Profiling PostgreSQL: perf, Flame Graphs, and eBPF Tools in Practice

Debugging PostgreSQL Performance Beyond `EXPLAIN`

Jan Nidzwetzki @ Nile
https://thenile.dev

Stuff We Will Talk About

- Debug PostgreSQL performance beyond `EXPLAIN`.
- Analyze a deliberately slow is_odd function.
- Profile CPU usage with perf.
- Visualize performance data with Flame Graphs.
- Use eBPF tracing tools for deeper insights.

Hi, I'm Jan 👋

- **Now**: Founding Engineer at Nile. PostgreSQL for multi-tenant applications.
- **Formerly**: Database researcher focused on handling multidimensional data (*spatial and spatio-temporal*).
- **Started:** First sysadmin contract with PostgreSQL in 2002.

pg_slow: a Deliberately Slow PostgreSQL Extension

PostgreSQL Extension **pg_slow**

- **pg_slow** is a PostgreSQL extension created for this presentation.
- It provides the function: `boolean is_odd_slow(int)`.

➡ We want to **understand** why the function is slow and what to optimize.

➡ This presentation focuses on **profiling** and **tracing tools**, not on C code optimization.

A Very Slow `is_odd` Function

```
db1=# CREATE EXTENSION pg_slow;
db1=# CREATE TABLE data(value INT);
db1=# INSERT INTO data(value) SELECT generate_series(1, 50000);
```

```
db1=# SELECT value, is_odd_slow(value) FROM data;
 value | is_odd_slow
-------+-------------
     1 | t
     2 | f
     3 | t
     4 | f
[...]
 49997 | t
 49998 | f
 49999 | t
 50000 | f
(50000 rows)
Time: 41194.986 ms (00:41.195)
```

EXPLAIN is Not Enough

```
db1=# EXPLAIN (VERBOSE, ANALYZE) SELECT value, is_odd_slow(value) FROM data;
```

EXPLAIN is Not Enough

```
db1=# EXPLAIN (VERBOSE, ANALYZE) SELECT value, is_odd_slow(value) FROM data;

                                   QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on public.data  (cost=0.00..847.00 rows=50000 width=5) (actual
time=0.165..40774.297 rows=50000.00 loops=1)
   Output: value, is_odd_slow(value)
   Buffers: shared hit=222
 Planning Time: 0.898 ms
 Execution Time: 40775.788 ms
(5 rows)

Time: 40784.256 ms (00:40.784)
```

Functions are a black box for EXPLAIN

is_odd_slow Implementation

```
Datum is_odd_slow(PG_FUNCTION_ARGS)
{
    bool result;
    int32 val = PG_GETARG_INT32(0);

    switch (BLACK_BOX_DECIDER(val))
    {
        case 0:
            result = is_odd_1(val);
            break;
        case 1:
            result = is_odd_2(val);
            break;
        case 2:
            result = is_odd_3(val);
            break;
        default:
           pg_unreachable();
    }

    PG_RETURN_BOOL(result);
}
```

- Where is the time spent?
- What should we optimize?

Debugging Performance Beyond `EXPLAIN` — Profiling with perf

Profiling

Profiling captures dynamic program behavior (e.g., function-call durations).

- `perf` is a profiler for Linux. We use it for CPU profiling.
- Employs **sampling** to provide estimated results rather than precise measurements.
- Identifies **frequently executed** or **slow** code paths.

➡ **Goal:** Focus optimization efforts on **real** bottlenecks.

Side Note: Debug Builds

- A debug build of PostgreSQL is used in the examples (`--enable-debug` / `-ggdb -O0 -g3 -fno-omit-frame-pointer`).

✅ No function inlining and clear Flame Graphs.

❌ `Asserts` and `USE_ASSERT_CHECKING` sections will run.

❌ No code optimization.

❌ Slower execution times.

