Garbage Collector Write Barriers and Just-in-Time Compilers

Calum Snowdon
u6044174

A report submitted for the course
COMP3770 - Individual Research Project

Supervised by Steve Blackburn and Ben Titzer
The Australian National University

July 2020
# Contents

1 Background
   1.1 Language Virtual Machines ................................................................. 2
   1.2 Garbage Collectors ................................................................................. 2
   1.3 Compilers ................................................................................................. 3

2 Write Barriers in a JIT
   2.1 Performance .............................................................................................. 4
   2.2 Dependencies .............................................................................................. 4
   2.3 Correctness ................................................................................................. 4

3 Current Implementations
   3.1 Taxonomy .................................................................................................... 4
   3.2 Existing Techniques
      3.2.1 Function Calls to GC ........................................................................... 5
      3.2.2 Function Calls to Patchable Code ......................................................... 5
      3.2.3 Machine Code Templates ..................................................................... 5
      3.2.4 Compiler IR Stubs ............................................................................... 6
      3.2.5 Input Language Expression ................................................................. 6
      3.2.6 LIL ...................................................................................................... 6
      3.2.7 XIR ..................................................................................................... 6

4 Our Design
   4.1 The Compiler .............................................................................................. 7
   4.2 The Garbage Collector ............................................................................... 7
   4.3 The Intermediate Language ...................................................................... 7

5 Exploration in V8
   5.1 Write Barriers in V8 .................................................................................. 8
   5.2 Goals .......................................................................................................... 8
   5.3 Plan ............................................................................................................. 8
   5.4 Attempted Implementation ....................................................................... 8

6 Conclusion
   6.1 Future Work ............................................................................................... 9

A Project Description ..................................................................................... 10
B Study Contract ............................................................................................. 10
C Artefact Description ..................................................................................... 10
D README ...................................................................................................... 10
Abstract

Garbage collectors perform automated memory management in language runtime environments. Memory management is orthogonal to most other features of runtime environments, so it’s desirable from a software engineering perspective to develop garbage collectors as stand-alone entities which may be integrated with arbitrary runtime environments. An important class of garbage collection algorithms require write barriers - small pieces of code which must be executed before (or after) pointer store operations. If a runtime environment decides to compile a section of code, the compiler must insert machine code which implements the logic of the write barriers into the compiled code at the correct places. To do this the compiler must have some understanding of the write barrier semantics - providing this in a portable, performant, correct manner is a non-trivial problem. In this report we investigate current write barrier implementations in compilers within several VMs. We then explore a new method which expresses write barrier semantics in an intermediate language provided by the garbage collector and processed by the compiler. We also explore the possibility of implementing our solution in V8, a production JavaScript engine. We don’t produce a functional implementation but we develop a plan for implementation and attempt to carry out this plan, gaining insight into the accompanying challenges. This report provides a reference for future research into this problem and lays the groundwork for a potential implementation.

1 Background

We’ll begin with an overview of the background knowledge, context, and motivation required to make sense of the body of this report.

1.1 Language Virtual Machines

Some programming languages are designed to be compiled directly from source code to machine code, which is then executed directly by the CPU. Another approach is to design a language for execution on an architecture-independent virtual machine (VM) rather than directly on the CPU. Such VMs are similar to CPUs in the sense that they have instruction sets which they can execute, but the VM itself may be implemented on multiple CPU architectures, making the VM’s language platform-independent. For example, Java is compiled into platform-independent Java bytecode which is executed by a Java Virtual Machine.

Execution within a VM provides the opportunity for advanced run-time management of program execution. For example a VM could perform run-time analysis to determine performance critical sections of code and dynamically compile these sections into machine code using a just-in-time compiler (JIT) for faster execution. This technique allows VM-managed languages to be competitive with compiled languages in terms of performance. A VM usually will implement a garbage collector (GC) to manage the memory required by the program.

