  print(@mmap);

  printf("\nmunmap calls by process:\n");
  print(@munmap);

  printf("\nDifference (potential leaks if large and positive):\n");
  @diff = @mmap;
  @diff -= @munmap;
  print(@diff);
}
```

A large positive difference between mmap and munmap counts may indicate a memory leak, especially in long-running processes.

Heap Analysis Script

```
#!/usr/bin/env bpftrace

// Comprehensive heap analysis for process $1

// libc malloc/free function addresses - adjust for your system
BEGIN {
  printf("Tracing heap activity for PID %d\n", $1);
}

// Track malloc calls
uprobe:/lib/x86_64-linux-gnu/libc.so.6:malloc /pid == $1/ {
  @malloc_sizes = hist(arg0);           // Histogram of allocation sizes
  @malloc_stack[ustack] = count();      // Stack traces of allocations
  @bytes_allocated += arg0;             // Running total
  @malloc_calls++;                      // Count total calls
}

// Track free calls
uprobe:/lib/x86_64-linux-gnu/libc.so.6:free /pid == $1/ {
  @free_calls++;
}

// Print summary every 10 seconds
interval:s:10 {
  time("%H:%M:%S Heap stats:\n");
  printf("Total malloc calls: %d\n", @malloc_calls);
  printf("Total free calls: %d\n", @free_calls);
  printf("Current difference: %d\n", @malloc_calls - @free_calls);
  printf("Estimated heap bytes: %d\n", @bytes_allocated);
}

END {
  printf("Allocation size distribution:\n");
  print(@malloc_sizes);

  printf("Top allocation stack traces:\n");
  print(@malloc_stack, 10);
}
```

Usage: bpftrace heapanalysis.bt 1234

Page Fault Tracking

Page Fault Types

- **Minor fault:** Page exists in memory but not mapped in MMU
- **Major fault:** Page must be loaded from disk

Major page faults are particularly expensive as they involve disk I/O.

```
# Track major page faults by process
bpftrace -e '
software:major-faults {
  @faults[comm, pid] = count();
}
'

# Track minor page faults
bpftrace -e '
software:minor-faults {
  @faults[comm, pid] = count();
}
'
```

High page fault rates can significantly impact application performance, especially when memory is under pressure.

Comprehensive Page Fault Analysis

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Tracing page faults... Press Ctrl-C to end.\n");
}

// Track major page faults with stack trace
software:major-faults {
  @major[pid, comm, kstack, ustack] = count();
  @major_total[pid, comm] = count();
}

// Track minor page faults with stack trace
software:minor-faults {
  @minor[pid, comm] = count();
}

// Print summary every 5 seconds
interval:s:5 {
  time("%H:%M:%S Page fault summary:\n");
  printf("Top processes with minor faults:\n");
  print(@minor, 5);
  printf("Top processes with major faults:\n");
  print(@major_total, 5);
}

END {
  printf("Major fault stack traces:\n");
  print(@major, 10);
  clear(@major);
  clear(@minor);
  clear(@major_total);
}
```

This script provides real-time monitoring of page fault patterns and detailed stack traces for major faults, helping identify code paths causing excessive disk I/O.

