The bpftrace Language: A Comprehensive Guide

bpftrace is a high-level tracing language for Linux that leverages eBPF (extended Berkeley Packet Filter) technology. This presentation will guide you through the core concepts, syntax, and capabilities of the bpftrace language, helping you understand how to write effective tracing programs.

Concepts covered

01

Language Structure

Program structure, preamble, and action blocks

02

Probes

Different probe types and their usage

03

Variables and Data Types

Scratch variables, maps, and supported types

04

Control Flow

Conditionals, loops, and predicates

05

Advanced Features

Structs, pointers, type conversion, and BTF support

06

Configuration and Best Practices

Config variables, error handling, and architecture support

Program Structure

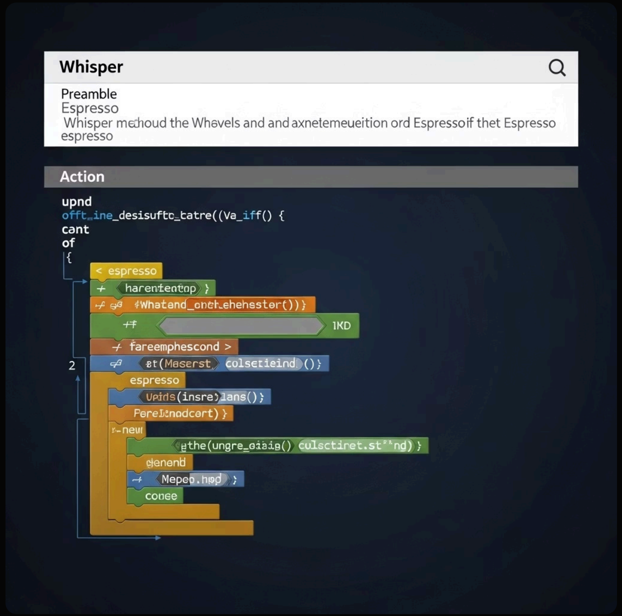
Every bpftrace script consists of two main parts:

Preamble

- Preprocessor definitions
- Type definitions
- Config block

Action Blocks

- Probes
- Predicates (optional)
- Actions



The structure is inspired by the D language used by dtrace, making it familiar for those with dtrace experience.

Action Blocks

Each action block consists of three parts:

```
probe[,probe] /predicate/ { action }
```

Probes

Specifies the event and event type to attach to (e.g., kprobe, tracepoint)

Predicate (Optional)

A condition that must be met for the action to be executed

Action

Programs that run when an event fires (and the predicate is met)

A semicolon-separated list of statements enclosed by brackets {}

Basic Script Example

```
BEGIN {
  printf("Tracing open syscalls... Hit Ctrl-C to end.\n");
}

tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
  printf("%-6d %-16s %s\n", pid, comm, str(args.filename));
}
```

This script has two action blocks and a total of 3 probes:

- The first action block uses the special BEGIN probe, which fires once during bpftrace startup
- The second action block uses two probes (open and openat) and prints the file being opened along with the process ID and name

Probe Types

Built-in Events

- BEGIN/END - Start/end of program
- self - Events in bpftrace process

Hardware/Software

- hardware - Processor-level events
- software - Kernel software events

Kernel Tracing

- kprobe/kretprobe - Kernel function start/return
- fentry/fexit - Kernel functions with BTF support
- tracepoint - Kernel static tracepoints

User Space

- uprobe/uretprobe - User-level function start/return
- usdt - User-level static tracepoints

Each probe type has a full name and a short name (e.g., kprobe:func and k:func are identical)

BEGIN/END Probes

These are special built-in events provided by the bpftrace runtime:

- **BEGIN** is triggered before all other probes are attached
- **END** is triggered after all other probes are detached

Note that specifying an END probe doesn't override the printing of 'non-empty' maps at exit. To prevent printing, all used maps need to be cleared in the END probe:

```
END {
  clear(@map1);
  clear(@map2);
}
```

Alternatively, you can set the configuration:

```
config = {
 print_maps_on_exit=0
}
```

Hardware and Software Probes

Hardware Probes

Pre-defined hardware events provided by the Linux kernel:

- cpu-cycles or cycles
- instructions
- cache-references
- cache-misses
- branch-instructions or branches
- branch-misses

```
hardware:cache-misses:1e6 {
  @[pid] = count();
}
```

Software Probes

Pre-defined software events provided by the Linux kernel:

- cpu-clock or cpu
- task-clock
- page-faults or faults
- context-switches or cs
- cpu-migrations
- minor-faults

```
software:faults:100 {
 @[comm] = count();
}
```

Interval and Profile Probes

Interval Probes

Fires at a fixed interval on one CPU at a time:

- interval:count (nanoseconds)
- interval:us:count (microseconds)
- interval:ms:count (milliseconds)
- interval:s:count (seconds)
- interval:hz:rate (frequency)

```
interval:1s {
  print(@syscalls);
  clear(@syscalls);
}
```

Profile Probes

Fires on each CPU on the specified interval:

- profile:count (nanoseconds)
- profile:us:count (microseconds)
- profile:ms:count (milliseconds)
- profile:s:count (seconds)
- profile:hz:rate (frequency)

```
profile:hz:99 {
 @[tid] = count();
}
```

Kernel Function Probes

kprobe/kretprobe

Dynamic instrumentation of kernel functions:

- kprobe[:module]:fn - Entry to function
- kprobe[:module]:fn+offset - Specific instruction
- kretprobe[:module]:fn - Return from function

```
kprobe:tcp_reset {
  @tcp_resets = count()
}
```

fentry/fexit

Similar to kprobe/kretprobe but with BTF support:

- fentry[:module]:fn - Entry to function
- fexit[:module]:fn - Return from function

```
fentry:x86_pmu_stop {
  printf("pmu %s stop\n",
    str(args.event->pmu->name));
}
```

Function arguments are available through argN for kprobe and through the args struct for fentry/fexit.