Example Program

```
int main(int argc, char *argv[]) {

    for (uint64_t i = 0; i < 10000000; i++) {
        /* Burn some CPU cycles */
        func1(i);

        /* func2 uses ~2 times more cycles than func1  */
        func2(i);

        /* func3 uses ~2 times more cycles than func2  */
        /* It calls func3a internally that needs       */
        /* ~50% of the CPU cycles.                     */
        func3(i);
    }

    return 0;
}
```

Profiling with `perf`

```
# Capture data...
perf record -g -F 111 -- ./myprog

# Generate report
perf report -n --stdio

# Text user interface (TUI)
perf report -n
```

`perf` for our Example Program

```
$ perf report -n --stdio
```

```
100.00%     0.00%     0  flamegraph-work  flamegraph-workload  [.] main
            |
            ---main
               |
               |--64.98%--func3
               |          |
               |           --29.62%--func3a
               |
               |--25.26%--func2
               |
                --9.76%--func1

  [...]
```

`perf` for our Example Program

$ perf report -n

```
Samples: 1K of event 'task-clock:ppp', Event count (approx.): 10342342332
  Children      Self       Samples  Command          Shared Object        Symbol
+  100.00%     0.00%             0  flamegraph-work  flamegraph-workload  [.] _start
+  100.00%     0.00%             0  flamegraph-work  libc.so.6            [.] __libc_start_main
+  100.00%     0.00%             0  flamegraph-work  libc.so.6            [.] 0x0000ffffa262221c
+  100.00%     0.00%             0  flamegraph-work  flamegraph-workload  [.] main
+   64.98%    35.37%           406  flamegraph-work  flamegraph-workload  [.] func3
+   29.62%    29.62%           340  flamegraph-work  flamegraph-workload  [.] func3a
+   25.26%    25.26%           290  flamegraph-work  flamegraph-workload  [.] func2
+    9.76%     9.76%           112  flamegraph-work  flamegraph-workload  [.] func1
```

`perf` **for** `is_odd_slow`

Running perf for is_odd_slow

```
$ perf report -n --stdio | wc -l
5092
```

⚠️ Returns more than **5000** lines for this small example.

➡ `perf` output can be too verbose.

perf for is_odd_slow

$ perf report -n

```
Samples: 1K of event 'task-clock:ppp', Event count (approx.): 11234234223
  Children      Self       Samples  Command   Shared Object       Symbol
+  100.00%     0.00%             0  postgres  postgres            [.] _start
+  100.00%     0.00%             0  postgres  libc.so.6           [.] __libc_start_main
+  100.00%     0.00%             0  postgres  libc.so.6           [.] 0x0000ffffb88d221c
+  100.00%     0.00%             0  postgres  postgres            [.] main
+  100.00%     0.00%             0  postgres  postgres            [.] PostmasterMain
+  100.00%     0.00%             0  postgres  postgres            [.] ServerLoop
+  100.00%     0.00%             0  postgres  postgres            [.] BackendStartup
+  100.00%     0.00%             0  postgres  postgres            [.] postmaster_child_launch
+  100.00%     0.00%             0  postgres  postgres            [.] BackendMain
+  100.00%     0.00%             0  postgres  postgres            [.] PostgresMain
+  100.00%     0.00%             0  postgres  postgres            [.] exec_simple_query
+  100.00%     0.00%             0  postgres  postgres            [.] PortalRun
+  100.00%     0.00%             0  postgres  postgres            [.] FillPortalStore
+  100.00%     0.00%             0  postgres  postgres            [.] PortalRunUtility
+  100.00%     0.00%             0  postgres  postgres            [.] ProcessUtility
+  100.00%     0.00%             0  postgres  postgres            [.] standard_ProcessUtility
+  100.00%     0.00%             0  postgres  postgres            [.] ExplainQuery
+  100.00%     0.00%             0  postgres  postgres            [.] ExplainOneQuery
+  100.00%     0.00%             0  postgres  postgres            [.] standard_ExplainOneQuery
```