Memory Pressure Indicators

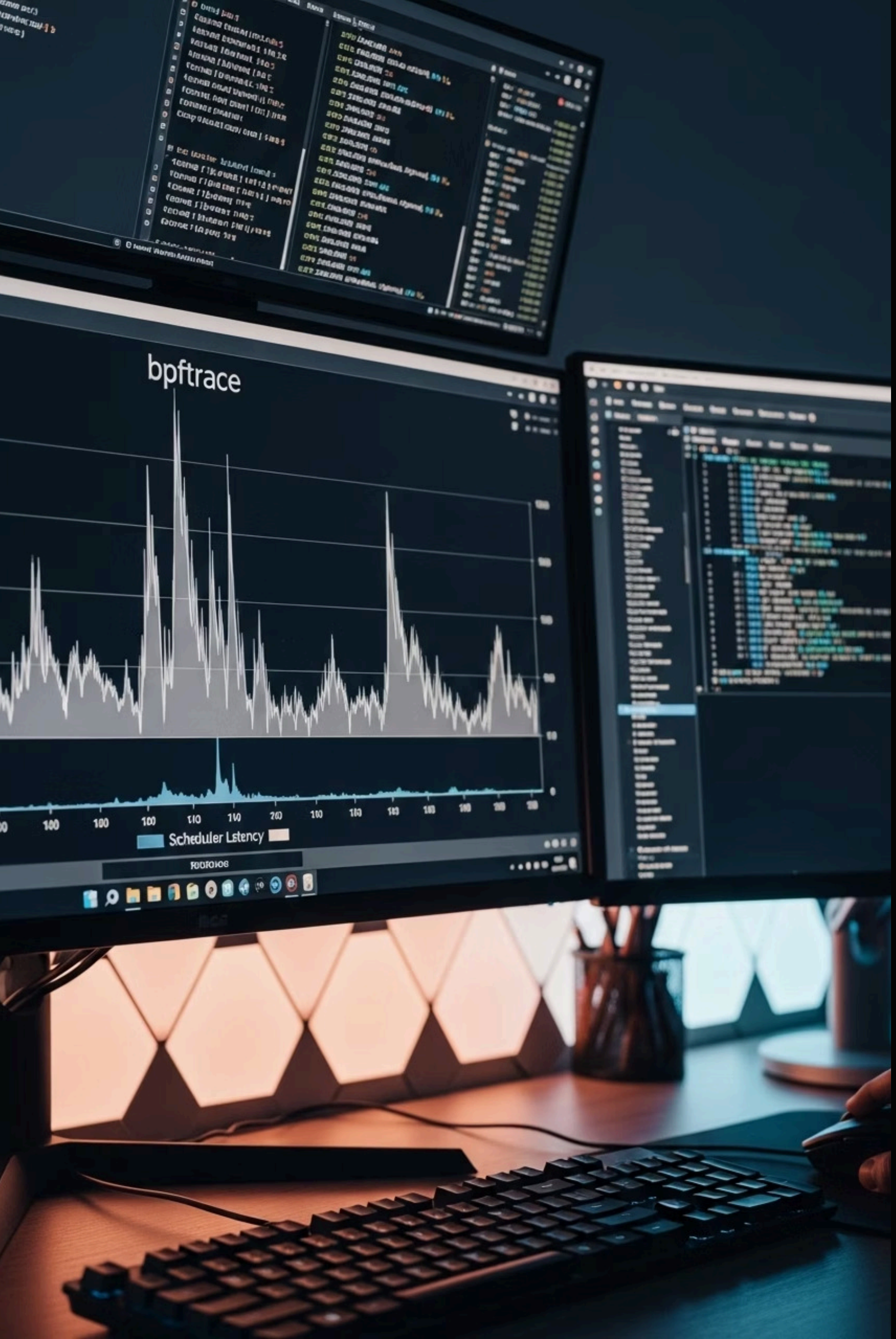
Track direct reclaim events (system under memory pressure)

```
# Monitor when system starts reclaiming memory
bpftrace -e '
kprobe:direct_reclaim_begin {
  printf("Memory pressure: direct reclaim started at %u\n", nsecs/1000000);
  @reclaims[kstack] = count();
}
'
```

Track compaction events (system trying to create larger contiguous pages)

```
# Monitor memory compaction events
bpftrace -e '
kprobe:compact_zone {
  printf("Memory compaction event at %u\n", nsecs/1000000);
  @compactions[kstack] = count();
}
'
```

These events indicate the system is under memory pressure, which can degrade performance across all applications.



Scheduler Latency: Introduction

Scheduler latency refers to the delay between when a process is ready to run and when it actually gets CPU time.

High scheduler latency can cause:

- Application responsiveness issues
- Missed deadlines in real-time systems
- Increased transaction processing time
- Poor interactive performance

bpftrace can measure and analyze these delays by instrumenting key scheduler events.

Measuring Scheduler Latency

Basic scheduler latency measurement:

```
# Measure time between process wakeup and execution
bpftrace -e '
// Record timestamp when a task becomes runnable
tracepoint:sched:sched_wakeup {
 // Store the timestamp for this PID
 @wakeup[args->pid] = nsecs;
}

// Calculate delay when the task actually runs
tracepoint:sched:sched_switch {
 $pid = args->next_pid;
 $ts = @wakeup[$pid];

 // If we have a wakeup timestamp, calculate the delay
 if ($ts) {
 $delay = nsecs - $ts;
 // Only show delays > 1ms
 if ($delay > 1000000) {
 printf("%-16s %-6d %16llu ns\n", args->next_comm, $pid, $delay);
 @delay_us = hist($delay / 1000);
 }
 delete(@wakeup[$pid]);
 }
}
'
```