Generally a VM may have a number of major internal components (such as JITs, GCs etc.) to improve performance or provide utility. Such components rarely have fundamental dependencies on the VM - for example compilers and garbage collectors are complex objects worthy of independent study and development, so ideally we’d be able to maintain each of these components individually without having to worry about dependencies between them. If the VM makes strong assumptions about the GC, it could be impossible to experiment with different GC algorithms without rewriting the VM. Unfortunately many VMs adopt - or often, evolve - a monolithic structure with non-trivial assumptions made about the internals of the various components, impeding future advancements and modifications. The main purpose of this report is to investigate one of these dependencies - GC write barriers - in detail, and attempt to figure out if current techniques for addressing it can be improved upon.

1.2 Garbage Collectors

A garbage collector is a component of a language implementation which performs memory management, allocating memory when it is requested and automatically deallocating when it is no longer being used. Without a garbage collector we need to perform manual memory management, which is highly error-prone (especially in large systems) and can lead to subtle bugs which may not manifest until long after a system is deployed, such as use-after-free bugs and memory leaks. Garbage collectors provide a safety net against such bugs, completely eliminating the former and improving the latter, and lift the burden of manual memory management from developers.

A fundamental problem behind GC design is de-
terminating when a portion of memory is no longer being used, or is no longer “live”. Most garbage collectors approximate liveness through reachability, considering an object safe to collect if there is no way for it to be accessed. Different GC algorithms approximate reachability in different ways. Reference-counting algorithms work by tracking the number of existing references to each object, considering an object safe to collect if no references to it exist. Tracing garbage collectors use a different approach, considering an object safe for collection if it is unreachable by tracing references from a set of root references. This is meaningfully different from reference-counting. In the case of a cycle, it’s possible for two objects to hold references to each other while no other object holds a reference to either of them, making them unreachable even though references exist to each of them.

We might naively design a reference-counting garbage collector as follows. Have the garbage collector record every object it allocates. When it runs out of heap space to allocate it checks all of the references in all existing objects and frees any objects which it doesn’t find a reference for. This method would work, but would incur substantial latency when a collection is triggered - this would add substantial unpredictable delays to program execution.

We could alternatively have the collection performed dynamically - the garbage collector could keep a “reference count” as per-object metadata, and whenever a pointer to an object is removed the garbage collector decrements the reference count, and whenever a pointer to an object is created it increments the reference count. If the reference count is ever decreased to 0, the object is freed and its references are recursively decremented. This method adds a small, predictable latency to every pointer operation, which may be preferable to the large, unpredictable delays of the previous method.

This dynamic method imposes a requirement on the runtime environment - every time a pointer operation is performed, the runtime must update the reference count. Other more sophisticated garbage collectors may require the runtime to perform other simple operations alongside reference stores, or even reference reads or reads/writes to primitive data types. These operations are collectively referred to as read/write barriers. Most production garbage collectors require barriers of some form, but the requirements of these barriers may differ between garbage collection algorithms.

While barriers of all kinds do exist, the most common form of GC barrier is the object reference write barrier, which must be executed along with stores to object reference fields. For the remainder of this report we’ll refer to these simply as “write barriers”, but the majority of our content can be applied to other barriers too.

1.3 Compilers

In this section we’ll cover the basics of compiler internals. A compiler is a program which consumes source code in some language and translates it into executable machine code (or perhaps some other medium such as bytecode). Compilers may also optimize the input program. Code optimization is an interesting problem - many optimizations require consideration of a program’s structure rather than just individual lines of code, so to apply non-trivial optimizations we need to analyze the control flow of the program in some way. The textual source code representation of a program is a serial data format, and thus isn’t amenable to structural analysis or modification. The structure of a program is often better represented as a control flow graph. So compilers will typically define one or more graph data structures internally to represent the input program. These data structures are commonly known as internal representations (IR’s). It may be useful to have multiple IR’s, for example compilers commonly have a high-level IR (HIR) which can represent high-level language features such as object operations or control structures along with a low-level IR (LIR) which can represent lower-level operations such as memory accesses or branches, which are closer to the underlying machine architecture. Certain optimizations may be more appropriate for application at the HIR or LIR stages. Note that the compiler will have to translate (lower) the HIR into LIR at some point during compilation.

