Trace-processing software application-specific performance optimization: the case of Babeltrace 2

Tracing summit 2018 (25 October 2018)

Philippe Proulx <pproulx@efficios.com>
eepp on GitHub/IRC/SO

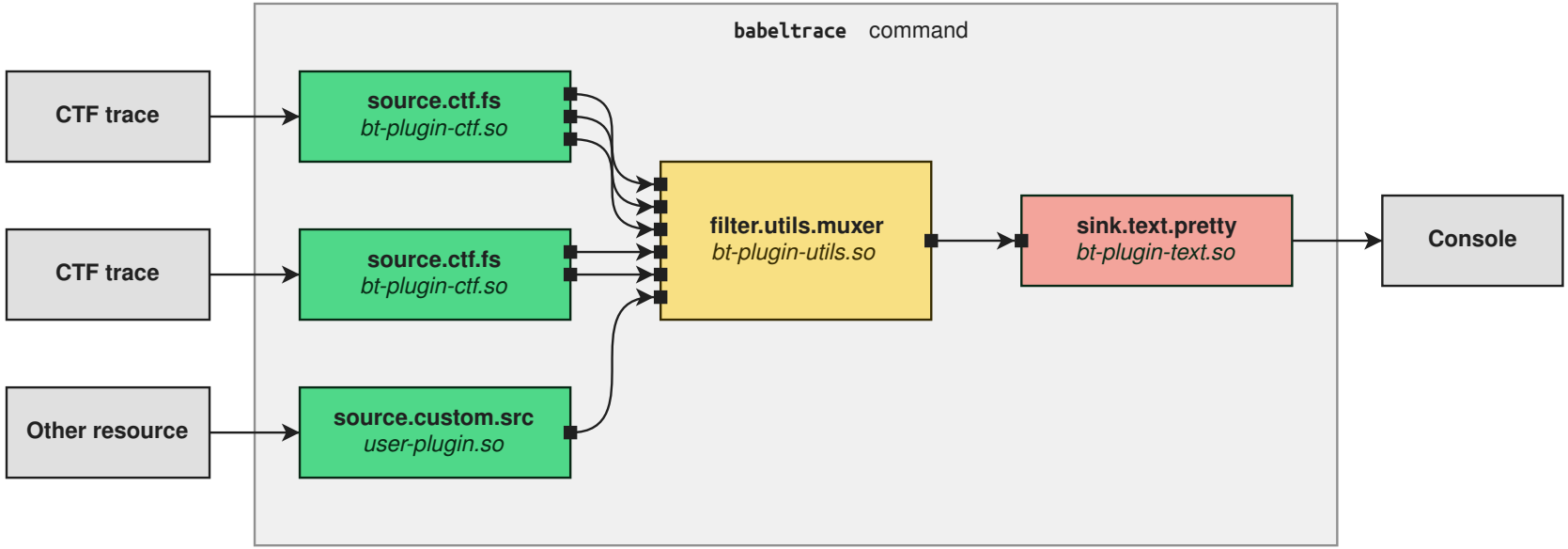
Contents

What is Babeltrace 2?

High level overview:

- Plugin-based trace converter and processor library/CLI.
- A plugin is a shared object (C/C++) or a Python script.
- Stable C and Python APIs.
- Cross-platform (Linux, macOS, Windows).

What is Babeltrace 2?



See also: ***Babeltrace 2: Tailor-made trace analyses*** by Jérémie Galarneau (Tracing Summit 2017)
<https://tracingsummit.org/w/images/2/23/TS17-bt2.pdf>

What is Babeltrace 2?

Relevant project's initial design goals:

- API must be **uniform**: all objects are shared thanks to reference counting.
- API must be "**bulletproof**": functions fail with invalid arguments, not altering any object, and return an error code; functions validate all arguments (deeply).

Problem

Once Babeltrace 2 was feature complete, we compared it to Babeltrace 1: **it was ~13x slower!**

Current users expect at least the same performance from a major update.

Goal

From this point, before releasing Babeltrace 2, so as to be able to modify the library's API, we entered an optimization phase.

New goal: reach Babeltrace 1's performance.

Methodology: measure

```
$ perf record -F100 --call-graph=dwarf babeltrace -o dummy /path/to/trace
$ perf report --no-children
```

```
Samples: 4K of event 'cycles:uppp', Event count (approx.): 165247363102
  Overhead  Command      Shared Object                Symbol
+    7.45%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_put
+    5.31%  babeltrace   libc-2.27.so                 [.] __libc_calloc
+    5.22%  babeltrace   libc-2.27.so                 [.] _int_malloc
+    4.98%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_get
+    4.81%  babeltrace   libc-2.27.so                 [.] _int_free
+    4.49%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_hash_table_lookup
+    3.92%  babeltrace   babeltrace-plugin-ctf.so     [.] bt_btr_start
+    2.77%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_field_type_get_type_id
+    1.93%  babeltrace   babeltrace-plugin-utils.so   [.] muxer_notif_iter_next
+    1.87%  babeltrace   libc-2.27.so                 [.] cfree@GLIBC_2.2.5
+    1.80%  babeltrace   babeltrace-plugin-ctf.so     [.] read_basic_int_and_call_cb
+    1.76%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_field_create
+    1.61%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_hash_table_lookup_extended
     1.33%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_notification_iterator_next
+    1.27%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_slice_alloc
+    1.20%  babeltrace   babeltrace-plugin-ctf.so     [.] get_next_field
+    1.05%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_get@plt
+    1.03%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_field_structure_freeze
     1.01%  babeltrace   babeltrace-plugin-ctf.so     [.] bt_notif_iter_get_next_notification
+    1.01%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_ptr_array_foreach
+    0.99%  babeltrace   babeltrace-plugin-ctf.so     [.] btr_unsigned_int_common
+    0.92%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_slice_free1
+    0.88%  babeltrace   babeltrace-plugin-ctf.so     [.] btr_unsigned_int_cb
+    0.86%  babeltrace   libc-2.27.so                 [.] malloc
```

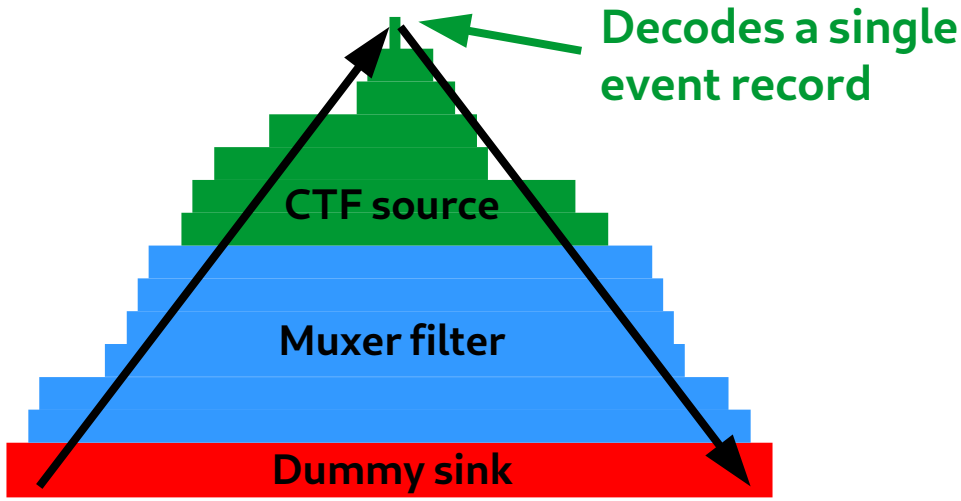
Reference count change

Allocation/ deallocation

Methodology: measure

```
$ gdb --args babeltrace -o dummy /path/to/trace
```

```
#0  bt_btr_start
#1  read_dscope_begin_state
#2  read_event_header_begin_state
#3  handle_state
#4  bt_notif_iter_get_next_notification
#5  ctf_fs_ds_file_next
#6  ctf_fs_iterator_next
#7  ensure_queue_has_notifications
#8  bt_notification_iterator_next
#9  muxer_upstream_notif_iter_next
#10 validate_muxer_upstream_notif_iter
#11 validate_muxer_upstream_notif_iters
#12 muxer_notif_iter_do_next
#13 muxer_notif_iter_next
#14 ensure_queue_has_notifications
#15 bt_notification_iterator_next
#16 dummy_consume
#17 bt_component_sink_consume
#18 consume_graph_sink
#19 consume_sink_node
#20 bt_graph_consume_no_check
#21 bt_graph_run
```



Call stack: up and down for each event record.

