Android Chrome Unwind Info v2

This Document is Public

Authors: wittman@chromium.org, etiennep@chromium.org
August 2021

Overview

Summary

Replace the current custom unwind information and the unwinder used to unwind stack frames within native code compiled into Android Chrome. The current unwind info representation can only unwind 55-65% of stack samples collected during active execution of native code in Chrome, which is well short of the 95+% required for aggregated samples to be fully useful. The new, more comprehensive unwinder will exceed this threshold.

Platforms

Android

Bug

https://crbug.com/1122124

Code affected

Sampling profiler, heap profiler.

Background

Stack unwinding

Stack unwinding is the process of 'undoing' the function calls on a thread's stack, starting at the frame associated with the currently executing function, and ending at some ancestor caller frame. This process is used for exception handling, where the registers are restored to what they were in the ancestor caller frame in order to continue execution where the exception is caught.

The process is also used for statistical profiling, where a thread's stack is periodically sampled by recording the functions associated with its stack frames. The sampled stacks are aggregated together to provide an indication of how much runtime is spent within

different functions within the executable. Chrome's sampling profiler[1] makes use of stack unwinding in this form to generate aggregate profiles of Chrome execution.

Stack unwinding may occur via multiple mechanisms. For the purposes of native code in Android Chrome, stack unwinding conceptually occurs via a large table mapping function addresses within Chrome to operations on the register state when executing within the function. The operations describe how to restore the caller frame's instruction pointer and register state given the callee's instruction pointer and register state. This information is collectively referred to as *unwind info*. It's customarily generated by the compiler and linked within the executable.

Stack unwinding within Java code uses a different mechanism and relies on support within the `libunwindstack` library. This document focuses solely on native code unwinds.

Chrome unwind info

Chrome ships without compiler-generated unwind info on Android however, due to binary size concerns: the unwind info would add 1MB to the stable binary size. The benefit of having the unwind info in those releases is not considered worth the costs in terms of user bandwidth, reduced update rate, memory, and disk usage.

Instead, a bespoke unwind info representation is generated at build time[2], and is used by a stack unwinder[3] specifically built for the representation. The unwind info is shipped with Chrome in canary and dev channels, where binary size concerns are not as pressing. (The unwind info will be shipped out-of-band via a dynamic feature module to a subset of clients on beta and stable to avoid size issues on those channels, but this has not been implemented yet.)

Why not MiniDebugInfo?

Chrome uses bespoke unwind info rather than Android's standard MiniDebugInfo for several reasons. The operating support for unwinding using MiniDebugInfo depends on the MiniDebugInfo being present within the executable's ELF file itself. This is incompatible with shipping it out-of-band on later channels to reduce binary size. The alternative of shipping out-of-band but using a custom unwinder to interpret the DWARF format embedded in MiniDebugInfo is also not viable because it is prohibitively costly in terms of code

---

1 https://source.chromium.org/chromium/chromium/src/+/main:base/profiler/stack_sampling_profiler.h;drc=e6c6bb38f04dafba6eb55bf95006dd14a503b58a

2 https://source.chromium.org/chromium/chromium/src/+/main:build/android/gyp/extract_unwind_tables.py;drc=e6c6bb38f04dafba6eb55bf95006dd14a503b58a

3 https://source.chromium.org/chromium/chromium/src/+/main:base/profiler/chrome_unwinder_android.h;drc=e6c6bb38f04dafba6eb55bf95006dd14a503b58a

complexity and maintenance.

MiniDebugInfo is designed for crash reporting, not stack sampling. The format is effectively reduced and compressed DWARF and is prohibitively costly to decompress and store at runtime. For comparison, libandroid_runtime.so's MiniDebugInfo is 84k and takes 20-30ms CPU time to decompress on a Pixel 3. Chrome's 1MB MiniDebugInfo would take hundreds of ms. When uncompressed the in-memory representation is 12MB which needs to be maintained in each process' memory for the duration of the profiling — 30s during startup and periodically thereafter.

The team has consulted with Android engineers working in this space on the MiniDebugInfo issues. They recognize the problems for Chrome but don't have near term plans to address them. When addressed they would roll out only in newer Android releases.

Chrome's current bespoke format is 2.2MB in size, small in comparison to the uncompressed MiniDebugInfo, and is directly usable from the memory-mapped resource file.