Advanced Scheduler Latency Script

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Tracing scheduler latency... Press Ctrl-C to end.\n");
}

// Track when tasks are woken up (become runnable)
tracepoint:sched:sched_wakeup,
tracepoint:sched:sched_wakeup_new {
  // Store wake-up time and the CPU that did the wakeup
  @wakeup[args->pid] = nsecs;
  @waker[args->pid] = cpu;
  @task_comm[args->pid] = args->comm;
}

// Track when tasks are actually scheduled
tracepoint:sched:sched_switch {
  $next_pid = args->next_pid;
  $ts = @wakeup[$next_pid];

  if ($ts) {
    $delay = nsecs - $ts;
    $comm = @task_comm[$next_pid];

    // Record detailed stats for delays > 5ms
    if ($delay > 5000000) {
      $waker_cpu = @waker[$next_pid];
      printf("Latency %10.3f ms for %s [%d]: waker CPU %d, current CPU %d\n",
             $delay / 1000000, $comm, $next_pid, $waker_cpu, cpu);

      // Was it scheduled on the same CPU that woke it?
      @cross_cpu[$waker_cpu != cpu] = count();
    }

    // Build latency histograms by process
    @latency_us[$comm] = hist($delay / 1000);

    // Clear the tracking maps
    delete(@wakeup[$next_pid]);
    delete(@waker[$next_pid]);
    delete(@task_comm[$next_pid]);
  }
}

END {
  printf("Scheduler latency histograms by process (microseconds):\n");
  print(@latency_us);

  printf("\nCross-CPU scheduling events (potential NUMA effects):\n");
  print(@cross_cpu);
}
```

Usage: bpftrace schedlat.bt

Scheduler Run Queue Analysis

Analyzing run queue length helps understand scheduler load:

```
# Monitor run queue length by CPU
bpftrace -e '
// Sample run queue length every 10ms
profile:hz:100 {
  $len = 0;
  // Parse task_struct runqueue information
  $len = *(int32*)(((void *)curtask) + 0x440);  // Offset to nr_running
  @runqlen[cpu] = hist($len);
}
'
```

Note: The 0x440 offset may vary by kernel version - check your system's struct definitions.

Long run queues indicate potential CPU contention and increased scheduler latency. Generally, values consistently above 1-2 tasks per CPU suggest the system might be CPU bound.

CFS Scheduler Analysis

Monitor CFS task selection decisions

```
# Track which tasks are picked by the scheduler
bpftrace -e '
kprobe:pick_next_task_fair {
  @pick[comm, pid] = count();
}
'

# Track task latency in the CFS scheduler
bpftrace -e '
kprobe:update_curr {
  // Extract task info from function args
  $curr = (struct task_struct *)arg0;
  $pid = $curr->pid;
  $comm = $curr->comm;

  // Get task's runtime
  $delta_exec = (uint64)$curr->se.sum_exec_runtime;

  // Keep track of total runtime
  @runtime_ns[$comm, $pid] = $delta_exec;
}
'
```

These scripts provide insights into how the Completely Fair Scheduler (CFS) is distributing CPU time among competing tasks.

Lab: Identifying Scheduler Latency

Setup

Let's create a simple test to observe scheduler latency under load:

```
# Terminal 1: Create artificial CPU load
for i in $(seq 1 $(nproc)); do
  stress-ng --cpu 1 --timeout 60s &
done

# Terminal 2: Create a sensitive test process
while true; do
  read -t 0.001 || echo -n ".";
  sleep 0.01;
done
```

Now we'll use our bpftrace script to detect scheduler latency for this process.