Tracepoint Probes

Tracepoint

Hooks into static events in the kernel:

- tracepoint:subsys:event

```
tracepoint:syscalls:sys_enter_openat {
  printf("%s %s\n", comm,
    str(args.filename));
}
```

Rawtracepoint

Similar to tracepoint but with better performance:

- rawtracepoint[:module]:event

```
rawtracepoint:vmlinux:kfree_skb {
  printf("%llx %llx\n", arg0, args.skb);
}
```

Tracepoint arguments are available in the args struct which can be inspected with verbose listing:

```
# bpftrace -lv "tracepoint:*"
tracepoint:xhci-hcd:xhci_setup_device_slot u32 info u32 info2 u32 tt_info u32 state ...
```

User Space Probes

uprobe/uretprobe

Dynamic instrumentation of user functions:

- uprobe:binary:func - Entry to function
- uprobe:binary:offset - Specific instruction
- uretprobe:binary:func - Return from function

```
uprobe:/bin/bash:readline {
  printf("arg0: %d\n", arg0);
}
```

USDT (User Statically-Defined Tracing)

Static tracepoints in user applications:

- usdt:binary_path:probe_name
- usdt:binary_path:[probe_namespace]:probe_name

```
usdt:/root/tick:loop {
  printf("%s: %d\n", str(arg0), arg1);
}
```

When tracing libraries, it's sufficient to specify the library name instead of a full path:

```
uprobe:libc:malloc {
 printf("Allocated %d bytes\n", arg0);
}
```

Special Probe Types

Iterator Probes

Allow iteration over kernel objects:

- iter:task
- iter:task_file
- iter:task_vma

```
iter:task {
  printf("%s:%d\n",
    ctx->task->comm,
    ctx->task->pid);
}
```

Watchpoint Probes

Memory watchpoints provided by the kernel:

- watchpoint:address:length:mode
- watchpoint:function+argN:length:mode

```
watchpoint:0x10000000:8:rw {
  printf("hit!\n");
}
```

Iterator probes can't be mixed with any other probe type. Watchpoint modes include read (r), write (w), and execute (x).

Variables in bpftrace

Scratch Variables

- Names start with $ (e.g., $myvar)
- Kept on the BPF stack
- Limited to lexical block scope
- Can be declared with let

```
$a = 1;
if ($a == 1) {
  $b = "hello";
  $a = 2;
}
```

Map Variables

- Names start with @ (e.g., @mymap)
- Use BPF maps
- Exist for the lifetime of bpftrace
- Accessible from all action blocks

```
@count = 0;
@bytes[pid] = count();
@stats[comm, pid] = sum(bytes);
```

The data type of a variable is automatically determined during first assignment and cannot be changed afterward.

Map Types and Usage

Maps without Explicit Keys

Values can be assigned directly to maps without a key:

```
@name = expression
```

Note: You can't iterate over these maps as they don't have an accessible key.

Maps with Keys

Single value map keys:

```
@name[key] = expression
```

Multiple value map keys (tuples):

```
@name[(key1,key2)] = expression
@name[key1,key2] = expression
```

Per-thread variables can be implemented as a map keyed on the thread ID:

```
kprobe:do_nanosleep {
 @start[tid] = nsecs;
}

kretprobe:do_nanosleep /has_key(@start, tid)/ {
 printf("slept for %d ms\n", (nsecs - @start[tid]) / 1000000);
 delete(@start, tid);
}
```

Data Types

Integer Types

uint8	Unsigned 8-bit integer
int8	Signed 8-bit integer
uint16	Unsigned 16-bit integer
int16	Signed 16-bit integer
uint32	Unsigned 32-bit integer
int32	Signed 32-bit integer
uint64	Unsigned 64-bit integer
int64	Signed 64-bit integer

Other Types

- Strings
- Pointers
- Structs
- Arrays
- Tuples

Note: Floating-point numbers are not supported by BPF and therefore not by bpftrace.

Integers are by default represented as 64-bit signed but that can be changed by either casting them or explicitly specifying the type upon declaration.

Literals

Integer Literals

- Decimal: 123
- Octal: 0123
- Hexadecimal: 0x10 or 0X10
- Scientific: 2e3 (= 2000)

Underscores can be used as field separators: 1_000_123_000

Duration suffixes: ns, us, ms, s, m, h, d

```
$a = 1m; // 60,000,000,000 nanoseconds
```

String Literals

Defined by enclosing characters in double quotes:

```
$str = "Hello world";
```

Escape sequences:

- \n - Newline
- \t - Tab
- \0nn - Octal value nn
- \xnn - Hexadecimal value nn

Character literals are not supported; use the corresponding ASCII code instead:

```
BEGIN { printf("Echo A: %c\n", 65); }
```

Control Flow: Conditionals

If/Else Statements

```
if (condition) {
  // if block
} else if (condition) {
  // else if block
} else {
  // else block
}
```

Ternary Operator

```
condition ? ifTrue : ifFalse
```

Both the ifTrue and ifFalse expressions must be of the same type.

```
$a == 1 ?
 print("true") :
 print("false");

$b = $a > 0 ? $a : -1;
```

Control Flow: Loops

For Loops

Iterate over elements in a map:

```
for ($kv : @map) {
  print($kv.0); // key
  print($kv.1); // value
}
```

Iterate over a range of integers:

```
for ($i : start..end) {
  print($i);
}
```

While Loops

```
while (condition) {
  // block
}
```

Supports break and continue statements.

```
interval:s:1 {
  $i = 0;
  while ($i <= 100) {
    printf("%d ", $i);
    if ($i > 5) {
      break;
    }
    $i++
  }
  printf("\n");
}
```

Loop unrolling is also supported with the unroll statement:

```
unroll(3) {
 print("Unrolled")
}
```

Filters/Predicates

Filters (also known as predicates) can be added after probe names. The probe still fires, but it will skip the action unless the filter is true.

```
kprobe:vfs_read /arg2 < 16/ {
  printf("small read: %d byte buffer\n", arg2);
}

kprobe:vfs_read /comm == "bash"/ {
  printf("read by %s\n", comm);
}
```

Predicates are powerful for filtering events based on specific conditions, reducing the amount of data processed and output generated.

