Profiling framepointer-less code with elfutils stacktrace

Serhei Makarov

serhei@serhei.io // smakarov@redhat.com

2024

available at http://serhei.io/files/cauldron2024-slides.pdf

In this talk....

The elfutils `eu-stacktrace` prototype uses elfutils' unwinder and `.eh_frame` CFI to process a stream of stack samples for profiling.

To explain the context of this project, we need to review the story of `-fomit-frame-pointer` and why this optimization was recently disabled in several Linux distributions....

In this talk....

1. History of `-fomit-frame-pointer` and profiling
2. Limitations of framepointer unwinding
3. Design of `eu-stacktrace`
4. Evaluation of `eu-stacktrace`+Sysprof prototype
5. Outline of further work

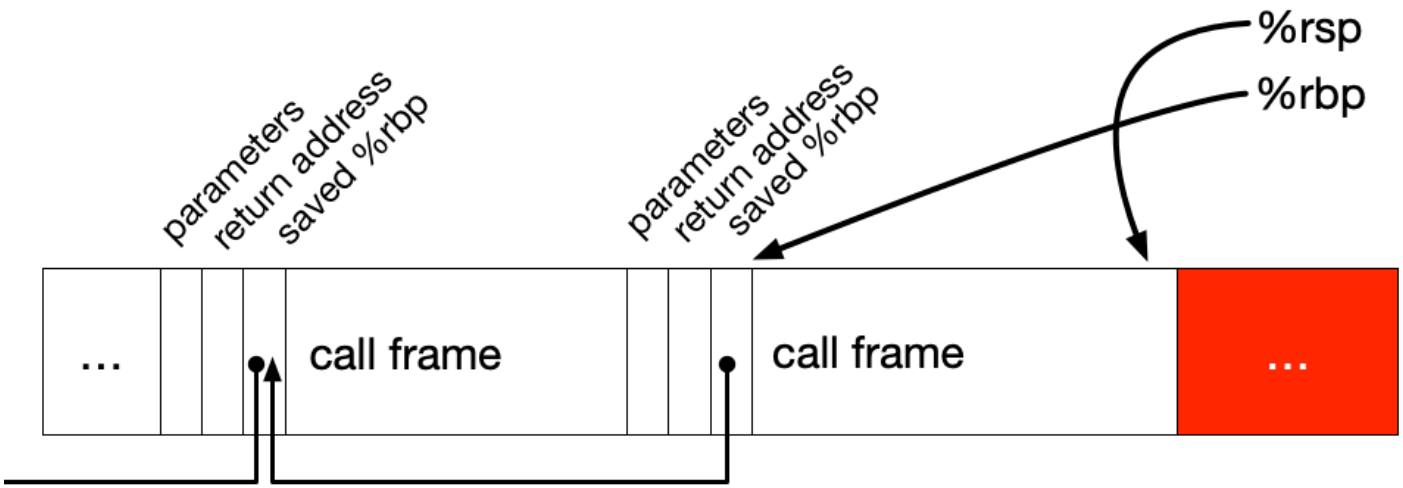
available at http://serhei.io/files/cauldron2024-slides.pdf

History of `-fomit-frame-pointer` and Profiling

1. Calling conventions with and without framepointers
2. DWARF and `.eh_frame` CFI
3. `-fomit-frame-pointer` compiler defaults
4. Shift from debugging to profiling
5. `-fomit-frame-pointer` defaults in Linux distros