```
# Terminal 3: Run the scheduler latency script
sudo bpftrace schedlat.bt
```

Look for patterns of increased latency for the shell process running our test.

Block I/O Analysis: Introduction

Storage I/O is often a major bottleneck in system performance.

With bpftrace, you can track:

- I/O latency from request to completion
- I/O sizes and patterns
- Per-process, per-file, and per-device metrics
- Disk queue depths and saturation
- Filesystem-level operations

Basic Block I/O Tracing

Track block I/O requests and completions:

```
# Trace block I/O request start
bpftrace -e '
tracepoint:block:block_rq_issue {
 printf("%s %s %s %d sectors\n",
 probe, args->comm, args->rwbs, args->nr_sector);
}
'

# Monitor I/O latency
bpftrace -e '
BEGIN {
 printf("Tracing block I/O... Hit Ctrl-C to end.\n");
}

tracepoint:block:block_rq_issue {
 // Start time for I/O request
 @start[args->dev, args->sector] = nsecs;
 @type[args->dev, args->sector] = args->rwbs;
}

tracepoint:block:block_rq_complete {
 $start = @start[args->dev, args->sector];
 $type = @type[args->dev, args->sector];

 if ($start) {
 $duration = nsecs - $start;
 // Create histogram of I/O latency by type (read/write)
 @usecs[$type] = hist($duration / 1000);
 delete(@start[args->dev, args->sector]);
 delete(@type[args->dev, args->sector]);
 }
}
'
```

bytesize.bt: Detailed I/O Analysis

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Tracing block I/O... Output every 1 second.\n");
}

tracepoint:block:block_rq_issue {
 // Track issue time, process, and device
 @start[args->dev, args->sector] = nsecs;
 @process[args->dev, args->sector] = comm;
 @bytes[args->dev, args->sector] = args->nr_sector * 512;
 @type[args->dev, args->sector] = args->rwbs;
}

tracepoint:block:block_rq_complete {
 $start = @start[args->dev, args->sector];

 if ($start) {
 $duration = nsecs - $start;
 $process = @process[args->dev, args->sector];
 $bytes = @bytes[args->dev, args->sector];
 $type = @type[args->dev, args->sector];

 // Update histograms
 @size_bytes[$type] = hist($bytes);
 @latency_us[$type] = hist($duration / 1000);
 @process_io[$process, $type] = sum($bytes);

 // Cleanup tracking maps
 delete(@start[args->dev, args->sector]);
 delete(@process[args->dev, args->sector]);
 delete(@bytes[args->dev, args->sector]);
 delete(@type[args->dev, args->sector]);
 }
}

// Print statistics every second
interval:s:1 {
 time("%H:%M:%S I/O summary:\n");
 printf("I/O size distribution (bytes):\n");
 print(@size_bytes);
 clear(@size_bytes);

 printf("I/O latency distribution (microseconds):\n");
 print(@latency_us);
 clear(@latency_us);

 printf("Per-process I/O (bytes):\n");
 print(@process_io);
 clear(@process_io);
}
```

Block I/O Stack Analysis

Identify which code paths are generating I/O:

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Tracing block I/O with stack traces... Ctrl-C to end.\n");
}

// Capture stack trace for I/O requests
tracepoint:block:block_rq_issue {
  // Only trace process given as argument (or all if no arg)
  if ($1 == 0 || $1 == pid) {
    @bytes[kstack, ustack, comm] = sum(args->nr_sector * 512);
    @count[kstack, ustack, comm] = count();
  }
}

// Print top 10 stack traces by I/O volume
END {
  printf("Top 10 stacks by I/O volume:\n");
  print(@bytes, 10);

  printf("\nTop 10 stacks by I/O frequency:\n");
  print(@count, 10);
}
```

Usage: `bpftrace iostacks.bt [PID]` (PID is optional)

This script reveals exactly which application code is triggering storage I/O, including through the page cache.

Filesystem-level I/O Analysis

Track file operations by process:

```
# Monitor file opens with path info
bpftrace -e '
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
  @files[comm, str(args->filename)] = count();
}
'

# Track read/write sizes by file
bpftrace -e '
tracepoint:syscalls:sys_exit_read /args->ret > 0/ {
  @reads[comm, pid] = hist(args->ret);
}