Deficiencies in Chrome Unwind Info v1

The 35-45% of unsuccessful native code unwinds on the renderer main thread with unwind info v1[4] are mostly due to deficiencies in the unwind info format itself. In particular it lacks support for:

- Stacks containing frames in functions that use dynamic stack allocation (as used to implement `alloca(3)`, for example).
- Stacks with frames in V8 generated code followed by frames in Chrome code.
- Stacks containing frames in functions that are too large in various dimensions (e.g. frame size, function size) to fit in the encoded info.
- Stacks with some leaf functions that don't alter the stack pointer (due to a bug in the info generator).
- Stacks where the executing code is in a function epilogue.

Design

Addressing the deficiencies in unwind info v1 boils down to four focus areas:

1. Lifting restrictions on the sizes of supported functions.
2. Lifting restrictions on the types of supported functions.
3. Restoring callee-save registers.
4. Supporting unwinds from epilogue instructions.

[4] build/android/gyp/extract_unwind_tables.py documents the format.

For (1) we need to change from the current format that uses fixed bit widths for characterizing sizes to a format that allows arbitrary sizes (within reason).

For (2) we need to add support for dynamic stack allocation which saves the stack pointer register into a callee-save register. The unwind info must be able to encode an unwind instruction to restore the stack pointer from the register.

For (3) we need to add support for restoring callee-save registers beyond the `lr` register. Unwinds through functions using dynamic stack allocation and through V8 require correct values in callee-save registers.

(4) is somewhat complicated because clang itself doesn't currently generate unwind information for epilogue instructions[5]. Several options with varying costs and benefits exist for working around this.

Function unwind encoding

Of the four focus areas, 1-3 can be largely addressed through the use of ARM's compact unwind encoding instructions[6], with some augmentation. These handle all possible unwind scenarios for the architecture in a compact format — at most three bytes per function for most functions. The unwind instructions may be synthesized from the DWARF unwind info produced by the compiler.

The compact unwind encoding instructions are designed for exception-driven synchronous unwinds, where all code in each function can be unwound by the same instructions. This is the case because exception-driven unwinds are always initiated at an instruction after the function prologue, which sets up the stack for the function execution, and before the epilogue, which tears it down. The sampling profiler uses asynchronous unwinds, where the unwind can take place from any instruction. Unwinds from within the prologue or epilogue need execute only a subset of the encoded unwind instructions, based on which code instructions have been executed. To support partial unwind instruction execution, we must additionally store the addresses corresponding to the code instructions which change the unwind state, along with a pointer to the specific unwind instructions to execute.

Example

As an example, let's look at the start of the `std::istream::operator>>(double&)` function:

```
009ef4dc <std::__1::basic_istream<char, std::__1::char_traits<char>
>::operator>>(double&)>:
```

[5] https://bugs.llvm.org/show_bug.cgi?id=47142
[6] Exception handling ABI for the ARM architecture ABI, § 9.3

```
9ef4dc:       b570            push    {r4, r5, r6, lr}
9ef4de:       b086            sub     sp, #24
9ef4e0:        4604            mov     r4, r0
9ef4e2:        2000            movs    r0, #0
9ef4e4:        9005            str     r0, [sp, #20]
9ef4e6:        a804            add     r0, sp, #16
9ef4e8:        460d            mov     r5, r1
9ef4ea:        4621            mov     r1, r4
9ef4ec:        2200            movs    r2, #0
...
```

The instructions in bold are the function's prologue.

When a function call is made on ARM, the calling convention[7] dictates that the caller passes arguments in registers `r0-r3` and optionally on the stack. Then the `pc` (program counter) register is copied to the `lr` (link) register, and execution branches to the first address in the function. At this point the procedure for unwinding the call is simply to copy `lr` back to `pc`. For the example this is the unwind procedure when the `pc` is at `9ef4dc`.

Registers `r4-r12` are callee-save registers on ARM on Android, meaning that if a function uses them for computation it must push them on the stack at the start of the function and pop them off at the end. This restores the values to what they were in the caller. An unwinder must do the same to restore the register contents.