Operators

Arithmetic Operators

+ (addition), - (subtraction), * (multiplication), / (division), % (modulo)

Logical Operators

&& (AND), || (OR), ! (NOT)

Bitwise Operators

& (AND), | (OR), ^ (XOR), << (left shift), >> (right shift)

Relational Operators

< (less than), <= (less than or equal), > (greater than), >= (greater than or equal), == (equal), != (not equal)

Assignment Operators

=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=

Increment/Decrement

++ (increment), -- (decrement)

Structs

C-like structs are supported by bpftrace. Fields are accessed with the . operator. Fields of a pointer to a struct can be accessed with the -> operator.

Custom structs can be defined in the preamble:

```
struct MyStruct {
  int a;
  char b[10];
}
```

Using structs in action blocks:

```
kprobe:dummy {
  $ptr = (struct MyStruct *) arg0;
  $st = *$ptr;
  print($st.a);
  print($ptr->a);
}
```

Note: Constructing structs from scratch is not supported. They can only be read into a variable from a pointer.

Tuples

bpftrace has support for immutable N-tuples (n > 1). A tuple is a sequence type where every element can have a different type.

Tuples are a comma-separated list of expressions, enclosed in brackets:

```
$a = (1, 2);
$b = (3, 4, $a);
```

Individual fields can be accessed with the . operator. Tuples are zero-indexed like arrays:

```
print($a); // (1, 2)
print($b); // (3, 4, (1, 2))
print($b.0); // 3
print($b.2.1); // 2
```

Type Conversion

Basic Type Conversion

Integer and pointer types can be converted using explicit type casting:

```
$y = (uint32) $z;
$py = (int16 *) $pz;
```

Integer casts to a higher rank are sign extended. Conversion to a lower rank is done by zeroing leading bits.

Array Casts

It's possible to cast between integer arrays and integers:

```
$a = (uint8[8]) 12345;
$x = (uint64) $a;
```

Both the cast and the destination type must have the same size. When casting to an array, it is possible to omit the size which will be determined automatically.

This feature is especially useful when working with IP addresses:

```
fentry:tcp_connect {
 if (args->sk->__sk_common.skc_daddr == (uint32)pton("127.0.0.1"))
 // ...
}
```

Arrays

bpftrace supports accessing one-dimensional arrays like those found in C.

The [] operator is used to access elements:

```
struct MyStruct {
  int y[4];
}

kprobe:dummy {
  $s = (struct MyStruct *) arg0;
  print($s->y[0]);
}
```

Constructing arrays from scratch is not supported. They can only be read into a variable from a pointer.

Array casts allow conversion between byte arrays and integers:

```
BEGIN {
 $a = (int8[8])12345;
 printf("%x %x\n", $a[0], $a[1]);
 printf("%d\n", (uint64)$a);
}
```

Command Line Parameters

Positional Parameters

Custom options can be passed to a bpftrace program via positional parameters:

- Accessed via $1, $2, ..., $N
- $# returns the number of arguments supplied

```
BEGIN {
  printf("I got %d, %s (%d args)\n",
    $1, str($2), $#);
}
```

Running the program:

```
# bpftrace -e 'BEGIN {
  printf("I got %d, %s (%d args)\n",
    $1, str($2), $#);
}' 42 "hello"

I got 42, hello (2 args)
```

If a parameter is used that was not provided, it will default to zero for a numeric context, and "" for a string context.

Comments

Both single line and multi-line comments are supported:

```
// A single line comment
interval:s:1 {
  // can also be used to comment inline

  /*
   * a multi line comment
   */

  print(/* inline comment block */ 1);
}
```

Comments help make your code more readable and maintainable, especially for complex scripts that may be shared with others or revisited later.

Config Block

To improve script portability, you can set bpftrace config variables via the config block, which can only be placed at the top of the script (in the preamble) before any action blocks:

```
config = {
  stack_mode=perf;
  max_map_keys=2
}

BEGIN {
  ...
}
```

The names of the config variables can be in the format of environment variables or their lowercase equivalent without the BPFTRACE_ prefix:

- BPFTRACE_STACK_MODE
- STACK_MODE
- stack_mode

Note: Environment variables for the same config take precedence over those set inside a script config block.

Config Variables (1/2)

cache_user_symbols

Controls caching of user symbols (PER_PROGRAM, PER_PID, NONE)

cpp_demangle

Enables/disables C++ symbol demangling in userspace stack traces (true/false)

lazy_symbolication

Controls whether to symbolicate on-demand (true) or ahead of time (false)

license

The license bpftrace will use to load BPF programs into the kernel (default: "GPL")

log_size

Log size in bytes (default: 1000000)

max_bpf_progs

Maximum number of BPF programs that bpftrace can generate (default: 1024)

Config Variables (2/2)

max_map_keys

Maximum number of keys that can be stored in a map (default: 4096)

max_probes

Maximum number of probes that bpftrace can attach to (default: 1024)

max_strlen

Maximum length for values created by str(), buf() and path() (default: 1024)

missing_probes

Controls handling of probes which cannot be attached (error, warn, ignore)

stack_mode

Output format for ustack and kstack builtins (bpftrace, perf, raw)

print_maps_on_exit

Controls whether maps are printed on exit (true/false)

BTF Support

If the kernel has BTF (BPF Type Format) data, all kernel structs are always available without defining them:

```
kprobe:vfs_open {
  printf("open path: %s\n",
    str(((struct path *)arg0)->dentry->d_name.name));
}
```

To allow users to detect this situation in scripts, the preprocessor macro BPFTRACE_HAVE_BTF is defined if BTF is detected.

