NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory

Thomas Shull, Jiho Choi, María J. Garzarán, Josep Torrellas
University of Illinois at Urbana-Champaign

February 19, 2019
HPCA-25 Session 5B

JavaScript Performance is Lagging

- JavaScript is widely used in Industry
  - Websites
  - Server-Side Applications
  - Desktop Applications
- Performance has greatly improved over the last decade
  - 10x improvements since 2008
- Performance still lags behind C/C++

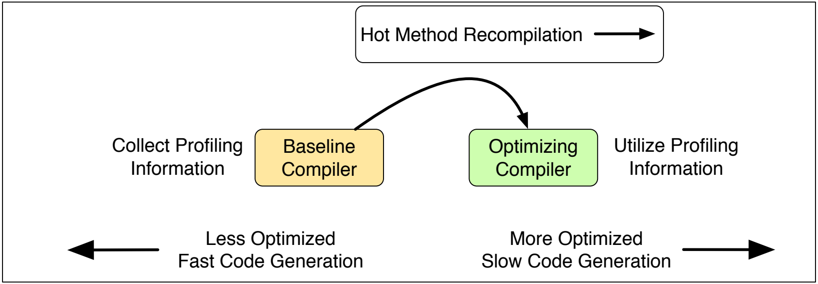
Attaining High Performance in JavaScript

- Two important performance techniques for fast JavaScript execution:
  - Multi-Tiered Just-in-Time (JIT) Compilation
  - Code Specialization
- Our work identifies bottlenecks in current approach
  - These two techniques require:
    - Many checks
    - Metadata (called Stack Map Points) which restrict compiler optimizations
  - Our work's contribution is to reduce this overhead

Multi-Tiered JIT Compilation

- Conflicting compiler goals:
  - Fast start-up time
  - High quality code generation
- Solution: use multiple compilers
  - Lower tier compilers (used initially):
    - Generate code quickly
  - Higher tier compilers (used later):
    - Only recompile “hot” code regions (i.e., methods frequently invoked)

Multi-Tiered JIT Compilation Process



JavaScript Has Complicated Language Semantics

- JavaScript is difficult to optimize due to its many control paths
- Example: $x + y$

Type_x	Type_y
Int	Int
Double	Double
Double	Int
Array	String
Date	Array

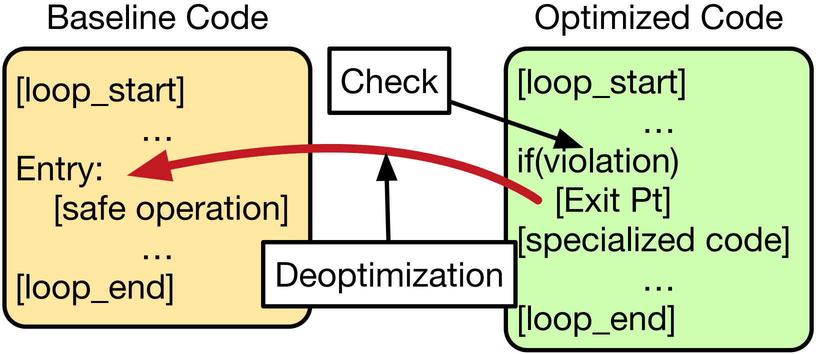
Code Specialization

- Solution: Code specialization – optimize code for the expected behaviors
  - Assume arithmetic operation's operands will be of a specific type
  - Assume array accesses will be inside existing bounds
- Leverage multi-tiered JIT compilation for accurate specializations
  - Lower tiers observe which behaviors occur
  - Higher tiers utilize profiling results to specialize the code and make it efficient

Handling Code Specialization

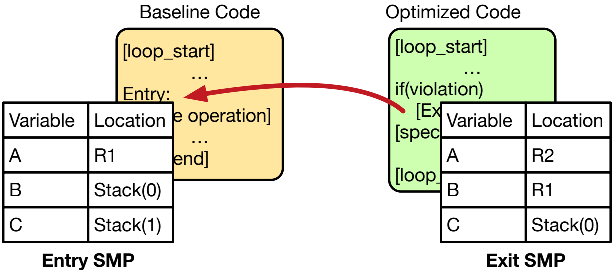
- Code specialization:
  - \+ Greatly improves performance
  - – Unsafe: no guarantee that assumptions made will be always true
- Solution: Deoptimization – jump back to “safe” version of code if assumptions are violated
  - “Safe” code covers all possible JavaScript behaviors
- How? Insert Checks and **Deoptimization Exit Points** to ensure correct execution
  - Deoptimization Exit Points: places where execution can jump out of code

