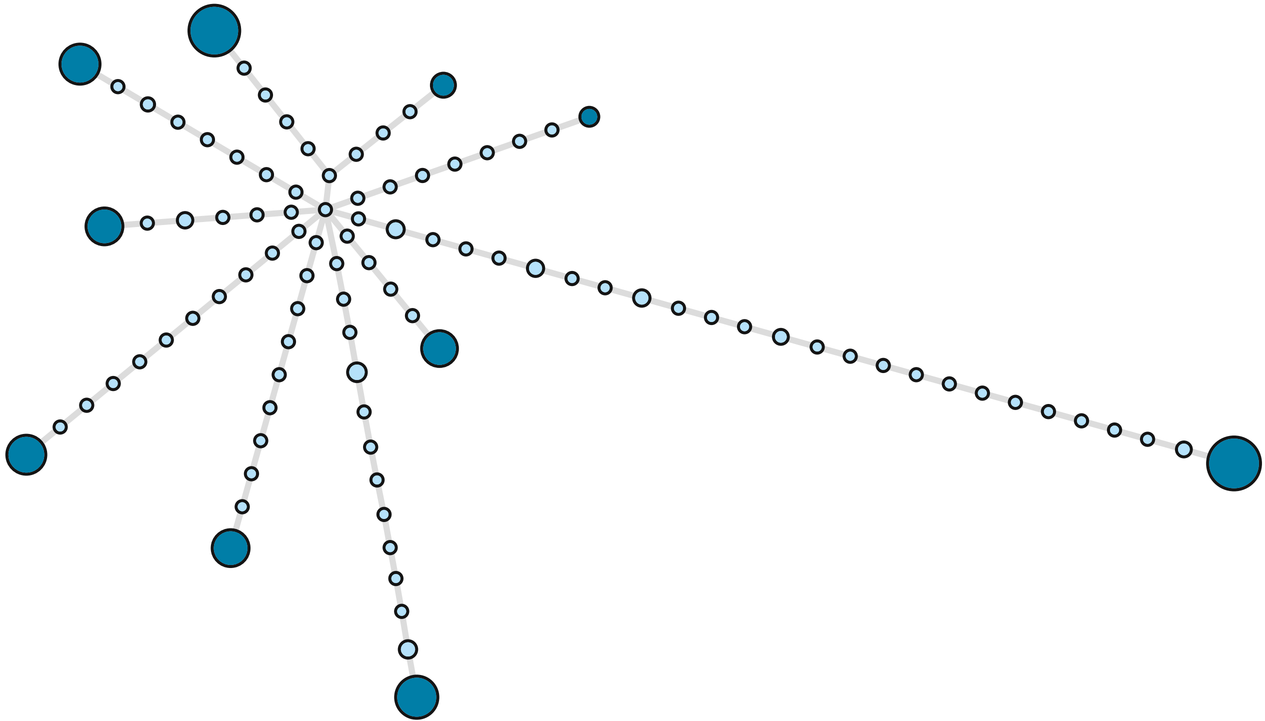
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Quick Access

Optimizing AOT Compiled Dynamic Programming Languages

Master’s Thesis in Computer Science and Engineering

Eli Adelhult

Carl Forsinge

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2025

Master's Thesis 2025

Quick Access

Optimizing AOT Compiled Dynamic Programming Languages

Eli Adelhult
Carl Forsinge

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden, 2025

Quick Access
Optimizing AOT Compiled
Dynamic Programming Languages

Eli Adelhult
Carl Forsinge

© Eli Adelhult, Carl Forsinge, 2025

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31-772 10 00

Typeset in Typst
Gothenburg, Sweden, 2025

Quick Access
Optimizing AOT Compiled
Dynamic Programming Languages

Eli Adelhult
Carl Forsinge
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Research on dynamic programming languages typically focus on *just-in-time* (JIT) compilation. However, JIT techniques are not always viable due to hardware limitations or security concerns, making it valuable to study *ahead-of-time* (AOT) compilation of dynamic languages as well.

The *GameMaker* game engine targets mobile devices, game consoles, and WebAssembly — all platforms poorly suited for JIT compilation. Instead, GameMaker supports AOT compilation of JavaScript and its own language, GameMaker Language.

In this thesis, we extend the engine with new profiling tools to analyze the memory usage and runtime performance of *property accesses*, a frequent and performance-critical operation in dynamic languages. Using our new tools, we identify several opportunities for improvement in both the compiler and runtime environment.

Finally, we propose and implement solutions to the identified areas of improvement. These include polymorphic property caches, a pipeline for profile-guided optimizations, caching of accessor properties, and cache pools designed to facilitate cache invalidation. In certain benchmarks, our solutions achieve speedups with factors ranging from 1.5 to 2.9.

Keywords: Ahead-of-time compilation, Hidden class, Property cache

Acknowledgements

Our thesis has been a collaboration with the people at Opera and YoYo Games. We would especially like to thank Andrew Martin and Luke Brown for their friendly guidance and technical feedback, as well as Erik Möller and Russell Kay for making this project possible.

We also wish to express our gratitude to Magnus Myreen for offering to supervise our thesis and his thoughtful feedback on our writing, without him this report would be a lot less interesting to read. We also direct a thank you to Alejandro Russo for acting as examiner.

Lastly, we would like to thank our friends and families for their support.

Eli Adelhult, Carl Forsinge, Gothenburg, 2025-06-02

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **15**
1.1 Problem . . . 15
1.2 Scope . . . 16
1.3 Outline . . . 16

2 Background **17**
2.1 Hidden classes . . . 17
2.2 Property caches . . . 18
2.3 GameMaker runtime and compiler . . . 19
2.3.1 GameMaker intermediate representation . . . 19
2.3.2 From GMIR to a running program . . . 20
2.3.3 Runtime libraries . . . 20
2.4 AOT compilation of JavaScript . . . 21
2.4.1 Profile-guided optimization . . . 22

3 Methods **23**
3.1 Building profiling tools . . . 23
3.2 Visualizing data . . . 23
3.3 Benchmarking suite . . . 24
3.4 Ensuring correctness . . . 26

4 Initial Findings **27**
4.1 Monomorphic property caches . . . 27
4.2 Cache invalidation and dynamic inheritance . . . 28
4.3 To cache or not to cache? . . . 29
4.3.1 Built-in properties on instances . . . 29
4.3.2 Methods . . . 29
4.4 Naive property cache allocation . . . 31
4.5 Hidden class graph . . . 32

5 Experiments & Results **35**
5.1 Polymorphic property caches . . . 35
5.1.1 Established designs . . . 35
5.1.2 Our implementation . . . 35
5.1.3 Using profiling information . . . 37
5.1.4 Speed versus space . . . 37

5.2 Grouping hidden class nodes ... 39
5.3 Caching accessor properties ... 39
5.3.1 Our implementation ... 40
5.3.2 Issues with inherited properties ... 41
5.3.3 An incomplete solution ... 41
5.4 Grouping property caches ... 42
5.4.1 Our implementation ... 42
5.4.2 Sharing property caches ... 43
5.5 Invalidating property caches ... 43
5.6 Bringing it all together ... 44

6 Discussion **47**
6.1 Limitations ... 47
6.1.1 Benchmarks ... 47
6.1.2 Binary size ... 48
6.2 Future work ... 48
6.2.1 Making better use of property caches ... 48
6.2.2 Investigating polymorphic property caches ... 49
6.2.3 Making further use of PGO ... 49
6.2.4 Integrate PGO pipeline in development workflow ... 49

Bibliography **51**

Appendix A GMIR Overview **55**

Appendix B Hidden Class Graph **61**

List of Figures

Figure 1 Object memory representations 17
Figure 2 Memory representation when using hidden classes 18
Figure 3 Overview of the GMRT compilation pipeline. 19
Figure 4 GameMaker Language compiled down to GMIR. 20
Figure 5 Overview of the layered runtime library architecture. 21
Figure 6 Raw profiling data translated into inline code annotation. 24
Figure 7 Reconstructed hidden class graph based on profiling data. 24
Figure 8 Some of the games used to measure performance and cache behavior. 25
Figure 9 Example of a polymorphic function. 27
Figure 10 The compiler fails to reason about the return value of `instance_create_layer`. .. 30
Figure 11 GMIR generated for pre-increment expression. 31
Figure 12 Accessor property modifying the shape of an object. 32
Figure 13 Hidden class caching for the example from Figure 2. 32
Figure 14 C++ definition of the polymorphic cache. 36
Figure 15 C++ definition of a space-efficient polymorphic cache. 38
Figure 16 A visual representation of the compact cache layout. 38
Figure 17 Property cache data type extended to support accessor properties. 40
Figure 18 Objects sharing hidden class but having different prototypes. 41
Figure 19 Caching accessor properties while making use of dynamic inheritance. 41
Figure 20 Code generation of cache pools. 42
Figure 21 Inheritance chain effected by adding or removing properties on a prototype. ... 44
Figure 22 Polymorphic cache structure supporting both accessor and value properties. ... 45
Figure 23 Implementation of the *GetProperty* function for the merged cache. 45

List of Tables

Table 1 Specifications of the computer used in benchmarks. 26
Table 2 Hit rates for games when using monomorphic property caches. 28
Table 3 Generated GMIR instruction for calls to different types of methods. 30
Table 4 Benchmarking different approaches to vector methods. 31
Table 5 Comparing the performance of polymorphic and monomorphic caches. 37
Table 6 Comparing memory usage for uniform and profile-guided cache sizes. 37
Table 7 Comparing the performance of the two polymorphic caches. 38
Table 8 Comparing execution times of the accessor cache. 40
Table 9 Sharing caches between call sites. 43

1

Introduction

Dynamic programming languages are used in a wide array of domains, including web browsers, data science, and games. Historically, the research on optimizing the runtime performance of such languages has focused on *just-in-time* (JIT) compilation — dynamically generating machine code while the program is running. This includes the pioneering work done in the 1980s and 1990s on the languages Smalltalk [1] and Self [2], and more recently on JavaScript [3].

As a result, the impact and potential of optimization techniques for *ahead-of-time* (AOT) compilation of dynamic programming languages are not as well explored. Quoting Manuel Serrano at INRIA, "very little effort is invested in studying static compilation of these [dynamic] languages; JIT studies, however, are countless" [4].

Unfortunately, just-in-time compilation is not without its limitations. For instance, compiling during runtime is undesirable in serverless computing applications where programs have a very short lifespan [5].

In some other domains, JIT compilation is unfeasible due to hardware constraints or security concerns [4]. This is the case when targeting smartphones, gaming consoles or WebAssembly[1]. A prime example of this is the *GameMaker* game engine, which targets all of the mentioned platforms and must therefore make use of AOT compilation techniques.