perf **for** is_odd_slow

```
$ perf report -n
```

Samples: 1K of event 'task-clock:ppp', Event count (approx.): 11234234223

	Children	Self	Samples	Command	Shared Object		Symbol
+	100.00%	0.00%	0	postgres	postgres	[.]	ExplainOnePlan
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecutorRun
+	100.00%	0.00%	0	postgres	postgres	[.]	standard_ExecutorRun
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecutePlan
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecProcNode
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecProcNodeInstr
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecSeqScanWithProject
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecProject
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecEvalExprNoReturnSwitchContext
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecEvalExprNoReturn
+	100.00%	0.00%	0	postgres	postgres	[.]	ExecInterpExpr
+	99.92%	0.00%	0	postgres	pg_slow.so	[.]	is_odd_slow
+	83.72%	0.88%	11	postgres	pg_slow.so	[.]	is_odd_1
+	64.80%	1.04%	13	postgres	pg_slow.so	[.]	decrementString
+	43.95%	2.81%	35	postgres	postgres	[.]	pg_snprintf
+	40.58%	2.17%	27	postgres	postgres	[.]	pg_vsnprintf
+	38.97%	8.74%	109	postgres	postgres	[.]	dopr
+	27.35%	12.67%	158	postgres	postgres	[.]	fmtint
+	16.20%	16.20%	202	postgres	pg_slow.so	[.]	is_odd_2

Debugging Performance Beyond `EXPLAIN` — Flame Graphs

Flame Graphs: Visualizing Performance Hotspots

Flame Graphs are *"a visualization of hierarchical data that I created to visualize stack traces of profiled software so that the most **frequent code-paths** can be **identified quickly** and accurately."*

Brendan Gregg: https://www.brendangregg.com/flamegraphs.html

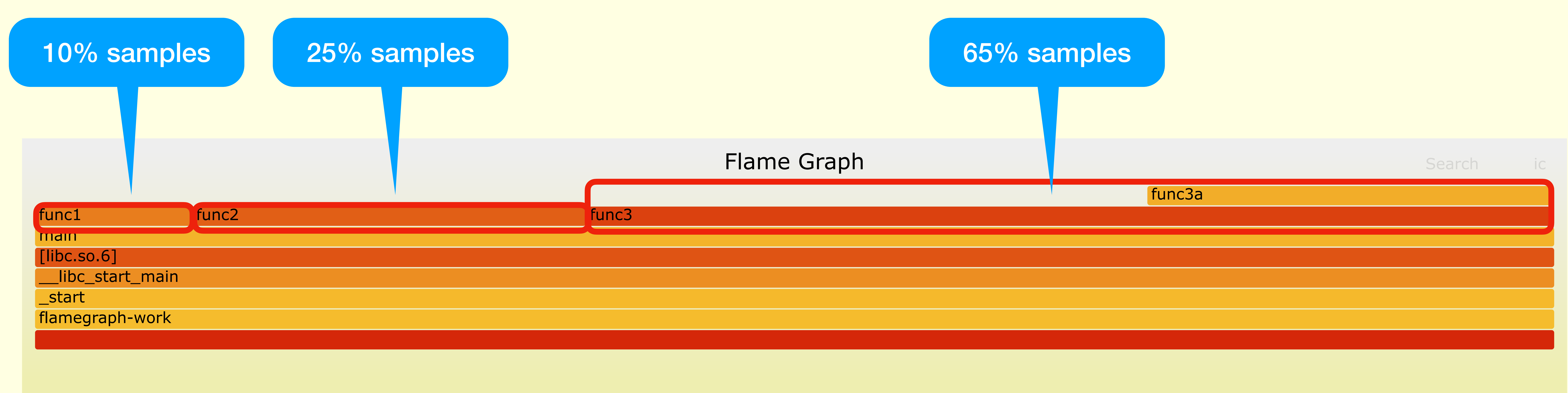
Paper: Brendan Gregg. 2016. The Flame Graph: This visualization of software execution is a new necessity for performance profiling and debugging. ACM Queue 14, 2, 91–110.