Deoptimization Exit Points



Handling Deoptimization

- Deoptimization requires consistent program state at the Exit Point and destination
- Register allocator may assign different locations for variables in each version of generated code
- **Stack Map Points (SMPs)** contain mapping of variables to registers and stack at a given point



Variable	Location
A	R1
B	Stack(0)
C	Stack(1)

Variable	Location
A	R2
B	R1
C	Stack(0)

Recap: Current JavaScript Optimization Techniques

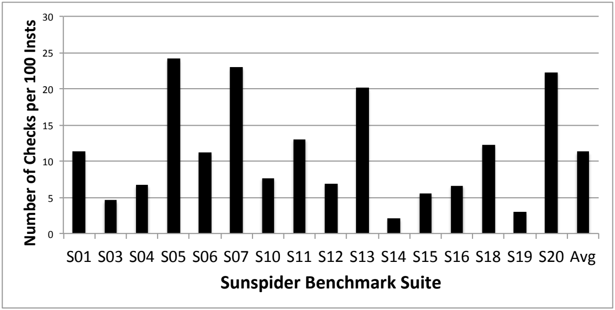
- Two techniques used to improve JavaScript performance
  - Multi-Tiered JIT Compilation
  - Code Specialization
- These techniques require extra safeguards:
  - Checks to verify code specializations are correct
  - SMPs needed to perform deoptimizations

Contribution: NoMap

- Discover the code specialization checks are very frequent in optimized code
- Discover that Stack Map Points (SMPs) significantly inhibit the performance of JavaScript
  - Hamper compiler optimizations by preventing code movement
  - Associated with checks – every deoptimization exit point has a SMP
- Propose to use Hardware Transactional Memory (HTM) to reduce check and SMP overhead
- Improve native performance of JavaScript by 16.7% using an industrial-strength compiler

Frequency of Checks

- We instrumented Safari's optimized compiler to determine frequency of code specialization checks
- Used Pin to measure the number of checks per 100 instructions



1 check for every 11.3 instructions

Overhead of Checks and SMPs

- Checks add instruction overhead
  - Must verify assumptions made
- SMPs stiffle conventional compiler optimizations
  - Program state must be consistent at Deoptimization Exit Point and destination
  - Hard to reorder code across SMPs
    - Would have to then redo/undo operations
  - SMPs very frequent – One SMP for each check

Frequency of Deoptimizations

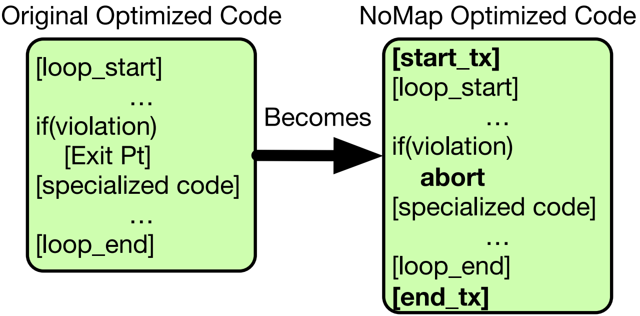
- Checks & SMPs are needed to safeguard against incorrect code specializations
- Very rarely are assumptions violated
- However, cannot remove them due to remote chance of deoptimization

Insight: Use Hardware Transactional Memory

- Idea: Leverage Hardware Transactional Memory (HTM)
- Surround check & SMP heavy codes with transactions
- Inside TM region one can:
    - Remove SMPs ⇒ enhances efficiency of conventional compiler optimizations
    - Compiler leverages HTM to reduce number of checks
        - Combine array-bounds checks
        - Eliminate overflow checks

