WebAssembly Out of Bounds Trap Handling

V8 currently enforces memory safety in WebAssembly by using explicit bounds checks. This has several downsides:

- The compiler generates extra nodes for each memory reference, leading to longer compile times due to the additional processing time needed for these nodes.
- In turn, these extra nodes lead to lots of extra code being generated, making WebAssembly modules bigger than they ideally should be.
- This extra code, particularly the compare and branch before every memory reference, incurs a significant runtime cost.

This document describes how to safely replace the bounds checks with a signal handler. Here we will focus on memory references on Linux for x86-64. Future design documents will consider other platforms and also handling the stack overflow case.

Change History

- 24 October 2016 - Changed the design of the signal handler so that the V8 embedder must install the signal handler itself and call a function provided by V8 to determine whether to handle the fault. Made small changes to the fault location table and added some discussion about using the source code layout to protect the code. Added a discussion about the changes needed to allocate Wasm memories with guard pages.
- 25 October 2016 - Added a section about moving V8-generated code to a separate region of memory.

Overview

The overall goal is to surround the WebAssembly memory with large guard regions that are marked PROT_NONE and then install a signal handler that traps attempts to access memory in the guard region. The signal handler is then responsible for determining where the fault happened and throwing an appropriate JavaScript exception. There are several components needed to make this happen:

- Guard pages around the WebAssembly memory space
- A signal handler that will respond to faults at runtime
- A fault location table that the signal handler uses to determine if the fault originated from WebAssembly code and where
- Compiler changes to populate the fault location table

Guard Pages

Wasm memory instructions include a base address and an offset, such as in this example:

```
(i32.load offset=12 (get_local $x))
```

Executing this instruction would compute the sum of the offset (12), and the base, which here is stored in a local variable `$x`, and then load from the resulting address. This roughly corresponds to the following x64-64 instruction[1]**:**

```
mov   eax, dword ptr [rdi + 12]
```

Wasm uses unsigned arithmetic for address calculations. The base and offset are each 32-bit values but added together using infinite precision arithmetic.[2] Thus, the largest possible memory address is 0xffffffff + 0xffffffff = 0x1fffffffe, or just under 8GB.

The minimum requirements to ensure safety, therefore, are to allocate an 8GB chunk of address space, use the first portion of it for the Wasm linear memory and mark the remainder as inaccessible (e.g. PROT_NONE). This will only work on 64-bit systems, so we will need to do something different for 32-bit. We assume, and will need to ensure, that mapping and accessing the guard pages does not lead to paging of unused memory.

For defense in depth, it makes sense to also allocate guard pages in front of the Wasm memory. This is to defend against potential code generation bugs where one or both of the operands are sign extended to 64 bits when they should have been zero extended. Protecting against this will require an 8GB guard region in front of the Wasm memory. The diagram below shows the layout of the memory under this design.

[1] See http://goo.gl/6knvFe to explore this example further.
[2] See https://github.com/WebAssembly/design/blob/master/AstSemantics.md#linear-memory for the relevant parts of the Wasm spec.

One potential concern is the amount of virtual address space available. Most 64 bit operating systems give processes 128TB of address space.[3] If we require 16GB per Wasm module, then a 128TB address space allows for a maximum of 16,000 Wasm modules per process. 16k ought to be enough for anybody.

This change will likely let us simplify `grow_memory`'s implementation for 64-bit systems. There is no longer any reason to move the memory, because the guard pages mean we have already reserved enough memory. Instead, we simply need need to make the newly added pages accessible.

Allocating Wasm Memory

See https://codereview.chromium.org/2396433008/ for a work-in-progress CL to add guard regions to Wasm memories.

Creating guard regions around Wasm memory happens during module instantiation. There are several cases we need to consider.

The easiest case is the a new memory is created for the module instance. In this case, we allocate enough space for the Wasm memory and its surrounding guard pages. We use the appropriate OS functions (such as `mprotect`) to make the guard regions inaccessible. Then we wrap this buffer in a new `ArrayBuffer`. Because we are managing allocation of this buffer ourselves, we mark it as external. A finalizer will free the buffer when the last reference to the `ArrayBuffer` has gone away.

A trickier case is when an existing `ArrayBuffer` is given. In this case, we still allocate a new region of memory and set up memory protection on the guard regions. However, we must also copy the contents of the previous buffer into the new one. Once this copy is done, we will use the V8 allocator to free the old buffer, and then update the backing store pointer for the `ArrayBuffer` to point to the new store.