1.1 Problem

In this master's thesis we aim to implement and evaluate a set of AOT compiler optimizations inside GameMaker as part of its new runtime and compiler toolchain. More specifically, with a focus on optimizations of *property accesses* (e.g. `myObj.someProperty`), which is a common and relatively costly operation in dynamic programming languages. This task is split into two distinct sub-problems:

- **Investigate the performance** of the existing systems. This includes the development of new tools to profile and visualize the inner workings of the runtime environment.
- **Improve the performance** by implementing and evaluating relevant ahead-of-time optimization techniques.

[1]WebAssembly uses the Harvard architecture, storing data and instructions separately, making JIT compilation difficult. However, there have been attempts to dynamically link multiple modules to achieve dynamic code generation [6].

1.2 Scope

The GameMaker compiler toolchain currently includes two compiler front-ends and a shared intermediate language. One front-end for JavaScript and another for their own dynamic scripting language GameMaker Language (GML) [7]. By exclusively focusing our efforts on the intermediate language and runtime libraries, we narrow the scope of the project while still granting users of both GML and JavaScript access to all potential performance improvements.

To further narrow the scope, all benchmarks are to be written in GML since it is the primary language used in the GameMaker engine, while JavaScript is only a recent addition with experimental support.

1.3 Outline

The rest of this thesis is structured as follows. Chapter 2 *Background* introduces terminology and related research and gives a brief introduction to the relevant parts of the GameMaker engine. Chapter 3 *Methods* explains our approach to performance benchmarking and the profiling tools that were developed. Chapter 4 *Initial Findings* highlights areas of potential improvement in the current implementation of the engine. Chapter 5 *Experiments & Results* presents our attempts to improve the engine, with results of each to measure their applicability. Finally, Chapter 6 *Discussion* provides a more analytical review of our work, including shortcomings and possible future work.

2

Background

This chapter introduces two of the most relevant optimization techniques — *hidden classes* and *property caches*. We also provide a technical overview of the new GameMaker runtime and a brief introduction to some more recent research.

2.1 Hidden classes

Both JavaScript and GameMaker Language[2] have similar object models and make use of the idea of *prototypal inheritance*. Unlike most other object-oriented languages, classes are not used to define the structure of objects. Instead, new objects are created by making a shallow clone of an existing one, referred to as a prototype. Objects also inherit properties from their prototype.

In a naive implementation of a prototypal language, each object would have to contain all the names and values of properties as well as metadata associated with them. As seen in Figure 1, this is less memory efficient compared to statically typed languages, where it suffices to store the values of properties, and the names can simply be mapped to a memory offset during compilation.

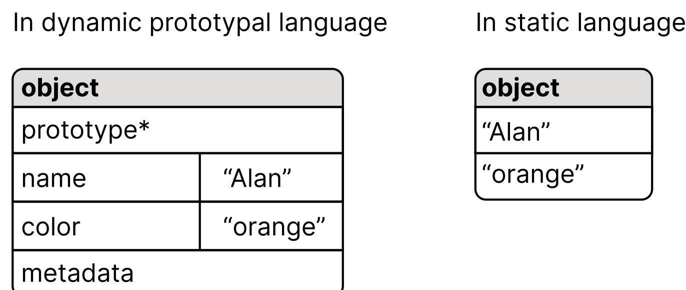


Figure 1 : Object memory representation in dynamic prototypal language versus statically typed language.

[2] At least more recent versions of GML. JavaScript-like objects were introduced in 2020 [8].

Fortunately, the implementers of the Self language [2] observed that typically, many objects have the same shape and are cloned from the same prototype. The runtime environment can track this information using a technique called *hidden classes*[3]. In modern language runtimes, hidden classes are implemented as a finite automata that stores the memory offsets needed to access properties as well as related metadata [9], [10].

When using this technique, each object only contains the values of its properties and a pointer to its hidden class. This approach leads to a much more efficient object-storage model that helps make better use of the hardware cache locality principle. A concrete example of a hidden class is found in Figure 2.

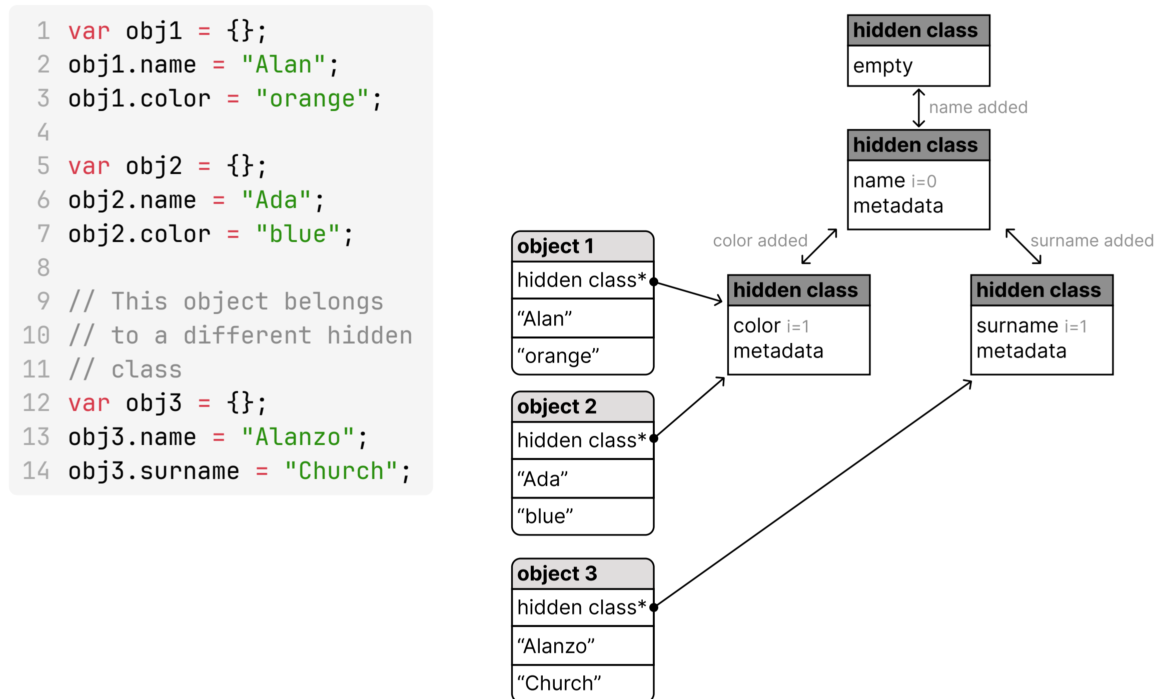


Figure 2 : Memory representation of three example objects when using hidden classes. The pointers between hidden classes allow them to be reused when an object dynamically adds or removes properties. Note that this is a simplified visualization and omits details around caching and where an object’s prototype relation is stored.

2.2 Property caches

While hidden classes work well for improving memory efficiency, they are also useful for other kinds of optimizations. For languages such as Self and JavaScript, hidden classes are commonly paired with *property caches* in order to achieve faster method calls and property lookups [2], [10].

Method calls and property lookups are relatively expensive operations in dynamic (and especially prototypal) languages. To resolve the memory offset that a property corresponds to, it may be required to traverse the hidden class graph, especially if the property is inherited from another object.

[3]The implementers of Self call it a *map*.

Typically, one property cache is used per call site, which in most cases means one cache per property access in the source code. The property cache saves the memory offset (or memory address) when a property is accessed. Using the cached memory offset directly on subsequent accesses makes it possible to avoid pointer indirections and potentially fetching data from slower memory. However, it is also important to store the hidden class to ensure that the shape of the object is the same, otherwise the cache needs to be invalidated.

Typically, JIT compilers refer to property caches by the name *inline caches* (IC), first invented for the Smalltalk language [1]. We consider this term slightly more general since it in some contexts refer to caching of more than just strictly property accesses [5], [11].

Property caches can either be monomorphic or polymorphic[4] [12]. The former can only hold one hidden class at a time, normally the most recently used one. The latter is more common and can contain many or even all hidden classes used at the call site. Polymorphic caches naturally require more memory but are often favored because they offer better performance for code with higher degrees of polymorphism [3], [10].

2.3 GameMaker runtime and compiler

The GameMaker game engine has evolved for more than 25 years. Recently, a rewritten runtime environment known as *GMRT* was made publicly available. This new runtime and compiler toolchain consists of many individual libraries and tools working together. The pipeline shown in Figure 3 showcases a few of the most relevant compiler tools.

The two scripting languages currently supported are GameMaker Language (GML) and a subset[5] of JavaScript. Both of these languages are compiled into a shared intermediate representation known as *GameMaker Intermediate Representation* (GMIR).

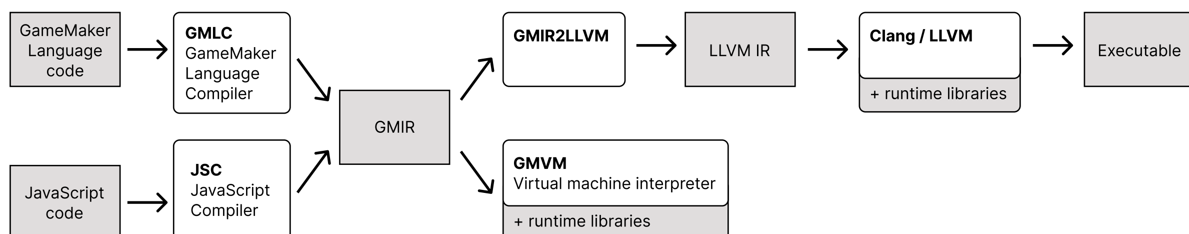


Figure 3 : Overview of the GMRT compilation pipeline.

2.3.1 GameMaker intermediate representation

GMIR is a register-based instruction set containing around 60 high-level instructions. For a reference of the complete instruction set, see Appendix A. Figure 4 contains an example of GML source code compiled down to GMIR. Notice that the compiler attempts to add type annotations when possible.

[4]Commonly abbreviated PIC.

[5]GameMaker aims to support the majority of the ECMAScript specification [13], with the exception of functions like `eval` that require dynamic evaluation.

GameMaker Language example code
```
/* In the scope of a
GameMaker object. Top-level
assignments are attached to
said object. */
my_prop = 0;
var my_var = my_prop + 10;
show_debug_message(my_var);
```

Generated GMIR
```
r4 : Double = LDC(c0(0))
PSET(r4 : Double, self_r0, c1(`my_prop`))
r5 = PGET_A(self_r0, c1(`my_prop`))
r6 : Double = LDC(c2(10))
my_var_r3 = ADD(r5, r6 : Double)
r7 = STATIC_CALL(
  c3(external function show_debug_message),
  self_r0,
  my_var_r3)
```

Figure 4 : GameMaker Language compiled down to (a textual representation of) GMIR.

