ShortCut: Architectural Support for Fast Object Access in Scripting Languages

Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ISCA 2017
Overheads of Scripting Languages

• Scripting languages are widely used
  – Designed for productivity
  – Dynamic type system: Difficult to generate efficient code

• Many overheads include:
  – Slow interpreter
  – Dynamic type check (e.g., integer or float?)
  – **Slow object access** ← Our Focus
  – Garbage collection
Traditional Languages: Fast Object Access

- **Type declaration** tells compiler the shape of an object
  - Properties and their offsets within an object

```c
struct T {
    int x;
    int y;
};

int get_x(T o) {
    return o.x;
}

T a;
get_x(a);
```

Compile:

```
[l]d R, &o[0]
```

*What if there is no type declaration?*
Scripting Languages: Slow Object Access

- **No type information** available ahead of execution
- A naïve approach requires an expensive dictionary lookup

```javascript
function get_x(o) {
    return o.x;
}

a = {}; // new obj
a.x = 0;
get_x(a);
```

Object shape unknown

Compile

[Code for get_x]
call Runtime

Dictionary lookup for every object access is prohibitively expensive.
Scripting Languages: Slow Object Access

- Current software solution: Dynamically generate a specialized handler for each object type and reuse it for the same type later

```javascript
function get_x(o) {
    return o.x;
}

a = {}; // new obj
a.x = 0;
get_x(a);
get_x(a);

b = {}; // new obj
b.y = 1; b.x = 2;
get_x(b);
get_x(b);
```

Compile

[Code for get_x]
call Dispatcher

[Code for Dispatcher]
if obj type is previously seen:
    jump to a specialized handler
else:
    jump to Runtime

[Handler_a]
1d R_D, &o[0]
ret R_D

[Handler_b]
1d R_D, &o[1]
ret R_D

Program → Dispatcher → Handler
Scripting Languages: Slow Object Access

- This software code structure is called **Inline Cache (IC)**

- We find that IC still has major overheads:
  - **At least 14 instructions per dispatcher invocation** to choose a handler
  - **22%** of total instructions executed are in the dispatcher
  - **46%** of branch mispredictions are in the dispatcher
Contributions

• Characterization of performance bottlenecks in IC operation in a state-of-the-art JavaScript engine

• Proposed two levels of HW/SW optimization to improve the efficiency of object access in scripting languages

• Implemented our proposal in multi-tier Google V8 compiler and reduced the average execution time:
  – by 30% running under the base tier
  – by 11% with the advanced tier enabled
Outline

• Motivation and background
• Contribution

• Our solution: ShortCut
  – Key idea
  – Design
  – Compiler integration

• Evaluation
• Summary
Key Idea

• **Plain ShortCut** transforms the call to the dispatcher into a call to the correct handler

• **Aggressive ShortCut** transforms the call to the dispatcher into an actual object access in place
A program calls the dispatcher at an object access site
- A BTB entry holds the dispatcher address
- The dispatcher chooses a handler
A program directly calls a handler at an object access site

- **IC_Call** takes an additional operand: object type
- A BTB entry holds a handler address

A new hardware table, **ICTable**, validates the BTB prediction

- Falls back to the dispatcher upon ICTable miss

**Plain ShortCut Design**

![Diagram](image-url)
Aggressive ShortCut Design

- A program performs an object access in place
  - **IC_Load** and **IC_Store** perform load and store, respectively
  - A BTB entry holds the next address
- Extend ICTable to store the offset of the property to access
- The property of the object is read or written using the offset value from ICTable.
Compiler Integration

• Replace the call to the dispatcher with the new instructions
  – \texttt{IC\_Call} in Plain ShortCut
  – \texttt{IC\_Load} or \texttt{IC\_Store} in Aggressive ShortCut

• Load the incoming object type and pass as an operand to \texttt{IC\_Call/Load/Store}

• More details are in the paper
Outline

• Motivation and background
• Contribution
• Our solution: ShortCut

• Evaluation
  – Experimental setup
  – Simulation Results

• Summary
Experimental Setup

- Modified Google V8 JavaScript JIT compiler
  - Implemented in the base tier of the compiler
  - Application to the advanced tier is future work

- Extended SniperSim to model ShortCut hardware

- Benchmark Suites: Octane and SunSpider
Evaluated Configurations

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Baseline: Unmodified V8</td>
</tr>
<tr>
<td>I</td>
<td>Ideal: Baseline with perfect BTB for the IC</td>
</tr>
<tr>
<td>PS</td>
<td>Plain ShortCut</td>
</tr>
<tr>
<td>AS</td>
<td>Aggressive ShortCut</td>
</tr>
</tbody>
</table>

- Ideal (I) serves as an upper bound for BTB
- Aggressive ShortCut is currently limited to a simple form of \texttt{IC\_Load}
Instruction Count Breakdown

- Octane benchmark result
  - SunSpider benchmark result is in the paper

- On average 26% of total instructions executed in the dispatcher running under the base tier

- Plain ShortCut reduces the average instruction count:
  - by 21% running under the base tier
  - by 8% with the advanced tier enabled
Branch Prediction

- 54% of branch mispredictions are in the dispatcher
- ShortCut reduces branch MPKI by 41% (from 14.4 to 8.5)
Overall Performance Improvement

- Plain ShortCut reduces the average execution time
  - by 37% running under the base tier
  - by 13% with the advanced tier enabled

- ShortCut outperforms perfect BTB ($I$).