In general, the backing store for an `ArrayBuffer` can come from anywhere. For example, it could possibly be allocated in GPU memory. In general, we cannot handle every possible backing store. The safe thing then is probably to check whether the `ArrayBuffer` is allocated by V8 and if not, fail to instantiate the module. Note that this probably is not unique to guard regions, but is also present with the way `grow_memory` is currently implemented, since `grow_memory` might need to move the backing store too. The newly allocated backing store will also use a finalizer to ensure it is freed at the right time.

---

[3] See, for example, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366778(v=vs.85).aspx#memory_limits.

Signal Handler

Because V8 can be embedded in many different hosts (such as Chrome, Android WebView, NodeJS, etc.) we will not implement our own signal handler. The various hosts may provide their own signal handler, and it is nearly impossible to ensure that everything works correctly together. Instead, hosts must explicitly opt in to use signal handlers for out-of-bounds checking.

V8 will provide a new API to enable signal handling. If this API is not used, V8 will continue to generate code with bounds checks as before. Once signal handling is enabled, the host is responsible for installing their own SIGSEGV handler. We will provide an additional function to call from the signal handler that will determine whether this fault was due to an out of bounds memory access in WebAssembly.

We will provide a signal handler in the d8 shell, which can also serve as an example for other V8 embedders. For enabling this feature in Chrome, it would be best to integrate with Breakpad and Crashpad.

Handling the fault

This section provides an overview of the V8-provided signal helper. For more details, with a particular focus on the security and correctness concerns, see this document. We must be cautious in the signal handler given that we cannot assume that memory has not been corrupted. Thus, we will use a sequence of checks of increasing complexity to rule out possible failures while determine whether a fault is one V8 can recover from. Below is pseudocode for the the V8 fault handler helper.

```
bool MaybeHandlerWasmFault(byte* code_addr,
                           byte* data_addr,
                           byte** target_addr)
    // code_addr: the address of the instruction that faulted.
    // data_addr: the address the instruction was trying to access.
    // target_addr: an output parameter for the address to return to
    //              if V8 decides to handle the fault.
    // returns true if V8 can recover from the fault, false otherwise[4]
{
  if (!thread_in_wasm_code) {
    return false;
  }
  thread_in_wasm_code = false;
```

[4] In Chrome, for example, if this function returns false then Chrome would send a crash report.

```
  restore_signal_mask();

  if (acquire_spinlock()) {
    for(code_object in gAllWasmCodeObjects) {
      if (code_object.contains(code_addr)
          && code_object.heap.contains(data_addr) {

        for(instruction in code_object.protected_instructions) {
          if (instruction.addr == code_addr) {
            *target_addr = code_addr + instruction.landing_pad_offset;
            thread_in_wasm_code = true;
            release_spinlock();
            return true;
          }
        }
      }
    }
  }

  thread_in_wasm_code = true;
  release_spinlock();
  return false;
}
```

This function first checks a thread-local variable indicating whether the thread was executing Wasm code (we will call this the TLS check). If it was not, we tell the caller to report a crash.

Next, it restores the signal mask to guard against crashes in the signal handler itself. Because we have cleared the `thread_in_wasm_code` flag, the TLS will fail and this would be reported as a crash.

Then we attempt to acquire the spinlock protecting the fault location table. The normal case for why this would fail is if the `thread_in_wasm_code` flag was not clear. We may also choose to add a timeout here to protect against deadlocks.

Having acquired the spinlock, we search a list of Wasm code objects for one whose code and associated data range include the faulting instruction and the address it was trying to access. If one was found, we pass the address of the instruction’s landing pad to the calling signal handler and return true.