The main focus of this thesis is optimizing code related to property accesses. The most relevant instructions are therefore the property get (`PGET`) and set (`PSET`) instructions. It is worth noting that GMIR has variants of these instructions with an `_A` suffix. They serve the same purpose but include an additional assertion to ensure that the property exists, and throw an exception when the assertion fails.

2.3.2 From GMIR to a running program

Once the compiler front-end has generated GMIR there are two options for how to proceed. The intermediate representation can either be directly interpreted in a virtual machine (for a faster development cycle) or compiled into machine code (for better performance). This thesis is exclusively interested in the latter approach.

In order to compile GMIR to an executable file, GMRT makes use of the Clang [14] compiler and the LLVM [15] compiler back-end. This is accomplished by further compiling GMIR into LLVM's own intermediate representation (LLVM IR), and the tool that does so is named *GMIR2LLVM*.

When type information is available, efficient instructions will be generated. For instance, in the example above, the constant 10 is known to be a double so the LLVM IR instruction `store double` is generated. For situations where the compiler fails to infer the type, a call to a function from a runtime library will be generated. For example, in the case of the addition `ADD(r5, r6: Double)` the type of `r5` is unknown so the instruction will be translated into a call to a C++ function named `vm_GML_ADD` that can handle arbitrary values.

2.3.3 Runtime libraries

The crucial core of GMRT is the set of runtime libraries that are used when compiling GML and JavaScript. Everything that is compiled in a GameMaker project uses the runtime libraries in one way or another. For example, adding a sprite to an in-game object uses the `gmsprite` library, and fetching user keyboard input uses the `input` library. As seen in Figure 5, the libraries are designed to be modular and to be grouped into layers, minimizing coupling.

For our work, the scripting layer is the most important. It is this layer, along with the GMIR-to-LLVM compilation tool, that will be affected the most by our changes to the code. The scripting layer is primarily responsible for implementations of individual GMIR instructions, with behavior mostly conformant to the ECMAScript specification [13].

Layer	Description
Modules	User-facing APIs
Systems	Physics, collision, rendering…
Resources	Textures, fonts, sprites…
Framework	Multimedia abstractions: audio, rendering…
Scripting	Scripting language runtime
Core	Core engine concepts: rooms, objects…
Platform / Utils	Platform abstractions and utilities

Figure 5 : Overview of the layered runtime library architecture.

To better explain what we will be working on within the runtime, we can use the `PGET` instruction as an example. Getting the property of an object in GML or JavaScript such as `myObj.prop` will generate a call to the function `vm_GML_PGET_CACHED`. This function receives an object, property name and a cache and returns a property value. It does so by first trying to use the property cache if it is valid, otherwise it moves on to execute a C++ implementation of the ECMAScript `OrdinaryGet` abstract operation [13].

Our work will mainly focus on improving the property cache that is passed to the `PGET` and `PSET` functions in the scripting library. The caches are allocated as global variables in the LLVM code generation step, and passed as pointers when used in functions. As a result, changes to the property cache in the scripting library will need to be reflected in the LLVM code generation, by generating structures with perfectly matching layouts.

2.4 AOT compilation of JavaScript

The research on the optimization of dynamic (and especially prototype based) languages has mainly focused on just-in-time compilation. The fastest JavaScript engines all make use of JIT techniques and are developed outside of academia by large industry actors such as Google [16], Apple [17], Mozilla [18], Microsoft[6] [19], and Oracle [20].

A notable exception to this trend is Manuel Serrano, who conducts research on performant AOT compilation of JavaScript. Throughout the past two decades, Serrano has been developing a multi-tier JavaScript environment called *Hop* [4], [21], [22]. A central part of the Hop environment is *Hopc*, which is an AOT JavaScript compiler utilizing a Scheme backend. In practice, this means that Hopc employs JavaScript-specific optimizations such as hidden classes and property caches, before compiling the JavaScript code into Scheme. By doing so, it is able to reduce the complexity of the compiler implementation at the cost of introducing some limitations from converting the code to Scheme. However, Scheme-to-C compilation (the final part of the Hopc pipeline) has also been extensively researched, and Hopc is able to take advantage of the optimizations implemented there as well.

[6]Note that Microsoft ended support for its engine ChakraCore in 2021 when moving to the Edge browser to Chromium.

Another attempt at AOT compilation of JavaScript was conducted by Samsung [23], where researchers developed a type system and type inference algorithm designed to facilitate AOT optimizations. However, in contrast to the Hop compiler, it only works for a relatively restricted subset of JavaScript. Nonetheless, within this subset, the AOT compiler shows promising results in benchmarks for resource-constrained hardware.

Yet another interesting and different approach to AOT compilation of JavaScript was recently investigated by Chris Fallin at Fastly [5]. He makes use of a partial-evaluation technique known as the first Futamara projection [24], specializing an interpreter (namely, Mozilla's SpiderMonkey) for some given JavaScript bytecode and getting a WebAssembly program in return.

2.4.1 Profile-guided optimization

A great benefit of JIT compilation is the ability to optimize based on profiling information collected during runtime. Such *profile-guided optimizations* (PGO) can also be used in more limited forms in AOT compilers by running the compiler and program once to collect information and then again, using that information.

An example of such PGO techniques is applied in Hopc. The compiler makes use of profiling information to omit fast paths for JavaScript getters and setters if those language features are not used, thus avoiding runtime checks and shrinking the size of the executable [10].

Another example is research on shrinking the memory footprint of the hidden class graph [25]. To minimize the graph, profile runs are used to identify and remove intermediate states that are only accessed when initially creating an object, both optimizing the hidden class graph and improving the hit rate of inline caches. This research project made use of eJSVM, a framework for generating JavaScript virtual machines for embedded systems [26].

Similarly, researchers at the University of Illinois have used profiling between runs to reduce the startup time of JavaScript programs [27]. Normally, inline caches will at least have cold misses, which means missing on the very first access due to the cache being empty. This is avoided by pre-populating every inline cache based on information collected during previous runs.

3

Methods

The starting point of the project was to develop tools for profiling the behavior of hidden classes and property caches. Developing these tools first had two main benefits: it helped understand the intricacies of the runtime environment, and it provided a baseline to compare future improvements against.

3.1 Building profiling tools

The most important statistics to track are hit rates of property caches, characteristics of the hidden class graph, and memory consumption for both. In order to track these statistics, we adapted the *GMIR2LLVM* compiler to translate `PGET` and `PSET` instructions into calls to special-purpose profiling functions.

The custom profiling functions are fed slightly more information than their non-profiled counterparts, such as line number and file name. This information, along with the result of the property access, is collected and can be outputted continuously while a game is running, or just before terminating a game as aggregated data.

It is important to note that a self-imposed constraint was set to ensure that the profiling can be enabled without rebuilding any of the C++ runtime libraries. This allows end-users to make use of the tools, and more importantly, potentially use profile-guided optimizations to improve the performance of subsequent compilations of their games.

3.2 Visualizing data

We built an accompanying tool to interpret and visualize the output from profiled runs. Shown in Figure 6 is an example of how the collected cache information is used to generate reports with inline annotations. The annotation includes the overall hit rate, access count, and the number of unique hidden classes seen by this cache.

```
1 {
2   "filename": "/example.gml",
3   "line": 3,
4   "col": 26,
5   "hits": 9,
6   "misses": 1,
7   "property": "prop",
8   "unique_classes": 1
9 },
```

a) Captured property cache statistic.

```
1 function example() {
2   var obj = {prop: "Hello"};
3   show_debug_message(obj.prop)
      prop, Accessed: 10, hit rate: 90%,
      unique hidden classes: 1
4 }
```

b) Generated report with inline annotation.

Figure 6 : Raw profiling data translated into inline code annotation.

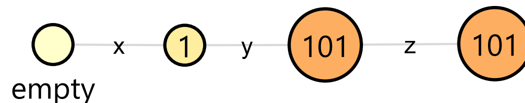
To visualize the hidden class graph, the profiling tool captures events that describe what happens to the graph. The tool can then be used to reconstruct an interactive version of the graph, as seen in Figure 7. The graph for a much larger program can be found in Appendix B.

```
1 var v2 = {x: 1, y: 2};
2 var v3 = {x: 1, y: 2, z: 3};
3 for (var i = 0; i < 100; i++) {
4     show_debug_message(
5       v2.x + v3.z
6     );
7 }
```

a) GML source code.

```
1 [{
2   "event": "new child",
3   "parent_name": "x",
4   "property_name": "y",
5   "hidden_class": "0x...",
6   "child_hidden_class": "0x..."
7 }, ...]
```

b) Captured hidden class event.



c) Generated graph. The nodes are scaled based on the number of accesses.

Figure 7 : Reconstructed hidden class graph based on profiling data.

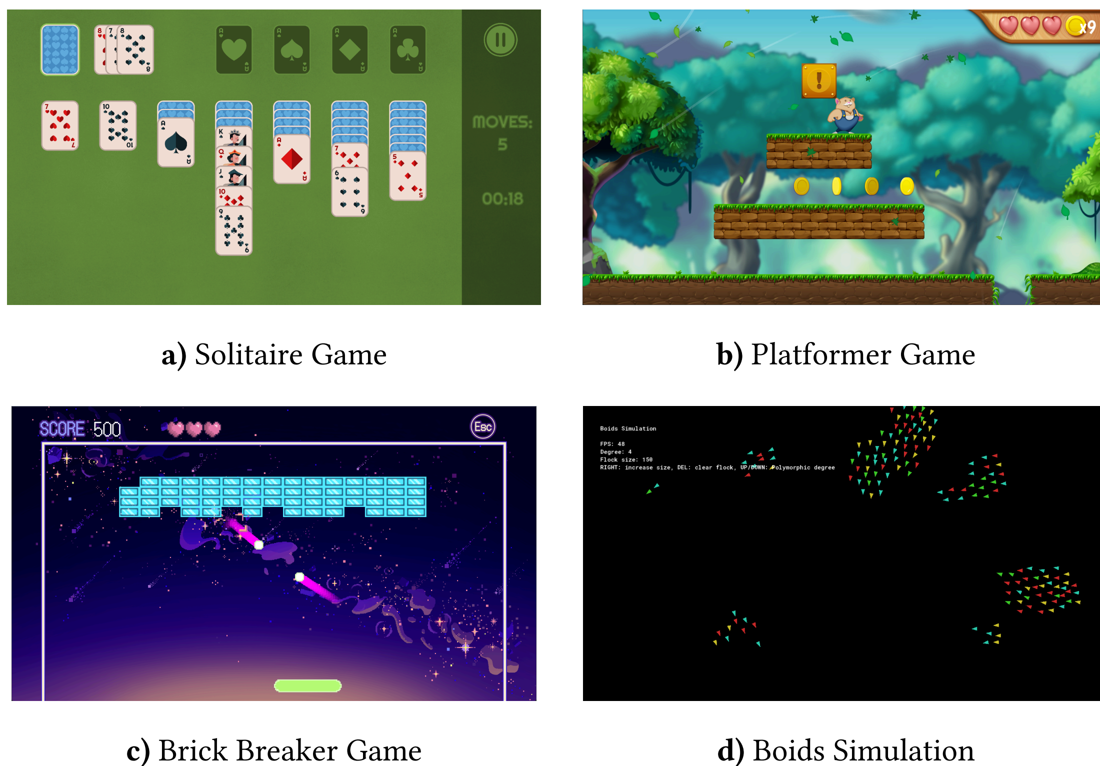
3.3 Benchmarking suite

One source of benchmarks was a small framework that Andrew Martin at GameMaker had set up prior to our thesis project. It includes GML ports of two programs from *The Computer Language Benchmarks Game* [28], namely *n-body* and *spectral-norm*. It also includes *simple-primes*, an original program which performs primality tests. Finally, there is also *simple-get* which focuses solely on the `PGET` instruction by repeatedly accessing a property in a loop, monomorphically.