Requirements for BTF

- For vmlinux:
  - Linux 4.18+ with CONFIG_DEBUG_INFO_BTF=y
  - Building requires dwarves with pahole v1.13+
- For kernel modules:
  - Linux 5.11+ with CONFIG_DEBUG_INFO_BTF_MODULES=y
  - Building requires dwarves with pahole v1.19+

Address Spaces

Kernel and user pointers live in different address spaces which, depending on the CPU architecture, might overlap.

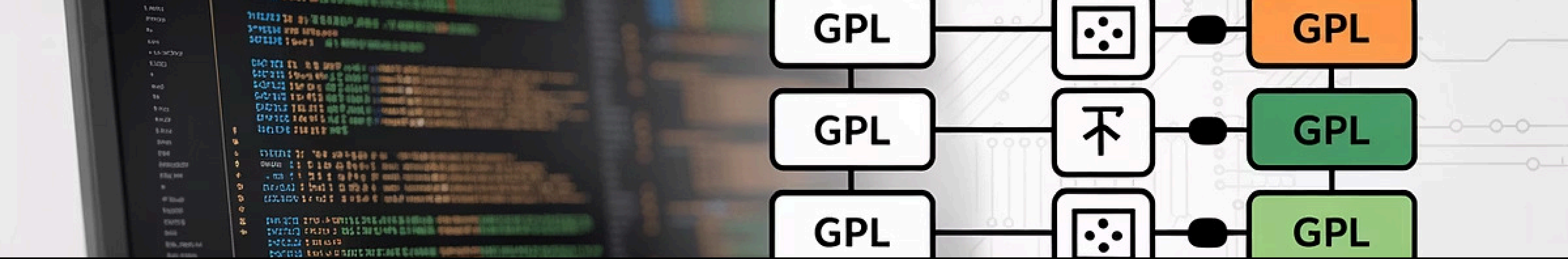
Trying to read a pointer that is in the wrong address space results in a runtime error:

```
stdin:1:9-12: WARNING: Failed to probe_read_user:
Bad address (-14)
BEGIN { @=*uptr(kaddr("do_poweroff")) }
```

bpftrace tries to automatically set the correct address space for a pointer based on the probe type, but might fail in cases where it is unclear.

The address space can be changed with the kptr and uptr functions:

- kptr() - Convert to kernel pointer
- uptr() - Convert to user pointer



BPF License

By default, bpftrace uses "GPL", which is actually "GPL version 2", as the license it uses to load BPF programs into the kernel.

Some other examples of compatible licenses are:

- "GPL v2"
- "Dual MPL/GPL"

You can specify a different license using the "license" config variable:

```
config = {
  license="Dual BSD/GPL"
}
```

The license affects what kernel functions your BPF program can call. Some functions are only available to GPL-compatible programs.

Clang Environment Variables

bpftrace parses header files using libclang, the C interface to Clang. Thus environment variables affecting the clang toolchain can be used.

For example, if header files are included from a non-default directory, the CPATH or C_INCLUDE_PATH environment variables can be set to allow clang to locate the files:

```
export CPATH=/path/to/headers
bpftrace script.bt
```

Other useful environment variables:

- BPFTRACE_KERNEL_SOURCE - Override default kernel source path
- BPFTRACE_KERNEL_BUILD - Specify out-of-tree Linux kernel build
- BPFTRACE_NO_CPP_DEMANGLE - Disable C++ symbol demangling
- BPFTRACE_LOG_SIZE - Set log size

Common Errors

BPF Stack Limit Exceeded

"Looks like the BPF stack limit of 512 bytes is exceeded"

- Reduce the size of data used in the program
- Use fewer map keys
- Split your program over multiple probes

Kernel Headers Not Found

bpftrace requires kernel headers for certain features

- Default search: /lib/modules/$(uname -r)
- Override with BPFTRACE_KERNEL_SOURCE

Probe Attachment Failures

Probes may fail to attach if they don't exist or there are permission issues

- Use -v for verbose output
- Set missing_probes=warn to continue despite failures

Map Printing

By default, when a bpftrace program exits it will print all maps to stdout. There are two ways to control this behavior:

Option 1: Config Variable

```
config = {
  print_maps_on_exit=0
}

BEGIN {
  @a = 1;
  @b[1] = 1;
}
```

Option 2: Clear Maps in END

```
BEGIN {
  @a = 1;
  @b[1] = 1;
}

END {
  clear(@a);
  clear(@b);
}
```

Both approaches will result in no maps being printed when the program exits.

PER_CPU Types

For bpftrace PER_CPU types, you may coerce (and thus force a more expensive synchronous read) the type to an integer using a cast or by doing a comparison.

This is useful when you need an integer during comparisons, printf(), or other operations.

```
BEGIN {
 @c = count();
 @s = sum(3);
 @s = sum(9);

 if (@s == 12) { // Coerces @s
 printf("%d %d\n",
 (int64)@c, // Coerces @c
 (int64)@s); // Coerces @s and prints "1 12"
 }
}
```

Supported Architectures

x86_64

Intel and AMD 64-bit processors

arm64 / arm32

ARM 64-bit and 32-bit processors

s390x

IBM Z mainframe architecture

mips64

MIPS 64-bit processors

ppc64

PowerPC 64-bit processors

riscv64

RISC-V 64-bit processors

bpftrace supports a wide range of architectures, making it a versatile tool for system tracing across different platforms.

Systemd Support

If bpftrace has been built with -DENABLE_SYSTEMD=1, you can run bpftrace in the background using systemd:

```
# systemd-run --unit=bpftrace --service-type=notify \
  bpftrace -e 'kprobe:do_nanosleep { \
    printf("%d sleeping\n", pid); \
  }'
```

In this example, systemd-run will not finish until bpftrace has attached its probes, ensuring that all following commands will be traced.