A typical compiler will have 3 components:

- The front-end, which parses source code into the compiler’s IR.
- The middle, which manipulates the IR form of the program. For example it may apply optimizations such as in-lining or loop unrolling to the IR graph. This is typically the most complex section of a compiler.
- The back-end, which translates the optimized IR into machine code.
There are a few interesting points worth noting about this architecture. Firstly, the middle and back-end shouldn’t depend on the source language, just on the IR. So if we write a compiler for language A, all we have to do to get it to compile language B is write a suitable front-end which translates language B into the compiler IR (provided language B is compatible with the compiler IR). Further, the middle and front-end shouldn’t depend on the target machine architecture, so we can compile to multiple architectures simply by writing a new back-end for each architecture.

2 Write Barriers in a JIT

Within a language runtime environment, write barriers present a non-trivial dependency between the GC and the JIT. When the JIT compiles a section of code, the compiled code will need to perform write barriers as long as the source includes reference writes. This means that the semantics of the write barrier will have to be implemented within the JIT at some point during compilation. We could simply hard-code the write barrier into the JIT for it to insert whenever it needs to, but the write barrier semantics depend on the GC algorithm, so this introduces coupling between the GC and the JIT. This kind of coupling is highly undesirable because it inhibits experimentation with new compiler or GC methods.

Implementation of GC write barriers in the JIT is a non-trivial software engineering problem. In this section we’ll explore the accompanying design considerations.

2.1 Performance

Object reference stores occur very frequently in object-oriented programs, thus write barriers are executed with a similar frequency, making them performance-critical pieces of code [3]. Write barriers are good targets for many compiler optimizations - for example they may be lifted out of loops, barriers which are close together may use the same registers for common constants, etc. so we could potentially improve performance by allowing the compiler to apply optimizations to the write barriers.

Another approach to achieving performance is to optimize the barriers as much as we can by hand, manually producing machine code for the barriers rather than compiling them. The performance improvement with this method will rarely be as potent as compiler optimizations, but will often be satisfactory. As we’ll discuss later, this method is currently used by the .NET CLR and V8 JITs.

2.2 Dependencies

There are many attractive approaches which introduce tight coupling between the GC and the JIT. In today’s agile software ecosystem, the ability to quickly implement and assess new techniques is highly valued. Especially for the purposes of designing a standalone garbage collector, we don’t want to impose any requirements on the internal structure on the compiler.

2.3 Correctness

Applying optimizations of any form to write barriers is a delicate operation. Write barriers may have atomicity requirements which could be violated by certain optimizations. For example, if the compiler interleaves the write barrier code with other object-related code, a GC sweep might be triggered in the middle of the write barrier, which could invalidate the barrier. So some care is required in controlling which optimizations will be applied to the write barriers.

3 Current Implementations

We’ve compared JIT write barrier implementations across a variety of VMs, ranging from hardcore production VMs to legacy VMs to VMs constructed for research purposes, in order to gain understanding of existing techniques for achieving efficiency, modularity and portability in various settings.

3.1 Taxonomy

An enumeration of methods for handling runtime support routines is given in [1]. Since this paper was written, current methods have diversified in meaningful but subtle ways. Some current methods are very similar to each other, but still differ enough in some aspects to warrant separate discussion. As such we’ve developed a more fine-grained scheme to classify current methods, in the form of the following criteria.
1. Duplication - Does the write barrier need to be re-written in any form other than the original GC's implementation?

Code duplication isn’t ideal from a software engineering perspective, but it may be an attractive option if it can give an easy performance boost.

2. Dependency - Does the method rely on assumptions about the JIT or GC internals?

This is another software engineering feature. Dependencies should ideally be avoided for ease of modification and experimentation with new techniques, but they can be difficult to eliminate without sacrificing performance.

3. Rigidity - Are the barriers given as static code snippets, or are they dynamically generated?

This is a performance point. If the write barrier exists as a static code stub, such as a pre-compiled function or hand-written assembly, it will have a calling convention which must be followed. This necessitates moving registers around and perhaps spilling onto the stack. When the barrier itself may only be a few machine instructions, this is a big performance reduction. These are “rigid” implementations. Non-rigid implementations have some means to counteract this, such as having multiple equivalent stubs with different calling conventions or expressing the barriers at a higher level so the compiler can inline them.

