ORC unwinder

Overview

The kernel CONFIG_UNWINDER_ORC option enables the ORC unwinder, which is similar in concept to a DWARF unwinder. The difference is that the format of the ORC data is much simpler than DWARF, which in turn allows the ORC unwinder to be much simpler and faster.

The ORC data consists of unwind tables which are generated by objtool. They contain out-of-band data which is used by the in-kernel ORC unwinder. Objtool generates the ORC data by first doing compile-time stack metadata validation (CONFIG_STACK_VALIDATION). After analyzing all the code paths of a .o file, it determines information about the stack state at each instruction address in the file and outputs that information to the .orc_unwind and .orc_unwind_ip sections.

The per-object ORC sections are combined at link time and are sorted and post-processed at boot time. The unwinder uses the resulting data to correlate instruction addresses with their stack states at run time.

ORC vs frame pointers

With frame pointers enabled, GCC adds instrumentation code to every function in the kernel. The kernel's .text size increases by about 3.2%, resulting in a broad kernel-wide slowdown. Measurements by Mel Gorman [1] have shown a slowdown of 5-10% for some workloads.

In contrast, the ORC unwinder has no effect on text size or runtime performance, because the debuginfo is out of band. So if you disable frame pointers and enable the ORC unwinder, you get a nice performance improvement across the board, and still have reliable stack traces.

Ingo Molnar says:

> "Note that it's not just a performance improvement, but also an instruction cache locality improvement: 3.2% .text savings almost directly transform into a similarly sized reduction in cache footprint. That can transform to even higher speedups for workloads whose cache locality is borderline."

Another benefit of ORC compared to frame pointers is that it can reliably unwind across interrupts and exceptions. Frame pointer based unwinds can sometimes skip the caller of the interrupted function, if it was a leaf function or if the interrupt hit before the frame pointer was saved.

The main disadvantage of the ORC unwinder compared to frame pointers is that it needs more memory to store the ORC unwind tables: roughly 2-4MB depending on the kernel config.

ORC vs DWARF

ORC debuginfo's advantage over DWARF itself is that it's much simpler. It gets rid of the complex DWARF CFI state machine and also gets rid of the tracking of unnecessary registers. This allows the unwinder to be much simpler, meaning fewer bugs, which is especially important for mission critical oops code.

The simpler debuginfo format also enables the unwinder to be much faster than DWARF, which is important for perf and lockdep. In a basic performance test by Jiri Slaby [2], the ORC unwinder was about 20x faster than an out-of-tree DWARF unwinder. (Note: That measurement was taken before some performance tweaks were added, which doubled performance, so the speedup over DWARF may be closer to 40x.)

The ORC data format does have a few downsides compared to DWARF. ORC unwind tables take up ~50% more RAM (+1.3MB on an x86 defconfig kernel) than DWARF-based eh_frame tables.

Another potential downside is that, as GCC evolves, it's conceivable that the ORC data may end up being *too* simple to describe the state of the stack for certain optimizations. But IMO this is unlikely because GCC saves the frame pointer for any unusual stack adjustments it does, so I suspect we'll really only ever need to keep track of the stack pointer and the frame pointer between call frames. But even if we do end up having to track all the registers DWARF tracks, at least we will still be able to control the format, e.g. no complex state machines.

ORC unwind table generation

The ORC data is generated by objtool. With the existing compile-time stack metadata validation feature, objtool already follows all code paths, and so it already has all the information it needs to be able to generate ORC data from scratch. So it's an easy step to go from stack validation to ORC data generation.

It should be possible to instead generate the ORC data with a simple tool which converts DWARF to ORC data. However, such a solution would be incomplete due to the kernel's extensive use of asm, inline asm, and special sections like exception tables.

That could be rectified by manually annotating those special code paths using GNU assembler .cfi annotations in .S files, and homegrown annotations for inline asm in .c files. But asm annotations were tried in the past and were found to be unmaintainable. They were often incorrect/incomplete and made the code harder to read and keep updated. And based on looking at glibc code, annotating inline asm in .c files might be even worse.

Objtool still needs a few annotations, but only in code which does unusual things to the stack like entry code. And even then, far fewer annotations are needed than what DWARF would need, so they're much more maintainable than DWARF CFI annotations.

So the advantages of using objtool to generate ORC data are that it gives more accurate debuginfo, with very few annotations. It also insulates the kernel from toolchain bugs which can be very painful to deal with in the kernel since we often have to workaround issues in older versions of the toolchain for years.

The downside is that the unwinder now becomes dependent on objtool's ability to reverse engineer GCC code flow. If GCC optimizations become too complicated for objtool to follow, the ORC data generation might stop working or become incomplete. (It's worth noting that livepatch already has such a dependency on objtool's ability to follow GCC code flow.)

If newer versions of GCC come up with some optimizations which break objtool, we may need to revisit the current implementation. Some possible solutions would be asking GCC to make the optimizations more palatable, or having objtool use DWARF as an additional input, or creating a GCC plugin to assist objtool with its analysis. But for now, objtool follows GCC code quite well.

Unwinder implementation details

Objtool generates the ORC data by integrating with the compile-time stack metadata validation feature, which is described in detail in tools/objtool/Documentation/stack-validation.txt. After analyzing all the code paths of a .o file, it creates an array of orc_entry structs, and a parallel array of instruction addresses associated with those structs, and writes them to the .orc_unwind and .orc_unwind_ip sections respectively.

The ORC data is split into the two arrays for performance reasons, to make the searchable part of the data (.orc_unwind_ip) more compact. The arrays are sorted in parallel at boot time.

Performance is further improved by the use of a fast lookup table which is created at runtime. The fast lookup table associates a given address with a range of indices for the .orc_unwind table, so that only a small subset of the table needs to be searched.

Etymology

Orcs, fearsome creatures of medieval folklore, are the Dwarves' natural enemies. Similarly, the ORC unwinder was created in opposition to the complexity and slowness of DWARF.

"Although Orcs rarely consider multiple solutions to a problem, they do excel at getting things done because they are creatures of action, not thought." [3] Similarly, unlike the esoteric DWARF unwinder, the veracious ORC unwinder wastes no time or siloconic effort decoding variable-length zero-extended unsigned-integer byte-coded state-machine-based debug information entries.

Similar to how Orcs frequently unravel the well-intentioned plans of their adversaries, the ORC unwinder frequently unravels stacks with brutal, unyielding efficiency.

ORC stands for Oops Rewind Capability.

[1] https://lore.kernel.org/r/20170602104048.jkkzssljsompjdwy@suse.de

[2] https://lore.kernel.org/r/d2ca5435-6386-29b8-db87-7f227c2b713a@suse.cz

[3] http://dustin.wikidot.com/half-orcs-and-orcs