To stop tracing, run:

```
systemctl stop bpftrace
```

For debugging early boot issues, bpftrace can be invoked via a systemd service ordered before the service that needs to be traced:

```
[Unit]
Before=service-i-want-to-trace.service

[Service]
Type=notify
ExecStart=bpftrace -e 'kprobe:do_nanosleep { \
 printf("%d sleeping\n", pid); \
}'
```

Complex Tools

bpftrace can be used to create powerful one-liners and simple tools. For complex tools, which may involve:

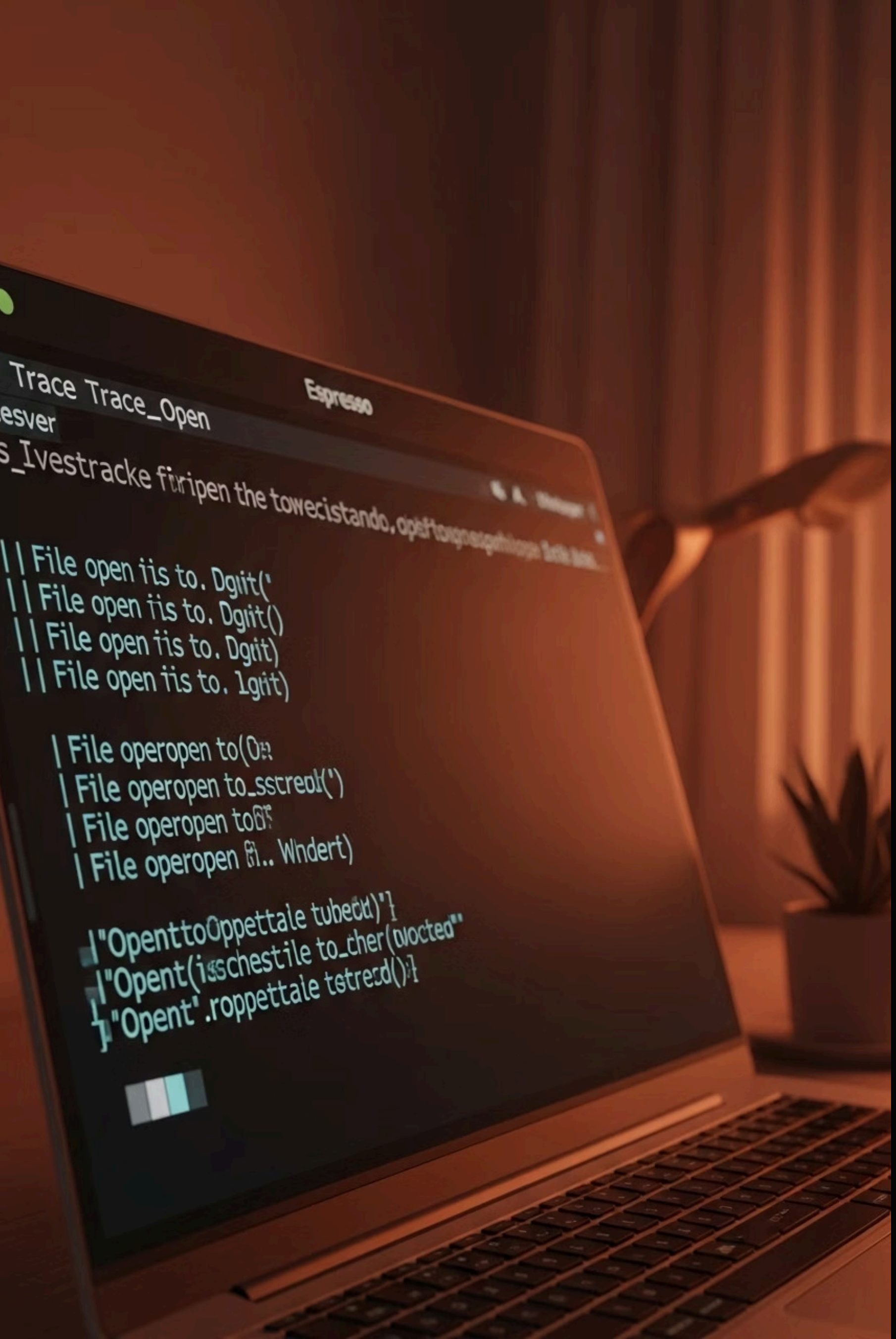
- Command line options
- Positional parameters
- Argument processing
- Customized output

Consider switching to bcc, which provides Python (and other) front-ends, enabling usage of all the other Python libraries (including argparse), as well as direct control of the kernel BPF program.

An expected development path would be:

1. Exploration with bpftrace one-liners
2. Ad hoc scripting with bpftrace
3. Advanced tooling with bcc

For example, the bpftrace xfsdist.bt tool also exists in bcc as xfsdist.py. Both measure the same functions and produce the same summary of information, but the bcc version supports various arguments and is more verbose (131 lines vs. 22 lines).