In the example, `r4-r6` are pushed on the stack in the instruction at address `9ef4dc`. `lr` Is also pushed in anticipation of calling another function from within `std::istream::operator>>(double&)`, which would require overwriting `lr`. To support unwinding from address `9ef4de` the unwind info records that those four registers must be popped from the stack.

The instruction at `9ef4de` further affects the stack pointer by moving it down by 24 bytes. This is done to provide space for local variables and/or spilled arguments for function calls made by `std::istream::operator>>(double&)`. To support unwinding from address `9ef4e0` and later the unwind info records that the stack point must be incremented by 24 bytes.

In summary the information required to unwind the example is:

Address	Unwind instructions
`9ef4dc`	Copy `lr` to `pc`.

[7] https://developer.arm.com/documentation/ihi0042/j/?lang=en

`9ef4de`	Pop `r4-r6,lr`.
`9ef4de`	Increment `sp` by 24.

If the `pc` is at the given address in the function or later, its unwind instruction and prior unwind instructions are executed to perform the unwind.

The ARM compact unwind encoding instructions encode the state in the second column, but we must separately augment with the address within the function in the first column to allow unwinds from the prologue instructions.

The epilogue generally works analogously to the prologue but performs the inverse instructions in the reverse order.

Unwind info: a conceptual representation

A conceptually complete representation of the unwind info involves two tables:

1. An address table to map `pc` addresses to their corresponding functions.
2. A function table listing the addresses within each function and their corresponding unwind instructions, analogous to the previous example.

To perform an unwind, look up the `pc` in the address table to find the function, then find the greatest address less than or equal to the `pc` in the function, and execute its and all prior unwind instructions.

The conceptual representation has several issues that prevent it from being of practical use:

- Storing the full address within the function unwind encoding is wasteful of space because it requires 4 bytes per address. Chrome's unwind info contains 816,686 addresses[8] so this would use 3.3MB.
- Chrome functions exhibit very little variation in unwind instructions, so storing dedicated unwind instructions per function (rather than deduplicating them) is wasteful. It would use 1.5MB, compared to 15k for the deduplicated instructions.
- The unwind instructions use a variable length encoding in order to represent stack pointer increments/decrements of varying sizes. As a result the function table entries cannot be a fixed size and must use a variable length encoding for both addresses and unwind instructions. The unwind instructions would additionally require an explicit length representation which would double their storage costs.
- The address table requires 4 bytes per function to encode each function's start address, plus another 4 bytes to index into the function table. (This assumes functions are contiguous, i.e. the start address of the following function is

[8] As of https://crrev.com/893260, `chrome_modern_public_bundle` official build.

immediately after the end address of the current function, which is substantially true.) Chrome's unwind info contains 389,694 functions, so this would use 3.1MB.

Naïvely the conceptual representation would use around 9.4MB which is considerably larger than desirable. A previous Google-internal study found that even a 5MB increase in memory consumption on low end devices has a measurable negative impact on page rendering times[9].