A reoccurring issue with these micro-benchmarks was unexpected differences in execution time from small, seemingly insignificant changes made to the runtime libraries. This is likely due to unpredictable link-time optimizations made by the LLVM linker[7] since both the runtime libraries and the user's code are combined into a single executable. Consequently, the results collected from the micro-benchmarks may not entirely generalize to larger programs and games.

Real games generally require user input and exhibit non-deterministic behavior, this makes them far from ideal when it comes to systematically measuring performance. However, we have adapted a few of the tutorial games included with the engine (Figure 8) and made use of them when measuring performance and profiling property access patterns. To the extent possible, we used fixed random seeds and provided easily reproducible user inputs.

We also developed a simulation based on Craig Reynolds famed *Boids* flocking algorithm [29]. Notably, our simulation allows the degree of polymorphism to be adjusted by creating bird-oid objects with different hidden classes. Additionally, we created a crude fluid simulation to measure the impact of accesses to built-in properties. As will be detailed in Section 4.3.1, these properties do not utilize caching in the current runtime.



a) Solitaire Game **b)** Platformer Game

c) Brick Breaker Game **d)** Boids Simulation

Figure 8 : Some of the games used to measure performance and cache behavior.

All benchmarks were run on a Windows laptop with the specifications listed in Table 1. Only release builds of the runtime libraries were used, and all benchmarks exclusively focus on games compiled to executables using LLVM, as opposed to running them in the virtual machine. Note that tables of benchmarks should not be compared against each other; they are only intended to highlight relative differences between individual entries in the same table.

[7] Note that this is not an inherent limitation of LLVM, only a side effect of how the GMRT runtime libraries and compiler have been architected.

Table 1 : Specifications of the computer used in benchmarks.

Laptop	Acer Nitro 5
OS	Windows 11 24H2
Power mode	Best Performance
Processor	Intel Core i5-12500H
Graphics Unit	Nvidia Geforce RTX 4050
Memory	Samsung M425R1GB4BB0-CQKOL 2x8 GB, 4800 MT/s

3.4 Ensuring correctness

GMRT is designed to execute GML and JavaScript in a way that conforms to the ECMAScript standard. To help ensure that our changes do not deviate from that behavior, the *Test262* suite [30] is used. The test suite itself is organized into subdirectories covering different parts of the ECMAScript language specification. Of most relevance to GMRT is the *built-ins* subdirectory. It consists of over 10 000 individual tests and focuses on the behavior of built-in objects such as arrays and strings.

The GMRT repository includes a continuous integration workflow that can be triggered to run the entire test suite, which is useful to control that the pass rate does not decrease as a result of our changes to the code. At the time of writing, GMRT passes 73.2% of the tests in *built-ins*. Other subdirectories are not tested. Test262 only helps ensure correct behavior for JavaScript code. Sadly, the test suite for GML is much smaller and is an area of future improvement.

4

Initial Findings

Before evaluating further optimizations, it is essential to first establish an understanding of the current implementation of GMRT. This chapter presents our initial observations, highlighting areas for potential improvement.

4.1 Monomorphic property caches

The property caches found in GMRT are monomorphic — they only contain a single property index and associated hidden class pointer. Thus, if the object shape alternates between accesses, the cache will never produce a hit. Figure 9 contains an example program that demonstrates this behavior. Calling the function with the argument `N=2` results in a cache hit rate of 0%, while the function is called with `N=1` the hit rate will be 100% (excluding the initial miss when the cache is first populated).

```
1  function readA(obj) {
2      return obj.a;
         a, Accessed: 1000, hit rate: 0%, unique hidden classes: 2
3  }
4
5  function myFunc(iterations, N) {
6      var objects = [{a: 1}, {a: 1, b: 2}];
7      var s = 0;
8      for (var i = 0; i < iterations; i++) {
9          var obj = objects[i % N];
10         s = readA(obj);
11     }
12     return s;
13 }
14
15 myFunc(1000, 2);
```

Figure 9 : Example of a polymorphic function. Adapted from Figure 2 in *Property Caches revisited* [10].

While examples such as the one found in Figure 9 are effective at highlighting use cases for polymorphic caches, they do not necessarily reflect the type of code found in GameMaker games. Shown in Table 2 are property cache hit rates for the four games mentioned in Section 3.3, with two different configurations for the boids simulation. The hit rates show that monomorphic code is used almost exclusively, with our polymorphic custom boids simulation being a large outlier. However, the Solitaire game does show that there is still some performance to be gained from efficient polymorphic caches, assuming misses are more costly than hits.

Table 2 : Hit rates for games when using monomorphic property caches.

Benchmark name	Cache hit rate
Solitaire Game	94.31%
Platformer Game	99.26%
Brick Breaker Game	99.96%
Boids Simulation, `degree=1`	99.99%
Boids Simulation, `degree=2`	81.87%

4.2 Cache invalidation and dynamic inheritance

Mature JavaScript engines usually make assumptions about what code users will actually author, aiming to optimize only a subset of the programs that the language semantics allow for. A prime example of this is changing an object's prototype during runtime, a feature known as *dynamic inheritance* [2].

In JavaScript, dynamic inheritance can be accomplished through the static method `Object.setPrototypeOf` [31]. Most engines assume that using this language feature is a rare occurrence and allows it to be very inefficient in order to otherwise improve general performance. Quoting the MDN Web Docs, making use of this feature is "by the nature of how modern JavaScript engines optimize property accesses, currently a very slow operation in every browser and JavaScript engine" [31]. Similar difficulties arise if an objects prototype is mutated by adding or removing properties since it affects inheritance.

It is worth noting that this issue is not only limited to JavaScript code used in GameMaker but to GameMaker Language as well. Regardless if GML actually exposes the same control over prototypes, it may still access and operate on objects created in JavaScript that does.

GMRT implements `Object.setPrototypeOf` but there is limited support for traversing and invalidating nodes in the hidden class graph and property caches. Unless these capabilities are implemented, GMRT will be limited to a more conservative set of optimizations than what is typically found in other implementations of JavaScript.

4.3 To cache or not to cache?

Currently, the property cache and hidden class optimizations are not fully utilized for all objects in the engine. For mainly historical reasons, GameMaker features two kinds of objects[8]. The first being *instances* which are created from an object template, similarly to class-based object-oriented programming. The other, more recent, addition are GML *structs* which share the same semantics as a JavaScript object.

4.3.1 Built-in properties on instances

One of the major differences between instances and structs is that the former always hold some built-in properties, the most notable being `x` and `y`. This brings some performance benefits for C++ code within the runtime libraries since they may directly access these properties at a known memory location. However, it also negatively impacts the performance of any GML or JavaScript code authored by the user.

The compiler always attempts to use the cache when accessing a property using the `PSET` or `PGET` GMIR instructions. This adds additional overhead when accessing built-in properties on an instance because they will never populate the cache. Instead, they will fail to find a matching entry in the cache and afterwards use an *accessor property* [13], more commonly referred to as getters and setters.

We have implemented micro-benchmarks to compare the runtime performance when using an accessor property for a built-in such as `x`, versus any other property that can make use of the cache. The runtime performance penalty of using a built-in property is as much as a 1000–1500% time increase. This is very significant, especially when taking into consideration that the built-in properties are some of the most commonly accessed in a typical GameMaker game.

In the “Brick Breaker” game from Figure 8, 47% of all property accesses are for built-in properties. For the “Platformer” game, 54% of accesses are for built-in properties. The high usage of property accessors could be considered atypical in comparison to other JavaScript environments. For instance, Serrano [10] fails to find any standard JavaScript benchmarking suites that include programs using accessor properties.

4.3.2 Methods

A method can be attached to both instances and structs. They are also found in stand-alone *script assets*[9], and many methods are built-in as part of the runtime libraries. Table 3 summarizes what GMIR instruction will be generated when calling a method in each of these scenarios.

[8] At least this is how the concept is exposed to users writing GML code. Technically, all objects in the scripting runtime are ECMAScript objects but with an additional relation to some *GameMaker Resource* that may contain both data and special behavior.

[9] Arguably these could be considered *functions* instead of methods. The GameMaker Language reference [7] refers to them as “script functions”.

Table 3 : Generated GMIR instruction for calls to different types of methods.

Method type	**GMIR instruction**	**Uses property cache**
Built-in methods	Static call	—
Script function	Static call	—
Instance	Static call or property get	✓
Struct, owned	Property get	✓
Struct, static	Property get	×

Calls to built-in methods and methods in script assets make use of the GMIR instruction `STATIC_CALL`, directly providing the address of the method and the attached object (referred to as `this` in JavaScript and `self` in GML). The compiler attempts to do the same for methods attached to an instance, however, it may fail to do so, especially if the instance is referenced from another object. Figure 10 displays an example where the compiler must first use a property get instruction to find the memory address of the method.

GML source code

```
1 // Creating an instance of
2 // 'some_object' at
3 // position (0, 0)
4 var my_instance =
5   instance_create_layer(
6     0,
7     0,
8     "Instances",
9     some_object);
10 // Method call
11 my_instance.my_method();
```

Generated GMIR

```
1 r4 : Double = LDC(c0(0))
2 r5 : Double = LDC(c0(0))
3 r6 = LDC(c1("Instances"))
4 r8 = STATIC_CALL(
5   c3(external function instance_create_layer),
6   self_r0,
7   r4 : Double,
8   r5 : Double,
9   r6,
10  r2_some_object)
11 my_instance_r3 = MOV(r8)
12 r9 = PGET_A(my_instance_r3, c4(`my_method`))
13 r10 = CALL(r9, my_instance_r3)
```

Figure 10 : The compiler fails to reason about the return value of `instance_create_layer` (stored in `r8`) and must resort to a property get operation to find `my_method`.

