Code Translation

- Interpretation
  - Line-by-line execution of a program
    - If a statement is in a loop, the statement is processed repeatedly
  - Interpreter typically written in a lower level language (C) then compiled --- *making the interpreter itself portable (retargetable)*
  - Great for fast prototyping of new languages/instructions
  - Can be used to define the operational semantics of a language (e.g. Ruby)

- Just-in-time compilation
  - Compile a method the first time you encounter it
  - Save off the binary and “call” its address if executed it again
Compilation

- Dynamic compilation
  - Compilation of methods at runtime at any time & repeatedly
    - Table-base compilation precludes the need for code patching
      - Adds a level of indirection (memory access) for table lookup
    - Can be performed in the background
  - Enables multi-mode (mixed mode) compilation
    - Interpret first, compile when hot (= executed repeatedly)
    - Fast-compile (no opts) first (Just-In-Time compilation)
      - Optimize (perhaps repeatedly) over time
  - Enables adaptive compilation
    - (Re-)compile hot methods and optimize based on their execution profile, behavior
  - Profiling/sampling required to identify when/how to do so
Compilation

- Trace-based
  - Same thing, only for a series of instructions (e.g. a path through a method) -- not an entire method
    - Store *traces* in a **code cache**
  - Link traces together when possible to avoid going back and forth between executing code and the runtime

- Used in
  - Binary translators/instrumentors: Pin, virtualization systems
  - Binary optimizers: Dynamo, Sun Studio, SOLAR, ADORE
  - VMs for high level languages
    - Javascript: Mozilla’s *SpiderMonkey*, SPUR
    - Java: Yeti, HotpathVM, Hotspot TraceJIT, Testerossa, Dalvik
    - Python: PyPy
    - Lua: LuaJIT
Translated Code Blocks

- Emulator translates a block at a time: *dynamic* basic blocks
  - What are static BBs?

- Dynamic BBs - determined by actual flow of executing program
  - Execute the program
  - Form basic blocks and connect them as you discover the edges
  - You CANNOT go back (one pass only!)
Static Basic Blocks versus Dynamic Basic Blocks

**Static Basic Blocks**

```
add . . . BB1
load . . .
store . . .
```

```
loop:
load . . .
ad . . . BB2
store . . .
brcond skip
```

```
load . . . BB3
sub . . .
```

```
skip:
add . . .
store . . . BB4
brcond loop
```

```
add . . . BB5
store . . .
jump indirect
```

```
... . . .
```

**Dynamic Basic Blocks**

```
add . . . BB1
load . . .
store . . .
```

```
loop:
load . . .
ad . . .
store . . .
brcond skip
```

```
load . . .
sub . . .
```

```
skip:
ad . . . BB2
store . . .
brcond loop
```

```
loop:
load . . . BB3
add . . .
store . . .
brcond skip
```

```
skip:
add . . . BB4
store . . .
brcond loop
```

```
... . . .
```
Translated Code Blocks

- Emulator translates a block at a time: *dynamic* basic blocks
- Dynamic BBs - determined by actual flow of executing program
  - Traces are series of dynamic basic blocks
    - Single entry, multiple exit
  - Note that traces can go *across* function call boundaries
    - Essentially *inlining* on-demand
Code Cache

- Translated code blocks (traces) are stored in a code cache for reuse
  - Old, unused blocks are replaced when the cache fills
Code Cache

- Translated code blocks (traces) are stored in a code cache for reuse
  - Old, unused blocks are replaced when the cache fills
- As the translator finds code blocks that are related in terms of control flow
  - A block is the fall through of one in the cache
  - A block is the jump target of one in the cache
  - A block jumps or falls through to a block already in cache

- Links them together in some cases, as discovered
  - When it can avoid jumping back into the interpreter
    - For frequently taken branches, or sequential bb’s
  - Called “chaining”, or “stitching” in our paper
Incremental Predecoding and Tracing

- Translate binary **while** the program is executing (**dynamically**)
  - Predecode or translate new instructions when reached by program
    - *Incrementally*
  - SPC: source (virtual) program counter
  - TPC: target program counter

![Diagram showing the process of incremental predecoding and tracing]

- Control flow
- Data access
- Byte or source code
- Binary traces
Code Cache Management

- **Differences from memory/hw cache**
  - Traces do not have a fixed size
  - Presence and locations of the blocks are dependent on one another b/c of chaining
  - There is no copy of the cached blocks in a `backing store`
Code Cache Management

- **Differences from memory cache**
  - Blocks do not have a fixed size
  - Presence and locations of the blocks are dependent on one another b/c of chaining
  - There is no copy of the cached blocks in a `backing store`

- Two key code cache operations
  - SPC to TPC mapping (find block in code cache given an SPC)
  - Given a TPC in code cache, find the corresponding SPC
    - This is needed to find the source when an exception occurs
Code Cache Management

- **Differences from memory cache**
  - Blocks do not have a fixed size
  - Presence and locations of the blocks are dependent on one another b/c of chaining
  - There is no copy of the cached blocks in a `backing store`