tracepoint:syscalls:sys_exit_write /args->ret > 0/ {
  @writes[comm, pid] = hist(args->ret);
}
'
```

These one-liners help identify which processes are performing I/O and what their access patterns look like, which can guide optimization efforts.

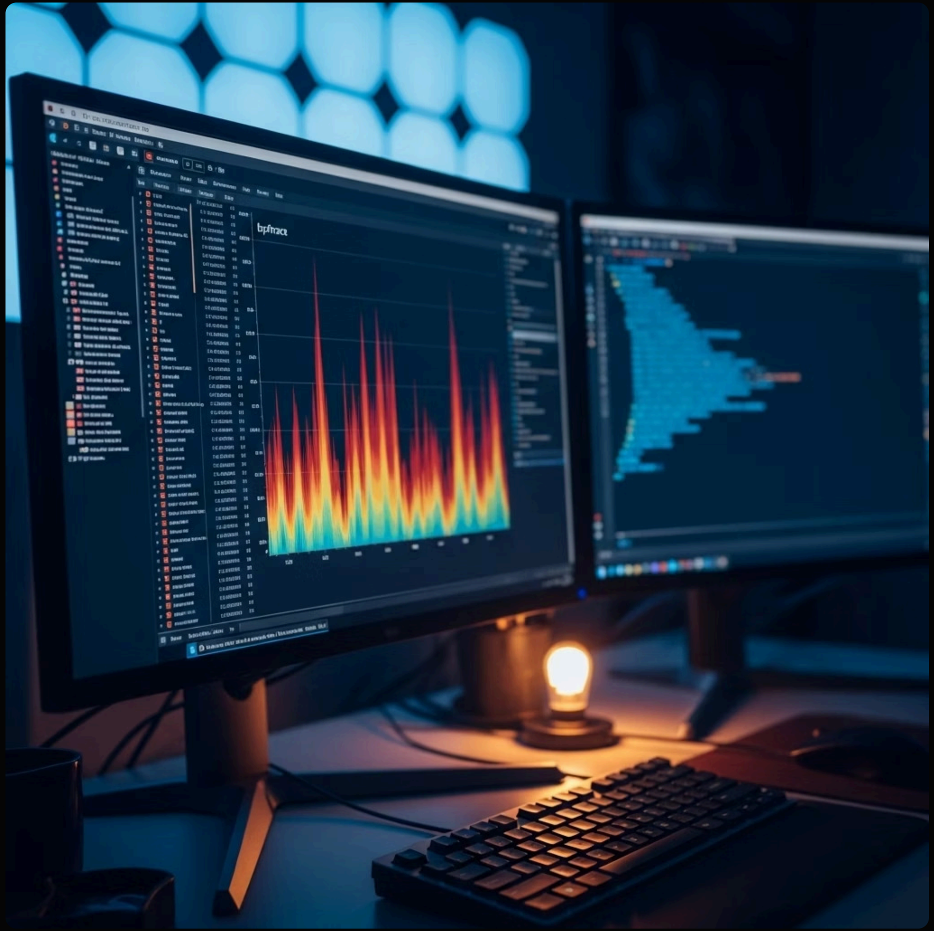
Detecting Synchronous I/O Issues

Synchronous I/O can block application threads and cause latency spikes:

```
# Detect direct synchronous writes
bpftrace -e '
kprobe:sync_page {
  @sync_by_proc[comm] = count();
}

kprobe:fsync {
  @fsync_by_proc[comm] = count();
  @fsync_stacks[ustack, comm] = count();
}
'
```

High sync operations suggest application changes might be needed - consider switching to async I/O patterns or buffering.



Comprehensive File Access Analysis

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Tracing file operations... Press Ctrl-C to end.\n");
}

// Track file opens
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
  $filename = str(args->filename);

  // Skip temporary and system files
  if ($filename != "" &&
      strncmp($filename, "/proc", 5) != 0 &&
      strncmp($filename, "/sys", 4) != 0 &&
      strncmp($filename, "/dev", 4) != 0) {

    printf("%s opened %s\n", comm, $filename);
    @opens[comm, $filename] = count();

    // Store the name for lookup on file descriptor ops
    @fdnames[pid, args->fd] = $filename;
  }
}

// Track file reads
tracepoint:syscalls:sys_exit_read /args->ret > 0/ {
  $name = @fdnames[pid, args->fd];
  if ($name != "") {
    @read_bytes[$name, comm] = sum(args->ret);
    @read_calls[$name, comm] = count();
  }
}

// Track file writes
tracepoint:syscalls:sys_exit_write /args->ret > 0/ {
  $name = @fdnames[pid, args->fd];
  if ($name != "") {
    @write_bytes[$name, comm] = sum(args->ret);
    @write_calls[$name, comm] = count();
  }
}

// Clean up on file close
tracepoint:syscalls:sys_enter_close {
  delete(@fdnames[pid, args->fd]);
}

END {
  printf("Files opened:\n");
  print(@opens);

  printf("\nBytes read by file:\n");
  print(@read_bytes);

  printf("\nBytes written by file:\n");
  print(@write_bytes);
}
```

This script provides a complete picture of which files are being accessed and how.

