Code Generation for Data Processing

Lecture 11: JIT Compilation and Sandboxing

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

JIT Compilation

- Ahead-of-Time compilation not always possible/sufficient
- "Dynamic source" code: pre-compilation not possible
    - JavaScript, `eval()`, database queries
    - Binary translation of highly-dynamic/JIT-compiled code
- Additional verification/analysis or increased portability desired
    - (e)BPF, WebAssembly
- Dynamic optimization on common types/values
    - Run-time sampling of frequent code paths, allows dynamic speculation
    - Relevant for highly dynamic languages – otherwise prefer PGO[55]

[55] Profile-Guided Optimization; GCC: `-fprofile-generate` to store information about branches/values; `-fprofile-use` to use it

JIT Compilation: Simple Approach

- Use standard compiler, write shared library
- Can write compiler IR, or plain source code
- `dlopen` + `dlsym` to find compiled function

- Example: libgccjit

+ Simple, fairly easy to debug
− Very high overhead, needs IO

JIT: Allocating Memory

- `malloc()` – memory often non-executable
- `alloca()` – memory often non-executable
- `mmap(PROT_READ|PROT_WRITE|PROT_EXEC)` – $W \oplus X$ may prevent this
  - $W \oplus X$: a page must never be writable and executable at the same time
  - Some OS's (e.g. OpenBSD) and CPUs (Apple Silicon) strictly enforce this

- For code generation: map pages read–write
  - NetBSD needs special argument to allow remapping the page as executable
- Before execution: change protection to (read–)execute

JIT: Making Code Executable

- Adjust page-level protections: `mprotect`
  - OS will adjust page tables
  - Typically incurs TLB shootdown
- Other steps might be needed, highly OS-dependent
  - Read manual

JIT: Making Code Executable

- Flush instruction cache
  - Flush DCache to unification point (last-level cache)
  - Invalidate ICache in *all* cores for virtual address range
    - After local flush, kernel might move thread to other core with old ICache
- x86: coherent ICache/DCache hierarchy – hardware detects changes
  - Also includes: transparent (but expensive) detection of self-modifying code
- AArch64, MIPS, SPARC, ... (Linux): user-space instructions
- ARMv7, RISC-V[56] (Linux), all non-x86 (Darwin): system call

- Skipping ICache flush: spurious, hard-to-debug problems

[56] RISC-V has user `fence.i`, but only affects current core

Code Generation: Differences AoT vs. JIT

	Ahead-of-Time	JIT Compilation
Code Model	Arbitrary	Large (or PIC with custom PLT)
Relocations	Linker/Loader	JIT compiler/linker
Symbols	Linker/Loader	JIT compiler/linker may need application symbols
Memory Mapping	OS/Loader	JIT compiler/linker
EHFrame	Compiler/Linker/Loader	JIT compiler/linker register in unwind runtime
Debuginfo	Compiler/Linker/Debugger	JIT compiler register with debugger

- JIT compiler and linker are often merged

JIT: Code Model

- Code can be located anywhere in address space
  - Cannot rely on linker to put in, e.g., lowest 2 GiB

- Large code model: allows for arbitrarily-sized addresses
- Small-PIC: possible for relocations inside object
  - Needs new PLT/GOT for other symbols
- Overhead trade-off: wide immediates vs. extra indirection (PLT)

- Further restrictions may apply (ISA/OS)

JIT: Relocations and Symbols

- JIT compiler must take care of relocations
  - Can try to directly process relocations during machine code gen.
  - Not always possible: cyclic dependencies
  - Option: behave like normal compiler with separate runtime linker
- Code may need to access functions/global variables from application
  - Option: JIT compiler "hard-codes" relevant symbols
  - Option: application registers relevant symbols
  - Option: application linked with `--export-dynamic` and use `dlsym`

JIT: Memory Layout

- *Never* place code and (writable) data on same page
  - $W \oplus X$; and writes near code can trigger self-modifying code detection
  - Avoid many small allocations with one page each
  - But: editing existing code pages is problematic
- Choose suitable alignment for code
  - Page alignment is too large: poor cache utilization
  - ICache cache line size not too relevant, decode buffer size is
    typical value: 16 bytes
  - Some basic blocks (e.g., hot loop entries) can benefit from 16-byte alignment

JIT: .eh_frame Registration (required for C++)

- Unwinder finds .eh_frame using program headers
- Problem: JIT-compiled code has no program headers
- Idea: JIT compiler registers new code with runtime

- libc provides __register_frame and __deregister_frame
  - Call with address of first Frame Description Entry (FDE)
  - Historically also called by init code

JIT: GDB Debuginfo Registration (optional)

- GDB finds debug info from section headers of DSOs
- Problem: JIT-compiled code has no DSO
- Idea: JIT compiler registers new code with debugger

- Define function `__jit_debug_register_code` and global var. `__jit_debug_descriptor`
  - Call function on update; GDB places breakpoint in function
  - Prevent function from being inlined
- Descriptor is linked list of in-memory object files
  - Needs relocations applied, also for debug info