Example: Tracing Open Syscalls

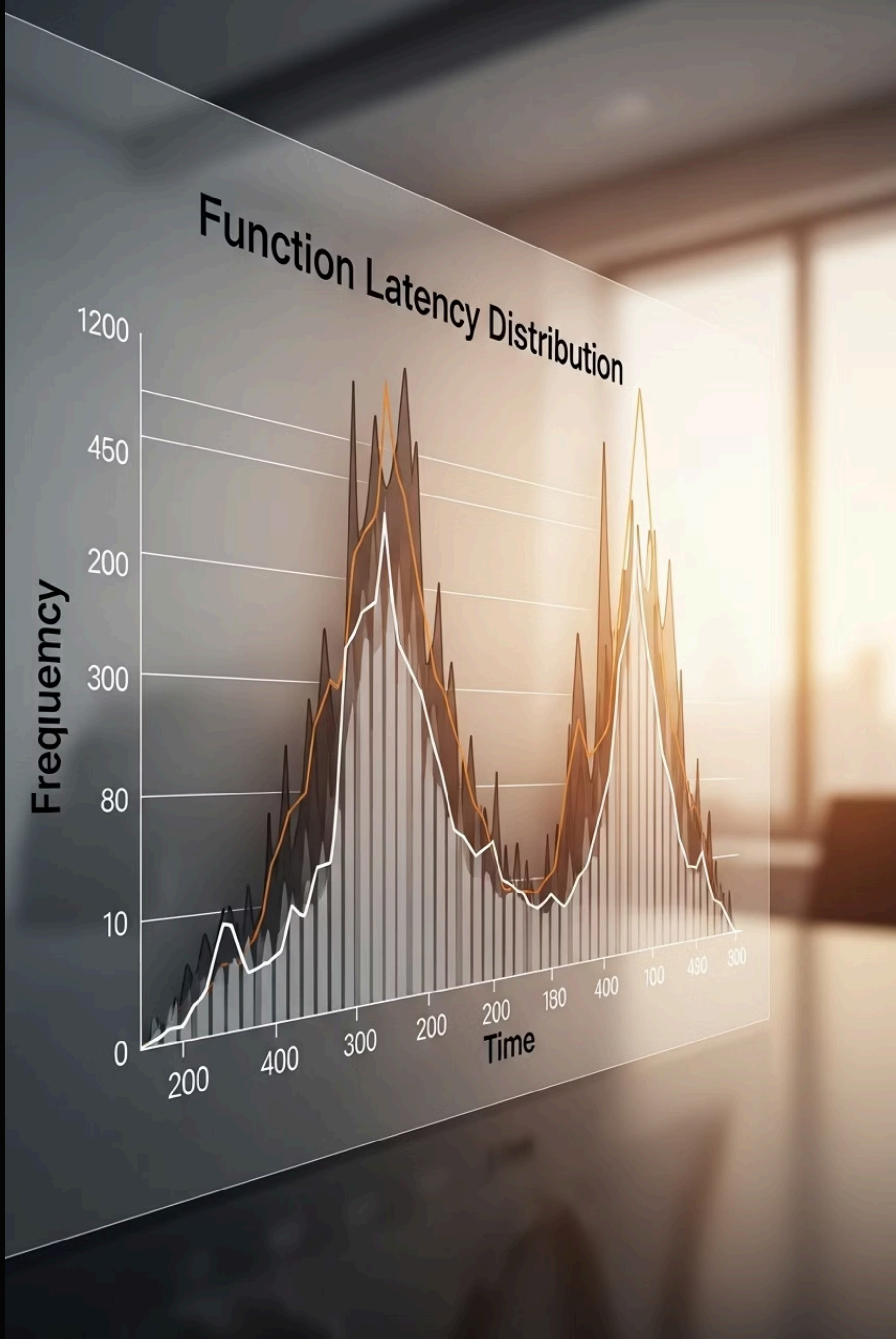
```
BEGIN {
  printf("Tracing open syscalls... Hit Ctrl-C to end.\n");
}

tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
  printf("%-6d %-16s %s\n", pid, comm, str(args.filename));
}
```

This script traces the open and openat system calls, printing the process ID, process name, and the filename being opened.

Example: Measuring Function Latency

```
kprobe:do_nanosleep {
  @start[tid] = nsecs;
}

kretprobe:do_nanosleep /@start[tid]/ {
  @sleep_time_ns = hist(nsecs - @start[tid]);
  delete(@start[tid]);
}
```

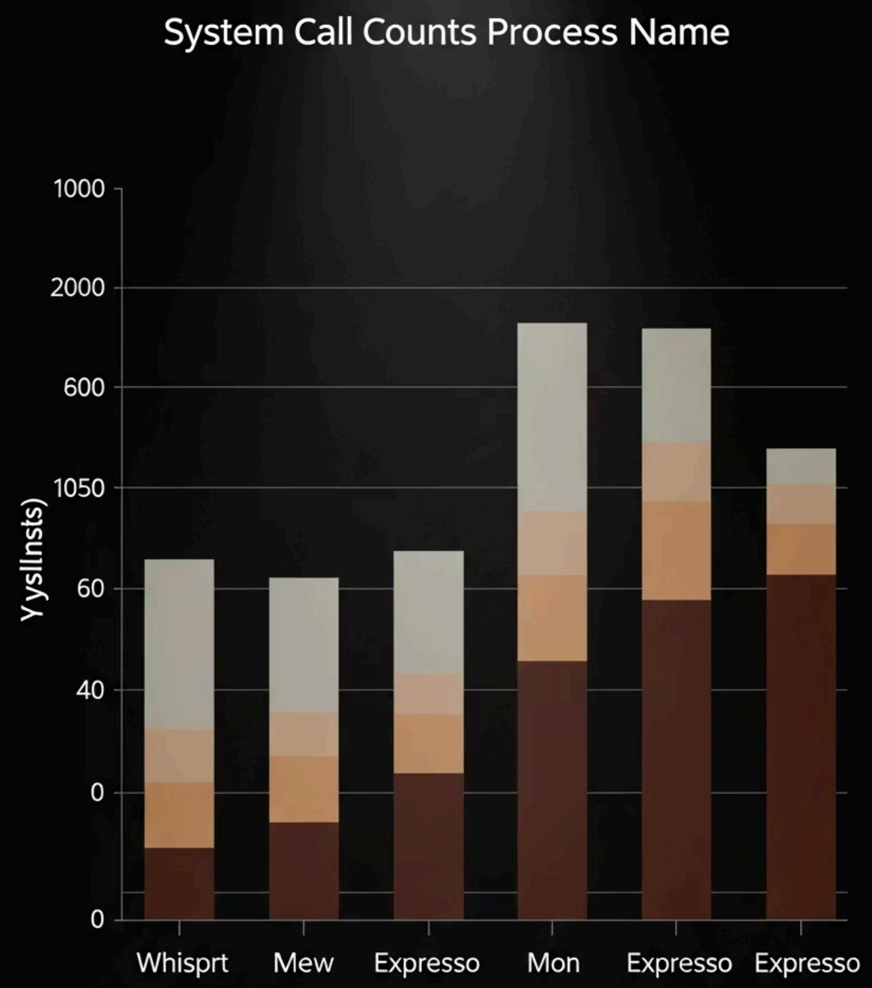
This script measures how long the do_nanosleep kernel function takes to execute and creates a histogram of the results.