4. Structure - Does the JIT insert the barriers in a structured (e.g. IR) or unstructured (e.g. pre-compiled) form?

This is another performance point. If the compiler receives the write barrier in a digestible form rather than an indecomposable form it can apply optimizations to write barrier code.

5. Portability - Is the method amenable to GCs/JITs written in different languages?

A final software engineering point. This criterion may be out of place in a general taxonomy, but it’s appropriate to consider for the purposes of this report.

These criteria are sufficient to meaningfully separate and assess each of the VMs we’ve analyzed.

### 3.2 Existing Techniques

In this section we’ll assess existing implementations against our criteria. All of the following implementations were found in one of the 8 VMs we analyzed.

#### 3.2.1 Function Calls to GC

This method has the write barriers written as functions in the GC which are pre-compiled, and the JIT is given a reference to the compiled write barrier. The JIT then inserts write barriers into generated code as calls to this function. This is the simplest method.

All the JIT needs to do here is insert function calls to pre-compiled code, so this method requires no code duplication, has no dependencies between the GC and the JIT, provides barriers in a rigid form and is compatible with cross-language implementations. It is, however, not amenable to compiler optimizations.

#### 3.2.2 Function Calls to Patchable Code

This is a method used by the .NET CLR. As above, write barriers are inserted as function calls. However, in this case the function is written in machine code, and the function code itself can be modified at run time. For example, if the write barrier requires access to some GC global variable whose location is subject to change, the write barrier would have to load it through a level of indirection. Alternatively, when the location of the variable changes we can modify the write barrier to load directly from the new location, removing the extra load operation.

This method is like inserting function calls to the GC, but also requires a machine code copy of the write barrier in the JIT, introducing a dependency and code duplication. In exchange, it does achieve performance befitting a production-quality VM.

#### 3.2.3 Machine Code Templates

With this method the JIT employs hand-written machine code templates which can be specialized and inserted when required. For example, the JIT may have a function which, given the register which will hold a runtime object address, emits machine code that executes the write barrier with that register as input.

This implementation requires the write barriers to be manually written in the JIT in machine code for
each architecture the VM supports, requiring code
duplication. Since the barriers are written in the JIT,
this is a dependency. This method is also non-rigid
because the templates can be specialized to operate
on any registers, and is unstructured because it di-
rectly inserts machine code. This approach would
work with cross-language implementations because
it doesn’t rely on any interface between the JIT and
GC.

This implementation is used by Google’s
JavaScript engine, V8, along with the Self and
Strongtalk VM’s. This method is highly performant,
allowing the exact sequence of machine instructions
required for the write barriers, making it suitable
for a high-performance production VM provided
the undesirable software engineering qualities are
tolerable.

3.2.4 Compiler IR Stubs

This implementation has the GC express its write
barriers in the compiler’s IR. This allows complete
compiler optimizations to be applied to write bar-
riers, but introduces a major dependency of the
GC on the JIT’s IR. Implementing this across lan-
guages would also require translation between the
language’s data structures.

This technique is used in Hotspot, the Java VM
used in OpenJDK, maintained by Oracle. It is non-
rigid and structured, facilitating high performance.
It requires duplication of the write barrier in the
compiler’s IR and thus introduces a dependency of
the GC on the JIT’s IR. The JIT’s IR is a data structure
in the source language of the JIT, so this method isn’t
practical for cross-language implementations.

3.2.5 Input Language Expression

This implementation has the GC express its write
barriers directly in the language which the JIT con-
sumes. For example, JikesRVM is a meta-circular
VM which is written in Java and consumes Java. The
GC here is already written in Java, so the JIT is free to
directly consume the write barrier bytecode without
any modification to the JIT or the GC.

This method features no code duplication, is non-
rigid, structured and has no dependencies between
the JIT and GC. It does, however, rely on the fact that
the GC is written in the input language of the com-
piler, meaning it isn’t suitable for cross-language im-
plementations.

3.2.6 LIL

The ORP VM addresses this problem with LIL [1],
a domain-specific language designed for expressing
low-level machine stubs in an architecture-neutral
form. The GC write barriers are expressed in the GC
as LIL templates, and when the compiler needs to
insert a write barrier it sends a request to the run-
time, which specializes a template to the context of
the request, compiles the LIL into machine code, and
sends the machine code back to the compiler, which
inserts it into the generated code.