Creating a Flame Graph

The ***old*** way using https://github.com/brendangregg/FlameGraph

```
# Prepare
git clone https://github.com/brendangregg/FlameGraph ~/fg

# Record perf
perf record -g -F 111 -o perf.data -- ./myprog

# Converting perf data to stack traces...
perf script -i perf.data > perf.stacks

# Collapsing perf stacks...
~/fg/stackcollapse-perf.pl perf.stacks > perf.fold

# Generating SVG...
~/fg/flamegraph.pl perf.fold > perf.svg
```

Flame Graph for the Example Program



Creating a Flame Graph

The ***new*** way using perf directly

```
# Execute workload
./myprog

# Capture data (alternative approach - profile PID)
perf record -g -F 111 -p $pid

# Generate flame graph
perf script report flamegraph --allow-download
```

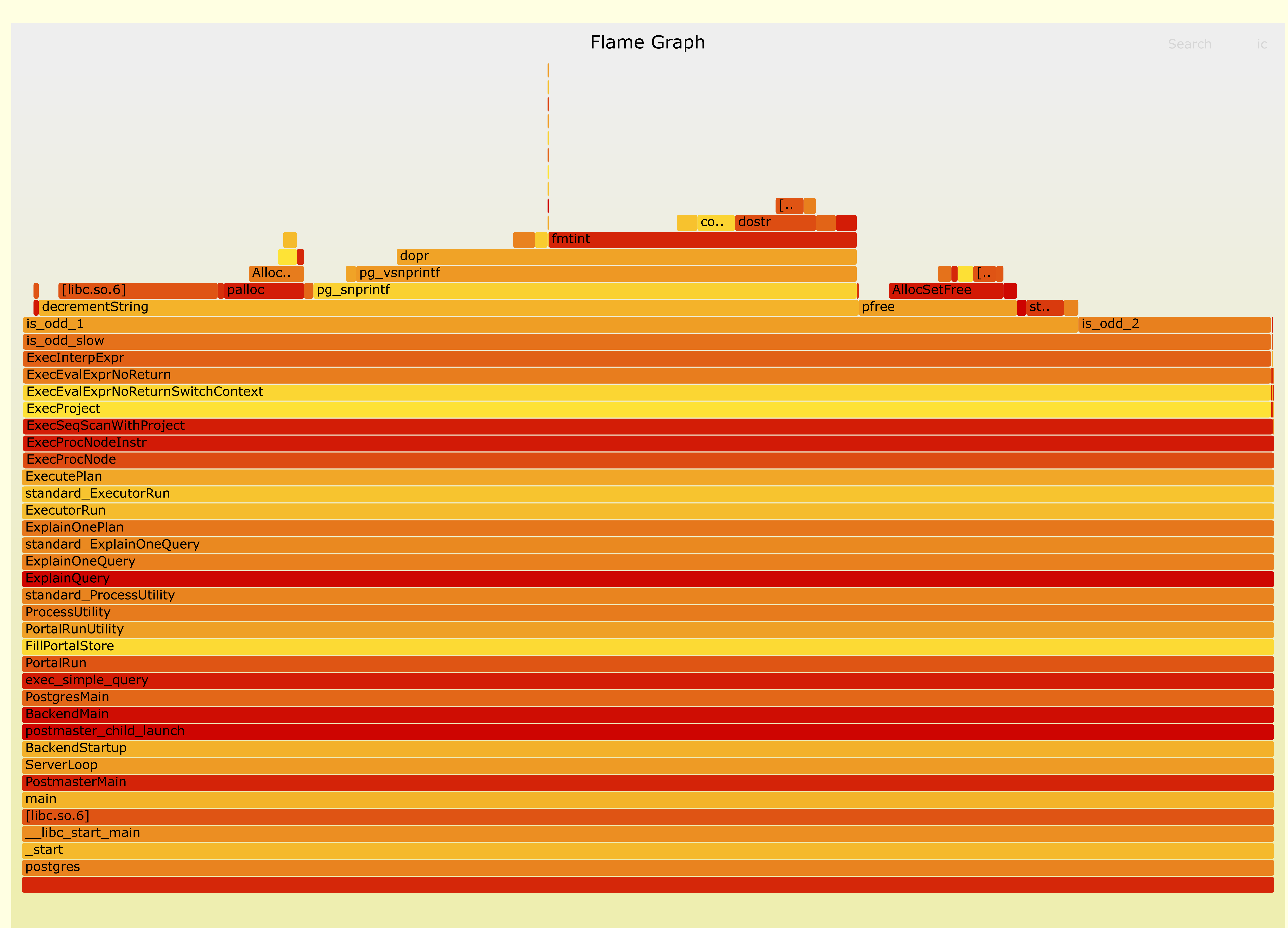
func3a
func3 func2 func1
main
[unknown]
__libc_start_main
_start
flamegraph-work (49134)
all