Example: Counting System Calls by Process

```
tracepoint:raw_syscalls:sys_enter {
  @syscalls[comm] = count();
}

interval:s:5 {
  print(@syscalls);
  clear(@syscalls);
}
```

This script counts system calls by process name and prints the results every 5 seconds, then clears the counters for the next interval.



Example: Tracing File I/O Size Distribution

```
tracepoint:syscalls:sys_exit_read,
tracepoint:syscalls:sys_exit_write {
  @bytes[probe] = hist(args.ret);
}

END {
  print(@bytes);
}
```

This script creates histograms of the sizes of read and write operations, showing the distribution of I/O sizes.

Example: Tracing TCP Connections

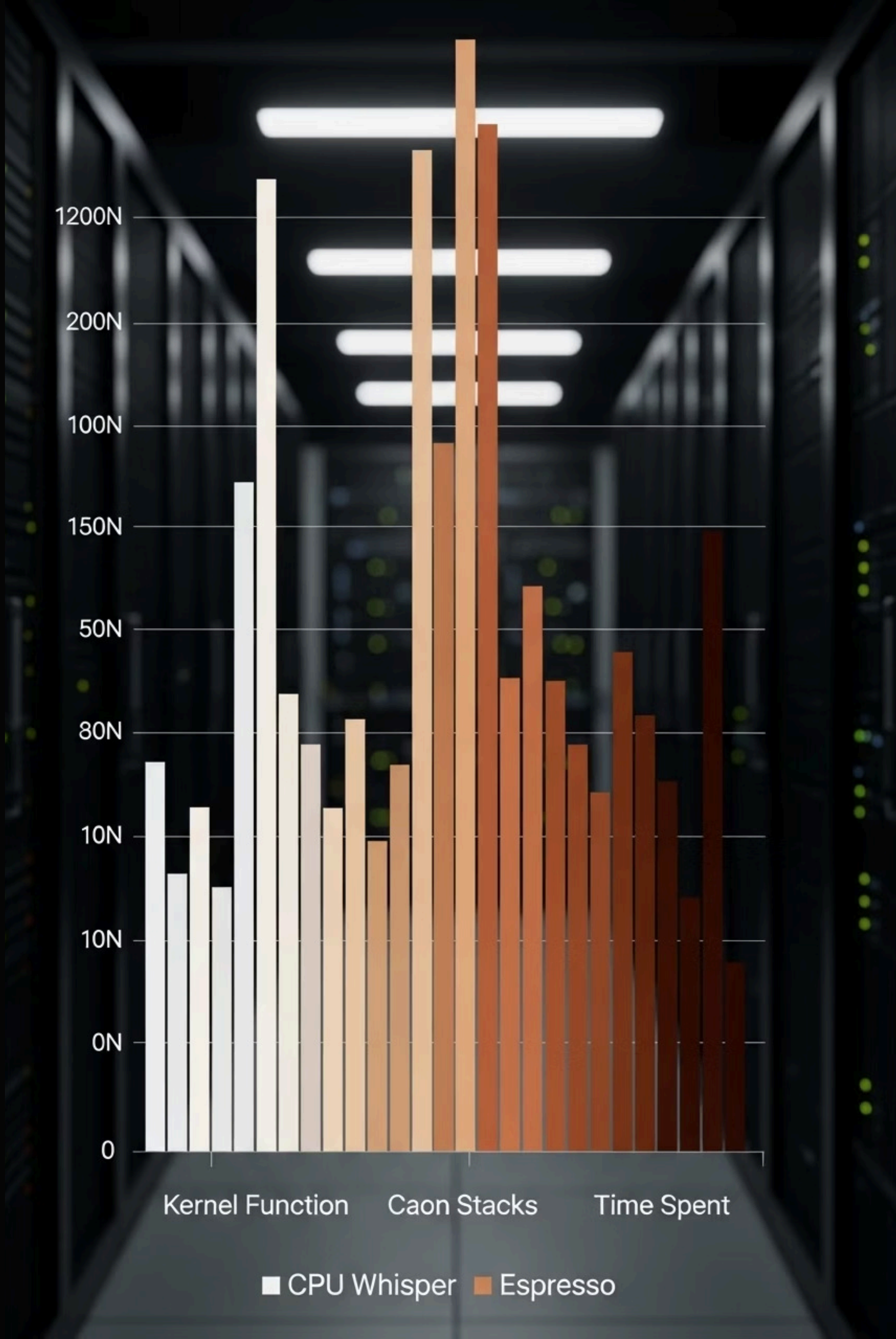
```
tracepoint:sock:inet_sock_set_state {
  $sk = (struct sock *)args.skaddr;
  $inet_family = $sk->__sk_common.skc_family;

  if ($inet_family == AF_INET || $inet_family == AF_INET6) {
    if (args.oldstate == TCP_SYN_SENT && args.newstate == TCP_ESTABLISHED) {
      printf("TCP connect: %s:%d -> %s:%d\n",
        ntop($inet_family, $sk->__sk_common.skc_rcv_saddr),
        $sk->__sk_common.skc_num,
        ntop($inet_family, $sk->__sk_common.skc_daddr),
        $sk->__sk_common.skc_dport);
    }
  }
}
```

This script traces TCP connection establishments, showing source and destination IP addresses and ports.

Example: CPU Flame Graph

```
profile:hz:99 {
  @[kstack] = count();
}
```

This one-liner samples the kernel stack trace at 99 Hz and counts the occurrences of each unique stack. The output can be piped to flamegraph.pl to generate a flame graph visualization.