This is similar to the use of machine code tem-
plates, except code duplication is reduced because
LIL is architecture-independent, so the barriers need
only be duplicated once. Also the LIL templates are
provided by the GC rather than hard-coded into the
JIT, so the dependency is removed. Thus this method
is non-rigid and dependency-free, but still requires
some code duplication and is unstructured. Further,
machine code is inherently serial, meaning it can eas-
ily be passed between programs written in different
languages.

3.2.7 XIR

The developers of the Maxine VM decided to port
Hotspot’s JIT for Maxine, for which they had to re-
move the runtime dependencies from the compiler.
This resulted in the development of XIR [2], an in-
termediate language used as a level of indirection
between the compiler and the runtime. Similarly
to LIL, XIR templates for the write barriers are pro-
grammed into the GC. When the compiler needs to
insert a write barrier it makes a request to the run-
time, which specializes an XIR template and returns
the resulting XIR snippet to the compiler. The com-
piler then inserts the XIR snippet as an LIR node and
can perform limited optimizations before code gen-
eration.

This is similar to LIL, with two key differences
with respect to our criteria. The data passed from
the runtime to the compiler here is an XIR snip-
net, which would require much more effort to prop-
erly translate between languages, making this ap-
proach less appropriate for cross-language im-
plementations. On the other hand, XIR is somewhat
structured in the sense that the compiler can perform
limited optimizations rather than just inserting ma-
chine code.
4 Our Design

From the current implementations we've examined, the most attractive on all fronts is the implementation by JikesRVM, which requires no code duplication and allows the full suite of compiler optimizations to be applied to the write barriers. However, this method is qualified by the requirement that the GC be written in the same language that the compiler consumes, which rules out the option of applying this to a standalone, universal garbage collector. However, we may be able to circumvent this requirement by drawing inspiration from ORP and Maxine, and using an intermediate representation.

We propose the following design for the expression of write barriers to the JIT.

1. The garbage collector designer selects an intermediate language which is sufficiently expressive to capture the semantics of arbitrary write barriers.

2. The garbage collector exposes a copy of the write barrier in the selected intermediate language.

3. The compiler implements a parser for the intermediate language.

If the compiler can parse and consume this intermediate language, we gain effectively the same benefits that we would if the barriers were written in the input language of the compiler. Further, because the intermediate language is agreed upon beforehand, there are no dependencies between the GC and compiler.

For the remainder of this section, we'll discuss some of the design considerations surrounding this method.

4.1 The Compiler

If a garbage collector follows this design pattern, a compiler can integrate this GC's write barriers by writing a parser which takes the chosen intermediate language and converts it into some form which is compatible with the compiler internals, then modifying the compiler to insert the translated write barrier in the required places. This gives the compiler designer a degree of freedom - they could compile the intermediate language directly into machine code and simply insert it into compiled code, or they could translate the intermediate language into its IR and apply optimizations before code generation, or they could do anything in between. There is, however, the requirement that they implement a parser for the intermediate language, which may not be simple.

4.2 The Garbage Collector

The requirements of this design on the garbage collector are light. We simply require that it provides a description of its write barrier in this intermediate language, so the garbage collector designers could simply produce the appropriate code by hand. They also have the option of developing a transpiler from the GC source language to the intermediate language and using it to automatically produce the intermediate form from the GC source code. This would remove the code duplication, but would require writing a compiler.

4.3 The Intermediate Language

The choice of intermediate language is critical to this design. The compiler designers will need to implement a translation from the chosen language into an internal form. If the language isn’t sufficiently simple, developing such a translation would be difficult and may not be worth the effort.

We could follow in ORP or Maxine’s footsteps and define this intermediate language ourselves, appealing to the simple nature of write barriers. GC barriers generally don’t require control structures more complex than conditional operations, so it would suffice to develop a primitive, semantically plain language to use as an intermediate.