- Two key code cache operations
  - SPC to TPC mapping (find block in code cache given an SPC)
  - Given a TPC in code cache, find the corresponding SPC
    - This is needed to find the source when an exception occurs

- Practical implementations limit the size of the code cache
  - We need a **replacement** policy for when/if cache fills
Code Cache Replacement Policies

- Least Recently Used (LRU)
  - Good idea, but hard to implement
    - Requires extra memory access per block to keep track
    - Unlinking gets complicated
      - All pointers into removed block must be updated to point to interpreter
      - Can use back pointers
    - Can lead to fragmentation (holes where nothing fits)
Code Cache Replacement Policies

- Least Recently Used (LRU)
  - Good idea, but hard to implement
    - Requires extra memory access per block to keep track
    - Unlinking gets complicated
      - All pointers into removed block must be updated to point to interpreter
      - Can use back pointers
    - Can lead to fragmentation (holes where nothing fits)

- Flush when full

- Preemptive flush
  - Flush when program behavior changes
Code Cache Replacement Policies

- Least Recently Used (LRU)
  - Good idea, but hard to implement
    - Requires extra memory access per block to keep track
    - Unlinking gets complicated
      - All pointers into removed block must be updated to point to interpreter
      - Can use back pointers
    - Can lead to fragmentation (holes where nothing fits)

- Flush when full

- Preemptive flush
  - Flush when program behavior changes

- Fine-grained FIFO
  - Manage code cache as circular buffer
  - Must still maintain backpointers to keep track of chaining
Code Cache Replacement Policies

- Least Recently Used (LRU)
- Flush when full
- Preemptive flush
- Fine-grained FIFO
- Course-grained FIFO
  - Partition cache into large blocks, each managed via FIFO
  - Simplifies backpointer problem b/c only cross-block pointers must be maintained
    - Backpointers point to blocks, no backpointers within blocks
Javascript Implementation

- Interpreter/runtime in browser (Node.js outside of browser)
- Lookups done by name at runtime b/c of dynamic typing -- variables have no types!
  - Variable access -- Lookup at runtime -- high overhead
  - Method calls -- Dynamic dispatch -- high overhead
Javascript Implementation

- Interpreter/runtime in browser (Node.js outside of browser)
- Lookups done by name at runtime b/c of dynamic typing -- variables have no types!
  - Variable access -- Lookup at runtime -- high overhead
  - Method calls -- Dynamic dispatch -- high overhead

- Complications
  - Inheritance – lookup name in table, then walk hierarchy doing the same
    - Local hash tables and global hashtables
  - Fields/methods can be added dynamically via use or call
    - Must always do the lookup to make sure you don’t miss something
Javascript Implementation

- Prototypes vs classes; prototype inheritance vs classical
  - See related readings for more on this & automatic type inference for JS

- Unboxing: decoding an object to determine its type / layout
  - High overhead

- Fat/high-level bytecodes (much work done in runtime for each)
  - Like superinstructions – only at the bytecode level
  - Unlike Java/GForth, similar to Python/Ruby – low dispatch rate
  - So traditional interpreter opts that target the dispatch loop have little impact
Today’s Paper

- Trace-based compilation (why? How?)
  - Mixed mode
  - Two assumptions
    - Most time spent in hot loops
    - Hot loops are mostly type-stable
  - Traces are extended bb’s – they have multiple exits
    - Single entry (at the top)
      - No joins within/into (**single entry**)
        - Entry only through the top
        - Makes optimizing much simpler
    - Functions inlined (a bit complex, b/c we have to still record frames)
      - Trace **across** function boundaries
Today’s Paper

- Trace-based compilation (why? How?)
  - Mixed mode

- Contributions
  - Focus on tracing loops only
    - Nested trace trees
    - Why? How does an outer get to an inner trace?
    - Option1: trace inner (hot first), continue tracing through outer back to inner (code duplication for outer for every side exit and type)
      - Will overflow the code cache (fixed-size memory area for traces)
    - Option2: give up and only trace inner’s
  - **New option**: keep track and separate inner and outer; have outer “call” inner
Today’s Paper

- Trace-based compilation (why? How?)
  - Mixed mode
- Contributions
  - Focus on tracing loops only
    - Nested trace trees
    - Why? How does an outer get to an inner trace?
  - Real implementation
- What loop is considered hot?
TraceMonkey State Machine

- Interpret Bytecodes
  - loop edge
  - cold/blacklisted loop/exit
- Monitor
  - compiled trace ready
  - hot loop/exit
  - finish at loop header
- Record LIR Trace
  - abort recording
- Compile LIR Trace
- Enter Compiled Trace
  - loop edge with same types
- Execute Compiled Trace
  - side exit, no existing trace
  - side exit to existing trace
- Leave Compiled Trace