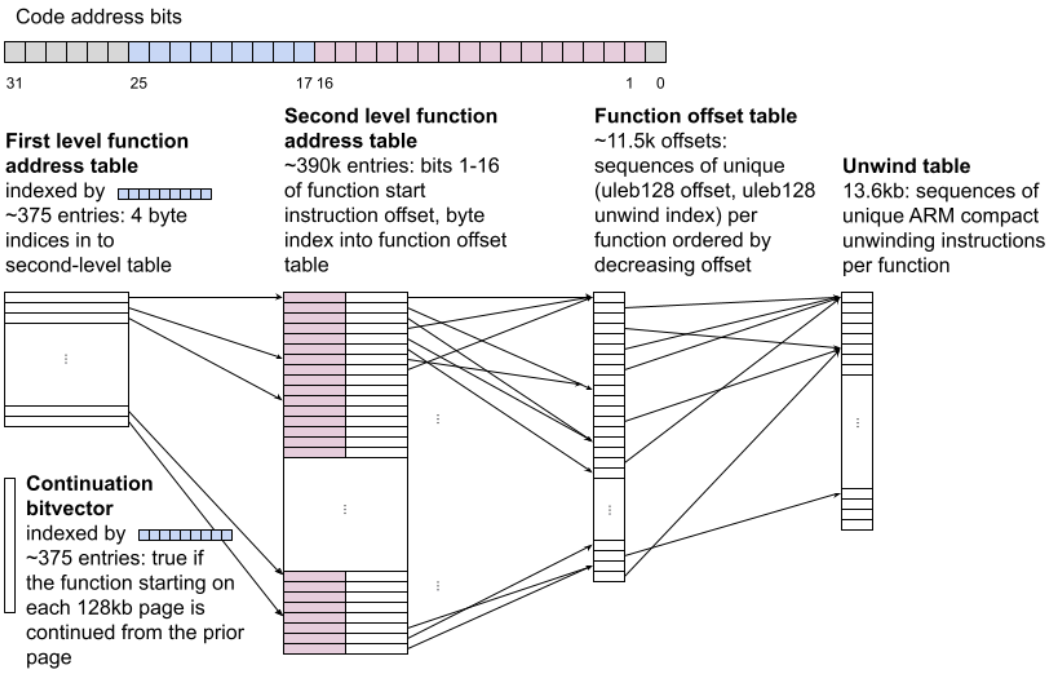
Unwind info: a practical representation

We apply several approaches to reduce the unwind info memory footprint and address the variable length encoding challenges.

- Use a two-level address table to reduce the function start address representation size from 4 bytes/address to ~2 bytes/address.
- Store address offsets from start of function in the function table, rather than full addresses, and use a variable length encoding to reduce offset representation size from 4 bytes/offset to 1 byte/offset.
- Move the unwind instructions to their own table and index into that from each offset. Each offset corresponds to a range of unwind instructions to execute, and would naïvely require storing pointers to the start and end instructions. But we can take advantage of the fact that the last unwind instruction to execute in each function implicitly marks the unwind as completed, and only store the start instruction to execute per offset. Using a variable length encoding of the index of the start unwind instruction (to match the offset variable length encoding) requires ~1.5 bytes/offset.
- Deduplicate function unwind state (both the offsets/instruction indices and the instructions) to reduce the function unwind encoding from 1.5MB to 43k.

With these approaches we can reduce the space requirements for the v2 unwind info from 9.4MB to 1.8MB. Note that this is smaller than the v1 unwind info at 2.2MB, even though it's a considerably more capable representation.

[9] Google-internal link: go/chrome-mem-ablation



Graphical representation of the unwind info v2 encoding

Pseudocode unwind algorithm

```
pc_offset = pc - text section base address
is_continued_from_prior_page = continuation_vector[pc_offset >> 17]
if is_continued_from_prior_page
 second_level_entry =
     second_level_table[first_level_table[pc_offset >> 17] - 1]
else
  page_start_index = first_level_table[pc_offset >> 17]
  next_page_start_index = first_level_table[(pc_offset >> 17) + 1]
  second_level_entry = /* binary search in second_level_table between
      page_start_index and next_page_start_index for the largest
      function start address less than (pc_offset >> 1) & 0xffff */

function_offset = ((pc_offset >> 1) & 0xffff) -
    second_level_entry.function_start_instruction_offset_from_page
curr_function_offset_table_position =
    second_level_entry.function_offset_table_index
```

```
while true
  unwind_instruction_offset, curr_function_offset_table_position =
      uleb128_decode(curr_function_offset_table_position)
  unwind_table_index, curr_function_offset_table_position =
      uleb128_decode(curr_function_offset_table_position)
  if unwind_instruction_offset <= function_offset:
    /* execute ARM compact unwinding instructions starting at
        unwind_table[unwind_table_index] */
    break
```