Creating a Flame Graph

Workflow for PostgreSQL

```
# Get the PID of the backend
db1=# SELECT pg_backend_pid();
 pg_backend_pid
----------------
          19236
(1 row)

# Run profiler
perf record -g -F 111 -o perf.data -p 19236

# Run query in PostgreSQL
[…]

# Converting perf data to stack traces...
perf script -i perf.data > perf.stacks

# Collapsing perf stacks and generate SVG...
~/fg/stackcollapse-perf.pl perf.stacks > perf.fold
~/fg/flamegraph.pl perf.fold > perf.svg
```

Flame Graph for `is_odd_slow`



Flame Graph for `is_odd_slow`

is_odd_slow Implementation

```
Datum is_odd_slow(PG_FUNCTION_ARGS)
{
    bool result;
    int32 val = PG_GETARG_INT32(0);

    switch (BLACK_BOX_DECIDER(val))
    {
        case 0:
            result = is_odd_1(val);
            break;
        case 1:
            result = is_odd_2(val);
            break;
        case 2:
            result = is_odd_3(val);
            break;
        default:
           pg_unreachable();
    }

    PG_RETURN_BOOL(result);
}
```

- Where is the time spent?
- What should we optimize? ✅

is_odd_slow Implementation

```
Datum is_odd_slow(PG_FUNCTION_ARGS)
{
    bool result;
    int32 val = PG_GETARG_INT32(0);

    switch (BLACK_BOX_DECIDER(val))
    {
        case 0:
            result = is_odd_1(val);       ~84% of the samples
            break;
        case 1:
            result = is_odd_2(val);       ~15% of the samples
            break;
        case 2:
            result = is_odd_3(val);
            break;
        default:
            pg_unreachable();
    }

    PG_RETURN_BOOL(result);
}
```

What is the distribution of the values returned by BLACK_BOX_DECIDER ?

Do we see so many samples because it is often called?

Debugging Performance Beyond `EXPLAIN` — eBPF Tools and Special Flame Graphs

- **eBPF** (extended Berkeley Packet Filter) allows you to run **sandboxed programs inside the Linux kernel.**
- **Verified:** a *verifier* checks the code before execution to prevent unbounded loops/recursion, deadlocks, or unauthorized memory access.
- **Event-Driven:** eBPF programs are triggered by events (e.g., kernel-space `kprobe` or user-space `uprobe`).
- **Data structures:** Maps (arrays, hashes, ring buffers, …) to persist data across events.
- **Communication:** Exchanges data between the kernel and user-space via these maps.

funccount-bpfcc for is_odd_slow

man funccount-bpfcc

“This tool is a quick way to determine which functions are being called, and at what rate. It uses in kernel eBPF maps to count function calls.”

Determine the function invocation counts

```
$ sudo funccount-bpfcc $(pg_config --pkglibdir)/pg_slow.so:
    'is_odd_slow|is_odd_1|is_odd_2|is_odd_3'

FUNC                                        COUNT
is_odd_1                                    16666
is_odd_2                                    16667
is_odd_3                                    16667
is_odd_slow                                 50000
Detaching...
```

Tracing not sampling!

is_odd_slow Implementation

```
Datum is_odd_slow(PG_FUNCTION_ARGS)
{
    bool result;
    int32 val = PG_GETARG_INT32(0);

    switch (BLACK_BOX_DECIDER(val))
    {
        case 0:
            result = is_odd_1(val);
            break;
        case 1:
            result = is_odd_2(val);
            break;
        case 2:
            result = is_odd_3(val);
            break;
        default:
            pg_unreachable();
    }

    PG_RETURN_BOOL(result);
}
```

What is the distribution of the values returned by BLACK_BOX_DECIDER ? ✅

16666 times / ~84% of samples

16667 times / ~15% of samples

16667 times / <0.1% of samples

`funclatency-bpfcc` for `is_odd_slow`

man funclatency-bpfcc

“This tool traces function calls and times their duration (latency), and shows the latency distribution as a histogram.”