Methodology: hypothesis #1

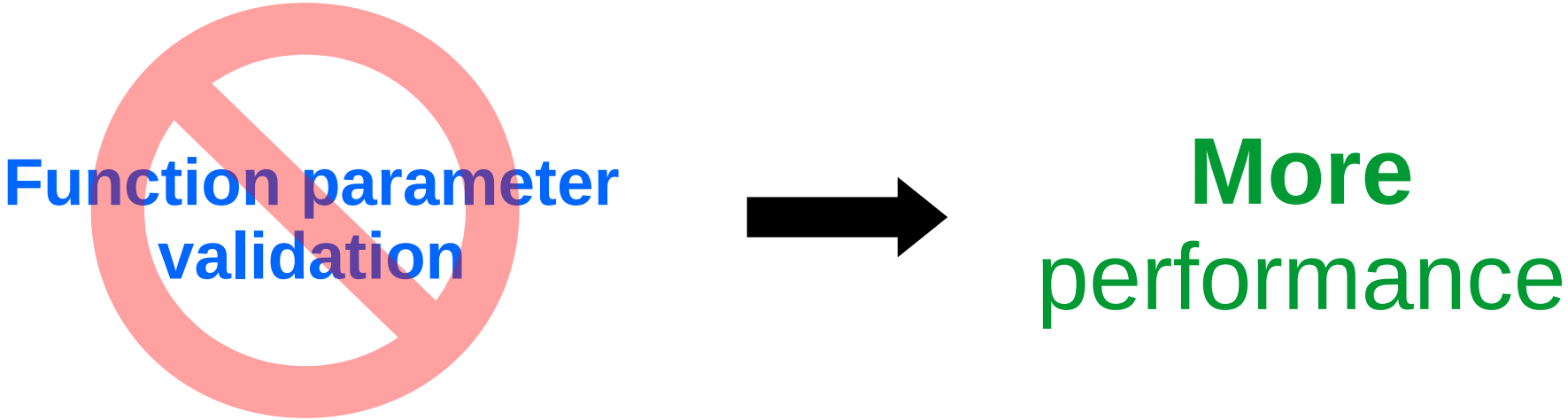
Given that reference counting and memory allocations/deallocations look very costly:

Less reference counting
Less memory alloc./dealloc.

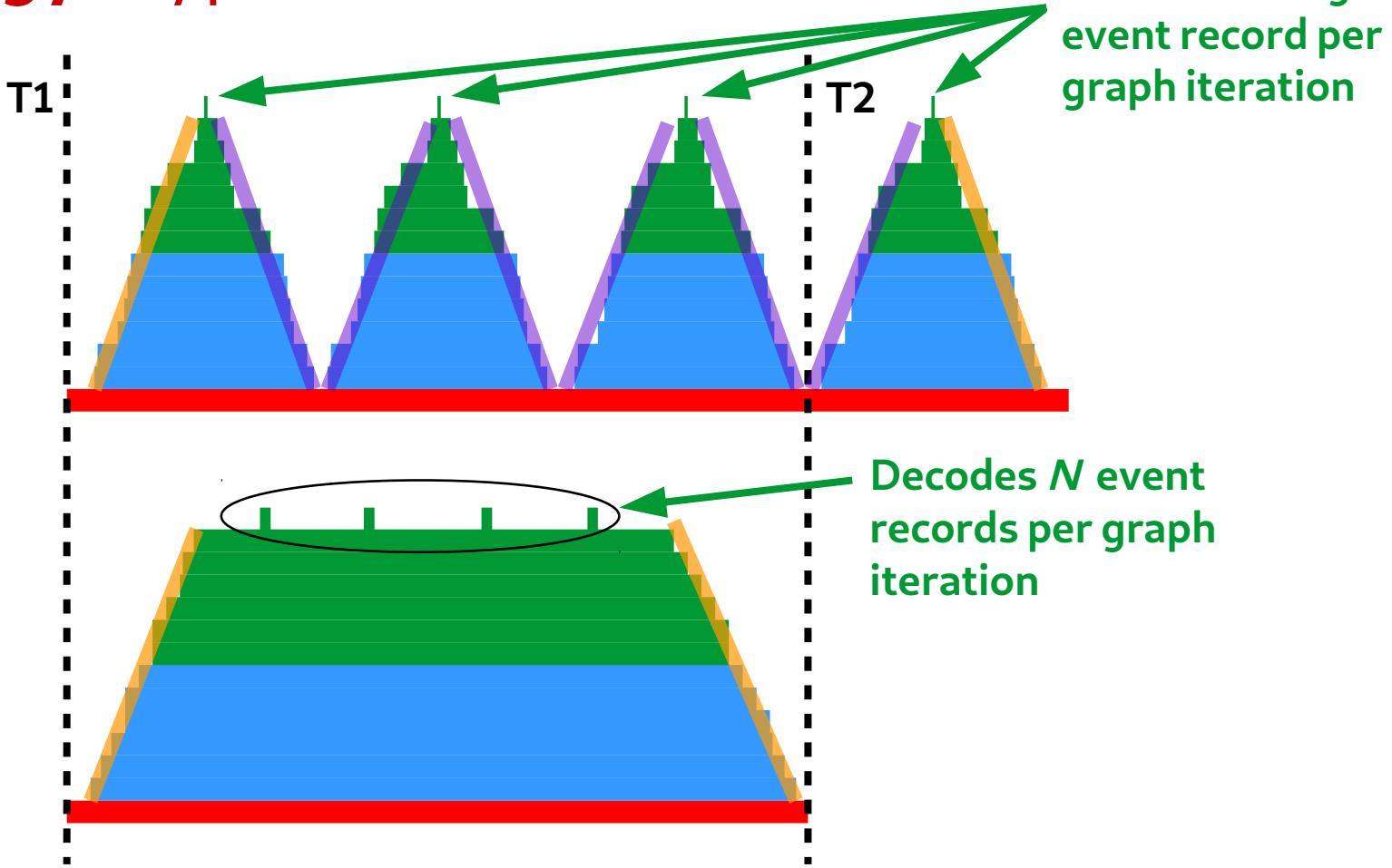
More performance

Methodology: hypothesis #2

Given that there is a lot of function parameter validation on the fast path (field, event, packet, and graph APIs):



Methodology: hypothesis #3



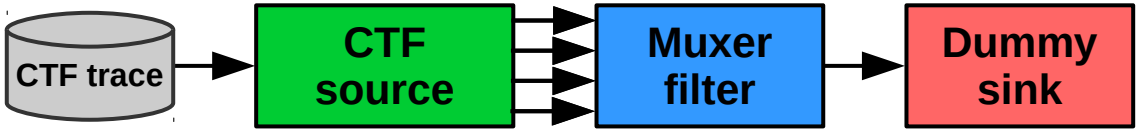
Methodology: hypothesis #3

Given that a single graph iteration involves many function calls:

More event records per graph iteration → **More** performance

Methodology: performance measurement

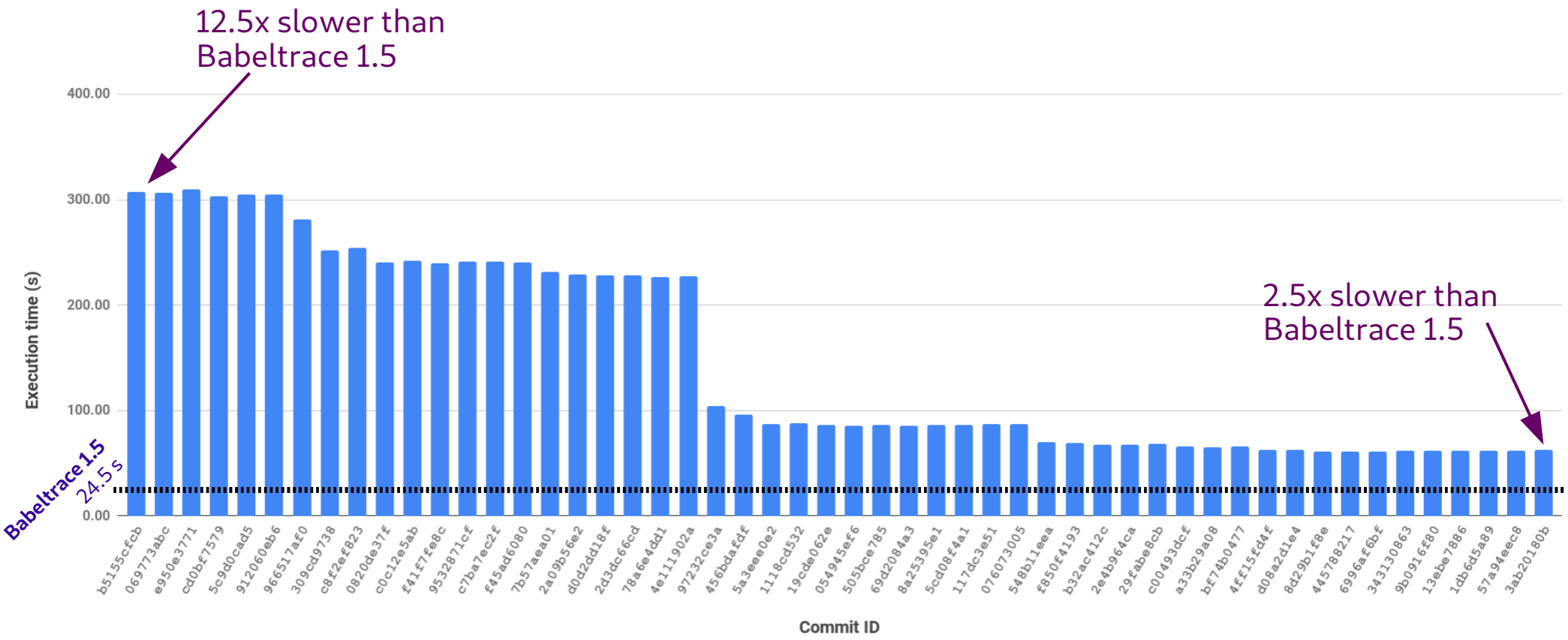
- **Environment**:
  - Intel(R) Core(TM) i7-3520M CPU @ 2.90GHz
  - Linux 4.17.5
  - Partly cloudy, 13 °C, 87 % humidity
- **Build**:
  - `CFLAGS='-O3 -DNDEBUG'`
  - Debug/developer mode and logging disabled
- **Execution**:
  - `# echo 3 > /proc/sys/vm/drop_caches`
  - `$ /usr/bin/time babeltrace /path/to/trace -o dummy`
  - Trace: LTTng 2.10 kernel trace, 1.4 GiB, 4 data streams, 187 packets
    Tracing all tracepoints/system calls for 5m37s on this system



Consumes and discards event records.

