moz://a

A tour of compiling a WebAssembly.Module

4.29.2021

Ryan Hunt
ryan@mozilla.com

Compiling a WebAssembly.Module

- Input: Module bytecode
- Output:
  - Loaded and linked machine code
  - Metadata about the code
  - Metadata about the module
- Goals:
  - Minimize latency of compile time
  - Maximize quality of compiled code
  - These goals are in tension
- Solution: Two compilers!
  - Baseline: optimized for latency
  - Ion: optimized for quality of compiled code
  - Compile with baseline first, swap in with Ion later

Selecting a strategy

- Not all compilers are always available
  - Platforms may have differing compiler support
  - New proposals may not be available on all compilers
  - Debugging is only supported in baseline
- Tiering is not always available
  - Must be able to compile in the background
- Tiering is not always beneficial
  - Must have enough executable memory for two copies of compiled code
  - Must have large enough module to justify overhead
  - Use heuristics based on size of code section
    - e.g. code bytes * 2.45 = estimated x64 machine code bytes
    - e.g. code bytes / 2100 = estimated ion compile milliseconds
    - Adjust for available parallelism

Streaming compilation

- Minimize latency by compiling functions as they are downloaded
- Decode and validate all sections before code into a ‘module environment’
  - Contains types, memories, globals, tables, etc.
- Estimate and pre-size buffers based off of module size
  - Re-use tiering heuristics
- Compile each function as its definition becomes available
- Merge compiled code into final code buffer

Parallel compilation

- Compile function definitions in parallel
- Depends on a shared immutable ‘module environment’
- Batch function definitions in each compile task
  - Compilation unit is still an individual function
  - Smoothes over overhead of scheduling a task, there can be many tiny functions
- Optimal batch size is *another* heuristic
  - e.g. Batch ~10_000 bytecode’s for baseline
  - e.g. Batch ~1000 bytecode’s for ion
- This may reorder functions in the final buffer, but that’s okay

Finishing the Module

- Generate stubs for interoperating with JS
  - Handle conversion between ABI and value representations
  - Entry stubs allow calling a function from the JS interpreter or JIT
  - Exit stubs allow calling an imported JS function
- Load and link code
  - Allocate executable memory and copy
  - Patch absolute addresses now that final destination is known
  - Patch addresses of runtime dependent data, such as VM functions
  - Write protect and enable execution
- Module code is shareable across instances and threads
- Just finished first tier? Run the whole process again in the background

Tiering up

- How do we switch from baseline to optimized code?
  - Baseline code may be shared across instances and threads!
  - Baseline code may be currently executing on different thread!
- Use a jump-table indexed by function definition
  - Every baseline function prologue loads from jump-table and jumps to address
  - Expensive no-op when optimized code hasn't arrived yet
  - Jumps into body of optimized code when jump-table is patched
  - Optimized and baseline functions must have compatible stack frames
  - Patching jump-table is not-atomic in presence of threads, but safe
- Pitfall: Only can tier up on function boundaries
- Pitfall: Baseline function pointers are not universally revoked
  - Persist in table entries and imported functions
- Pitfall: No freeing of baseline executable memory

Lazy stubs

- We generate entry stubs for each exported function
- Problem: Not every exported function will be called from JS
- Solution: Only generate stubs for a function when grabbed from the exports object
  - Problem: Some notable web content will touch every exported function, but never call them
- Solution II: Only generate stubs when a function is called
  - Problem: Allocating an individual executable page for each function is wasteful
- Tweak: Generate some stubs eagerly (following another *heuristic*)
- How does this interact with tiering?
  - Each tier gets own set of stubs; code pointers are hard-coded in
  - We must observe a locking protocol when generating lazy stubs or tiering up to avoid a race in jump-tables

Code caching (future)

- Why repeat all this work on page reload, if we've already compiled a module?
- Serialize code and metadata
  - Gecko can store optimized representations of network resources in 'alt-data'
  - Must use a Response object in the WebAssembly JS-API
- Deserialize from the 'alt-data' cache on reloads
- Currently unfinished due to some missing pieces in Gecko
  - Hope to finish this soon!

Summary

- Tiering allows us to achieve both low-latency and good code quality
  - The speed of baseline allows WebAssembly to compete with JS in startup time
  - We don’t want anyone to worry about including WebAssembly in their webapp
- Easy streaming/parallelism are enabled due to WebAssembly’s design
  - Lessons learned from JS and other bytecodes
  - Let’s keep that the case!
- Heuristics and performance tuning are still important

moz://a

Thank You