- Users: LLVM, Wasmtime, HHVM, . . . ; consumers: GDB, LLDB

JIT: Linux `perf` Registration (optional)

- perf tracks binary through backing file of `mmap`
- Problem 1: JIT-compiled code has no backing file for its `mmap` region
- Problem 2: after tracing, JIT-compiled code is gone
- Goal 1: map instructions to functions
- Goal 2: keep JIT-compiled code for detailed analysis

- Approach 1: dump function limits to `/tmp/perf-<PID>.map`[57]
  - Text file; format: `startaddr size name\n`
- Approach 2: *needs an extra slide*

[57] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jit-interface.txt

JIT: Linux perf JITDUMP format (optional)

- JIT-compiler dumps function name/address/size/code[58]
  - JITDUMP file: record list for each function, may contain debuginfo
  - File name must be `jit-<PID>.dump`
- JIT-compiler `mmap`s part of the file as executable somewhere
  - Only use: perf keeps track of executable mappings ⇝ mapping is JIT marker, s.t. perf can find the file later
- Need to run `perf report` with `-k 1` to use monotonic clock
- After profiling: `perf inject --jit -i perf.data -o jit.data`
  - Extracts functions from JITDUMP, each into its own ELF file
  - Changes mappings of profile to refer to newly created files
- `perf report -i jit.data` – Profit!

[58] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/jitdump-specification.txt

Compilation Time

- Problem: code generation takes time
  - Especially high-complexity frameworks like GCC or LLVM
- Compilation time of JIT compilers often matters
  - Example: website needing JavaScript on page load
  - Example: compiling database query
- Functions executed once are not worth optimizing
- But: often not known in advance

- Idea: adaptive compilation
- Incrementally spend more time on optimization

Compilation Time: Simple Approach

Caching

- Doesn't work on first execution

Adaptive Execution

- Execution tiers have different compile-time/run-time tradeoffs
  - Bytecode interpreter: very fast/slow
  - Fast compiler: medium/medium
  - Optimizing compiler: slow/fast
- Start with interpreter, profile execution
  - E.g., collect stats on execution frequency, dynamic types, …
- For program worth optimizing, switch to next tier
  - Depends on profile information, e.g. only optimize hot code
  - Compile in background, switch when ready

Adaptive Execution: Switching Tiers

- Switching only possible at compiler-defined points
  - Needs to serialize relevant state for other tier
- Simple approach: only switch at function boundaries
  - Simple, well-defined boundaries; unable to switch inside loop
- Complex approach: allow switching at loop headers/everywhere
  - Needs tracking of much more meta-information
  - All entry points need well-defined interface
  - All exit points need info to recover complete state
  - Severely limits optimizations; all loops become irreducible

- Using LLVM is possible, but not a good fit

Adaptive Execution: Partial Compilation and Speculation

- Observation: even in hot functions, many branches are rarely used
- Optimizing cold code is wasted time(/energy)

- Observation (JS): functions often get called with same data type
- Specializing on structure allows removing string lookup for fields

- Idea: speculate on common path using profiling data
- Add check whether speculation holds; if not, use side-exit
  - Side-exit can be patched later with actual code
- Side-exit must serialize all relevant state for lower tier
  - "Deoptimization"

Sandboxing

- Executing untrusted code without additional measures may harm system
- Untrusted input may expose vulnerabilities

- Goal 1: execute untrusted code without impacting security
  - Code in higher-level representation allows for further analyses but needs JIT compilation for performance
- Goal 2: limit impact potential of new vulnerabilities

- Other goals: portability, resource usage, performance, usability, language flexibility

Approach: Sandbox Operating System as-is

- Idea: put entire operating system in sandbox (“virtual machine”)
- Widely used in practice

- Virtualization needs hardware and OS support
  - CPU has hypervisor mode which controls guest OS;
    offers nested paging, hypercalls from guest OS to hypervisor

+ Good usability and performance
+ Strong isolation
– Rather high overhead on resource usage: completely new OS
– Inflexible and high start latency (seconds)

Approach: Sandbox Native Code as-is

- Idea: strongly restrict possibilities of native code

- Restrict system calls: seccomp
  - Filter program for system calls depending on arguments
- Separate namespaces: network, PID, user, mount, . . .
  - Isolate program from rest of the system
  - Need to allow access to permitted resources
- Limit resource usage: memory, CPU, . . .   cgroups

Approach: Sandbox Native Code as-is

- Frequently and widely used (“container”)

- $+$ Good usability and performance, low latency (milliseconds)
- $+$ Finer grained control of resources
- $\sim$ Resource usage: often completely new user space
- $-$ Weak isolation: OS+CPU often bad at separation
  - Kernel has a fairly large interface, not hardened against bad actors
  - Privilege escalation happens not rarely

Approach: Sandbox Native Code with Modification

- Idea: enforce limitations on machine code
  - Define restrictions on machine code, e.g. no unbounded memory access
  - Modify compiler to comply with restrictions
  - Verify program at load time

- Google Native Client[59], originally x86-32, ported to x86-64 and ARM
- Designed as browser extension
- Native code shipped to browser, executed after validation

[59] B Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *SP*. 2009, pp. 79–93.