Baseline: Babeltrace 1.5 (`4f10a4ad`), same parameters.

Results: summary



Conditional precondition check

A lot of fast path parameter validation, for example:

- NULL check
- Array bound check
- Value range check
- ...

Conditional precondition check

Different approaches:

- **System calls**: cannot (should not) cause kernel oops.
  This was our initial approach.
- `strlen(NULL)`: segmentation fault.

Can we find a compromise?

Conditional precondition check

Add precondition check macros:

- `BT_ASSERT_PRE(condition, fmt, ...)`
- `BT_ASSERT_PRE_NON_NULL(obj, obj_name)`
- `BT_ASSERT_PRE_VALID_INDEX(index, length)`

No-op in production build mode.

Conditional precondition check

Examples:

```
BT_ASSERT_PRE(size > 0,
              "Size is greater than zero: size=%u", size);

BT_ASSERT_PRE_NON_NULL(field_name, "Field name");
```

Conditional precondition check: before and after

```c
int bt_field_sequence_set_length(struct bt_field *field,
                 struct bt_field *length_field)
{
        int ret = 0;
        struct bt_field_type_integer *length_type;
        struct bt_field_integer *length;
        struct bt_field_sequence *sequence;
        uint64_t sequence_length;

        if (!field) {
                BT_LOGW_STR("Invalid parameter: field is NULL.");
                ret = -1;
                goto end;
        }

        if (!length_field) {
                BT_LOGW_STR("Invalid parameter: length field is NULL.");
                ret = -1;
                goto end;
        }

        if (field->frozen) {
                BT_LOGW("Invalid parameter: field is frozen: addr=%p",
                        field);
                ret = -1;
                goto end;
        }

        if (bt_field_type_get_type_id(length_field->type) !=
                        BT_FIELD_TYPE_ID_INTEGER) {
                BT_LOGW("Invalid parameter: length field's type is not an integer field type: "
                        "field-addr=%p, length-field-addr=%p, length-ft-addr=%p, length-ft-id=%s",
                        field, length_field, length_field->type,
                        bt_field_type_id_string(length_field->type->id));
                ret = -1;
                goto end;
        }

        length_type = container_of(length_field->type,
                struct bt_field_type_integer, parent);
        /* The length field must be unsigned */
        if (length_type->is_signed) {
                BT_LOGW("Invalid parameter: length field's type is signed: "
                        "field-addr=%p, length-field-addr=%p, "
                        "length-field-ft-addr=%p", field, length_field,
                        length_field->type);
                ret = -1;
                goto end;
        }
```

NULL check

NULL check

"Frozen" state check

Type check

Signedness check

```c
int bt_field_sequence_set_length(struct bt_field *field,
                 struct bt_field *length_field)
{
        int ret = 0;
        struct bt_field_integer *length;
        struct bt_field_sequence *sequence;
        uint64_t sequence_length;

        BT_ASSERT_PRE_NON_NULL(field, "Sequence field");
        BT_ASSERT_PRE_NON_NULL(length_field, "Length field");
        BT_ASSERT_PRE_FIELD_HOT(field, "Sequence field");
        BT_ASSERT_PRE_FIELD_HAS_TYPE_ID(length_field, BT_FIELD_TYPE_ID_INTEGER,
                "Length field");
        BT_ASSERT_PRE(!bt_field_type_integer_is_signed(length_field->type),
                "Length field's type is signed: %!+f", length_field);
        BT_ASSERT_PRE_FIELD_IS_SET(length_field, "Length field");
```

Conditional precondition check

Example: calling `bt_field_integer_signed_get_value()` with an unsigned field object.

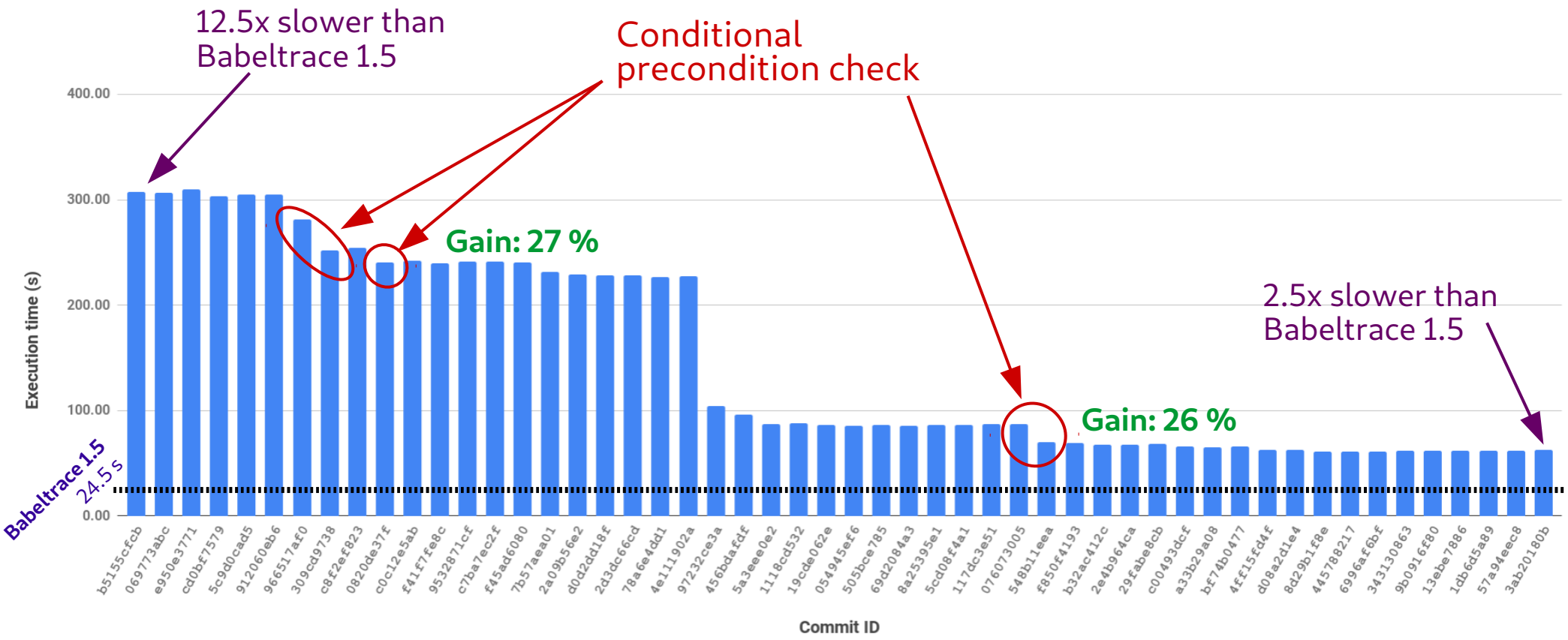
```
10-24 17:35:48.597 15796 15796 F FIELDS Library
precondition not satisfied:

10-24 17:35:48.597 15796 15796 F FIELDS @fields.c:594
Field's type is unsigned: addr=0x560ee6dc7ab0,
is-set=1, is-frozen=1, type-addr=0x560ee6e582d0,
type-id=BT_FIELD_TYPE_ID_INTEGER, value=2

10-24 17:35:48.597 15796 15796 F FIELDS Aborting...
```