Fault location table

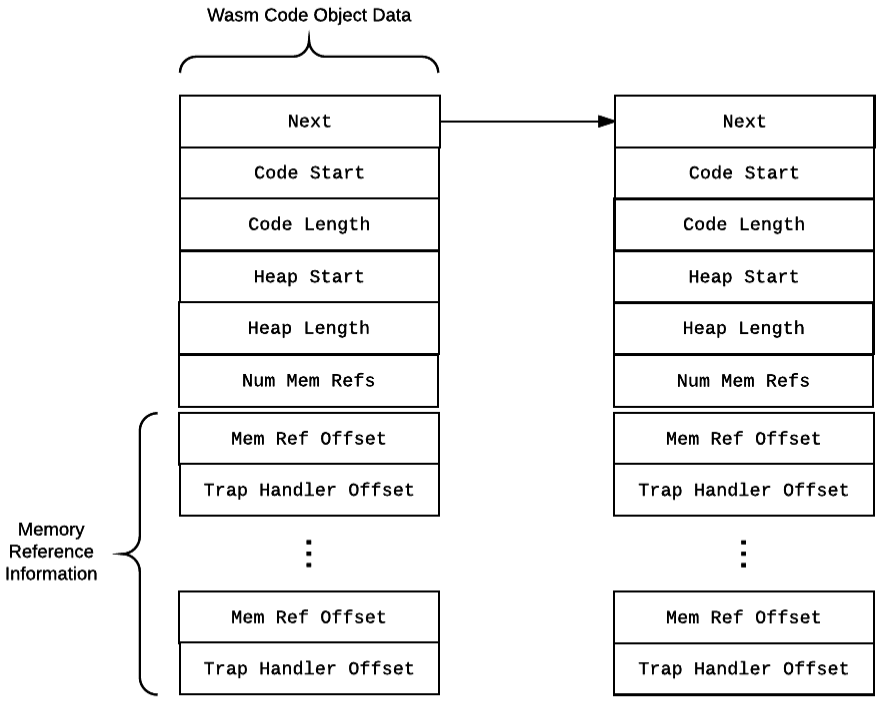
The fault location table is used by the signal handler to determine whether a fault originated for Wasm code, and to give enough information to continue with the JS exception throwing process. We'll consider two versions here: the simple version and an improved version. The simple version contains basically the smallest amount of information that could possibly work, while the improved version contains more information that will allow us to avoid generating too much code.

This table will be kept outside of the GC Heap for several reasons:

1. The GC heap is per-isolate, but the fault handler is per-process.
2. Keeping a separate table reduces the amount of code and data that needs to be audited to ensure the security of this feature.

Later on there is a section that discusses the possibility of implementing this feature in terms of existing data structures on the GC heap.

The diagram below shows the simple version:



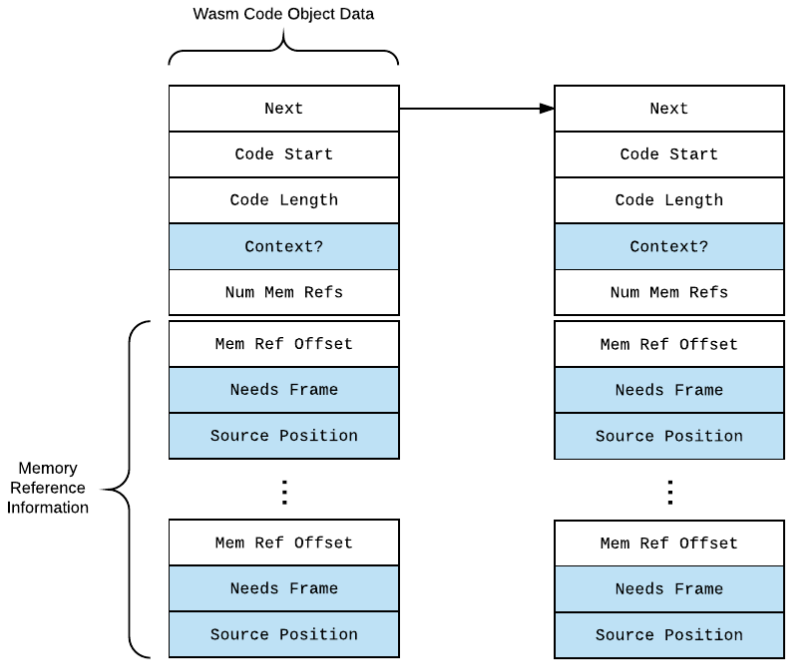
The simple version is linked list with information about the code objects generated by Wasm. The signal handler simply traverses the list and checks whether the faulting program counter

(PC) is within each code object. Each module's entry in the table also includes an array of offsets to memory instructions and offsets to trap code for each memory instruction.

This table will be allocated outside of garbage collected memory. The reason is that the table needs to stay in a well-known location for the signal handler to find it, which is difficult in the presence of a moving GC. A module's entry in the table will be removed when the corresponding module instance is destroyed. Since the GC may move the code object that is pointed to in this table, we will need to either add relocation information to update the table any time the code object is moved or make these changes directly in `Code::Relocate`. In order to simplify this part, we will only have one raw pointer for each object in the table, and any other pointers will be offsets relative to this pointer.