History: calling convention with frame pointers

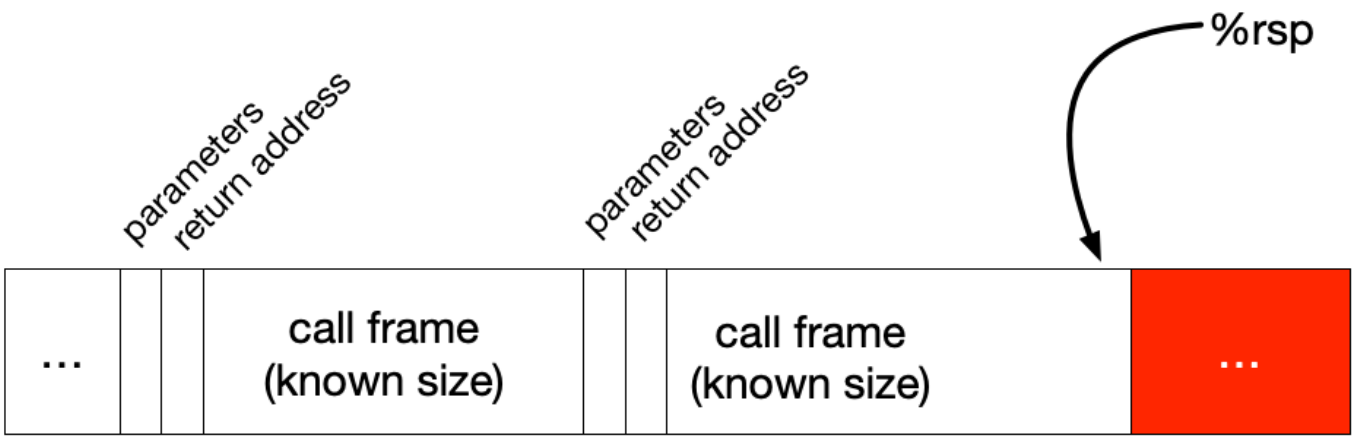
On x86, `%rsp` (stack pointer) points to end of stack, `%rbp` (frame pointer) points to beginning of stack frame:



Here, the saved `%rbp` values form a **backchain** through the entire call stack.

History: calling convention without frame pointers

On x86 with `-fomit-frame-pointer`, `%rsp` (stack pointer) points to end of stack:



Optimization: `%rbp` is not essential for finding the previous stack frame *when* we know the size of the stack frame; this knowledge can be compiled into the function epilogue *and* encoded in call-frame information (CFI) for external analysis.

History: DWARF and `.eh_frame` CFI

CFI in DWARF and `.eh_frame` format is widespread:

- In a DWARF `.debug_frame` section, bytecode specifies the Canonical Frame Address (CFA) for each instruction.
- `.eh_frame` section in executables is a size-optimized variant of the `.debug_frame` section.
- `.eh_frame` section is required in current Linux executables, used to support exception handling.

For more detailed info see:

- blog post on CFA and unwinding by Will Cohen.
- blog post on CFA and unwinding by Gwen Larsen.
- detailed review of unwind info by MaskRay.
- older posts on `.eh_frame`, `.eh_frame_hdr`, and GCC Exception frames (older approach) by Ian Lance Taylor.
- `dwarfstd.org`

History: `-fomit-frame-pointer` compiler defaults

Timeline of `omit-frame-pointer` optimization in GCC:

- GCC (~1993), architecture config: `CAN_DEBUG_WITHOUT_FP` target macro controls `-fomit-frame-pointer` optimization. (Now controlled by `TARGET_FRAME_POINTER_REQUIRED`.)
- GCC, PR13822 (2004): serious discussion to enable `-fomit-frame-pointer` by default for x86; at the time, Java runtime exception unwinding was not ready.
- GCC, PR13822 (2011): enabled by default on x86 in GCC 4.6.
- GCC, PR100811 (2023): a request to disable the `-fomit-frame-pointer` default; not granted.

GCC defaults on other architectures:

- PPC uses `-fomit-frame-pointer`, frame pointer not needed for backtracing.
- ARM uses `-momit-leaf-frame-pointer`, requires frame pointer in non-leaf functions.

LLVM defaults are fairly similar:

- `.eh_frame` format is generated on Linux cf llvm docs/ExceptionHandling.
- `-fomit-frame-pointer` enabled as default circa 2011 cf PR8186.
- (* Apple `aarch64` has a compact unwinding format aimed at exception handling but not debugging.)