Conditional precondition check

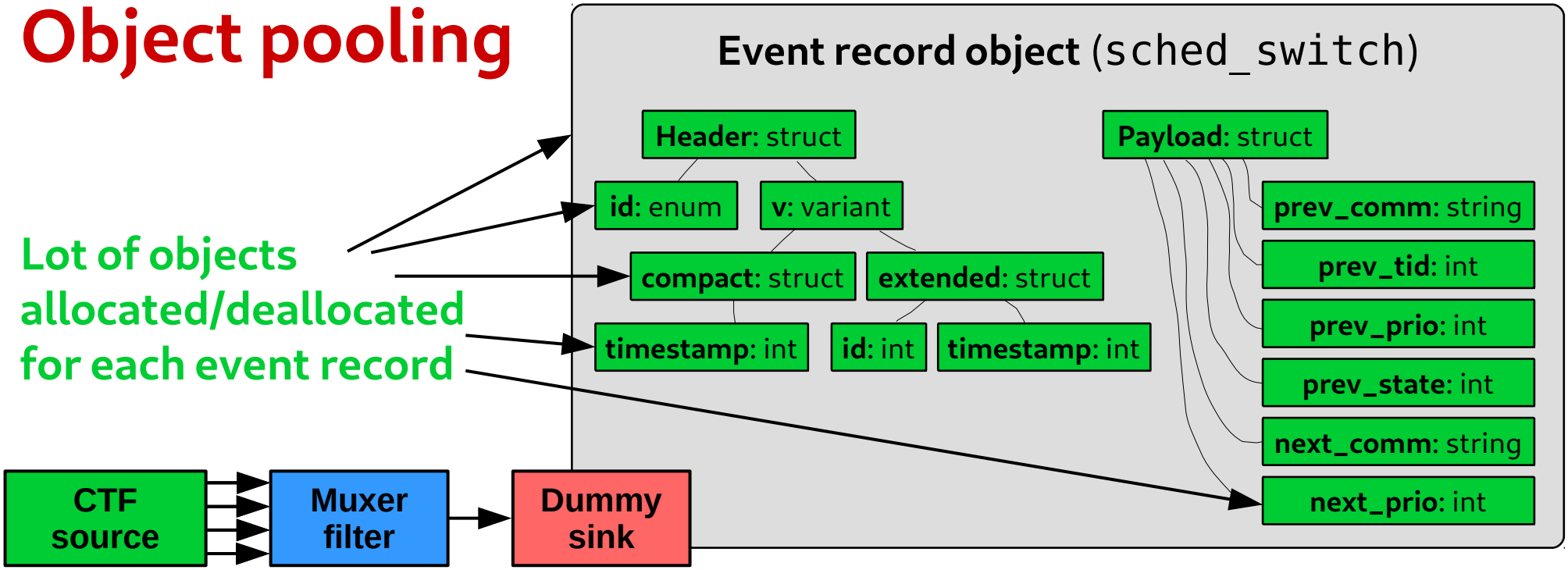
When a precondition is not satisfied:

```
abort();
```

Conditional precondition check: results



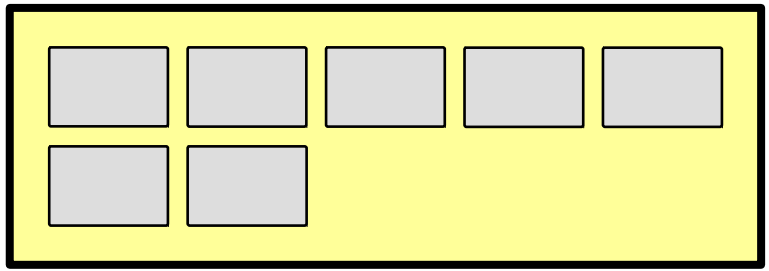
Object pooling



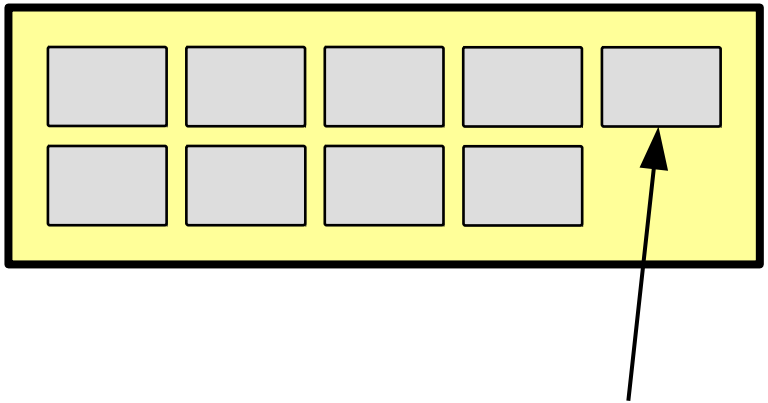

```
sched_switch: { prev_comm = "lttng-sessiond", prev_tid = 11738,
prev_prio = 20, prev_state = 4096, next_comm = "lttng-consumerd",
next_tid = 11853, next_prio = 20 }
```

Object pooling

`kmem_free` event record class's event record pool

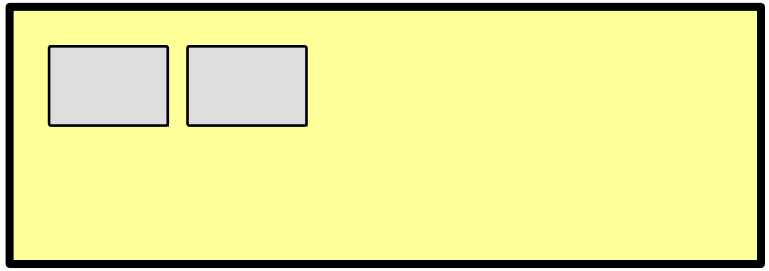


`sched_switch` event record class's event record pool

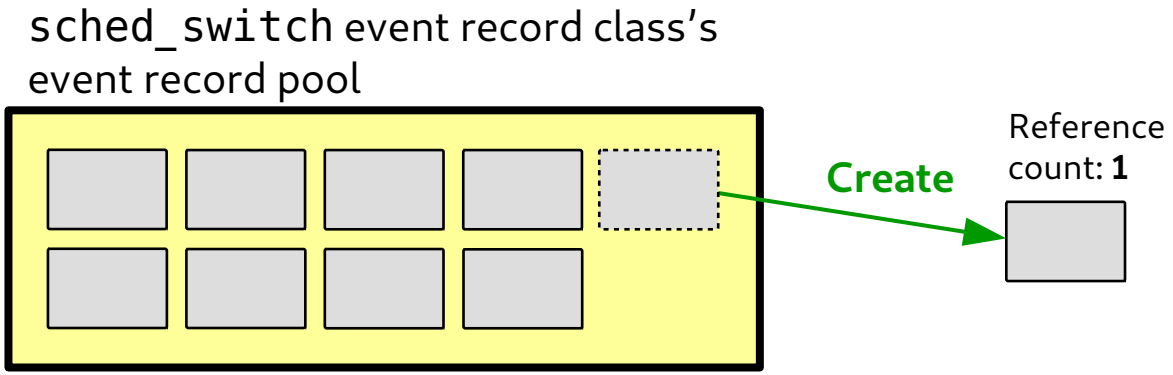


Free `sched_switch` event record (reference count: 0)

`timer_start` event record class's event record pool

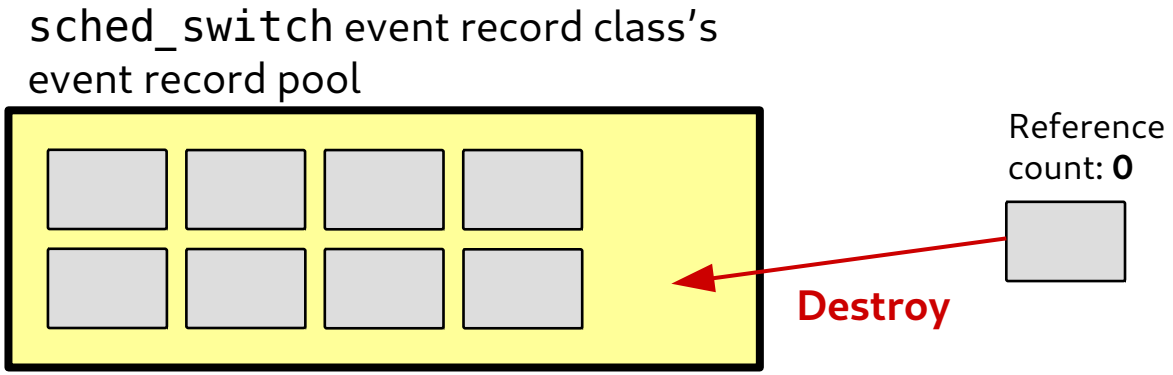


Object pooling



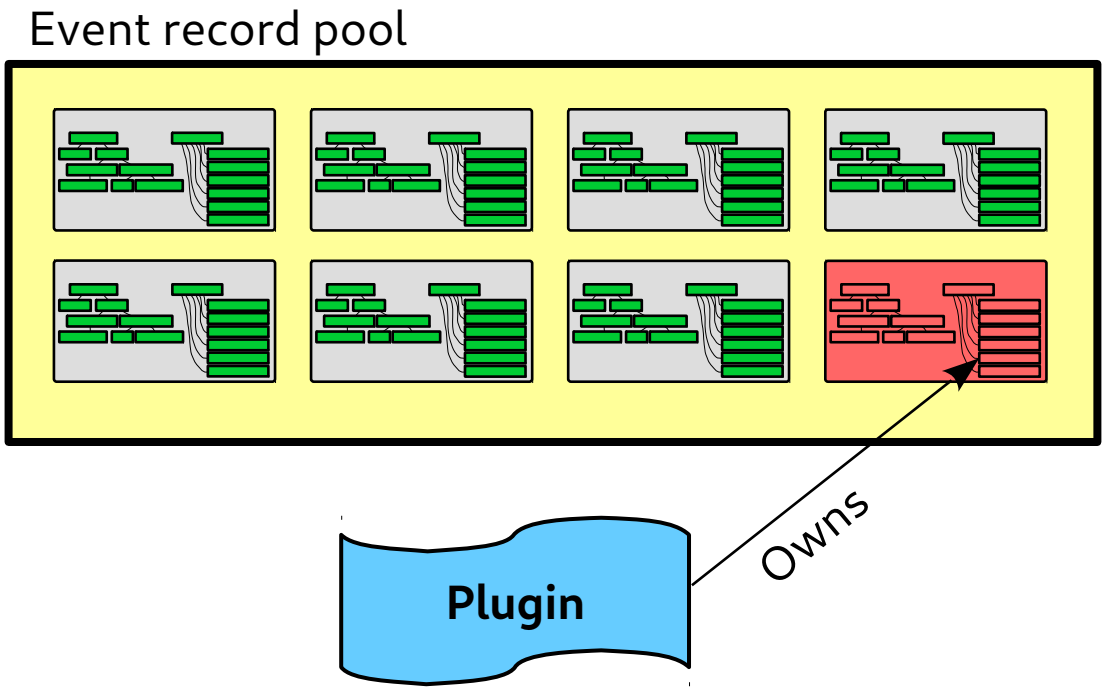
Free event record object becomes owned (out of the pool) on creation.