When calling a method attached to a struct, the additional property get operation will always be performed. Notice that Table 3 includes two types of methods on structs: *owned* and *static*. The property cache is utilized when accessing an owned method; for static methods, the cache may not be used for the reasons outlined in Section 4.2.

Note that a static method in GML does not refer to the same concept as static methods in other popular object-oriented languages such as Java. A static method is still tied to a struct and has access to `self`. Instead, it refers to the fact that only one method will be created and shared among all constructed structs instead of each and every struct owning a separate copy of the method. Methods defined in JavaScript code are represented as *owned methods* in the runtime.

To highlight the performance impact, we have created multiple versions of our Boids simulation program that utilizes functions, static methods, or owned methods for all vector operations. The execution time for running a simulation with some fixed parameters can be found Table 4.

Unsurprisingly, using functions is the most performant option. Owned methods perform the worst, even though they, unlike static methods, make use of the property cache. This is caused by the additional overhead of creating copies of methods for each and every struct used.

Table 4 : Benchmarking different approaches to vector methods.

Benchmark	Execution time
Free-standing function	265 ms
Static method	289 ms
Owned method	479 ms

4.4 Naive property cache allocation

As previously mentioned, property caches are created while translating GMIR to LLVM IR. Note that a separate cache will be created for every `PGET` and `PSET` instruction. This is a naive approach that results in more caches than needed. For instance, consider the pre-increment operator, which will be translated into multiple GMIR instructions. As seen in Figure 11, the expression `++myObj.myProp` will be translated into both a get and set operation, each of which will have a separate property cache. To shrink the size of a compiled binary, the GMIR2LLVM compiler should ideally perform a cache allocation pass, merging entries when possible.

GML source code

```
1 var myObj = {myProp: 0};
2 ++myObj.myProp;
```

Generated GMIR

```
1 // Object literal
2 r4 = LDC(c0(global `@@GML_NewObject@@`))
3 r5 = CALL(r4, self_r0)
4 r6 : Double = LDC(c1(0))
5 PSET(r6 : Double, r5, c2(`myProp`))
6 // Pre-increment operation
7 myObj_r3 = MOV(r5)
8 r7 = PGET_A(myObj_r3, c2(`myProp`))
9 r8 : Double = LDC(c3(1))
10 r9 = ADD(r7, r8 : Double)
11 PSET(r9, myObj_r3, c2(`myProp`))
```

Figure 11 : GMIR generated for pre-increment expression.

It is worth noting that it is relatively hard for a compiler to statically reason about property accesses since they may in fact be a JavaScript getter and perform any arbitrary side effect — including changing the shape of the object being accessed. Consider the JavaScript program in Figure 12. It is unfavorable if both accesses to the property `x` share a cache since the hidden class used by the object changes when accessing `y`. We argue that a reasonable constraint would be merging property caches only as long as no other property is accessed or function is called in-between[10].

[10] Even with this constraint some rare may still suffer from an additional cache miss. For instance, consider a getter function which the first time it is called performs an expensive operation and caches the result in another property, thus changing the hidden class of the object.

```
1  const myObj = {
2    get y() {
3      delete this.x;
4      this.z = 30;
5      this.x = 10;
6      return 20;
7    },
8
9    x: 10,
10 };
11
12 console.log(
13   myObj.x + // Shape of myObj, y: idx=0, x: idx=1
14   myObj.y + // Shape of myObj, y: idx=0, z: idx=1, x: idx=2
15   myObj.x,
16 );
```

Figure 12 : Accessor property modifying the shape of an object.

4.5 Hidden class graph

Throughout the execution of a program or game, GMRT builds a graph of hidden classes as new properties are added to objects. As seen in Figure 13, each node in the graph keeps information about its corresponding property. Additionally, each node has a cache that may also include all preceding properties (note that this is a different concept from the property cache). When a new node is created, the child takes ownership of the cache, leaving the parent with an empty cache. The parent's cache can then be repopulated if needed. This approach leverages the fact that most programs build up an object property by property and then interact only with the complete object. Intermediate nodes are never accessed again in this case. A hidden class graph demonstrating this pattern is shown in Appendix B. The graph shows access counts for hidden class nodes from a run of the Solitaire template game.

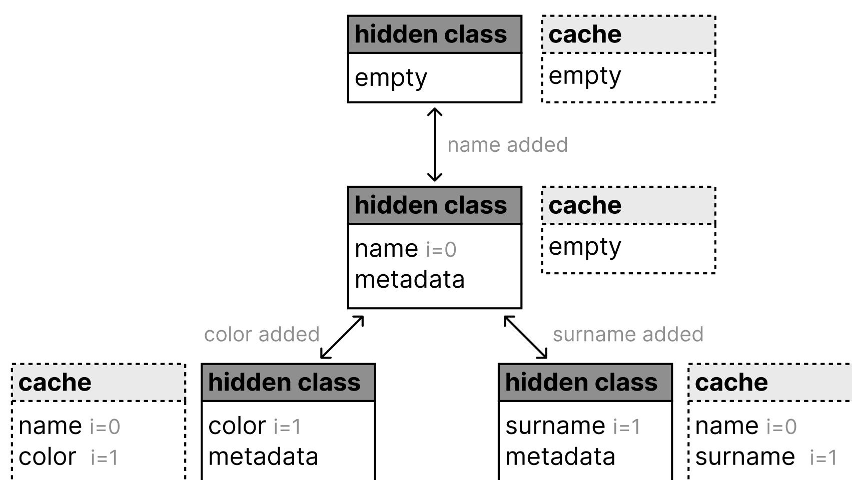


Figure 13 : Hidden class caching for the example from Figure 2.

The implementation in GMRT deviates from the described model when a branch reaches a depth of 32 properties. When this occurs, the leaf node is detached from the graph, marked as mutable, and its cache of properties can grow indefinitely without creating new nodes. In practice, we encounter this scenario if an object is used as a dictionary data structure. This avoids the need to pollute the graph with an additional node for each and every stored key-value pair.

There are more advanced implementations of hidden class graphs that leverage shared caches between nodes to a greater extent, such as in Google's V8 engine [32]. In their implementation, only leaf nodes keep caches of their properties. All parent nodes keep a reference to the leaf's cache instead, with an extra restriction of which properties it is allowed to read.

Additionally, the hidden class graph found in GMRT is not currently capable of tracking prototypes like Self [2] and most JavaScript engines do [33], instead a pointer to the prototype is stored directly on each and every object.

5

Experiments & Results

Based on our initial findings, we implement and investigate a set of independent of optimizations targeted at property access operations. This includes polymorphic property caches that utilize profiling information and caches that are extended to support accessor properties. We also present benchmarks to evaluate our designs, comparing against the existing implementation of GMRT.

5.1 Polymorphic property caches

The notion of a polymorphic cache was first introduced in 1991 by Hölzle, Chambers and Ungar for the Self language [12]. It aims to avoid costly property lookups in polymorphic code by storing several entries in a single property cache, thereby increasing the hit rate.

5.1.1 Established designs

The implementors of Self discuss different variations in their 1991 paper, but the core idea is to use a cache that can grow as needed so that entries will never be replaced. They do, however, suggest setting a maximum to avoid impractically large caches at call sites that are *megamorphic*, where the degree of polymorphism could be several orders of magnitude higher.

As for megamorphic call sites, Serrano and Feeley have made more recent developments in designing property caches suitable for AOT compilation [10]. To avoid an asymptotic complexity that scales with the degree of polymorphism, they introduce the idea of *virtual tables* in the Hop compiler. Doing so allows the cache to perform lookups of monomorphic accesses just as before, with a single comparison with hidden classes. In the case of a cache miss, the property will be resolved using a virtual table, meaning, the time complexity of lookups stays constant. One of the main drawbacks of this design is that the virtual tables are dynamically allocated during runtime (based on a cache miss threshold).

5.1.2 Our implementation

Ungar among others concludes that monomorphic code is very common and that monomorphic caches reach around 95% hit rate in Smalltalk systems [1], [34]. This largely aligns with the hit rates we have seen when initially benchmarking GameMaker games, which does not (yet) favor polymorphic programming patterns. However, since GameMaker makes use of ahead-of-time compilation, we need to decide whether a cache should be polymorphic before

running any code. The already high hit rates and lack of opportunities to fine-tune the caches during runtime require that the polymorphic cache has minimal memory overhead and negligible performance overhead.

With this requirement in mind, we designed a polymorphic cache with no additional overhead for cache hits. Given that hit rates are so high, it should be sufficient for matching the performance of the monomorphic cache.

Importantly, we also built a system that allows the size of each cache to be individually configured during compilation. This lets us use profiling information to fine-tune cache sizes, minimizing the memory overhead. This process is explained in further detail in Section 5.1.3.

A C++ definition for the polymorphic cache is shown in Figure 14. We use two fields `capacity` and `replaceIndex` to implement a lightweight ring-buffer. The latter is used to keep track of the next entry to replace in the case of a full cache miss. When this index reaches the end, it resets back to the start. Notably, we also put `entries` as the last field of the struct in order to use it as a flexible array member. The size of this array is not known when building the runtime libraries, instead we rely on the `capacity` field which is populated when compiling a game.

```
1  struct PropertyCacheEntry {
2      HiddenClass* hiddenClass;
3      int propertyIndex;
4  };
5
6  class PropertyCache {
7      int capacity;
8      int replaceIndex;
9      PropertyCacheEntry entries[];
10 }
```

Figure 14 : C++ definition of the polymorphic cache.

During execution of a cached `PGET` or `PSET` instruction, if we find a matching entry somewhere in the array, we will move it to index zero if needed. Similarly, when new entries are added, they are inserted at index zero and the previous first entry will replace the entry at `replaceIndex`. This allows us to unconditionally check the first entry before looping through the rest, which also proves to be sufficient for the vast majority of cache accesses given the high hit rates. Doing so circumvents the otherwise constant overhead of looping through the entries for every access.

Table 5 presents a comparison between the existing monomorphic cache and our polymorphic implementation. For the Boids simulation and the small polymorphic example, we see significant speedups along with better hit rates as expected. However, we also include benchmarks with monomorphic code to measure potential overhead introduced by polymorphic caches. We see mixed results in the monomorphic benchmarks included, as the n-body benchmark slows down while the boid simulation speeds up. As mentioned, small variances in execution times such as these could be attributed to differences in link-time optimizations rather than direct overhead, especially since the monomorphic cache was outperformed in the monomorphic boid simulation.

Table 5 : Comparing the performance of polymorphic and monomorphic caches.