Network Performance Analysis

Network operations can significantly impact performance:

```
# Monitor socket connections
bpftrace -e '
tracepoint:syscalls:sys_enter_connect {
  @connects[comm] = count();
}
'

# Track TCP retransmits
bpftrace -e '
kprobe:tcp_retransmit_skb {
  @retransmits[comm] = count();
}
'
```

Network Performance Bottlenecks

- Socket buffer sizes
- TCP retransmissions
- Connection establishment latency
- Packet drops
- Protocol overhead

TCP Connection Tracking

```
#!/usr/bin/env bpftrace

#include
#include

BEGIN {
  printf("Tracing TCP connections... Press Ctrl-C to end.\n");
  printf("%-6s %-16s %-16s %-16s\n", "PID", "COMM", "REMOTE_ADDR", "LATENCY(μs)");
}

// Track connection start
kprobe:tcp_v4_connect,
kprobe:tcp_v6_connect {
  // Store timestamp when connection initiated
  @connect_start[tid] = nsecs;
}

// Extract remote address
kretprobe:tcp_v4_connect {
  $sk = (struct sock *)arg0;
  $daddr = ntop($sk->__sk_common.skc_daddr);
  $dport = $sk->__sk_common.skc_dport;

  // Convert port from network to host byte order
  $dport = (($dport & 0xFF) << 8) | ($dport >> 8);

  @connect_addr[tid] = $daddr;
  @connect_port[tid] = $dport;
}

// Track connection completion
kprobe:tcp_finish_connect {
  $sk = (struct sock *)arg0;
  $start = @connect_start[tid];

  if ($start) {
    $addr = @connect_addr[tid];
    $port = @connect_port[tid];
    $latency = (nsecs - $start) / 1000; // microseconds

    printf("%-6d %-16s %-16s:%-5d %9d\n",
           pid, comm, $addr, $port, $latency);

    // Aggregate statistics
    @conn_latency_us[comm] = hist($latency);

    // Cleanup
    delete(@connect_start[tid]);
    delete(@connect_addr[tid]);
    delete(@connect_port[tid]);
  }
}

END {
  printf("\nTCP connection latency histograms by process:\n");
  print(@conn_latency_us);
}
```

This script provides detailed visibility into TCP connection establishment latency.

DNS Resolution Analysis

DNS lookups can cause unexpected application stalls:

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Tracing DNS queries... Press Ctrl-C to end.\n");
}

// Track getaddrinfo entry
uprobe:/lib/x86_64-linux-gnu/libc.so.6:getaddrinfo {
  // Store the query name and timestamp
  @dns_start[tid] = nsecs;
  @dns_query[tid] = str(arg0);
  @dns_process[tid] = comm;
}

// Track getaddrinfo return
uretprobe:/lib/x86_64-linux-gnu/libc.so.6:getaddrinfo {
  $start = @dns_start[tid];
  $query = @dns_query[tid];
  $process = @dns_process[tid];

  if ($start) {
    $latency = (nsecs - $start) / 1000000; // milliseconds

    // Only print if latency is over 10ms
    if ($latency > 10) {
      printf("DNS query for %s by %s took %d ms\n",
             $query, $process, $latency);
    }

    // Aggregate statistics
    @dns_latency_ms[$process, $query] = hist($latency);

    // Cleanup
    delete(@dns_start[tid]);
    delete(@dns_query[tid]);
    delete(@dns_process[tid]);
  }
}

END {
  printf("\nDNS resolution latency histograms (ms):\n");
  print(@dns_latency_ms);
}
```

Adjust the libc path for your specific system.

Custom Problem Detection Scripts

Creating alerting scripts for production use

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Starting performance anomaly detection...\n");
}

// Define thresholds
#define DISK_LATENCY_THRESHOLD_MS 100
#define CPU_RUNQ_THRESHOLD 10
#define PROCESS_OF_INTEREST "nginx"

// Monitor disk I/O latency spikes
tracepoint:block:block_rq_issue {
  @start[args->dev, args->sector] = nsecs;
}

tracepoint:block:block_rq_complete {
  $start = @start[args->dev, args->sector];
  if ($start) {
    $latency_ms = (nsecs - $start) / 1000000;
    if ($latency_ms > DISK_LATENCY_THRESHOLD_MS) {
      printf("ALERT: High disk latency detected: %d ms for dev %d\n",
             $latency_ms, args->dev);
    }
    delete(@start[args->dev, args->sector]);
  }
}

// Monitor process-specific anomalies
profile:hz:10 /comm == PROCESS_OF_INTEREST/ {
  @process_samples = count();
}

// Detect CPU run queue length spikes
profile:hz:100 {
  $len = *(*int32*)(((void *)curtask) + 0x440);  // Adjust offset as needed
  if ($len > CPU_RUNQ_THRESHOLD) {
    printf("ALERT: High run queue length on CPU %d: %d tasks\n", cpu, $len);
  }
}

// Print regular status updates
interval:s:30 {
  printf("Monitoring active, no critical alerts triggered\n");
}

END {
  printf("Performance anomaly detection ending.\n");
}
```