Object pooling



Owned event record object becomes free (back to the pool) on destruction.

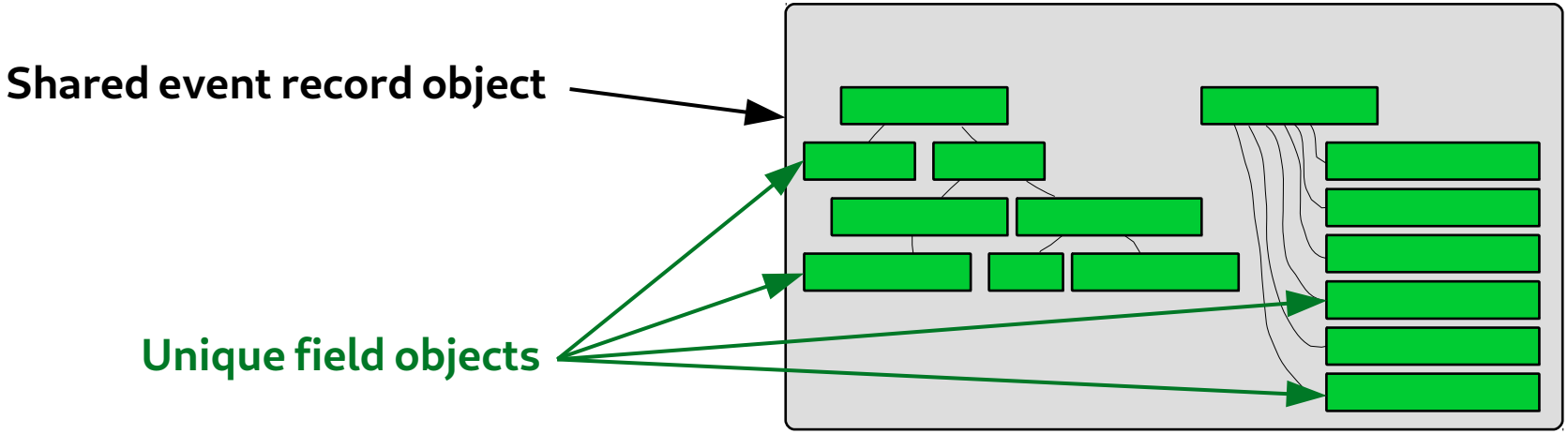
Object pooling



Bad scenario:

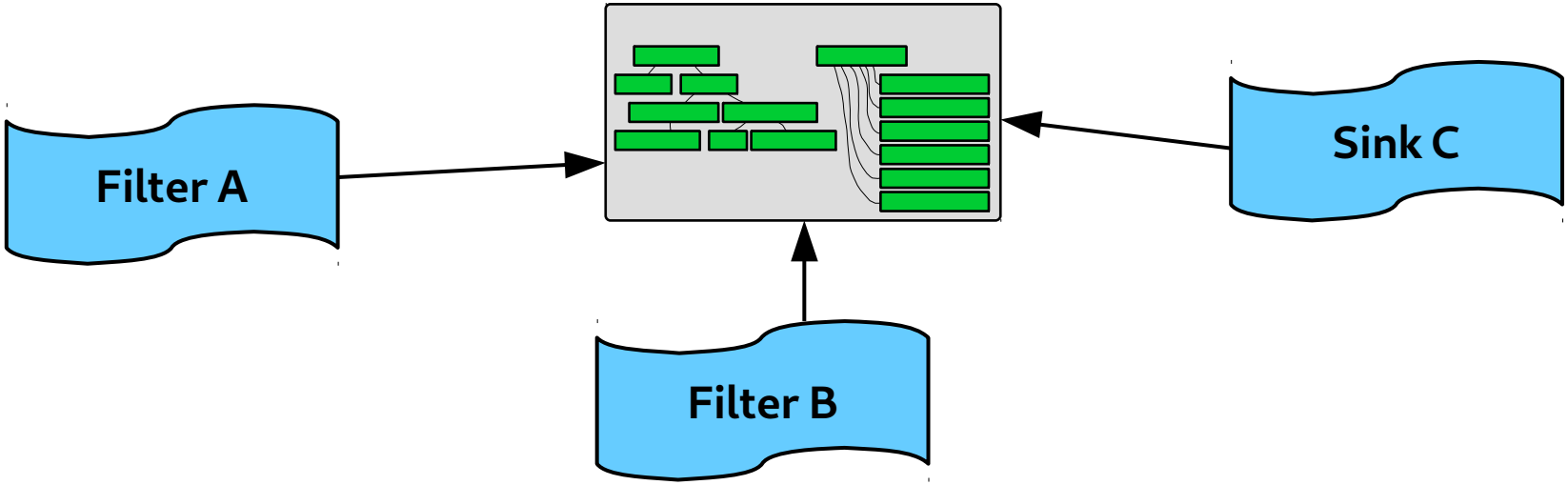
Pooled event record object is not really “free” because one of its fields is externally owned.

Object pooling



Object pooling

Event record objects are still shared for filters/sinks to keep them without copying.



Careful reference counting

Before:

```
void func(struct bt_stream_class *sc)
{
    // bt_stream_class_get_trace() calls bt_get() internally
    struct bt_trace *trace = bt_stream_class_get_trace(sc);

    // ...

    bt_put(trace);
}
```

After:

```
void func(struct bt_stream_class *sc)
{
    struct bt_trace *trace = bt_stream_class_borrow_trace(sc);

    // ...
}
```

Object pooling & less ref. count: before and after

```
Samples: 21K of event 'cycles:uppp', Event count (approx.): 770085577824
  Overhead  Command      Shared Object                Symbol
+    8.28%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_put
+    6.79%  babeltrace   libc-2.27.so                 [.] _int_free
+    6.44%  babeltrace   libc-2.27.so                 [.] __libc_calloc
+    6.34%  babeltrace   libc-2.27.so                 [.] _int_malloc
+    5.16%  babeltrace   babeltrace-plugin-ctf.so     [.] bt_btr_start
+    4.81%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_get
+    4.25%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_hash_table_lookup
+    2.74%  babeltrace   babeltrace-plugin-ctf.so     [.] read_basic_int_and_call_cb
+    2.68%  babeltrace   libc-2.27.so                 [.] cfree@GLIBC_2.2.5
+    2.22%  babeltrace   babeltrace-plugin-utils.so   [.] muxer_notif_iter_next
+    2.20%  babeltrace   babeltrace-plugin-ctf.so     [.] get_next_field
+    1.60%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_hash_table_lookup_extended
+    1.53%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_slice_alloc
```