- Aggressive ShortCut delivers marginal improvement over Plain ShortCut

ShortCut: Architectural Support for Fast Object Access in Scripting Languages
Conclusions

• Two main sources of slow object access in scripting languages
  – Instructions executed in the dispatcher
  – Hard-to-predict branches in the dispatcher

• Two levels of HW/SW optimization to accelerate object access
  – **Plain ShortCut**: Skips the dispatcher execution
  – **Aggressive ShortCut**: Skips even the handler execution
    • Emulates fast object access in traditional languages

• Implemented our solution in Google V8 and improved execution
  – by 30% running under the base tier
  – by 11% with the advanced tier enabled
ShortCut: Architectural Support for Fast Object Access in Scripting Languages

Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ISCA 2017
Multi-tier Compiler

• Even if the advanced tier is enabled, a significant fraction of the execution of programs uses code generated by the base tier:
  – It takes a while for the advanced tier to engage
  – If any assumption made by any optimization fails (e.g., unexpected object type is encountered), the base tier is re-invoked
  – There are some functions in a program that the advanced tier abstains from compiling, often based on heuristics; they include `eval` constructs and other complicated cases

• Execution time is short
Aggressive Shortcut

• IC_Store is not supported

• We note that of all the handler invocations
  – 75.7% are loads
    • 15.1% of them are covered by Aggressive ShortCut
  – 24.3% are stores
    • 17.2% of them can be covered by Aggressive ShortCut
Future Work

- Full implementation of Aggressive ShortCut
  - IC_Store
- Application to the advanced tier
  - using Aggressive ShortCut
- Application to interpreters
ISA

• Conventional: \texttt{Call \textit{Addr}_{Dispatcher}}

• Plain ShortCut: \texttt{IC\_Call \textit{Addr}_{Dispatcher} \ R_{Type}}
  – If it hits in ICTable, call the handler
  – Otherwise, call the dispatcher

• Aggressive ShortCut: \texttt{IC\_Load/\textit{Store} \textit{Addr}_{Dispatcher} \ R_{type}}
  – If it hits in ICTable, perform a load/store
  – Otherwise, call the dispatcher

• Both: \texttt{IC\_Update \ R_{PC} \ R_{Type}}
  – Installs an entry in ICTable and updates BTB

• Both: \texttt{IC\_Flush}
  – Flushes ICTable
Experimental Setup: Processor Architecture

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>4-wide out-of-order, 128-entry ROB, 2.66GHz</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Hybrid predictor</td>
</tr>
<tr>
<td></td>
<td>BTB: 4K entries, 4-way, RR replacement, 96b/entry</td>
</tr>
<tr>
<td></td>
<td>Branch misprediction penalty: 15 cycles</td>
</tr>
<tr>
<td>ICTable</td>
<td>512 entries, 4-way, RR replacement, 145b/entry</td>
</tr>
<tr>
<td>Caches</td>
<td>L1-I: 32KB, 4-way, 4-cycle latency</td>
</tr>
<tr>
<td></td>
<td>L1-D: 32KB, 4-way, 4-cycle latency</td>
</tr>
<tr>
<td></td>
<td>L2: 256KB, 4-way, 12-cycle latency</td>
</tr>
<tr>
<td></td>
<td>L3: 8MB, 16-way, 30-cycle latency</td>
</tr>
<tr>
<td></td>
<td>Block size: 64B, LRU replacement</td>
</tr>
<tr>
<td>Memory</td>
<td>120-cycle minimum latency</td>
</tr>
<tr>
<td></td>
<td>16 DRAM banks</td>
</tr>
</tbody>
</table>
Sensitivity Study

- **PS** outperformed **B** even with only 16 ICTable entries

- 512-entry ICTable is about 9 KB
Instruction Count Breakdown

ShortCut: Architectural Support for Fast Object Access in Scripting Languages
Branch MPKI Analysis

- ShortCut reduces branch MPKI from 10.8 to 6.9 running under the baseline compiler

- ShortCut avoids the hard-to-predict branch in the dispatcher
Overall Performance Improvement

ShortCut: Architectural Support for Fast Object Access in Scripting Languages