History: shift from debugging to profiling

Use cases expand from debugging (GDB) and runtime exception handling to system-wide profiling, typically with `perf_events`:

- Desktop: performance problems during interactions among many desktop components; *advocacy cf.* *chergert*
- Hyperscale enterprises (e.g. Google, Meta, Netflix): continuous profiling in production; *advocacy examples cf.* *Gregg*, *Grigorenko*, *rwmj*

History: `-fomit-frame-pointer` defaults in Linux distros

Requests for default-on framepointers shift from GCC to various commonly-used Linux distributions:

- Fedora 38 on x86_64:
  - Fedora wiki "Changes/fno-omit-frame-pointer"
  - FESCo issues #2817, #2923, mailing-list discussions, lwn
  - further complications in pull-requests:
    redhat-rpm-config #231, #235, python3.11 #95
- Ubuntu 24.04: blog post
- Arch Linux: RFC #0026, gitlab MR
- Debian: request in bug tracker #7756; not granted

Limitations of Framepointer Unwinding

1. Arguments over performance reduction
2. Function prologues and epilogues
3. Inline assembly-code functions in libraries
4. In summary: assumptions for an profiler project

Limitations: arguments over performance reduction

Change of defaults is unsatisfying, being a win-lose tradeoff.
A few points from the back-and-forth during the Fedora discussions:

- Framepointer performance reduction: consensus is around 1-2%, with ∼10% in problematic (but fixable) cases ...
- ... but the number of feature requests producing a 1-2% global slowdown is **numerous** (see: SPECTRE/Meltdown mitigations, security hardening, and now framepointers). Repeating 10x 1% slowdown approaches 10%.
- 1-2% performance ≈ 1-2 years focused work on optimizations.
- Profile-driven fixes promised to offset the global slowdown.
- Win-lose impacts: users most impacted by framepointer slowdown (dependent on compiler optimization performance) different from the users who benefit from profiling-driven fixes (dependent on system-wide tuning). Who is more important?

Limitations: function prologues and epilogues

Framepointer register is not accurate everywhere:

- During function prologues and epilogues, the framepointer register is being updated.
- Earliest validity of fp is around 8 bytes (x86, riscv) or 12 bytes (aarch64) after the entry point to the function.
- Prologue initialization code may be further expanded by compiler optimizations.

Minimal x86 function prologue:

```
3532c: f3 0f 1e fa  endbr64
35330: 55           push   %rbp
35331: 48 89 e5     mov  %rsp,%rbp <- updating fp
35334: 53           push   %rbx    <- fp valid here
```

Limitations: function prologues and epilogues

Will Cohen's (`wcohen@redhat.com`) setup to estimate the impact of prologues and epilogues on profiling:

```
# perf report --sort=sample,symoff
  | grep -E '0x[01234567]$' # first 8 bytes only
  | grep -v "[k]" | grep -v "@plt" # userspace only
```

... yields ∼5.2% of userspace samples falling within the first 8 bytes of the function, when the framepointer still reflects the caller's stack frame, *cf wcohen's perf RFE*.

(The high percentage is likely due to the sampler hitting freshly loaded functions after a TLB miss. This affects the validity of the profile when evaluating code layout optimizations.)

Limitations: inline assembly-code functions in libraries

Inlined assembly-code functions in glibc do not follow the conventions for `-fno-omit-frame-pointer`. Hence a framepointer unwind would skip over the immediate caller of an inlined function, e.g. `f()` → `g()` → `memmove()` would be reported as `f()` → `memcpy()`.

Retooling glibc inline assembly-code to maintain the framepointer register is not a trivial (or desirable) ask, *cf fedoraproject discussions*.

In Summary: assumptions for an profiler project

Can we make profiling work without needing framepointers everywhere?

1. `-fomit-frame-pointer` is a performance option, we would like to preserve the flexibility to use it (decided on application, language-runtime, distro, compiler level).
2. `.eh_frame` data is widely available, since languages must unwind for exception handling.
3. `.eh_frame` data and unwinding are **necessary** to cover the gaps in framepointer unwinding for prologues and epilogues.

Design of `eu-stacktrace`

1. Minimal adaptation of a `perf`-based profiler
2. Modifying Sysprof to collect stack samples
3. Passing the stack samples to `eu-stacktrace`
4. Maintaining a table of Dwfl data structures
5. The unwinding procedure
6. Passing the unwound callchains back to Sysprof

Design: minimal adaptation of a `perf`-based profiler

Trying to reinvent the wheel as little as possible:

1. Take an existing profiler tool based on `perf_events`.
2. Try to make a minimal patch.
3. Patch connects the profiler to a new `elfutils` tool, `eu-stacktrace`.
4. `eu-stacktrace` uses the existing elfutils unwinder and existing `.eh_frame` CFI.