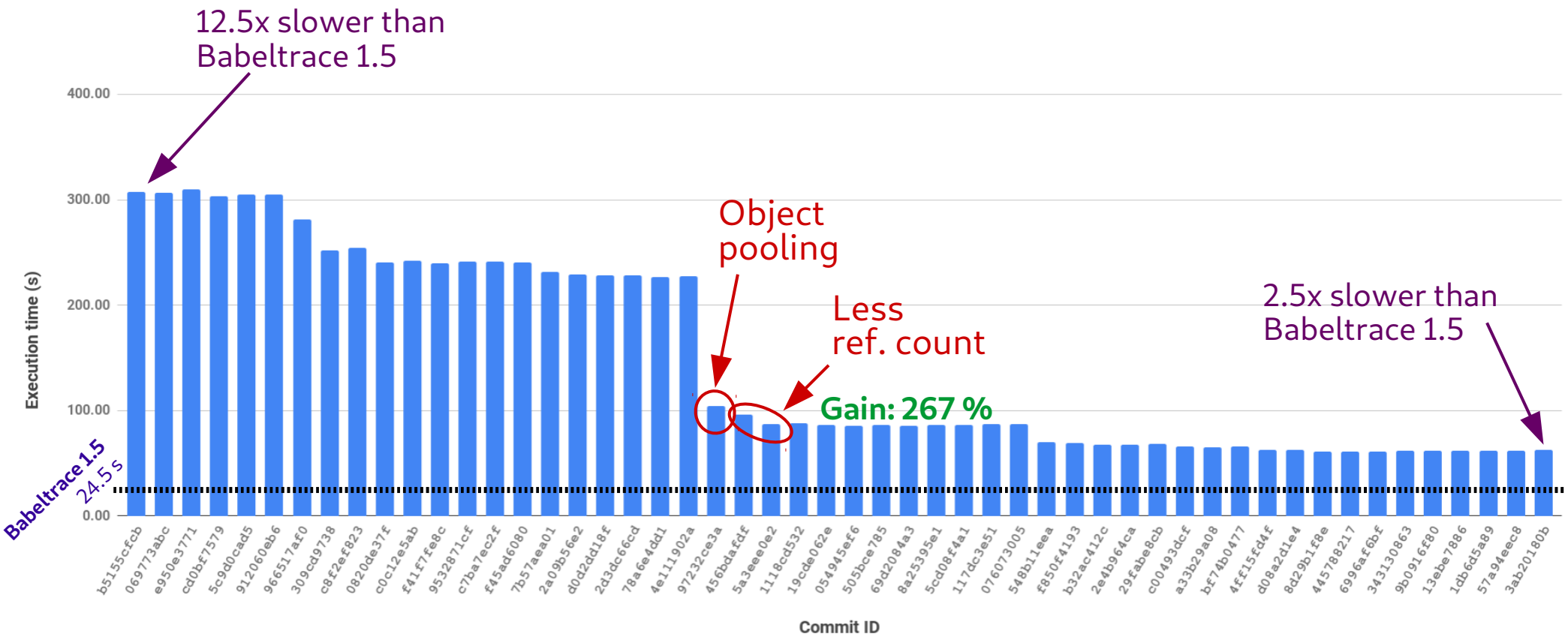
```
Samples: 8K of event 'cycles:uppp', Event count (approx.): 294764165876
  Overhead  Command      Shared Object                Symbol
+   11.69%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_hash_table_lookup
+   10.76%  babeltrace   babeltrace-plugin-ctf.so     [.] bt_btr_start
+    6.05%  babeltrace   babeltrace-plugin-ctf.so     [.] read_basic_int_and_call_cb
+    5.40%  babeltrace   babeltrace-plugin-utils.so   [.] muxer_notif_iter_next
+    3.51%  babeltrace   babeltrace-plugin-ctf.so     [.] borrow_next_field
+    3.49%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_hash_table_lookup_extended
+    2.98%  babeltrace   babeltrace-plugin-ctf.so     [.] btr_unsigned_int_cb
+    2.89%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_field_type_get_alignment
+    2.44%  babeltrace   babeltrace-plugin-ctf.so     [.] update_clock.isra.13
+    2.17%  babeltrace   libbabeltrace.so.2.0.0       [.] bt_field_type_common_get_alignment.part.23
+    2.16%  babeltrace   libglib-2.0.so.0.5600.1      [.] g_str_hash
```

Reference count change

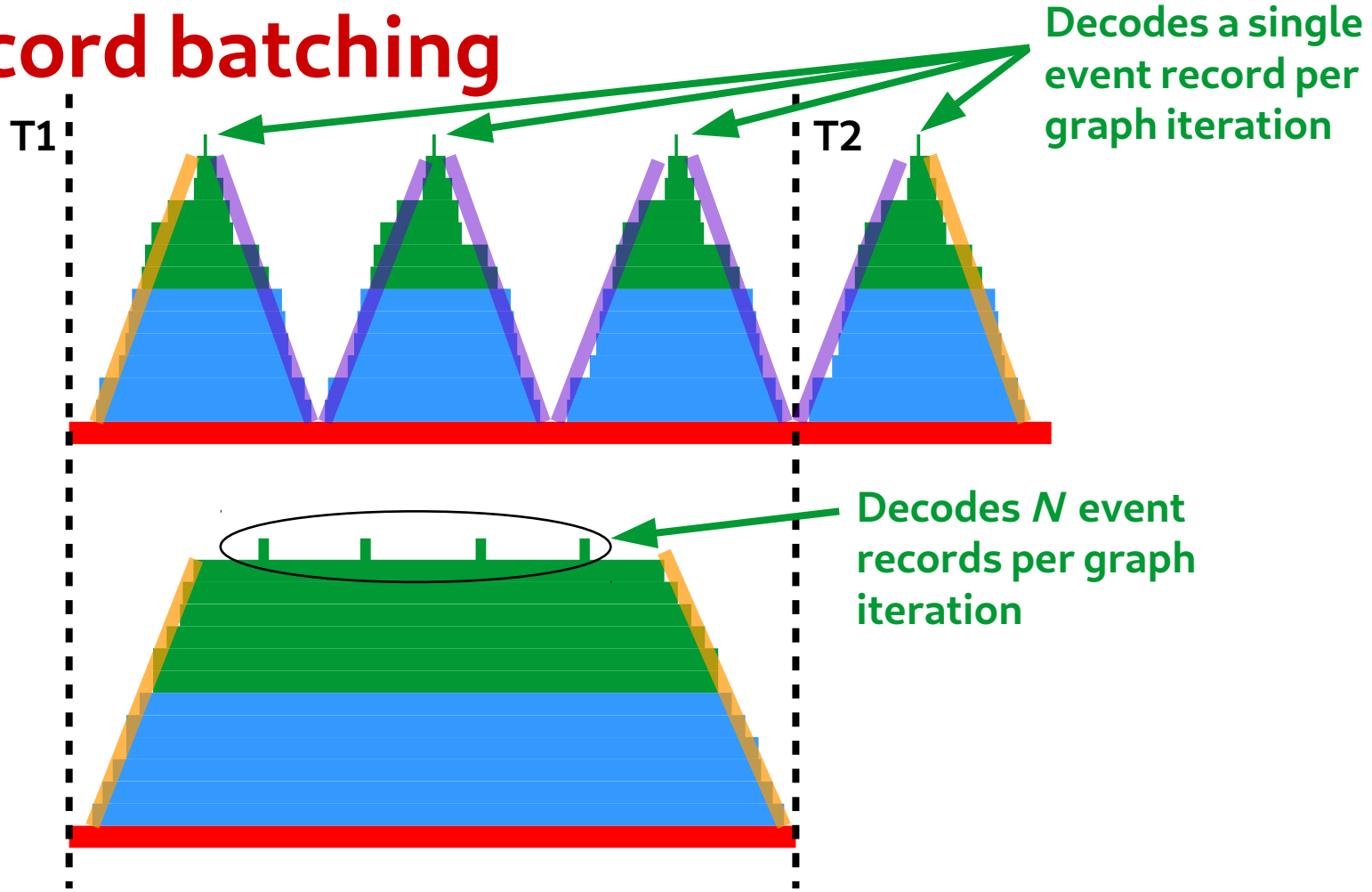
Allocation/ deallocation

CTF source functions

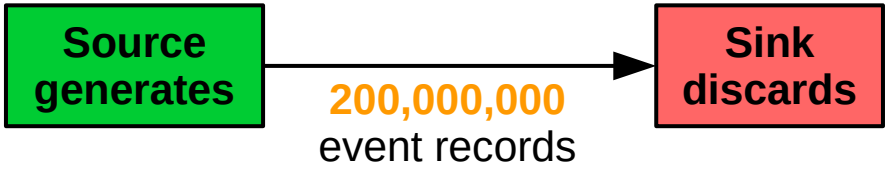
Object pooling & less ref. count: results



Event record batching



Event record batching



Testing configuration:

```
$ babeltrace run -c fake:src.utils.fake -p count=200000000 \
                 -c dummy:sink.utils.dummy -C fake:dummy
```

This tests the graph operations without doing anything useful:

- No file system access.
- No console output.
- No event record muxer.

Event record batching

```
$ perf stat -B -e ...
```

Batch size (event records)	Execution time (s)	Instructions (billion instructions)	LL cache references (thousand references)
1	16.61	106	535
2	13.93	88	520
5	12.50	77	792
10	12.10	73	655
20	11.70	71	1,251
30	**11.57**	**71**	**1,120**
50	12.25	70	4,393
100	12.12	70	87,400
200	12.64	69	508,467
400	14.24	69	1,965,072
500	15.11	69	2,160,902

Event record batching: results

Saves <u>constant</u> ~25 ms/million event records.

Is it beneficial?