NaCl Constraints on i386

- Problem: dynamic code not verifiable
  - ⇒ No self-modifying/dynamically generated code
- Problem: overlapping instructions
  - ⇒ All “valid” instructions must be reachable in linear disassembly
  - ⇒ Direct jumps must target valid instructions
  - ⇒ No instruction may cross 32-byte boundary
  - ⇒ Indirect jumps/returns must be `and eax, -32; jmp eax`
- Problem: arbitrary memory access inside virtual memory
  - ⇒ Separate process, use segmentation restrict accessible memory
- Problem: program can run arbitrary CPU instructions
  - ⇒ Blacklist “dangerous” instructions

NaCl on non-i386 Systems

- Other architectures[60] use base register instead of segment offsets
  - Additional verification required
- Deprecated in 2017 in favor of WebAssembly

+ Nice idea, high performance (5–15% overhead)

∼ Instruction blacklist not a good idea

– Not portable, severe restrictions on emitted code

– High verification complexity, error-prone

[60] D Sehr et al. "Adapting Software Fault Isolation to Contemporary {CPU} Architectures". In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

Approach: Using Bytecode

- Idea: compile code to bytecode, JIT-compile on host
  - Benefit: verification easy – all code generated by trusted compiler
  - Benefit: more portable

- Java applets
- PNaCl: bytecode version of NaCl

+ Fairly high performance, portable

~ Heavy runtime environment
  - Especially criticized for Java applets

– Very high complexity and attack surface

Approach: Subset of JavaScript: asm.js

- Situation: fairly fast JavaScript JIT-compilers present
- Idea: use subset of JavaScript known to be compilable to efficient code
  - All browsers/JS engines support execution without further changes

- asm.js[61]: strictly, statically typed JS subset; single array as heap
- JS code generated by compilers, e.g. Emscripten
- JavaScript has single numeric type, but asm.js supports int/float/double
  - Coercion to integer: `x|0`
  - Coercion to double: `+x`
  - Coercion to float: `Math.fround(x)`

[61] D Herman, L Wagner, and A Zakai. *asm.js*. 2014. 🌐.

asm.js Example

```
var log = stdlib.Math.log;
var values = new stdlib.Float64Array(buffer);
function logSum(start, end) {
  start = start|0; // parameter type int
  end = end|0; // parameter type int

  var sum = 0.0, p = 0, q = 0;

  // asm.js forces byte addressing of the heap by requiring shifting by 3
  for (p = start << 3, q = end << 3; (p|0) < (q|0); p = (p + 8)|0) {
    sum = sum + +log(values[p>>3]);
  }

  return +sum;
}
```

Example taken from the specification

Approach: Encode asm.js as Bytecode

- Parsing costs time, type restrictions increase code size
- Idea: encode asm.js source as bytecode

- First attempt: encode abstract syntax tree in pre-order
- Second attempt: encode abstract syntax tree in post-order
- Third attempt: encode as stack machine

- ... and WebAssembly was born

Approach: Using Bytecode – WebAssembly

- Strictly-typed bytecode format encoding a stack machine
- Global variables and single, global array as memory
- Functions have local variables
  - Parameters pre-populated in first local variables
  - No dynamic/addressable stack space! ⇝ part of global memory used as stack
- Operations use implicit stack
  - Stack has well-defined size and types at each point in program
- Structured control flow
  - Blocks to skip instructions, loop to repeat, if-then-else
  - No irreducible control flow representable

Approach: Use Verifiable Bytecode – eBPF

- Problem: want to ensure termination within certain time frame
- Problem: need to make sure *nothing* can go wrong – no sandbox!

- Idea: disallow loops and undefined register values, e.g. due to branch
  - Combinatorial explosion of possible paths, all need to be analyzed
  - No longer Turing-complete
- eBPF: allow user-space to hook into various Linux kernel parts
  - E.g. network, perf sampling, . . .
- Strongly verified register machine
- JIT-compiled inside kernel

JIT Compilation and Sandboxing – Summary

- JIT compilation required for dynamic source code or bytecode
- Bytecode allows for simpler verification than machine code, but is more compact
- Producing JIT-compiled code needs CPU, OS, and runtime support
- JIT compilers can do/need to do different kinds of optimizations
  adaptive execution is key technique to hide compilation latency
- Sandboxing can be done at various levels and granularities
- Virtualization and containers widely used for whole applications
- Bytecode formats popular for ad-hoc distribution of programs

JIT Compilation and Sandboxing – Questions

- When is JIT-compilation beneficial over Ahead-of-Time compilation?
- How can JIT-compilation be realized using standard compilers?
- How can code be made executable after writing it to memory?
- Why do some architectures require a system call for ICache flushing?
- How can JIT compilers trade between compilation latency and performance?
- Why is sandboxing important?
- What methods of deploying code for sandboxed execution are widely used?