Determine the function latency

```
$ sudo funclatency-bpfcc $(pg_config --pkglibdir)/pg_slow.so:'is_odd_1'
```

funclatency-bpfcc for is_odd_slow

```
$ sudo funclatency-bpfcc $(pg_config --pkglibdir)/pg_slow.so:'is_odd_1'

Function = [unknown] [181624]
     nsecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 0        |                                        |
       512 -> 1023       : 0        |                                        |
      1024 -> 2047       : 0        |                                        |
      2048 -> 4095       : 0        |                                        |
      4096 -> 8191       : 8        |                                        |
      8192 -> 16383      : 21       |                                        |
     16384 -> 32767      : 62       |                                        |
     32768 -> 65535      : 386      |*                                       |
     65536 -> 131071     : 1315     |******                                  |
    131072 -> 262143     : 2323     |***********                             |
    262144 -> 524287     : 4002     |********************                    |
    524288 -> 1048575    : 7981     |****************************************|
   1048576 -> 2097151    : 539      |**                                      |
   2097152 -> 4194303    : 23       |                                        |
   4194304 -> 8388607    : 6        |                                        |

avg = 551582 nsecs, total: 9192671940 nsecs, count: 16666
```

Differential Flame Graph

Differential Flame Graph

A differential flamegraph compares two execution profiles and visualizes the change in time spent across call stacks, making it easy to spot functions or code paths that became more expensive or cheaper between versions.

Example

```
~/fg/difffolded.pl old.folded new.folded | ~/fg/
flamegraph.pl --negate > diff.svg
```

Differential Flame Graph for `is_odd_slow`

funclatency-bpfcc for is_odd_slow

```
$ sudo funclatency-bpfcc $(pg_config --pkglibdir)/pg_slow.so:'is_odd_2'

Function = [unknown] [194278]
     nsecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 0        |                                        |
       512 -> 1023       : 12159    |****************************************|
      1024 -> 2047       : 1154     |***                                     |
      2048 -> 4095       : 11       |                                        |
      4096 -> 8191       : 4        |                                        |
      8192 -> 16383      : 2        |                                        |
     16384 -> 32767      : 4        |                                        |
     32768 -> 65535      : 0        |                                        |
     65536 -> 131071     : 0        |                                        |
    131072 -> 262143     : 0        |                                        |
    262144 -> 524287     : 3047     |**********                              |
    524288 -> 1048575    : 286      |                                        |

avg = 101829 nsecs, total: 1697188060 nsecs, count: 16667
```

500x slower

What is `bpftrace`?

https://github.com/bpftrace/bpftrace

"bpftrace is a general purpose tracing tool and language for Linux. It leverages eBPF to provide powerful, efficient tracing capabilities with minimal overhead."

Build your own tracing tool!

Probes

`uprobe` and `uretprobe` are user-space probes that are invoked when a function is entered or exited.

bpftrace **for** `is_odd_slow`

`uprobe` - **function enter**

`uretprobe` - **function return**

```
$ sudo bpftrace -e '
uprobe:/usr/local/postgresql-18.3/lib/pg_slow.so:is_odd_1 {
  printf("function enter\n");
}

uretprobe:/usr/local/postgresql-18.3/lib/pg_slow.so:is_odd_1 {
  printf("function exit\n");
}
'

Attaching 2 probes…

function enter
function exit
function enter
function exit
```

bpftrace for `is_odd_slow`

```
uprobe:/usr/local/postgresql-18.3/lib/pg_slow.so:is_odd_2
{
  @start[tid] = nsecs;
  @v[tid] = arg0;
}

uretprobe:/usr/local/postgresql-18.3/lib/pg_slow.so:is_odd_2
/@start[tid]/
{
  $dur = nsecs - @start[tid];
  $v = @v[tid];

  /* Assign input into 10000-width buckets */
  $bucket = ((uint64)$v / 10000) * 10000;

  @count[$bucket] = count();
  @avg_ns[$bucket] = avg($dur);

  delete(@start[tid]);
  delete(@v[tid]);
}

interval:s:5
{
  print(@count);
  print(@avg_ns);
}
```

`uprobe` - take current time

`uretprobe` - subtract current time from enter time

Create buckets of 10000-width and record observation

Show data every 5 seconds

bpftrace for `is_odd_slow`

Number of executions

```
@count[0]: 3333
@count[10000]: 3334
@count[20000]: 3333
@count[30000]: 3333
@count[40000]: 3334

@avg_ns[0]: 1210
@avg_ns[10000]: 889
@avg_ns[20000]: 903
@avg_ns[30000]: 973
@avg_ns[40000]: 529635
```

Average execution time.