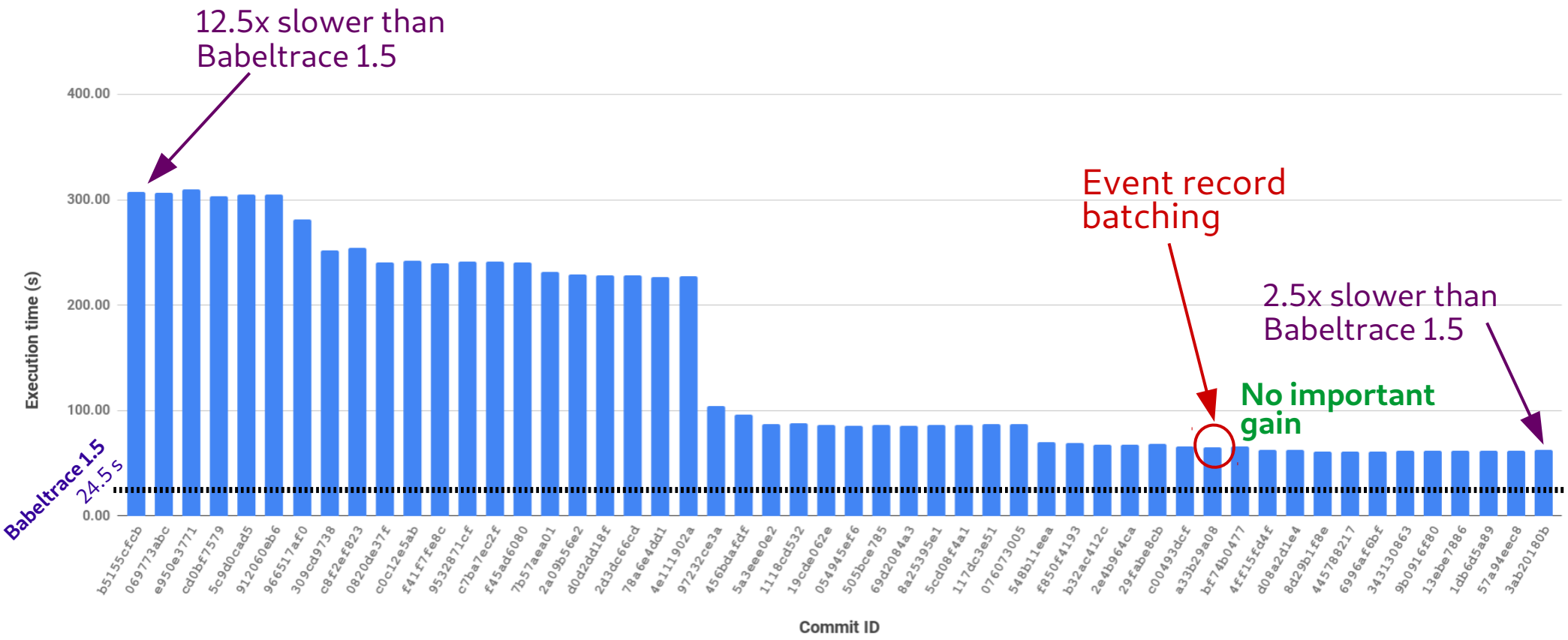
Initial run time (s)	**Gain** (%)
62.4	1.7
24.5	4.3

(1.4 GiB trace: ~42 million event records)

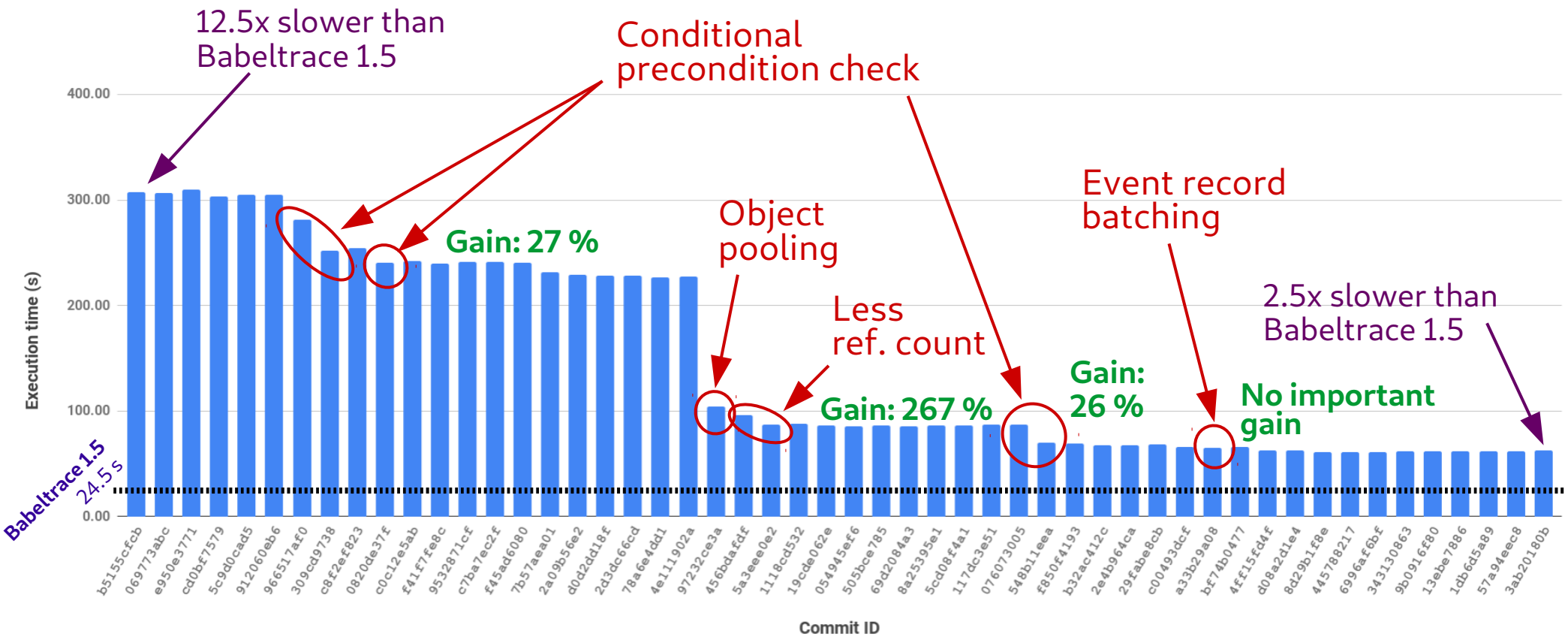
Currently: not so much.

Target: maybe?

Event record batching: results



Results


12.5x slower than Babeltrace 1.5
Conditional precondition check
Gain: 27 %
Object pooling
Less ref. count
Gain: 267 %
Event record batching
Gain: 26 %
No important gain
2.5x slower than Babeltrace 1.5
Babeltrace 1.5 24.5 s
Execution time (s)
400.00
300.00
200.00
100.00
0.00
b5155cfcb
069773abc
e950e3771
cd0bf7579
5c9d0cad5
912060eb6
966517af0
309cd9738
c8f2ef823
0820de37f
c0c12e5ab
f41f7fe8c
9532871cf
c7ba7ec2f
f45ad6080
7b57aea01
2a09b56e2
d0d2dd18f
2d3dc66cd
78a6e4dd1
4e111902a
97232ce3a
456bdafdf
5a3eee0e2
1118cd532
19cde062e
054945ef6
505bce785
69d2084a3
8a25395e1
5cd08f4a1
117dc3e51
076073005
548b11eea
f850f4193
b32ac412c
2e4b964ca
29fabe8cb
c00493dcf
a33b29a08
bf74b0477
4ff15fd4f
d08a2d1e4
8d29b1f8e
445788217
6996af6bf
343130863
9b0916f80
13ebe7886
1db6d5a89
57a94eec8
3ab20180b
Commit ID

What's next?

```
Samples: 8K of event 'cycles:uppp', Event count (approx.): 294764165876
  Overhead  Command      Shared Object               Symbol
+   11.69%  babeltrace   libglib-2.0.so.0.5600.1     [.] g_hash_table_lookup
+   10.76%  babeltrace   babeltrace-plugin-ctf.so    [.] bt_btr_start
+    6.05%  babeltrace   babeltrace-plugin-ctf.so    [.] read_basic_int_and_call_cb
+    5.40%  babeltrace   babeltrace-plugin-utils.so  [.] muxer_notif_iter_next
+    3.51%  babeltrace   babeltrace-plugin-ctf.so    [.] borrow_next_field
+    3.49%  babeltrace   libglib-2.0.so.0.5600.1     [.] g_hash_table_lookup_extended
+    2.98%  babeltrace   babeltrace-plugin-ctf.so    [.] btr_unsigned_int_cb
+    2.89%  babeltrace   libbabeltrace.so.2.0.0      [.] bt_field_type_get_alignment
+    2.44%  babeltrace   babeltrace-plugin-ctf.so    [.] update_clock.isra.13
+    2.17%  babeltrace   libbabeltrace.so.2.0.0      [.] bt_field_type_common_get_alignment.part.23
+    2.16%  babeltrace   libglib-2.0.so.0.5600.1     [.] g_str_hash
```