Symbol Key:
- Overhead
- Interpreting
- Native
Sieve example in Javascript, TraceMonkey LIR for Line 5 – tests/side exits are guards

```javascript
for (var i = 2; i < 100; ++i) {
  if (!primes[i])
    continue;
  for (var k = i + i; i < 100; k += i)
    primes[k] = false;
}
```

Assumptions: primes is an Array type, store to primes[k] completes without error

```lir
v0 := ld state[748] // load primes from the trace activation record
st sp[0], v0        // store primes to interpreter stack
v1 := ld state[764] // load k from the trace activation record
v2 := i2f(v1)      // convert k from int to double
  st sp[8], v1     // store k to interpreter stack
  st sp[16], 0     // store false to interpreter stack
v3 := ld v0[4]     // load class word for primes
v4 := and v3, -4    // mask out object class tag for primes
v5 := eq v4, Array  // test whether primes is an array
  xf v5            // side exit if v5 is false
v6 := js_Array_set(v0, v2, false) // call function to set array element
v7 := eq v6, 0     // test return value from call
  xt v7            // side exit if js_Array_set returns false.
```
X86 for Line 5 in the Sieve example

-- Guards are the *'d instructions

```assembly
mov edx, ebx(748)   // load primes from the trace activation record
mov edi(0), edx     // (*) store primes to interpreter stack
mov esi, ebx(764)   // load k from the trace activation record
mov edi(8), esi     // (*) store k to interpreter stack
mov edi(16), 0      // (*) store false to interpreter stack
mov eax, edx(4)     // (*) load object class word for primes
and eax, -4         // (*) mask out object class tag for primes
cmp eax, Array      // (*) test whether primes is an array
jne side_exit_1     // (*) side exit if primes is not an array
sub esp, 8           // bump stack for call alignment convention
push false           // push last argument for call
push esi             // push first argument for call
call js_Array_set    // call function to set array element
add esp, 8           // clean up extra stack space
mov ecx, ebx         // (*) created by register allocator
test eax, eax        // (*) test return value of js_Array_set
je side_exit_2       // (*) side exit if call failed
...
side_exit_1:
mov ecx, ebp(-4)     // restore ecx
mov esp, ebp         // restore esp
jmp epilog           // jump to ret statement
```
Compiler Optimizations and Blacklisting

- **Optimizations**
  - Floating point to integer emulation on non-fp architectures
  - CSE
  - Simplification, constant folding, strength reduction
  - Source-language optimizations (Double -> int)
  - Dead stack stores, call stack stores, code elimination

- **Blacklisting**
  - Some traces won’t finish - won’t reach header (exceptions)
    - So recording them is a waste
    - Set a counter so that you don’t record them for awhile
    - Try again, then if untraceable, mark in bytecode (bad loop header)
    - Trace lookup in blacklist is time consuming
Static Single Assignment Form

• LIR of a function (intermed. repr.)
  ■ Simplifies program analysis and optimization
  ■ Static Single Assignment (SSA) form of a program makes information about variable definitions and uses explicit

• A program is in SSA form if it satisfies:
  ■ each definition (def) has a distinct name; and
  ■ each use refers to a single definition.

• To make this work, the compiler inserts special operations, called $\phi$-functions, at points where control flow paths join (block E above)
Static Single Assignment Form

• LIR of a function (intermed. repr.)
  - Simplifies program analysis and optimization
  - Static Single Assignment (SSA) form of a program makes information about variable definitions and uses explicit

• A program is in SSA form if it satisfies:
  - each definition (def) has a distinct name; and
  - each use refers to a single definition.

• To make this work, the compiler inserts special operations, called $\phi$-functions, at points where control flow paths join

  - Although this is a key part of SSA, traces contain NO joins
    - In TraceMonkey
Register Allocation

- Linear scan over the instructions in a trace
  - From last to first, greedily
    
    v3 = add v1, v2

- Spilling
  - Heuristic: consider all values in registers at this point (for the instructions after this one)
    Find oldest register-carried value across them:
    - Consider values in registers for instruction after this one
    - Look at instructions before this instruction
      - Find last instruction that refers to (uses) each value
      - Spill the value that has the minimum (smallest index) instruction
    - Copy a value to/from memory (runtime stack location) when needed
      - Insert restore code just after this instruction
      - Insert spill code just after last use (a previous instruction)
Other Issues and Results

- Calling external functions
  - C-function: Foreign Function Interface (FFI)
- Access to global variables, exceptions, to the call stack
  - Aborts trace, but rarely occurs (note this when you code!)

Findings

- 9 of 26 benchmarks better than G8 and SFX (all are better than SpiderMonkey interpreter-only baseline)
  - Both G8 (hot method compilation) and SFX (opt’d interpretation)
  - Footprint not measured
Figure 10. Speedup vs. a baseline JavaScript interpreter (SpiderMonkey) for our trace-based JIT compiler, Apple’s SquirrelFish Extreme inline threading interpreter and Google’s V8 JS compiler. Our system generates particularly efficient code for programs that benefit most from type specialization, which includes SunSpider Benchmark programs that perform bit manipulation.