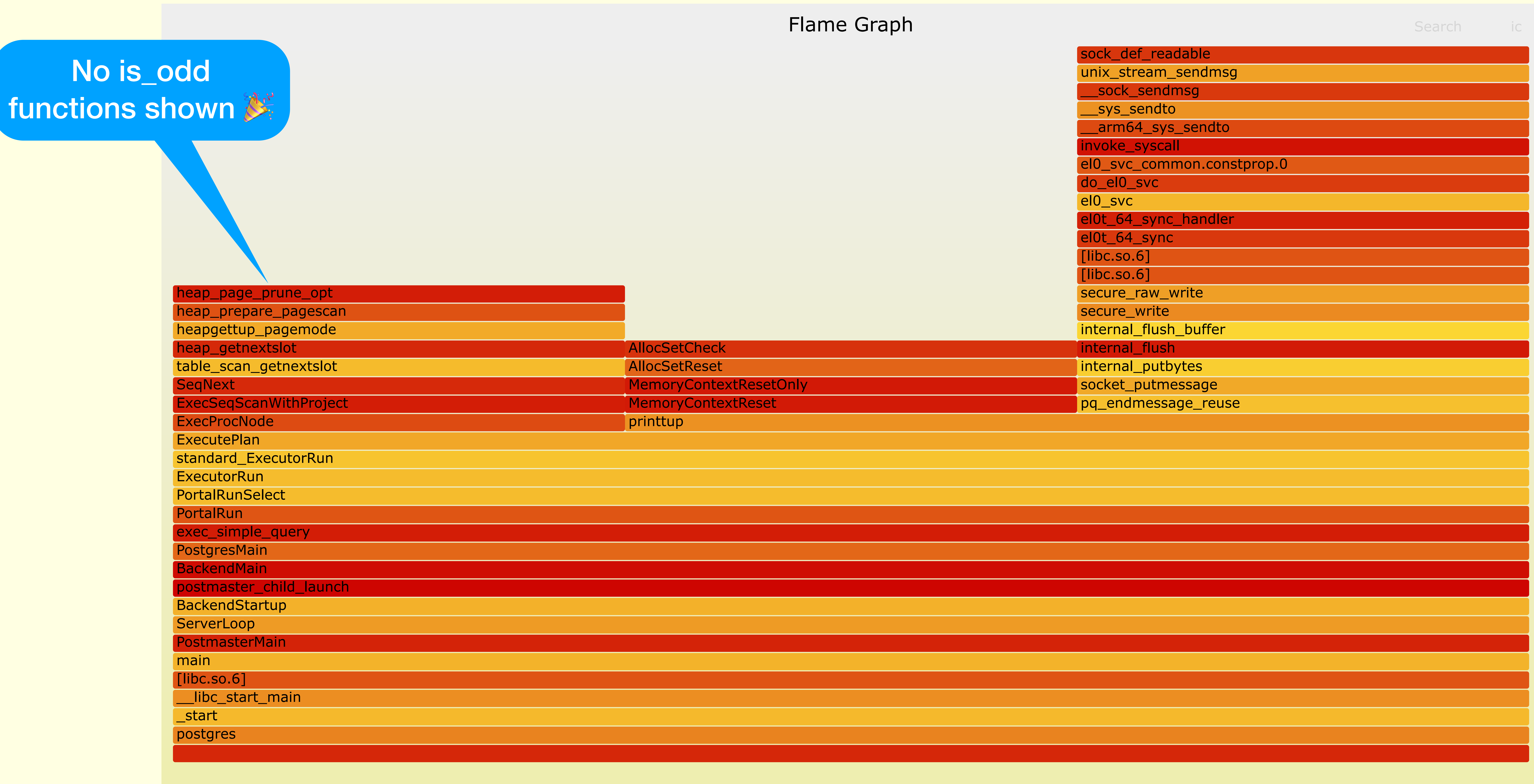
500x slower for values > 40000

pg_slow — `is_odd_2` Implementation

```
bool is_odd_2(int32 val)
{
    if (val > 40000)
    {
        int i;
        volatile int dummy;

        for(i = 0; i < 2000000; i++)
        {
            dummy = (i * i);
            (void) dummy; // Keeps the compiler quiet
        }
    }

    return (val & 1) != 0;
}
```