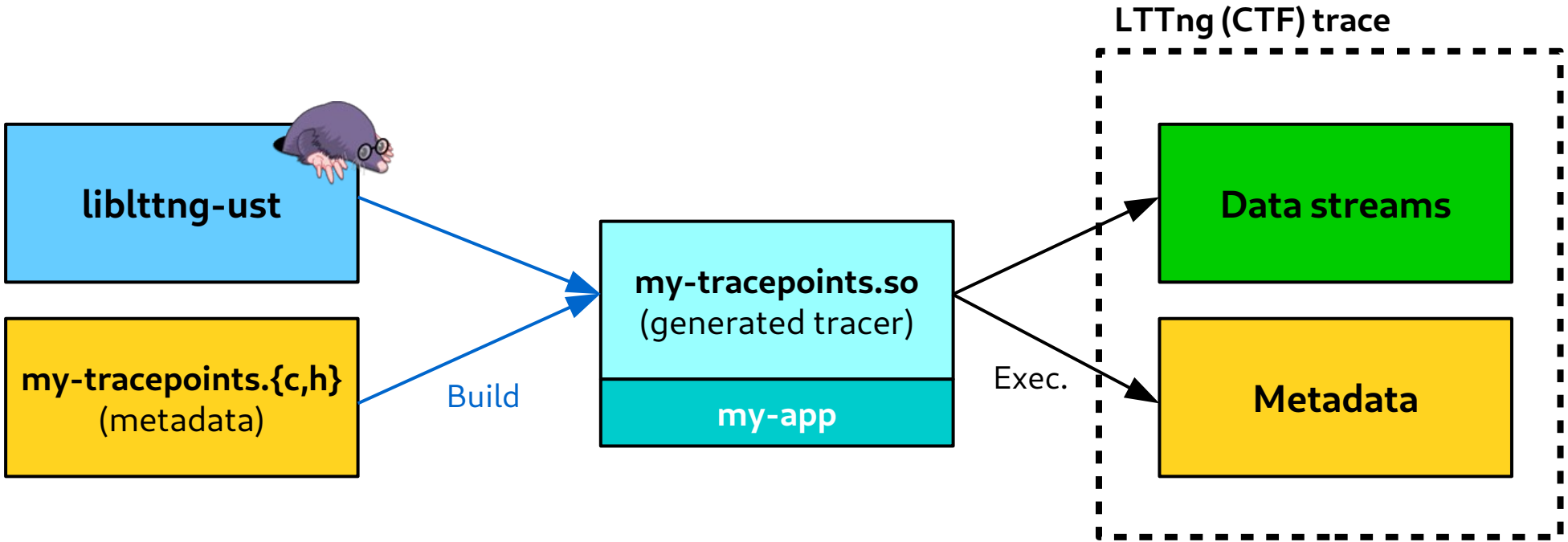
CTF source functions

Muxer filter function

- Focus on CTF source's optimization.
- Focus on muxer filter's optimization.
- Does not change the C API:
  can be done after 2.0 release.

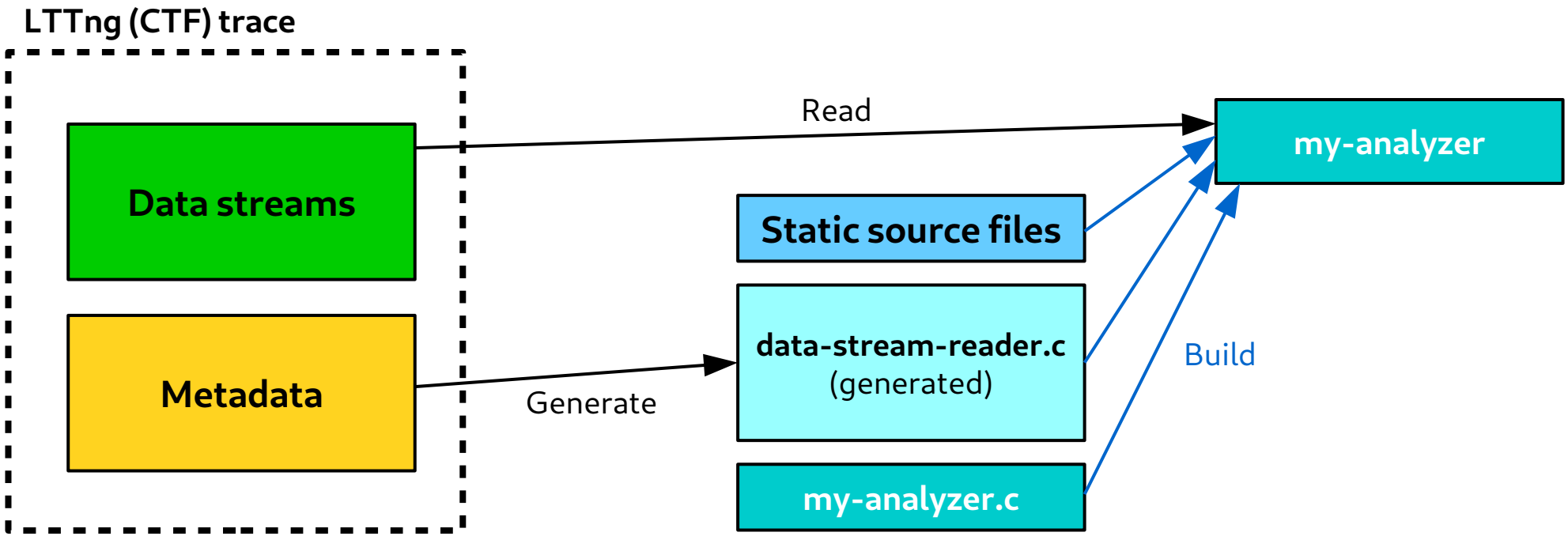
Future work ideas

LTTng-UST and LTTng-modules are tracer generators.



Future work ideas

Why not go the opposite way?

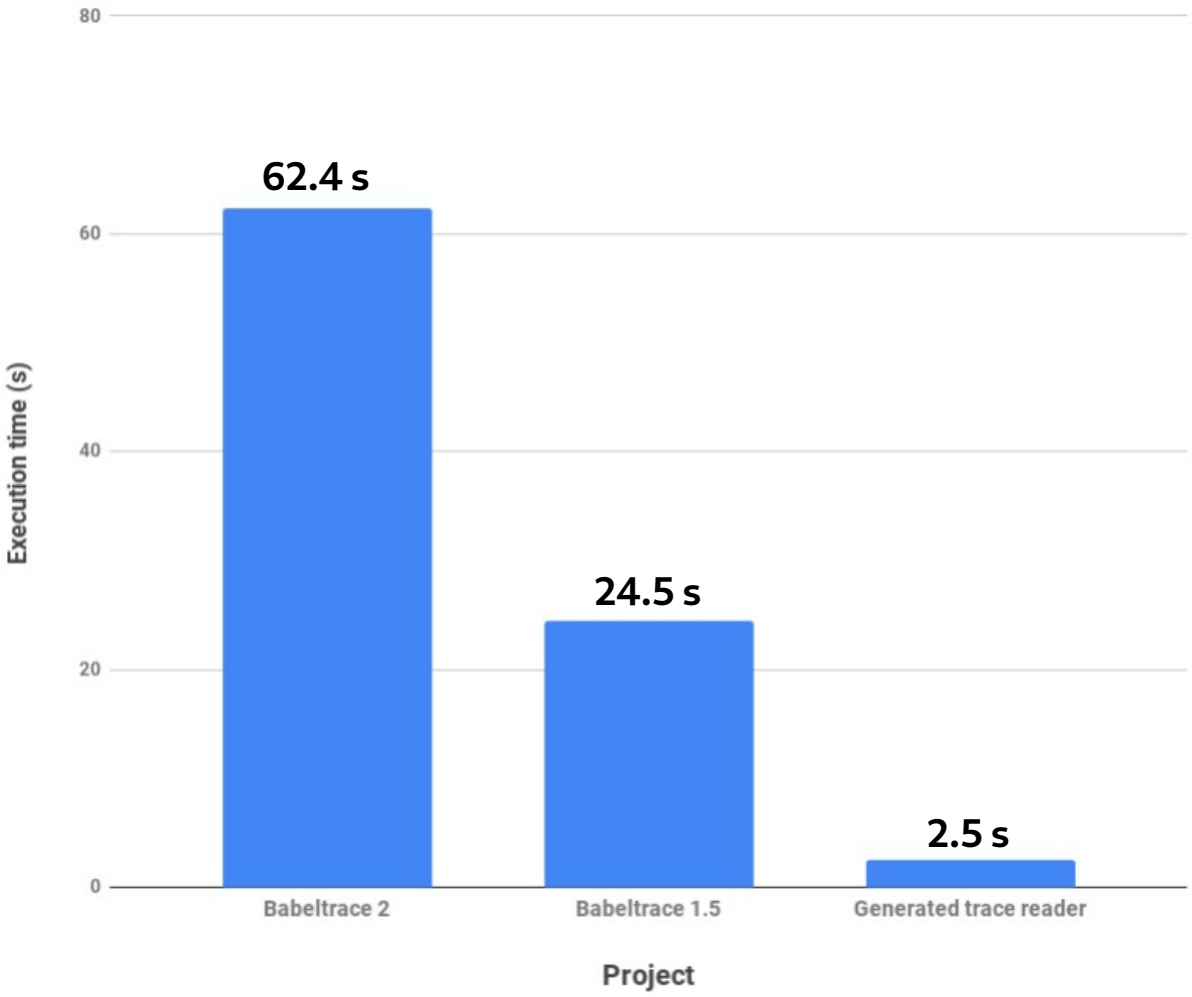


Future work ideas

Generated trace reader results:

Same build/run parameters:

- `CFLAGS='-O3 -DNDEBUG'`
- Dev./debug mode disabled
- Logging disabled
- Decode and discard event records
- Trace: 1.4 GiB, 4 data streams
- Drop caches before running



Future work ideas

- Generated trace reader has **no dynamic trace validation**: would be slower, but how much?
- Still, this experiment shows that **JIT-compiling** data stream decoders from the metadata is something to investigate for Babeltrace 2.

Conclusion

- Optimization results so far: **12.5x** → **2.5x** Babeltrace 1.5.
- Object pooling (vs. allocations/deallocations) increases performance when you create/destroy lots of object with the same layout.
- Not validating API function preconditions increases performance for fast path functions.
- Event record batching shows no significant performance gain currently, maybe when we optimize more.
- Trade-off for optimization techniques above is a **less uniform, riskier C API**.
- Future work: JIT-compiling trace reader shows good promise.