Struct representation for the unwind info header

```
// Represents each entry in the function table (i.e. the second level of the
// function address table).
struct FunctionTableEntry {
 // The offset into the 128kb page containing this function. Indexed by bits
 // 1-16 of the pc offset from the start of the text section.
 uint16_t function_start_address_page_offset;

 // The byte index of the first offset for the function in the function offset
 // table.
 uint16_t function_offset_table_byte_index;
};

// The header at the start of the unwind info resource, with offsets/sizes for
// the tables contained within the resource.
struct UnwindInfoHeader {
 // The offset in bytes from the start of the unwind info resource to the page
 // table (i.e. the first level of the function address table). The page table
 // represents discrete 128kb 'pages' of memory in the text section, each of
 // which contains functions. If the start address of the resource is
 // represented by uintptr_t resource_start_address, then the page table can be
 // accessed as
 //   const uint32_t* const page_table = reinterpret_cast<uint32_t*>(
 //       resource_start_address + page_table_byte_offset);
 // and indexed by bits 17 and greater of the pc offset from the start of the
 // text section.
 uint32_t page_table_byte_offset;
 // The number of entries in the page table.
 uint32_t page_table_entries;

 // The offset in bytes from the start of the unwind info resource to the
 // function continuation bitvector, which represents whether a the first
```

```
// function in a 128kb page is continued from the previous page. The
// continuation bitvector has the same number of entries as
// |page_table_entries| and can be accessed as
//   const uint8_t* const function_continuation_bitvector =
//       reinterpret_cast<uint8_t*>(resource_start_address +
//           function_continuation_bitvector_byte_offset);
// and the bits indexed by bits 17 and greater of the pc offset from the start
// of the text section. Bits in the vector are in little endian order.
uint32_t function_continuation_bitvector_byte_offset;
// The size of the bitvector in bytes.
uint32_t function_continuation_bitvector_size_in_bytes;

// The offset in bytes from the start of the unwind info resource to the
// function table (i.e. the second level of the function address table). The
// function table represents the individual functions within a 128kb page. The
// function table can be accessed as
//   const FunctionTableEntry* const function_table =
//       reinterpret_cast<FunctionTableEntry*>(resource_start_address +
//           function_table_byte_offset);
// the relevant entry for a pc offset from the start of the text section is
// the one with the largest function_start_address_page_offset <= (pc_offset
// >> 1) & 0xffff.
uint32_t function_table_byte_offset;
// The number of entries in the function table.
uint32_t function_table_entries;

// The offset in bytes from the start of the unwind info resource to the
// function offset table. The function offset table represents the pc offsets
// from the start of each function along with indices into the unwind
// instructions for the offsets. The pc offsets and unwind indices are
// represented as (ULEB128, ULEB128) pairs in decreasing order of offset.
// Distinct sequences of (offset, index) pairs are concatenated in the table.
// The function offset table can be accessed as
//   const uint8_t* const function_offset_table =
//       reinterpret_cast<uint8_t*>(resource_start_address +
//           function_offset_table_byte_offset);
uint32_t function_offset_table_byte_offset;
// The size of the function offset table in bytes.
uint32_t function_offset_table_size_in_bytes;

// The offset in bytes from the start of the unwind info resource to the
// unwind instruction table. The unwind instruction table represents distinct
// sequences of ARM compact unwind instructions[1] used across all functions
```

```
 // in Chrome. The compact unwind instructions byte-oriented variable length
 // encoding so are indexed by byte position. The unwind instruction table can
 // be accessed as
 //   const uint8_t* const unwind_instruction_table =
 //       reinterpret_cast<uint8_t*>(resource_start_address +
 //           unwind_instruction_table_byte_offset);
 //
 // 1. See Exception handling ABI for the ARM architecture ABI, §9.3.
 // https://developer.arm.com/documentation/ihi0038/b.
 uint32_t unwind_instruction_table_byte_offset;
 // The size of the unwind instruction table in bytes.
 uint32_t unwind_instruction_table_size_in_bytes;
};
```

Epilogue unwind info

clang currently doesn't generate unwind information for function epilogues on arm32, only for prologues, so it's difficult to support unwinding from epilogue instructions. The options available for addressing epilogue unwinds are:

1. Don't support epilogue unwinds.
2. Wait for clang support for epilogue unwind information.
3. Synthesize unwind information ourselves based on disassembly of the Chrome executable and
    a. include in the above representation (note: probably blows up encoded size substantially), or
    b. add a separate representation for epilogue unwind information.
4. In the unwinder, simulate epilogue instruction execution to effectuate a 'return' from the function if the pc appears to be at an epilogue instruction. (Note: applicable to arm32 architecture only since arm64 uses execute-only memory for text sections.)

In local testing only 1.5% of stack unwinds failed due to missing epilogue unwind information. That's within our 5% maximum failure rate target so we will not attempt to support epilogue unwinds initially. If epilogue unwinds prove more problematic than that measurement indicates, we will explore simulating epilogue instruction execution which should be low effort and high return.