Benchmark	Cache hit rate		Execution time	
	Monomorphic	Polymorphic	Monomorphic	Polymorphic
Boid Simulation, `degree=1`	100.0%	100.0%	302 ms	287 ms
Boid Simulation, `degree=4`	81.7%	100.0%	456 ms	309 ms
n-body	100.0%	100.0%	1848 ms	1921 ms
Figure 9 example, 10^6 iterations	0.0%	100.0%	435 ms	259 ms

5.1.3 Using profiling information

By utilizing profiling information gathered from previous runs, we can determine a suitable size for the cache at each individual call site. For example, if a cache encountered eight unique hidden classes during a complete run, we set the size to eight. If no profiling information is available for a given cache, we set it to be monomorphic by default.

This approach avoids unnecessarily large caches for the games that do not need it, while still offering optimized polymorphic accesses in the rare games that do. In practice, we also need to enforce an upper limit of cache capacity so that megamorphic call sites do not yield infeasibly large caches.

In Table 6 we highlight the difference in memory consumption between naively allocated caches (4 entries) and our profile-guided sizes. For a simple game with strictly monomorphic code such as Brick Breaker, we notice a 75% reduction as expected. However, even in games utilizing polymorphic code such as our Boid Simulation, we see a significant reduction in memory usage. Even when designing the program to be polymorphic, many call sites are still monomorphic and can save memory compared to an homogenous capacity configuration.

Table 6 : Comparing memory usage for uniform and profile-guided cache sizes.

Benchmark	Entry count	
	Homogenous `capacity=4`	Heterogenous `capacity=[1-4]`
Boid Simulation, `degree=4`	440	249
Brick Breaker Game	120	30
Solitaire Game	1012	327

5.1.4 Speed versus space

On certain architectures, it may be more valuable to prioritize a lower memory footprint. We therefore propose an alternative design that has neither of the additional fields seen previously. Instead it reserves two bits in the property index, one for marking the final entry, and one for marking that the entry should be replaced. A C++ definition for this design is shown in Figure 15, and a visual overview in Figure 16.

```cpp
struct PropertyCacheEntry {
    HiddenClass* hiddenClass;
    int propertyIndex;
};

class PropertyCache {
    PropertyCacheEntry entries[];
}

constexpr int terminateBit = 0b10000000000000000000000000000000;
constexpr int replaceBit   = 0b01000000000000000000000000000000;
constexpr int indexMask    = 0b00111111111111111111111111111111;
```

Figure 15 : C++ definition of a space-efficient polymorphic cache.

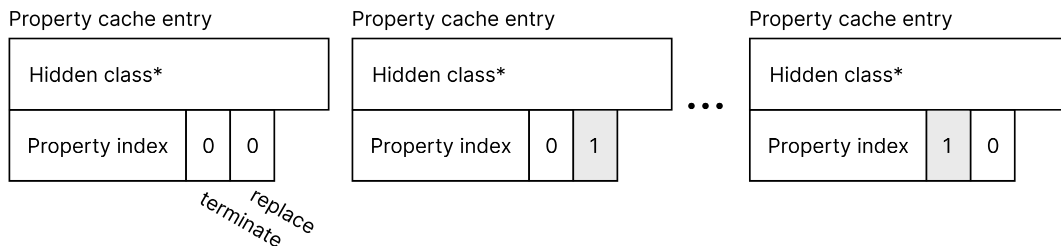


Figure 16 : A visual representation of the compact cache layout.

Similar to the speed-efficient design, we utilize the first entry for unconditional checks. However, even in the best case this cache has an overhead of one instruction per access. This is because the property index needs to be extracted using a bit-mask. On the other hand, the omitted fields save 64 bits per cache assuming typical memory padding and alignment. This equates to a third of the memory used by speed-efficient caches of size 1.

Table 7 presents a comparison between the cache designs for two runs of the Boid simulation, with and without polymorphic accesses. As expected, we see a slight decrease in execution time for the cache using separate fields, while the cache using reserved bits yields significant improvements in memory usage.

Table 7 : Comparing the performance of the two polymorphic caches.

Benchmark	Estimated bytes used		Execution time	
	Space	Speed	Space	Speed
Boid Simulation, `degree=1`	1792	2688	292 ms	287 ms
Boid Simulation, `degree=4`	3968	4864	314 ms	309 ms

5.2 Grouping hidden class nodes

In the current runtime, a memory allocation is made for every new node in the hidden class graph. As a result, pointers are used to identify and reference hidden classes. On most architectures today, pointers are 64 bits wide but a 32-bit integer would more than suffice to enumerate all hidden class nodes in any given game. It is therefore possible to save memory by instead using 32-bit integers to reference the nodes.

To realize this idea, we implement a global, dynamically growable, memory arena to store the entire hidden class graph. A node is then identified by its corresponding index in the arena, starting from the root which occupies index zero. Flattening a hierarchical structure and referencing its elements by index like this is a common compiler technique [35], [36].

The benefits of using an arena allocation are manyfold. Firstly, other data structures and in particular the property cache save memory. Due to padding and alignment, the original monomorphic cache effectively halves its size from 128 bits to 64 bits. Secondly, it avoids frequent yet costly heap allocations and possibly reduces memory usage further in cases where metadata is stored for each allocation. Thirdly, it opens the door for future profile-guided optimizations that require traversal of the hidden class graph, such as collecting information to remove intermediate nodes. Traversing the graph was previously not guaranteed to be possible, since certain nodes may be detached during runtime.

5.3 Caching accessor properties

As discussed in Section 4.3.1, accessing built-in properties such as `x` and `y` is a slow operation. This performance penalty is caused by the fact that built-in properties are implemented as *accessor properties* (getters and setters) which, unlike ordinary *value properties*, do not make use of the property cache.

There are multiple potential solutions to alleviate these performance issues. One approach would be to remove the special treatment of built-in properties and instead inherit them as value properties from some base prototype. This would arguably be an elegant solution but requires unifying the object model for GML structs and instances. It would also require a rewrite of all legacy runtime libraries that make use of these properties. Accomplishing such a rewrite without breaking compatibility with previous versions of the engine is a task deemed far beyond the scope of this thesis.

Another approach is allowing the compiler to generate code that accesses the property if it is known to be a built-in property for an instance object. However, as demonstrated in Figure 10, this will not work for situations where the compiler fails to statically reason about the type of an object. Both of these approaches also fail to improve the actual performance of accessor properties, which users may want to use for purposes other than built-in properties.

To solve the more general problem of slow accessor property lookups, Serrano and Feeley [10] suggest extending the property cache with an additional hidden class pointer `accessorClass`. When accessing a property, the general behavior remains the same, comparing hidden class pointers to ensure the object's shape matches the cached property index. It is only if the first

check fails an additional comparison is performed to see if the object's hidden class matches `accessorClass` and if so, the cached value can be assumed to be a valid accessor property. This approach only adds runtime overhead on cache misses, the only downside being additional memory consumption.

5.3.1 Our implementation

To support caching accessor properties, we have made modifications to the original monomorphic cache, as seen in the C++ struct definition shown in Figure 17. Unlike Serrano and Feeley's implementation [10] in the Hop compiler, accessor properties in GMRT are not always stored as owned properties on an object. For this reason, we are required to store a union between an index and a pointer to an accessor property. This requires additional memory compared to their implementation.

```
1 struct PropertyCache {
2     HiddenClass* valueClass;
3     HiddenClass* accessorClass;
4     union {
5         int index;
6         Accessor* accessor;
7     };
8 }
```

Figure 17 : Property cache data type extended to support accessor properties.

When measuring performance of the accessor cache, our first benchmark is a version of *simple-get* that was rewritten to use the built-in property `x`. The second benchmark is a crude particle fluid simulation, utilizing instances as particles along with their built-in `x` and `y` properties. Table 8 presents results from these benchmarks, where we notice speedups with factors of around 2.9 for *simple-get* and 2.1 for the fluid simulation, both of which make use of instance objects and built-in properties. The other benchmarking programs do not make use built-in properties or any other accessor properties and as expected we see no noteworthy performance penality.

Table 8 : Comparing execution times of the accessor cache.

Benchmark	**Execution time**	
	Original cache	Accessor cache
simple-get, built-in properties	1341 ms	458 ms
Fluid Simulation	1174 ms	543 ms
n-body	1916 ms	1920 ms
Boid simulation, `degree=1`	281 ms	275 ms
Boid simulation, `degree=4`	405 ms	396 ms

5.3.2 Issues with inherited properties

An issue with the presented design is that it allows objects with inherited accessor properties to utilize the property cache without verifying that they inherit from the same prototype. Figure 18 demonstrates an example with two objects sharing the same hidden class but inheriting the accessor property `foo` from different prototypes. This causes incorrect behavior when the two objects make use of a property cache after one another.

To address the problem, the hidden class graph in GMRT could be extended to incorporate prototype information. Alternatively, a slightly less performant but simpler solution is to store an `accessorClass` and an `accessorPrototype` in the property cache and validate them both against the accessing object.

```
1 let proto1 = { get foo() { return "Hey!"; } };
2 let proto2 = { get foo() { return "Bye!"; } };
3 // Same hidden class but different prototypes
4 let o1 = Object.create(proto1);
5 let o2 = Object.create(proto2);
6
7 const readFoo = (obj) => {
8     // The accessor will be cached here
9     show_debug_message(obj.foo);
10 }
11 readFoo(o1); // Expecting "Hey!"
12 readFoo(o2); // Expecting "Bye!"
```

Figure 18 : JavaScript example exposing an issue with objects sharing hidden class but having different prototypes.

5.3.3 An incomplete solution

Even with the changes mentioned in Section 5.3.2, the accessor property cache optimization is dependent on a cache invalidation mechanism to function correctly. For instance, consider the JavaScript program from Figure 19 that makes use of both accessor properties and dynamic inheritance. When the prototype changes, the cache would need to be invalidated to ensure that the correct accessor property is called.

```
1 const celestialPrototype = {
2   get greeting() { return "Hi! I'm a celestial object"; }
3 };
4 const dogPrototype = {
5     get greeting() { return "Woof! I'm a dog"; }
6 };
7 const pluto = Object.create(celestialPrototype);
8
9 const greet = () => { show_debug_message(pluto.greeting); } // prop cache
10 greet(); // Expecting "Hi! I'm a celestial object"
11 Object.setPrototypeOf(pluto, dogPrototype);
12 greet(); // Expecting "Woof! I'm a dog"
```

Figure 19 : JavaScript example exposing an issue when caching accessor properties while making use of dynamic inheritance.

5.4 Grouping property caches

Section 4.2 and Figure 19 above motivate the need for a mechanism that allows property caches to be invalidated if certain invariants are broken. A first step in this direction is to allow property caches to be traversed during runtime.