Improved Version

The simple version still requires code to be generated for each memory reference to throw a JS exception. This generated code mostly consists of plumbing some metadata around. In the improved fault location table, we try to store this information in the table instead. The diagram below shows the updated version, with the changes highlighted in blue.



Throwing a Wasm out of bounds exception in JavaScript[5] needs a few parameters: the exception type, a source position, and a context. Using the simple table, these parameters are

[5] Accomplished by calling `Runtime_ThrowWasmError`.

hardcoded in the trap code. In this improved case, we store this data in a table and use a single trap handler to throw the exception.
While the exception type and source position is straightforward, the context is a little less clear. One option is to store the context in the table, as shown in the diagram above. This seems problematic, however, as at the very least the GC will have more bookkeeping to do if the context moves. Another option is to see if we can reliably find a context on the stack.

The current plan is to implement the simple version of the table, since many of the pieces are already prototyped. Once that is done, we can switch to the improved version. The main benefit of the improved table will be significantly less generated code.

Safely Modifying the Table

There are several factors that complicate modifying the fault location table. Modifications are necessary when new Wasm modules are instantiated and destroyed. Some of the complicating factors are:

1. Faults may occur anywhere. Ideally they would only happen in Wasm code, but due to bugs they may occur elsewhere. We need to ensure the Wasm signal handler can always safely determine whether a fault came from Wasm.
2. There can be multiple threads. It's possible that one thread may be compiling a Wasm module while another is executing one. We need to make sure we are not modifying the table when a Wasm fault occurs.
3. There are multiple isolates in a process, but signal handlers are global. V8 is designed so there is minimal shared state between isolates, but we need to allow for Wasm modules in different isolates to safely interact with the fault location table.

See Mark Seaborn's document for some additional hazards involving signal handlers.

To overcome these challenges, the fault location table will be protected by a spinlock. The advantage of using a spinlock is that we can implement it without relying on any library state. The lock should very rarely be contended, so spinning should not be too expensive. Any code that modifies the fault location table will do so while holding the lock, and the signal handler will acquire the lock before accessing the table.

This spinlock introduces the possibility of deadlocks in these ways:

1. The code that is modifying the table faults while holding the lock.
2. The signal handler itself faults.

We will avoid these issues by ensuring that we only take the spinlock while the `thread_in_wasm_code` flag is clear. Because the signal handler immediate exits if the thread was not in Wasm code, it will never attempt to take the lock if the same thread is already holding it.

Alternate Data Structures

The design presented here uses an array of pointers to code object descriptors coupled with a linear search, primarily because of its simplicity.

A desirable feature for this data structure is that descriptors never move in memory until they are destroyed. The reason is that moving entries in the table requires updating the corresponding code object's relocation information so that the garbage collector knows the new location to update when code changes.

If linear time is unacceptable, it shouldn't be too much trouble to sort the array and use a binary search or to use a binary tree or even a hash table instead.

Compiler Changes

The compiler will need to be modified to generate the fault location table, among other things. The fault location table contains physical code addresses, so it cannot be generated until very late in the compiler. However, the Wasm frontend needs to communicate where its memory references are so the code generator knows when to generate this table. We will facilitate this by adding a new `ProtectedLoad` and analogous `ProtectedStore` instruction. These instructions work the same as the usual Load and Store instructions, except that they include extra parameters for the metadata needed for trap handling. The behavior of these instructions differs depending on whether we are using the simple or improved fault location table.

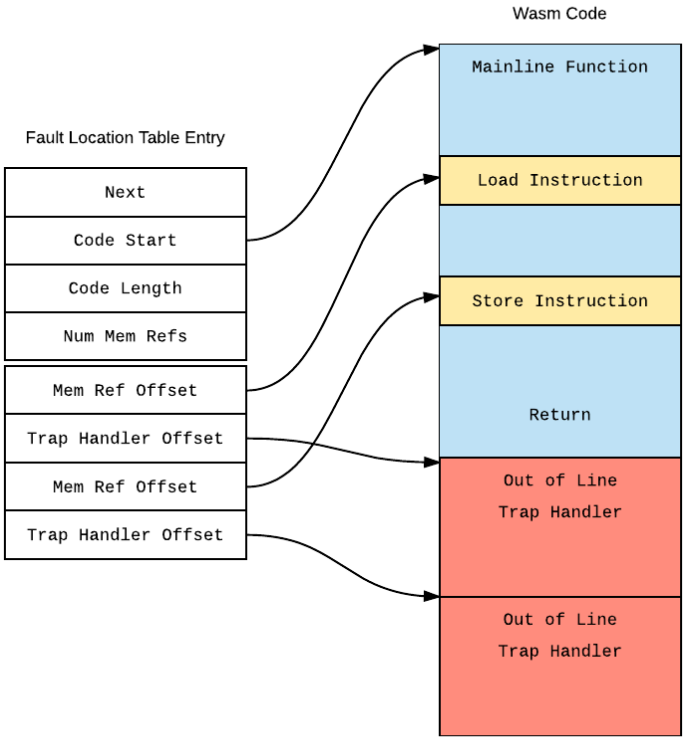
The fault location table is finally written out during the Wasm linking phase, since by this point all of the code addresses have been finalized.