Design: modifying Sysprof to collect stack samples

Prototype the idea with Sysprof, adapt to other profilers later:

- Sysprof collects sample data from the Linux `perf_events` interface and produces a stream of data packets.
- Currently, Sysprof uses `PERF_SAMPLE_CALLCHAIN` option to request `perf` to provide an array of ip's from framepointer unwinder.
- Instead, add an option `--sample-method=stack` to use `PERF_SAMPLE_REGS_USER`+`PERF_SAMPLE_STACK_USER` and collect stack samples for unwinding.
- Tuning: increase perf ring buffer size and stack sample size.
- Do not record the stack samples to disk! Instead, pass them to an elfutils helper program `eu-stacktrace`.

Design: modifying Sysprof to collect stack samples

Sysprof file format is defined in `<sysprof-6/sysprof-capture-types.h>`. Add:

```
struct SysprofCaptureStackUser { ...
  u64 size; i32 tid; u32 padding; u8 data[];
};
struct SysprofCaptureUserRegs { ...
  u32 n_regs; u32 abi; u64 regs[];
};
```

Design: passing the stack samples to `eu-stacktrace`

Stream the Sysprof sample packets through a filter program `eu-stacktrace`:

```
$ mkfifo test.fifo
$ sudo eu-stacktrace --input test.fifo \
    --output test.syscap
$ sysprof-cli --sample-method=stack \
    --use-fifo=test.fifo test.syscap
```

In practice, the above arrangement is handled by Sysprof:

```
$ sysprof-cli --use-stacktrace
```

Design: maintaining a table of `Dwfl` data structures

- The elfutils libs use a data structure Dwfl to represent information about a process. Hence, we need a table mapping `pid` to `Dwfl`, with one entry for each process in the profile.
- Each `Dwfl` must be initialized with a map of the objects loaded into the process.
- For now, we just use the existing `dwfl_linux_proc_report` code which obtains the info from `procfs`.
- More work needed here due to concerns about processes in-container, short-lived processes, late library loads, reuse of pid's for different executables....

Design: the unwinding procedure

The elfutils unwinder takes a set of callbacks that implement memory reads of the process. Prior implementations used ptrace or core file:

```
struct Dwfl_Thread_Callbacks {
  ...
  bool (*set_initial_registers) (thread, *thread_arg);
  bool (*memory_read) (Dwfl, addr, *result, *arg);
  ...
}
```

Memory reads are also easy to simulate with a stack sample at known base address (via `%rsp`).

Design: the unwinding procedure

Fun detail: on x86, the register file is scrambled around like eggs between hardware, `perf`, and DWARF formats.