Example: Using Conditionals and Loops

```
tracepoint:syscalls:sys_enter_openat {
  $filename = str(args.filename);

  if ($filename != "") {
    $count = 0;
    $len = strlen($filename);

    for ($i = 0; $i < $len; $i++) {
      if ($filename[$i] == '/') {
        $count++;
      }
    }

    @path_depth[$count] = count();
  }
}
```

This script counts the depth of file paths (number of / characters) being opened and maintains a histogram of path depths.

Example: Using Maps with Multiple Keys

```
tracepoint:block:block_rq_issue {
  @start[args.dev, args.sector] = nsecs;
}

tracepoint:block:block_rq_complete /@start[args.dev, args.sector]/ {
  @latency[args.dev, args.sector] = nsecs - @start[args.dev, args.sector];
  delete(@start[args.dev, args.sector]);
}
```

This script measures I/O latency for each device and sector, using a map with multiple keys to track start times and calculate latencies.

Resources for Learning More

Official Documentation

The bpftrace reference guide and language specification

https://bpftrace.org/docs/

GitHub Repository

Source code, examples, and issue tracking

https://github.com/iovisor/bpftrace

BPF Performance Tools Book

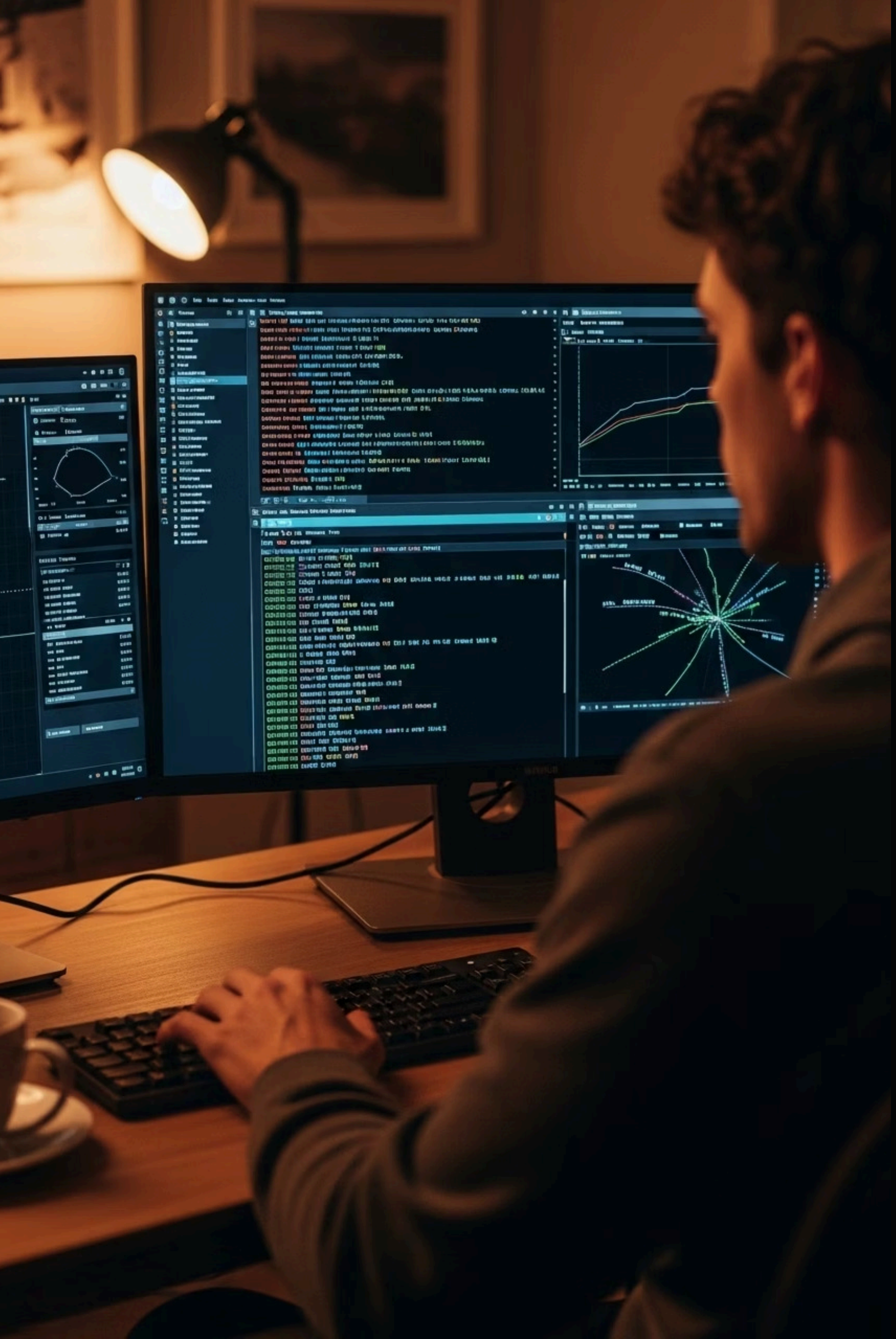
Comprehensive guide by Brendan Gregg

http://www.brendangregg.com/bpf-performance-tools-book.html

Example Tools

Collection of ready-to-use bpftrace scripts

https://github.com/iovisor/bpftrace/tree/master/tools



Summary

In this presentation, we've covered:

- The structure of bpftrace programs
- Various probe types and their usage
- Variables, maps, and data types
- Control flow with conditionals and loops
- Advanced features like structs and type conversion
- Configuration options and best practices

bpftrace provides a powerful yet concise language for Linux system tracing, enabling you to:

- Diagnose performance issues
- Monitor system behavior
- Understand application interactions with the kernel
- Create custom observability tools

With the knowledge gained from this presentation, you should be able to start writing your own bpftrace scripts to explore and analyze system behavior.