A different code path is used for values > 40000 (e.g., external sorting)

Flame Graph for `is_odd_slow`



A Very Slow `is_odd` Function

Before optimization

```
db1=# SELECT value, is_odd_slow(value) FROM data;
[...]
 49999 | t
 50000 | f
(50000 rows)
Time: 41194.986 ms (00:41.195)
```

After optimizing the 99% of samples

```
db1=# SELECT value, is_odd_slow(value) FROM data;
 value | is_odd_slow
-------+--------------
[...]
 50000 | f
(50000 rows)
Time: 27694.481 ms (00:27.694)
```

Off-CPU Flame Graph

- **Limitation:** Regular flame graphs show only active **On-CPU** time.
- **The Blind Spot:** They miss time spent waiting, blocked, or waiting for I/O.
- **Off-CPU Analysis:** Visualizes where a program is **waiting** rather than executing.
- **Key Insight:** Identifies **I/O bottlenecks** and **synchronization delays** (locks) rather than just CPU-bound hotspots.

```
# Sample data
sudo offcputime-bpfcc -df -p 1955 > offcpu.stacks

# Generate off-CPU flame graph
~/fg/flamegraph.pl --color=io --title="Off-CPU Flame Graph” < offcpu.stacks
> offcpu.svg
```

See https://www.brendangregg.com/FlameGraphs/offcpuflamegraphs.html as well

Off-CPU Flame Graph for `is_odd_slow`

Off-CPU Flame Graph for `is_odd_slow`

schedule_hrtimeout_range_clock
schedule_hrtimeout_range
do_epoll_wait
do_epoll_pwait.part.0
__arm64_sys_epoll_pwait
invoke_syscall
el0_svc_common.constprop.0
do_el0_svc
el0_svc
el0t_64_sync_handler
el0t_64_sync
-
__internal_syscall_cancel
__syscall_cancel
epoll_pwait
WaitEventSetWaitBlock
WaitEventSetWait
WaitLatch
is_odd_3
is_odd_slow
ExecInterpExpr
ExecEvalExprSwitchContext
ExecProject
ExecScan
ExecSeqScan
ExecProcNodeInstr
ExecProcNode
ExecutePlan
standard_ExecutorRun
ExecutorRun
ExplainOnePlan
standard_ExplainOneQuery
ExplainOneQuery
ExplainQuery

The Optimized `is_odd` Function 🎉

Before optimization

```
db1=# SELECT value, is_odd_slow(value) FROM data;
[...]
 49999 | t
 50000 | f
(50000 rows)
Time: 41194.986 ms (00:41.195)
```

After optimization

```
db1=# SELECT value, is_odd_slow(value) FROM data;
 value | is_odd_slow
-------+--------------
[...]
 50000 | f
(50000 rows)
Time: 16.268 ms
```

Conclusion

We employed multiple tools to analyze performance:

- `EXPLAIN` → plan shape and row estimates.
- `perf` → hot code paths by CPU time.
- `flame graph` → visual call-stack hotspot analysis.
- `funccount / funclatency` → call tracing and latency distribution using eBPF.
- `bpftrace` → general-purpose eBPF tracing tool.

Thank You

- Email: jan@thenile.dev / jnidzwetzki@gmx.de
- Blog: https://jnidzwetzki.github.io/
- GitHub: https://github.com/jnidzwetzki/
- LinkedIn: https://www.linkedin.com/in/jnidzwetzki/

Jan's Blog:

Thank You

Supplemental material / pg_slow source code:

https://github.com/jnidzwetzki/pg-conf-ger-2026