The current implementation of the GMIR2LLVM tool creates an LLVM global constant for each property cache, this makes the actual memory location of each constant unknown during code generation. The unknown placement makes it infeasible to traverse the caches during runtime, thereby ruling out cache invalidation.

5.4.1 Our implementation

Our solution was to ensure a contiguous linear memory layout of the caches by grouping the caches in *pools* with sizes known at compile-time. Each compiled file (LLVM module) contains an array for all property caches, with an extra prepended element that contains the length. These arrays are then registered in a global list during runtime, making the location of all caches known. See Figure 20 for a visual overview.

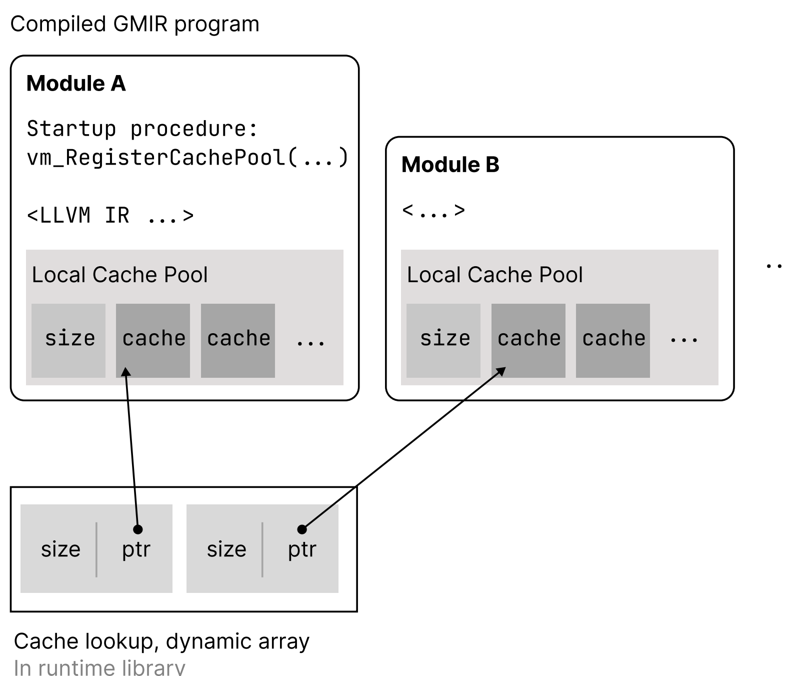


Figure 20 : Code generation of cache pools.

It is possible to avoid registering the cache pools during runtime, since all information is in fact known during compilation. For instance, one approach would be to share a single pool between all modules. However, having separate *local* pools has some clear benefits. In particular, it makes both caching and parallel compilation of files much easier to implement.

5.4.2 Sharing property caches

Having more control over cache allocation also opens the door to more novel optimizations. As described in Section 4.4, some property caches could in fact be shared with neighboring call sites. This has the potential to both decrease memory usage and increase cache hit rate. For example, encountering a new shape of `obj` in the expression `obj.a + obj.a + obj.a` would only produce a single cache miss, instead of the original three.

We implement this optimization in the same pass as when the size of the cache pool is determined. By including additional state that is updated for each visited GMIR instruction, it is possible to determine whether the previous generated cache can be shared with the current instruction.

We define a whitelisted subset of the GMIR instructions, encountering any instruction outside of this subset will require a new property cache to be used. As a result, a cache can be shared when the following two conditions are met: only whitelisted instructions have been seen since the last new cache, and the current cached instruction uses the same object and property name. In general, the whitelisted subset includes arithmetic, logic and comparison instructions. In the future this list may be extended, for instance, calls to some built-in functions could be allowed if their side-effects are known by the compiler.

Table 9 lists the number of caches needed with and without this optimization for the template games found in Figure 8. The amount of memory saved greatly differs depending on the game, but between 5%–13.5% of caches are removed for these examples.

Table 9 : Sharing caches between call sites.

Benchmark	**Number of caches**		**Caches removed**
	Original	With sharing	
Boid Simulation	317	301	5.0%
Solitaire Game	1039	971	6.5%
Platformer Game	1510	1396	8.1%
Brick Breaker Game	339	293	13.5%

5.5 Invalidating property caches

Thanks to the grouping concept described in the section above GMRT is capable of invalidating property caches when needed. We have implemented a simple invalidation policy inspired by the one found in Hopc [10]. All caches are invalidated when either:

(a) an object changes prototype (i.e. `setPrototypeOf` in JavaScript),
(b) a property is removed from a prototype, or
(c) a property is added to a prototype.

The need for (a) is demonstrated in Figure 19. The two other measures are taken to handle slightly different scenarios when the inheritance chain changes. For instance, consider the program in Figure 21 where an inherited property is removed and the property cache needs to be invalidated.

```
1 let p2 = { get foo() { return 23; } };
2 let p1 = { get foo() { return 43; } };
3 Object.setPrototypeOf(p1, p2);
4 let o = Object.create(p1);
5
6 const readFoo = (obj) => {
7   show_debug_message(obj.foo);
8 }
9
10 readFoo(o);
11 delete p1.foo; // Removing a property from
12                // the inheritance chain <invalidate cache>
13 readFoo(o);    // expected to search deeper and find 23.
```

Figure 21 : JavaScript program showcasing how the inheritance chain can be effected by adding or removing properties on a prototype.

This mechanism is the first step towards making better use of property caches in GMRT. Allowing both accessor properties and methods to be cached in the future. However, note that our implementation is limited compared to the ones found in Hopc [10], Google's V8 [33] and most other engines. These limitations are further discussed in Chapter 6.

5.6 Bringing it all together

Many of the optimizations previously discussed in this chapter are independent of each other and will require modifications to function together. This section presents a sketch of how the polymorphic property cache may be combined with the accessor property cache and arena allocated hidden class graph.

There are several potentially interesting approaches to designing a polymorphic cache with support for accessor properties. Given that the benchmarked examples show an even split between accessor and value properties, prioritizing one over the other is impractical. We therefore propose a solution that generalizes to a large variety of coding patterns by making further use of profile-guided optimization.

The implementation requires extending the PGO pipeline to also include statistics of unique hidden classes seen for accesses to accessor properties, instead of strictly tracking value properties. Using this information, we can design a cache that employs two separate ring buffers inside a single entry array. The C++ data types for such an implementation are shown in Figure 22. To avoid consuming more memory than necessary, we restrict the maximum cache capacity to 2^8 in order to use 8-bit integers for every field. Such a restriction should not result in any noticeable limitations on end-user code. In the rare case of megamorphic programs, it is undesirable to use a polymorphic cache anyway.

```
1 // Type safe wrappers for the hidden class nodes have
2 // been replaced with 'int'.
3
4 struct PropertyCacheEntry {
5   union {
6     int index;
7     Accessor* accessor;
8   };
9   int hiddenClass;
10 };
11
12 class PropertyCache {
13     unsigned char valCapacity;
14     unsigned char valReplace;
15     unsigned char accCapacity;
16     unsigned char accReplace;
17
18     PropertyCacheEntry* entries[];
19 }
```

Figure 22 : Polymorphic cache structure supporting both accessor and value properties.

When profiling information is unavailable, `valCapacity` and `accCapacity` both default to 1. This maintains its original functionality for value properties, as well as the functionality presented in Section 5.3. On the other hand, after a profiling run, it may be detected that a specific cache encounters five accessor properties but no value properties, for example. Using that information would allow the cache to contain the exact five accessor entries needed, while wasting no space for value properties.

In the implementation presented, it is also possible to check the first entries outside of any loops as proposed in Section 5.1, by placing recent entries at the start of the respective ring buffer. If those checks fail, two loops are used to check all possible entries, one for each type of entry. Separating the entries and loop logic in this way allows us to fetch the value through the accessor directly, instead of returning an accessor and conditionally executing it later on. Figure 23 presents an implementation of the *GetProperty* function, utilizing said loops to efficiently return a value.

```
1 std::optional<Value> GetProperty(int hiddenClass, Object* obj) {
2     for (int i = 0; i < valCapacity; i++) {
3       if (entries[i].hiddenClass == hiddenClass) {
4         return obj->properties[entries[i].index];
5       }
6     }
7     for (int i = valCapacity; i < valCapacity + accCapacity; i++) {
8       if (entries[i].hiddenClass == hiddenClass) {
9         return Call_Accessor_Get(obj, entries[i].accessor);
10      }
11    }
12    return {};
13 }
```

Figure 23 : Implementation of the *GetProperty* function for the merged cache.

6

Discussion

The primary accomplishment of the project has been our profiler which allows property caches and hidden classes in the GameMaker runtime to be examined in much greater detail. As described in Chapter 4, this helped us identify multiple limitations and potential improvements to the runtime and compiler toolchain. Additionally, we have implemented and benchmarked the following list of features.

- **Polymorphic property caches** *(Section 5.1.2)*
- **Pipeline for profile-guided optimizations** *(Section 5.1.3)*
- **Grouping hidden class nodes** *(Section 5.2)*
- **Accessor property caches** *(Section 5.3)*
- **Grouping property caches** *(Section 5.4.1)*
- **Sharing property caches** *(Section 5.4.2)*
- **Basic property cache invalidation** *(Section 5.5)*

6.1 Limitations

While our changes to the GameMaker engine have proved fruitful, there are aspects and potential drawbacks left for consideration. In this section, we discuss the reliability of our benchmarks, and the binary size of compiled games.

6.1.1 Benchmarks

The most glaring issue throughout the project has been the lack of reliability and consistency in benchmarks. While metrics such as hit rates and memory usage are completely accurate, the execution time depends on a number of factors outside of the scope of this thesis. Due to the high variance in execution time, it becomes much harder to analyze smaller impacts of code changes.

We suspect that link-time optimizations are responsible for most of the variance, and that the problem can be remedied by having a much larger benchmarking suite. Given enough benchmarks, the effect of link-time optimizations on the timing would average out enough to produce results more representative of real world usage. The reason for not including a larger benchmarking suite is that each benchmark needs to be translated to GML, and was therefore deemed too time-consuming for this project. Furthermore, the benchmarks themselves should

use coding patterns that are representative for GameMaker games, so as to not optimize for the wrong type of code. Finding such benchmarks is another task that proves to be time-consuming.

6.1.2 Binary size

Much of the focus of the thesis has revolved around reducing memory usage and runtime speed of the property caches. In practice however, *binary sizes* of compiled projects may also be relevant to end users, particularly for web targets where transfer speed can be an important consideration.

While reducing the number of caches has a definite positive impact on the memory usage, our changes to the code base as a whole may have resulted in a net increase in binary size. Resulting binary sizes were not tracked during the thesis work and is left as inconclusive as a result.