As an alternative, we could appeal to a widely-supported existing language. WebAssembly (wasm) is a simple, platform-independent language that can be used to express low-level code. Wasm is already widely supported, with existing compilers from several prominent languages to wasm, and with many production language runtimes having wasm support. So if our GC is written in a language with a wasm compiler, we can trivially translate the write barriers into wasm with no code duplication. If a compiler already has wasm support, it could be modified to accept write barriers as wasm stubs without having to write a new parser.

This isn’t a perfect solution - as it stands wasm programs run in a sandbox while write barriers may need to access GC data from memory outside the sandbox. We can address this by introducing a new
wasm opcode which can perform raw memory operations. This would require that the VM’s wasm compiler support this opcode, but implementing a new opcode is a simple affair so this shouldn’t be a serious obstacle for the compiler designers.

This second option provides a convenient choice for VMs that already have wasm compilers, but is prohibitive for those which don’t. To bridge this gap we could have the GC expose the write barrier in multiple forms, and allow the compiler to choose which form it will accept.

5 Exploration in V8

To assess our proposed design we attempted an implementation of wasm write barriers in Chromium’s JavaScript engine, V8, which already has wasm support. We hoped to obtain a faithful idea of the practicality of our design by experimenting within a production codebase. To this end we analyzed the V8 code base and identified modifications we could make to have V8’s JIT consume write barriers in wasm form.

5.1 Write Barriers in V8

The write barriers for V8’s garbage collector are simple. For each write of an object address to a field of an object, the write barrier checks some flags maintained in memory by the GC and calls a function in the GC if necessary. Normally, the flags indicate that no work is to be done, so the fast path is a few loads, a compare, and a branch that is not taken.

In V8’s optimizing compiler, Turbofan, the write barriers are inserted into compiled code at the latest possible stage - code generation. The code generator has a function which takes in the registers containing the objects in question and emits machine code performing the write barrier. In particular, the write barriers are hard-coded in assembly. Moreover, Turbofan has backends targeting 8 different architectures - so the write barriers are hard-coded in assembly 8 times.

It’s clear that V8 could benefit from our proposed architecture. A successful implementation would see duplication of the write barrier removed and a boost performance through insertion of the write barriers before optimization.

5.2 Goals

For our proposal to be useful we need production compilers to be amenable to our design. In this case we would need them to be able to, at the very least, accept a WebAssembly stub for use as a write barrier. So our main goal in experimenting with V8 was to assess this possibility. As a baseline, we just wanted to see what it would take to have the compiler accept the stub and work correctly - so we have little concern for the effect on performance at this stage.

5.3 Plan

Our selected method for inserting our WebAssembly write barriers is to first compile the WebAssembly code to machine code, then change the write barrier function so that it inserts our WebAssembly code rather than the hard-coded write barrier stubs. This introduces a problem - Turbofan’s existing implementation generates a stub given the required registers, but if we pre-compile our WebAssembly stub then the registers it uses are fixed; our design is rigid (it should be possible to compile the wasm into Turbofan IR rather than machine code, so this rigidity isn’t inherent to our design, just this implementation). So we’ll also have to add some code to move the operand registers into the correct positions, but Turbofan provides adequate facility for this.

Next we need to modify the code generator to insert our code stub rather than the pre-generated ones. Currently the write barriers are inserted in the architecture-specific code generators. All of the common features across the architecture-specific code generators are collected into a platform-independent “base” code generator class - since our version is platform-independent, it makes sense to include a function which inserts our stub here. Then we can modify the platform-specific operations to call this function rather than generating and inserting the pre-written stub as usual. There may be a nicer method, but this will get the job done and will suffice for exploratory purposes.

5.4 Attempted Implementation

A great deal of proficiency is required to perform non-trivial modifications to a large, industrial-strength, highly-optimized code base like V8. After many weeks studying the code base and attempting to make modifications, we determined that imple-
menting these planned changes would be impractical within the scope of this project.

6 Conclusion

Existing techniques in this area are certainly inadequate for the purposes on developing a performant, well-engineered standalone garbage collector. In theory the idea we’ve developed should be appropriate, but we have yet to verify this claim.

While the development of a working implementation of our solution was infeasible within the scope of this project, it certainly seems possible given adequate time and expertise.