Simple Fault Location Table

In this version, `ProtectedLoad` and `ProtectedStore` take a context parameter, representing the JS context, and a source position parameter, indicating which instruction in the Wasm module caused the fault.

The code generator emits a memory load or store instruction, but also adds and out of line landing pad to the end of the function that throws an exception. This code is essentially the same as was generated for a bounds check failure. Finally, the code generator records the location of the memory reference and an offset in the fault location table so that the signal handler can find the landing pad.

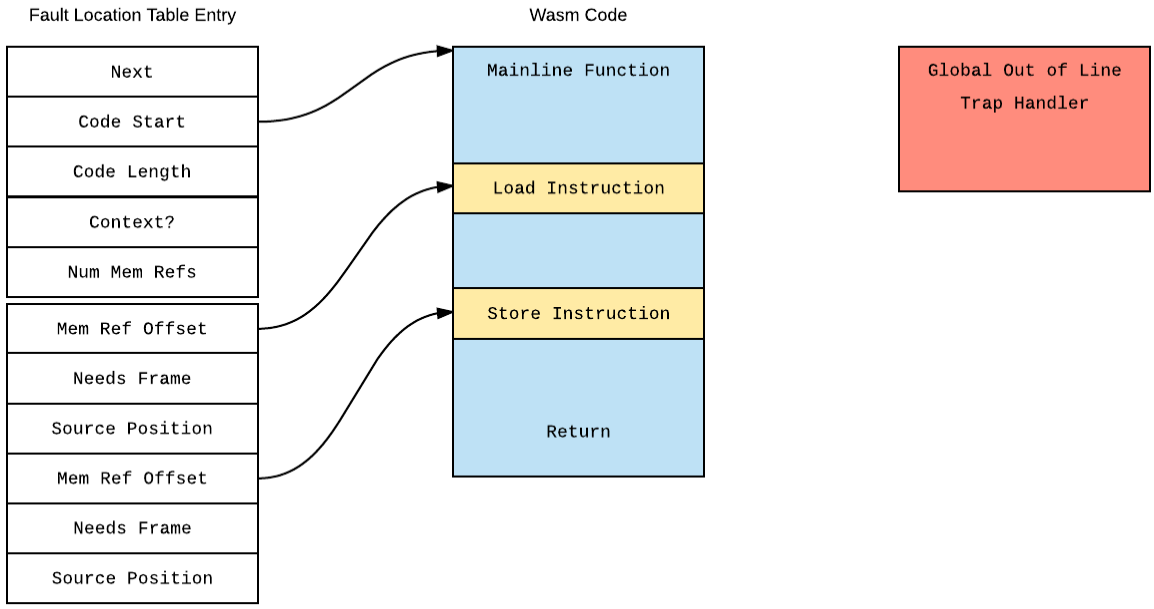
The diagram below shows the relationship between a fault location table entry and the Wasm code.



Each memory reference in the code object has a slot in the table entry, as well as an out of line trap handler/landing pad. In each case, the landing pad consists of code to call `Runtime_ThrowWasmError` with the correct context and source location.

Improved Fault Location Table

The improved fault location table includes enough metadata that we do not have to generate any per-memory reference code. Instead, the code generator records in the fault location table each memory reference and its Wasm source position. There will be a single global trap handler which will read the position information out of the fault location table and throw an exception. This arrangement is shown in the diagram below.



Source Code Layout

The code will be structured to make a clear separation between the rest of V8 and the code that accesses the fault handling data, including the signal handler. This will make it easier to audit the code and make sure the invariants that the signal handler relies on are preserved. We will follow a convention that code is allowed to call into the signal handler-related code, but the signal handler-related code cannot call out (other than to return to the correct landing pad). In addition, the signal handler-related code will be protected by an `OWNERS` file that requires a review from someone on the Chrome Security Team to change.