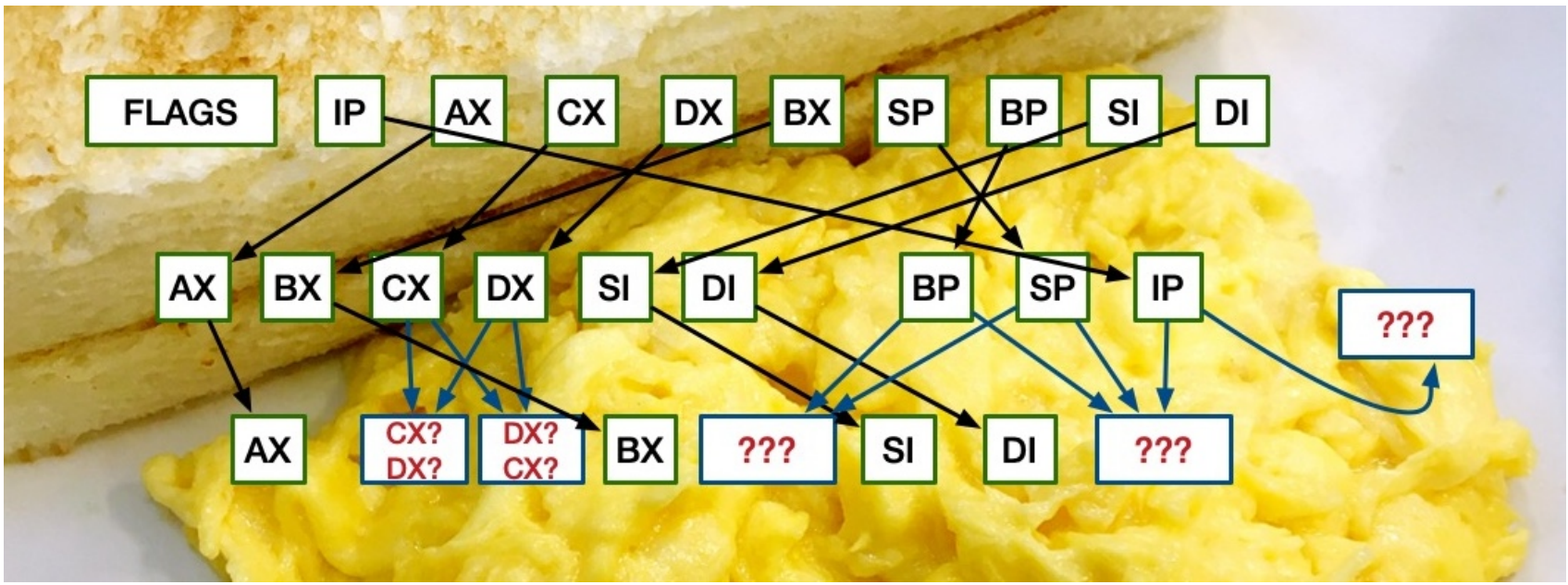


image source: https://commons.wikimedia.org/wiki/File:Hong_Kong_style_Scrambled_egg_sandwich.jpeg – CC BY-SA 4.0

Design: the unwinding procedure

Fun detail: on x86, the register file is scrambled around like eggs between hardware, `perf`, and DWARF formats.

- Hardware provides order seen in linux `tools/perf/util/intel-pt.c` `pebs_gp_regs` array.
  - `FLAGS`, `IP`, `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI`, `DI`, ...
- `perf_events` rearranges the registers to order seen in `arch/x86/include/uapi/asm/perf_regs.h` enum.
  - `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `BP`, `SP`, `IP`, ...
- Elfutils dwarf/eh unwinders require order seen in elfutils `backend/{x86_64,i386}_initreg.c`.
  - 32-bit: `AX`, `CX`, `DX`, `BX`, `SP`, `BP`, `SI`, `DI`, `IP`
  - 64-bit: `AX`, `DX`, `CX`, `BX`, `SI`, `DI`, `BP`, `SP`, `R8`, ... `R15`, `IP`

Many thanks to existing Qt `perfparser` project for illustrating how to manage this.

Design: passing the unwound callchains back to Sysprof

```
$ mkfifo test.fifo
$ sudo eu-stacktrace --input test.fifo \
    --output test.syscap
$ sysprof-cli --sample-method=stack \
    --use-fifo=test.fifo test.syscap
```

In this arrangement:

- Sysprof captures packets of type `SysprofCaptureStackUser+SysprofCaptureUserRegs` and passes them to `test.fifo`.
- Elfutils unwinds the samples and writes packets of type `SysprofCaptureSample`. These contain an array of ip's, the same as provided by `PERF_SAMPLE_CALLCHAIN`.
- Sysprof has a second annotation pass to find function symbols; this runs on the `test.syscap` file without any modifications.