6.1 Future Work

The first major piece of future work would be completing the implementation of our solution in V8. Following this, development could begin on implementing the GC side of our solution, writing the GC to compile its write barriers to wasm (or some other form) and expose them to the JIT. The ultimate goal is a working implementation of the technique we’ve proposed.

Beyond this, the taxonomy we’ve presented for classifying implementations of write barriers in JIT compilers is a fresh piece of work and could benefit from further development, and perhaps analysis by others who are more versed in the topic.
Appendices

A  Project Description

The project description found in the study contract, covers the planned goals we had for the project, but they weren’t fully fleshed out.

Once we actually started work, our agreed upon goal was for me to modify the V8 code base to accept write barriers in wasm form rather than as architecture-dependent machine code templates. Several weeks in we decided that it was infeasible within the remaining time, so we changed goals.

The final agreed upon goal was to investigate implementations of write barriers in JIT compiles within existing VMs to try to understand and classify existing methods, and to compare them to our proposed method I was also to develop the ideas behind our proposed solution.

B  Study Contract

Over the page. This is the contract signed at the start of the semester, it’s grown somewhat inaccurate as the project evolved, note the points in appendix A. Also it lists Ben as my examiner, which isn’t correct.

C  Artefact Description

In Lieu of a software artefact, I’ve performed a code review of 8 languages VMs (6 in full detail, 2 in light detail) to compare and contrast how they handle write barriers in their JITs.

I downloaded and read through the source code for each VM and figured out in what form the write barriers are presented to the JIT, and where/how they are inserted during compilation.

The document itself contains high-level explanations of the implementation in each VM along with links to the files in the repositories containing relevant code, and an explanation of the relevance of each file.

D  README

I haven’t submitted any software, so I haven’t provided a README.
INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the Honours/projects co-ordinator

SECTION A (Students and Supervisors)

Unid: u5644174
FAMILY NAME: Snowden
PERSONAL NAME(S): Calum
PROJECT SUPERVISOR (may be external): Steve Blackburn
COURSE SUPERVISOR (a RSCS academic): Steve Blackburn
COURSE CODE, TITLE AND UNIT: COMP3770 Individual Research Project, 6 units

SEMESTER ☑ S1 YEAR: 2020 ☐ S2 YEAR:
PROJECT TITLE: Portable Fast Path Inlining For Garbage Collectors

LEARNING OBJECTIVES:
- Develop a working understanding of
compilers and garbage collectors.
- Gain experience working with &
modifying large code bases.
- Develop research skills such as
literature analysis &
technical writing.

PROJECT DESCRIPTION:
- Write a literature survey
- Investigate options for translating a restricted subset of Rust into a suitable
intermediate representation which is digestable by the V8 JIT.
- Write software to automatically translate sufficiently minimal Rust functions
into said intermediate representation and feed them to the JIT, effectively
allowing simple Rust functions to be inlined in Javascript.
- Analyze performance compared to hard-coding said functions.
- Attempt to extend this method to other VMs.
- Write report
ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th></th>
<th>(fixed)</th>
<th>Proj (6-12 credit)</th>
<th>.... (fixed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assessed project components:</td>
<td>% of mark</td>
<td>Assessed project components:</td>
<td>% of mark</td>
</tr>
<tr>
<td>Thesis</td>
<td>(85%)</td>
<td>Thesis (reviewer mark)</td>
<td>50</td>
</tr>
<tr>
<td>Presentation</td>
<td>(10%)</td>
<td>Artefact (supervisor project mark)</td>
<td>40</td>
</tr>
<tr>
<td>Critical Feedback</td>
<td>(5%)</td>
<td>Presentation</td>
<td>(10%)</td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

Weekly

STUDENT DECLARATION: I agree to fulfil the above defined contract:

.......................................................... ..................................................
Signature                                      Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

.......................................................... ..................................................
Signature                                      Date

Reviewer:

Name: .............................................. Signature: ..................................

Reviewer 2: (for Honours only)

Name: .............................................. Signature: .................................

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Honours / Projects coordinator approval)

.......................................................... ..................................................
Signature                                      Date
References