Security

By the time a segmentation fault happens, there is very little that can safely be assumed about the state of the process memory. As such, this feature is security-sensitive. These security concerns are discussed in detail here. The main technique for mitigating the security threats is to use a sequence of checks progressing from simple to more complex, and designing each layer of checks to fail in favor of sending a crash report rather than handling a bogus out of bounds error.

Testing

Functionality Testing

There will be tests to exercise at least the following behaviors:

1. Valid loads and stores still work.
2. Invalid loads and stores throw an exception.
3. Faults from outside of Wasm code are ignored by the Wasm signal handler and fall back on the previous signal handler.
4. Modules clean up properly when destroyed.

Performance Testing

One of the main goals for making this change is to improve performance. We will evaluate the performance in three categories:

1. Runtime performance of Wasm code.
2. Size of generated Wasm code.
3. No penalty on non-Wasm code (including runtime memory usage).

We will use the v8-perf benchmarks to evaluate the performance. We want to show an increase in performance on the Wasm benchmarks for item (1), a decrease is code size on the Wasm benchmarks for item (2), and no reduction in compile- and runtime performance on the other benchmarks for (3).

Alternative Design: Fault Location Table in GC Heap

Much of the complexity in the design so far involves creating and maintaining the fault location table outside of the garbage collector's heap. Another option is to reuse a lot of the existing functionality in V8. The signal handler would do the following steps:

1. Get the current isolate from thread local storage. If the current isolate is NULL, invoke the previous signal handler.
2. Ask the isolate for the code object associated with the faulting instruction pointer. If none is found, invoke the previous signal handler.
3. Check if the resulting code object is a Wasm object. If not, invoke the previous signal handler.
4. Search for the exception information for the faulting address in the code object's relocation data. If not found, invoke the previous signal handler.
5. Return to the landing pad to throw the Javascript exception.

This approach would require additional relocation information be added that would include the same information about memory references as in the fault location table.

In many ways, this design is simpler, and it is able to reuse a lot of existing, well tested code in V8. However, there are several security concerns.

First, it greatly increases the amount of code we need to audit to ensure the trap handler is safe. Similarly, it couples the trap handler to potentially changing assumptions in other parts of V8. For example, a seemingly innocuous change elsewhere in V8 might lead to the signal handler depending on state that is unreliable during a signal handler.

Secondly, this approach depends on data structures that are near to the code and other V8 or JS heap data. This makes it more likely to be corrupted if something goes wrong.

Alternative Design: Wasm code in a separate GC space

Determining whether a fault occurred in Wasm code and maintaining the fault location table is complicated by the fact that Wasm code is intermingled with other JS code and subject to being moved by the garbage collector. If we could ensure Wasm code never moves then the GC would never have to update the fault location table once it has been created. If we could move the Wasm code to its own part of the heap then determine whether a fault came from Wasm code is simply a matter of checking whether the faulting instruction pointer is within the Wasm code region.

V8's garbage collector already has a notion of spaces. One of these is the large object space, which has a lot of the properties we would want here. Each object in the large object space is a single `mmap`ped object, and therefore never moves once it is created.

This proposal is to add a new space for Wasm code objects. This space would be set up so that the GC does not move objects in the the space. We could also modify the allocator to reserve a large block of address space from which to allocate Wasm objects, thereby simplifying the check for whether a fault was from Wasm code.

Despite these advantages, this change is a rather significant shift in the GC design, basically treating WebAssembly as a special case. Having immovable code objects also makes the heap prone to fragmentation and complicates the allocation bookkeeping. Some of the fragmentation issues could be alleviated by grouping Wasm code objects (i.e. functions) from each module into one large allocation from the Wasm code space.

Alternative Design: All V8-generated code in segregated address range

A less intrusive variant of putting the Wasm code in a separate GC space is to set aside a large chunk of address space for V8-generated code. This would allow us to add another extremely simple check to the signal handler before having to traverse the fault location table. V8 already keeps each isolate's code confined to a 2GB region. In this change, V8 would first reserve a large (say, several terabytes) region of address space and then each isolate would select a 2GB region out of this space for its own generated code.

The communication between isolates would be minimal. A bitmask would probably be needed to track which 2GB code regions are in use. Isolates would use either a lock or atomic operations to set a bit in the bitmask to claim a region, and then clear the bit to release it when the isolate shuts down. Once a region has been claimed, the isolate will be able to continue without any changes.