Garbage Collector Write Barriers

Calum Snowdon, u6044174
COMP3770
Supervised by Steve Blackburn & Ben Titzer
Portable Garbage Collectors

• Garbage Collectors (GCs) are responsible for memory management in language runtime environments.
  ➢ Reclaiming un-used memory, compaction etc.

• We’re interested in writing garbage collectors which are separate from the runtime, so people don’t have to write a new GC for every language.

• Problems arise.
Compiling Languages with GCs

• Compiler needs to inject GC write barriers into compiled code – but everything is written in a different language.
Desirable Qualities

• Performant
  ➢ Compiled code should not compromise on speed.

• Portable
  ➢ Should be able to re-use GC without substantial modification to either codebase.

• Safe
  ➢ Should be sufficiently easy to integrate without errors.
Existing Solutions

• All existing implementations have undesirable qualities.
  ➢ Have compiler insert cross-language function calls.
    ▪ Easy, portable, safe, but slow.
  
  ➢ Hard-code write barriers into the compiler.
    ▪ Fast, but introduces dependency between compiler and GC and has poor safety guarantees (code duplication).
  
  ➢ Implement GC in the language to be compiled – can just insert barriers into source code!
    ▪ Fast and safe, but not portable to other compilers.
# Existing Solutions

<table>
<thead>
<tr>
<th></th>
<th>Performant</th>
<th>Portable</th>
<th>Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard-Code</td>
<td>✓</td>
<td>~</td>
<td>x</td>
</tr>
<tr>
<td>Implement in same Language</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Insert Function Calls</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Our Work

• This semester we’ve proposed & investigated a compromise-free solution.

<table>
<thead>
<tr>
<th></th>
<th>Performant</th>
<th>Portable</th>
<th>Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard-Code</td>
<td>✓</td>
<td>?</td>
<td>✗</td>
</tr>
<tr>
<td>Implement in same Language</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Insert Function Calls</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Our Solution</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Our Proposed Architecture

• Choose an intermediate form to represent the write barrier in.

• Garbage Collector:
  ➢ Implement write barriers in the garbage collector.
  ➢ Compile write barrier into an intermediate form, expose to compiler.

• Compiler:
  ➢ Implement a decoder which translates the intermediate form into the compiler’s internal data format.
Analysis

Garbage Collector

Write Barrier

Write Barrier (intermediate)

Write Barrier (compiler IR)

Compiler
Analysis

No duplication of write barrier!
Maintain safety guarantees of GC!
Analysis

No duplication of write barrier!
Maintain safety guarantees of GC!

Indirection removes dependency!
Analysis

Garbage Collector

- Write Barrier

Indirection removes dependency!

No duplication of write barrier!
Maintain safety guarantees of GC!

Write Barrier (intermediate)

Write Barrier (compiler IR)

Compiler

Barrier expressed in compiler IR, can optimize!
Drawbacks

• GC designers have to write a compiler into the intermediate form.
  ➢ Could get away with hardcoding it, but this would be more error-prone (code duplication).

• Compiler designers have to write a compiler from the intermediate form to the compiler internal representation.
  ➢ Could be more effort than it’s worth.

• Have to design an appropriate intermediate form.
  ➢ Or do we?
WebAssembly

• WebAssembly (Wasm) is a simple, platform-independent language which can be used to express low-level code stubs.

• Compilers from C/C++ to Wasm already exist, and support is growing!
• Many production compilers can already consume Wasm!

➢ Implementation in these cases should be easy!
➢ Makes Wasm an attractive option.
Exploration in v8

• We attempted to modify v8, Google Chrome’s JavaScript engine, to use write barriers given in Wasm form.
  ➢ Technique would be very convincing if we could get it working in a real, production JavaScript engine.

• Not trying to get good performance, just trying to make it work.

• We didn’t get a working implementation, but we developed an implementation plan.
v8’s Current Write Barriers

- v8 currently inserts write barriers during platform-dependent machine code generation, at the latest possible stage in compilation.
- Write barriers have to be hard-coded for each architecture!
The Plan: Breaking Platform-Dependencies

- Lift the write barrier up to the platform-independent Base Macroassembler.
The Plan: Taking Wasm Input

- Compile the write barrier Wasm stub into machine code on startup, have the MacroAssembler insert this compiled code rather than the hard-coded platform-dependent barriers.

- More complex in practice (register allocation etc.) but this is the idea.
Implementation

• We developed a detailed plan of exactly what changes would need to be made to v8 for this to work.

• v8 is a complex codebase, any attempt at non-trivial modifications failed for difficult-to-diagnose reasons.
  – You can’t really work with this codebase without substantial experience or guidance.
Summary

• We develop a design to efficiently handle write barriers in standalone garbage collectors.

• We explore the possibility of implementing this design. We learn some things but don’t get far with the actual implementation.
Questions?