Evaluation of `eu-stacktrace`+Sysprof prototype

1. Adding diagnostics to the unwinding procedure
2. Tracking unwinder success/failure by process
3. Table of results

Evaluation: adding diagnostics to the unwinding procedure

- Elfutils unwinder uses `.eh_frame`, then `DWARF`, then fallback framepointer unwinding.
- A simple patch to `libdwfl/frame_unwind.c` records which method was used.
- This can be tracked for each frame in detailed debug output:

```
sysprof_init_dwfl pid 4775 (cached): size=16384
  pc=7f96874bdfec sp=7f9491fff5d0+(0)
* 0: pc_adj=7f96874bdfec sp=7f9491fff5d0+(0) unwound=eh_frame
* 1: pc_adj=5627fcdef13f sp=7f9491fff5d0+(20) unwound=eh_frame
* 2: pc_adj=5627fce9fc84 sp=7f9491fff5d0+(40) unwound=eh_frame
* 3: pc_adj=5627fce8b644 sp=7f9491fff5d0+(90) unwound=eh_frame
* 4: pc_adj=7f9687441896 sp=7f9491fff5d0+(c0) unwound=eh_frame
* 5: pc_adj=7f96874c88c3 sp=7f9491fff5d0+(170) unwound=eh_frame
sysprof_unwind_cb pid 4775 (qemu-system-x86): unwound 6 frames
```

Evaluation: tracking unwinder success/failure by process

For larger-scale analysis, we produce a report with one line per process:

```
1 systemd -- max 24 frames, received 756 samples,
  lost 12 samples (1.6%) (last eh_frame, worst unknown)
2157 gsd-screensaver -- max 13 frames, received 1 samples,
  lost 0 samples (0.0%) (last eh_frame, worst eh_frame)
117690 gnome-system-mo -- max 96 frames, received 5378 samp
  lost 62 samples (1.2%) (last eh_frame, worst unknown)
...
```

To estimate failed unwinds, any sample with $\leq$2 frames is counted as 'lost'. This may be pessimistic, so we apply the same estimate to the framepointer data for comparison.

Evaluation: table of results

benchmark	cfi load	lost	fifo load	lost	fp load
f39 stress-ng matrix	1.6%	0.09%	0.26%	0.15%	0.04%
f40 gnome3 30sec	4.1%	6.2%*	3.1%	0.5%	1.9%
f40 gnome3 5min	9.0%	25%*	2.0%	0.33%	3.4%
f40 libreoffice	5.1%	12%*	1.9%	0.35%	0.81%
deb gnome3 30sec	8.4%	3.2%*	2.1%	11%?	1.5%
deb gnome3 5min	15%	4.8%*	6.2%	13%?	3.3%
deb libreoffice	14%	9.3%*	3.4%	26%?	2.8%

Benchmark explanation:

- 'load' is % of samples found in `eu-stacktrace+sysprof-cli+sysprofd`
- 'lost' is % of samples with $\leq$2 frames
- *lost samples on GNOME3 systems were dominated by `pkla-check-auth`
- ?gray text indicates framepointer unwind on non-framepointer distribution
- 'cfi' profiles with `eu-stacktrace` CFI unwinding
- 'fifo' profiles with framepointer then passes samples thru fifo to `eu-stacktrace`
- 'fp' profiles with framepointer and does not use `eu-stacktrace`

Outline of Further Work

Things I'm considering to do:

- Improvements to error diagnostics. (Elfutils' errnos are not enough to understand why a CFI load didn't succeed.)
- Analyze perf ring buffer throughput to make sure stack samples aren't being dropped.
- Ensure the tool works with containerized code. (Integrate Sysprof's solutions for this into elfutils proc-report.)
- Portability: other architectures, distros (Ubuntu, Debian).
- Portability: work with other profiling tools esp. `perf`.
- Efficiency: library interface instead of fifo interface.
- Efficiency: reuse data between Dwfls referencing the same object files. (Issue seen on the `libreoffice` benchmark.)
- Correctness: handle short-lived processes, late library loads, PIDs reused by different programs.
- Implement a per-thread stack prefix cache to allow arbitrary-depth backtrace and reduce the stack sample size.

Q & A

Get the code:

- `git clone -b users/serhei/eu-stacktrace https://sourceware.org/git/elfutils.git`
- `git clone https://git.sr.ht/~serhei/sysprof-experiments sysprof`
- Build instructions: https://sourceware.org/cgit/elfutils/tree/README.eu-stacktrace?h=users/serhei/eu-stacktrace

Some related projects:

- Qt `perfparser`: analyzes `perf record` data and `elfutils`.
- Polar Signals unwinder: translates `.eh_frame` to BPF for unwinding.
- SFrame: CFI format for in-kernel unwinding of userspace programs.
- samply: single-process, unwinds `perf` stack samples in memory.
- Further down the pipeline: shadow stacks?