Deploy this script in production to get early warnings of performance issues.

Combining CPU, Memory, and I/O Analysis

For comprehensive system analysis, create an integrated script:

```
#!/usr/bin/env bpftrace

BEGIN {
  printf("Starting integrated performance analysis...\n");
}

// CPU profiling (sampling)
profile:hz:49 {
  @cpu_stacks[kstack, ustack, comm] = count();
}

// Memory allocations
kprobe:kmalloc /comm == str($1)/ {
  @kmalloc_bytes = hist(arg1);
}

// I/O tracking
tracepoint:block:block_rq_issue {
  @io_start[args->dev, args->sector] = nsecs;
  @io_type[args->dev, args->sector] = args->rwbs;
}

tracepoint:block:block_rq_complete {
  $start = @io_start[args->dev, args->sector];
  $type = @io_type[args->dev, args->sector];

  if ($start) {
    $latency = nsecs - $start;
    @io_latency_ms[$type, comm] = hist($latency / 1000000);
    delete(@io_start[args->dev, args->sector]);
    delete(@io_type[args->dev, args->sector]);
  }
}

// Print stats every minute
interval:s:60 {
  time("%H:%M:%S Periodic performance summary:\n");

  printf("\nI/O Latency distribution (ms):\n");
  print(@io_latency_ms);
  clear(@io_latency_ms);

  printf("\nMemory allocation sizes:\n");
  print(@kmalloc_bytes);
  clear(@kmalloc_bytes);

  printf("\nTop CPU stack traces:\n");
  print(@cpu_stacks, 5);
  clear(@cpu_stacks);
}

END {
  printf("Performance analysis complete.\n");
}
```

Usage: bpftrace integrated.bt [process_name]

Performance Tuning Methodology

01

Observe

Use bpftrace to gather real-time data about the system's behaviour

02

Benchmark

Establish baseline performance metrics using targeted bpftrace scripts

03

Hypothesize

Form theories about bottlenecks based on observation data

04

Change

Make one system modification at a time

05

Validate

Measure the impact using the same bpftrace scripts and benchmarks

Repeat this cycle until performance goals are met or diminishing returns are reached.

Best Practices and Caveats

Production Overhead

Keep scripts focused and limit the data collected to minimize impact

```
# Limit tracing to specific processes
bpftrace -e 'kprobe:kmalloc /comm ==
"nginx"/ { ... }'

# Use sampling instead of tracing
every event
bpftrace -e 'profile:hz:49 { ... }'
```

Kernel Version Compatibility

Be aware of structure offsets that may change between kernel versions

```
# Use BTF (BPF Type Format) when
available
bpftrace -e 'kprobe:kmalloc {
  $task = (struct task_struct *)curtask;
  printf("%d\n", $task->pid);
}'
```

Map Size Limits

bpftrace maps have size limits; use filtering to stay within bounds

```
# Add filtering to limit map growth
bpftrace -e 'kprobe:vfs_read
 /pid == 1234 && arg2 > 1024/ { ... }'
```

Resources for Further Learning

Documentation and References

- **bpftrace Reference Guide**
- **Brendan Gregg's eBPF Resources**
- **BCC Tools Collection**
- Linux Observability with BPF (O'Reilly book)

Practice Environments

- Linux Kernel Developers' Virtual Machine
- eBPF Playground (online interactive environment)
- Linux Performance Virtual Machine

Continue to build your skills by creating custom scripts for your specific system needs. The best learning comes from solving real performance problems.

Remember: The key to mastering bpftrace is combining deep Linux systems knowledge with creative application of tracing techniques.