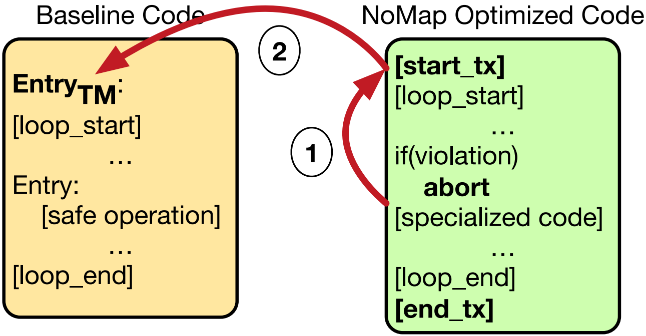
Eliminating SMPs

- Within transactions, replace Deoptimization Exit Points with aborts:
  - SMPs no longer needed



Check Failure (Deoptimization) Control Flow

Suppose deoptimization is necessary:



Combining Array-bounds Checks

- Using HTM, bounds checks can be moved out of loops

Baseline Code

Entry$_{TM}$:
[loop_start]
...
Entry:
    [safe operation]
...
[loop_end]

NoMap Optimized Code

[start_tx]
[loop_start]
...
~~**if(!in_bounds(data, idx))**~~
    ~~**abort**~~
sum += data[idx]
...
[loop_end]
[end_tx]

Eliminating Overflow Checks

- Using HTM, check for overflow only at transactional commit

Baseline Code

Entry$_{\mathbf{TM}}$:
[loop_start]
…
Entry:
    [safe operation]
…
[loop_end]

NoMap Optimized Code

[start_tx]
[loop_start]
…
sum += a
~~if(overflow(sum))~~
    ~~abort~~
…
[loop_end]
[end_tx]

NoMap's Light Hardware Requirements

- Light TM hardware
  - Only buffer speculative writes (not reads)
  - Transaction exit need not stall for write buffer drain
- Sticky Overflow Flag
  - Reset at transaction start
  - Automatically checked at transaction end
- Similar to support in IBM POWER 8/9
  - Rollback-Only Transaction (ROT) mode
- Much simplier than traditional HTM

Native Evaluation Environments

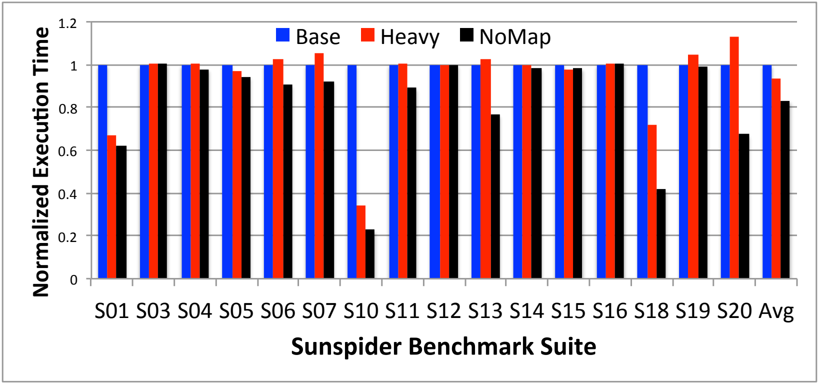
- Lightweight HTM: Emulated NoMap Support
  - Add fence on TX Start
  - Add short stall on TX End (for clearing Speculative Tags)
  - Performance verified against IBM POWER 8 System
- Heavyweight HTM: NoMap targeting Intel's Restricted Transactional Memory (RTM)
  - Many performance drawbacks
    - Monitors both *read* and write set
    - TX write footprint must fit in L1
    - Expensive commit

Evaluation Configurations

- We evaluate NoMap on the SunSpider and Kraken Benchmark Suites

Architecture	Explanation
Base	Unmodified compiler. No transactions.
Heavy	Using Heavyweight HTM: * Does not combine overflow checks.
NoMap	Proposed design. Using Lightweight HTM

Execution Time



- **Heavy improves execution time by 6.5%**
- **NoMap improves execution time by 16.7%**

Conclusions

- Identified the high frequency of checks and SMPs as a primary JavaScript performance bottleneck
- Proposed using HTM to eliminate this bottleneck
  - Convert SMPs to aborts $\Rightarrow$ compiler optimizations more effective
  - Combined array-bounds checks
  - Eliminated overflow checks via the Sticky Overflow Flag
- Improved native JavaScript performance by 16.7% by applying NoMap to an industrial-strength compiler

NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory

Thomas Shull, Jiho Choi, María J. Garzarán, Josep Torrellas
University of Illinois at Urbana-Champaign

February 19, 2019
HPCA-25 Session 5B