Our hypothesis is that that the profiling tool should not have had any noticeable effect given that the LLVM compiler removes unused code. Thus, parts such as the profiler will be excluded from compiled releases of games. Additionally, most of our implemented code is part of the compiler and not the runtime libraries. However, some of the runtime optimizations likely had an effect on code size. For instance, the additional checks for the accessor property cache may have a negative impact on code size, especially if LLVM chooses inline the code at every call site.

6.2 Future work

Apart from the improvements already implemented, there are many other areas worth exploring, both in GMRT and in regards to ahead-of-time compilation of dynamic languages in general. The following subsections mention some of the most noteworthy ideas.

6.2.1 Making better use of property caches

The far most impactful performance improvement is arguably to complete the cache invalidation mechanism. This mechanism would allow both accessor properties and methods to be cached to a much greater extent. As previously stated we have implemented a naive invalidation policy but there are still issues to resolve.

Most importantly, code generation in both compiler front-ends must be adapted to avoid breaking invariants which force caches to be discarded needlessly often. Currently, objects are first assigned a prototype and the prototype is then mutated, which means that the invalidation procedure is triggered for every property on the prototype. If the prototype is simply constructed first, then the cache would not need to be invalidated at all.

It is also well worth exploring and adapting more ideas from other runtime environments. For instance, Serrano and Feeley [10] include two extra fields[11] in each property cache, thus allowing inherited properties (including methods) to also make use of the cache. Another possible future direction is to store the prototype relation in the hidden class graph instead of in each object, this approach is used by Google´s V8 engine among others [33].

[11] To be specific, `owner` which points to the object which a property is inherited from and `pclass` which points to the hidden class that populated the cache.

6.2.2 Investigating polymorphic property caches

The polymorphic cache design presented in this thesis is not without its flaws. For instance, it may suffer from poor performance for megamorphic coding patterns where the size of the ring buffer is large and it needs to probe for the correct cache entry often. It would be interesting to further explore good heuristics for finding a suitable capacity based on available profiling information.

Another unfavorable coding pattern may arise when the shape of the underlying object changes for every access. In this case, swapping the matching entry to the first index is strictly worse for performance, making the first entry guaranteed to never hit. It may be possible to address this drawback by extending the profiler to track the number of entry swaps and occurrences of each hidden class. In practical terms, one could populate the cache entries in an optimal order given by the profiled information, and, if the number of entry swaps is above a certain threshold, disable entry swapping completely to maintain said order throughout the runtime of the game.

Conducting a more systematic comparison between polymorphic caches in an ahead-of-time compilation context would also be relevant. Re-implementing the design from Hopc [10] and comparing performance for a larger sample of GameMaker games would be a valuable effort.

6.2.3 Making further use of PGO

Our PGO pipeline can feasibly be extended to support other kinds of optimizations. Among those that would require the least amount of work to implement is pre-populating property caches on start-up [27], and shrinking the hidden class graph [25] (both described in Section 2.4.1). Most of the information required for these optimizations is already tracked and outputted by our profiler, in particular for the latter. In order to pre-populate caches, however, some additional work is needed to make sure hidden class identifiers persist in some way between runs, so that they do not strictly depend on the order in which the graph is built.

Yet another interesting idea that could improve the caches presented in Section 5.3 and Section 5.6 is to track whether accessor properties or value properties are more prevalent for a specific cache. This is similar to how Hopc [10] removes the property accessor runtime check if it is never used. However, given that accessor properties are unusually common in GameMaker and that the PGO pipeline is restricted to offline optimizations, it will likely not prove as fruitful. On the other hand, the information can be used to re-order the checks for value and accessor properties. For instance, if the default behavior is to check the value property first, it may be reordered to check the accessor property first and avoid the extra overhead.

6.2.4 Integrate PGO pipeline in development workflow

While the PGO pipeline that was developed by us has adequate functionality for existing and future optimizations, it is not very ergonomic for users. In its current state, it requires manual editing of undocumented configuration files for each compilation. Additionally, any changes to a GameMaker project invalidates all previous profiling information.

Ideally, the PGO pipeline should be a well integrated part of the development workflow in the GameMaker IDE. Such a feature would ease development for those working with the engine itself, as well as users creating games.

Bibliography

[1] L. P. Deutsch and A. M. Schiffman, “Efficient implementation of the Smalltalk-80 system,” in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1984, pp. 297–302.

[2] C. Chambers and D. Ungar, “Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language,” *ACM SIGPLAN Notices*, vol. 24, no. 7, pp. 146–160, 1989.

[3] M. Bynens and B. Meurer, “JavaScript Engines: The Good Parts,” 2018. [Online]. Available: https://www.youtube.com/watch?v=5nmpokoRaZI

[4] M. Serrano, “Of javascript AOT compilation performance,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. ICFP, pp. 1–30, 2021.

[5] C. Fallin, “wevaling the wasms: AOT JS Compilation to WebAssembly.” [Online]. Available: https://www.youtube.com/watch?v=_T3s6-C38JI

[6] A. Wingo, “Just-in-time code generation within webassembly.” [Online]. Available: https://wingolog.org/archives/2022/08/18/just-in-time-code-generation-within-webassembly

[7] “GML Code Overview.” [Online]. Available: https://manual.gamemaker.io/monthly/en/index.htm#t=GameMaker_Language%2FGML_Overview%2FGML_Overview.htm

[8] M. Alexander, “GameMaker Studio 2.3: New GML Features.” [Online]. Available: https://gamemaker.io/en/blog/gamemaker-studio-2-dot-3-new-gml-features

[9] S. Thompson, “Design elements,” 2015. [Online]. Available: https://web.archive.org/web/20170305155607/https://github.com/v8/v8/wiki/Design%20Elements#fast-property-access

[10] M. Serrano and M. Feeley, “Property caches revisited,” in *Proceedings of the 28th International Conference on Compiler Construction*, 2019, pp. 99–110.

[11] M. Gaudet, “An Inline Cache isn't Just a Cache.” [Online]. Available: https://www.mgaudet.ca/technical/2018/6/5/an-inline-cache-isnt-just-a-cache

[12] U. Hölzle, C. Chambers, and D. Ungar, “Optimizing dynamically-typed object-oriented languages with polymorphic inline caches,” in *ECOOP'91 European Conference on Object-Oriented Programming*, P. America, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 21–38.

[13] S. Guo, M. Ficarra, and K. Gibbons, Eds., “ECMA 262: ECMAScript® 2025 language specification.” 2025. [Online]. Available: https://262.ecma-international.org/15.0/index.html

[14] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD conference*, 2008, pp. 1–20.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[16] Google, "V8 JavaScript engine." [Online]. Available: https://v8.dev/

[17] Apple, "JavaScriptCore." [Online]. Available: https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html

[18] Mozilla, "SpiderMonkey JavaScript." [Online]. Available: https://spidermonkey.dev/

[19] Microsoft, "ChakraCore." [Online]. Available: https://github.com/chakra-core/ChakraCore

[20] Oracle, "GraalJS." [Online]. Available: https://www.graalvm.org/latest/reference-manual/js/

[21] M. Serrano, "Hop." [Online]. Available: https://github.com/manuel-serrano/hop

[22] M. Serrano, "JavaScript AOT compilation," *ACM SIGPLAN Notices*, vol. 53, no. 8, pp. 50–63, 2018.

[23] S. Chandra *et al.*, "Type inference for static compilation of JavaScript," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, in OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 410–429. doi: 10.1145/2983990.2984017.

[24] Y. Futamura, "Partial computation of programs," in *RIMS Symposia on Software Science and Engineering: Kyoto, 1982 Proceedings*, 1983, pp. 1–35.

[25] T. Ugawa, S. Marr, and R. Jones, "Profile Guided Offline Optimization of Hidden Class Graphs for JavaScript VMs in Embedded Systems," in *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, in VMIL 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pp. 25–35. doi: 10.1145/3563838.3567678.

[26] T. Ugawa, H. Iwasaki, and T. Kataoka, "eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems," *Journal of Computer Languages*, vol. 51, pp. 261–279, 2019, doi: https://doi.org/10.1016/j.cola.2019.01.003.

[27] J. Choi, T. Shull, and J. Torrellas, "Reusable inline caching for JavaScript performance," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, in PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 889–901. doi: 10.1145/3314221.3314587.

[28] "The Computer Language Benchmarking Game." [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[29] C. W. Reynolds, “Flocks, herds and schools: A distributed behavioral model,” in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 25–34.

[30] Ecma Technical Committee 39 (TC39), “Test262: ECMAScript Conformance Test Suite.” [Online]. Available: https://github.com/tc39/test262

[31] MDM Web Docs, “Object.setPrototypeOf().” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf

[32] Google, “Maps (Hidden Classes) in V8.” [Online]. Available: https://v8.dev/docs/hidden-classes

[33] M. Bynens and B. Meurer, “JavaScript engine fundamentals: optimizing prototypes.” [Online]. Available: https://mathiasbynens.be/notes/prototypes

[34] D. M. Ungar, “The design and evaluation of a high performance SMALLTALK system,” University of California at Berkeley, USA, 1986.

[35] A. Sampson, “Flattening ASTs (and Other Compiler Data Structures).” [Online]. Available: https://www.cs.cornell.edu/~asampson/blog/flattening.html

[36] M. Vollmer *et al.*, “Compiling Tree Transforms to Operate on Packed Representations,” in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, P. Müller, Ed., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 74. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 1–29. doi: 10.4230/LIPIcs.ECOOP.2017.26.

Appendix A
GMIR Overview

GameMaker intermediate representation (GMIR) is a high-level instruction set that can either be directly interpreted in a virtual machine (GMVM) or compiled into a lower-level language, such as LLVM IR. GMIR makes use of registers but is not written in static single-assignment form. Note that this reference only provides a brief overview of GMIR and should not be considered a full specification of the language nor the binary format.

Textual representation

Multiple examples of GMIR expressed in a textual format can be found in this thesis. The concrete syntax is explained below in Backus–Naur form.

A statement inside a function is either written as just an instruction or assigned to a register if it returns a value.

Stmt	::=	AssignStmt	
	|	Instr	
AssignStmt	::=	Register = Instr	
	|	Register : Type = Instr	*with type annotation*

Notice that registers are optionally annotated with a *type*. Statically knowing the type helps avoid costly type assertions at runtime and may in some cases allow for direct translation into more efficient machine instructions.

Type	::=	Gmval	*arbitrary value*
	|	Double	
	|	Object	
	|	Bool	
	|	Void	

An instruction is expressed as an *opcode* followed by a list of *arguments*.

Instr	::=	Opcode (ArgList)	
Opcode	::=	ADD	
	|	SUB	
	|	…	*see the rest under the "Instructions" heading*
ArgList	::=	Arg , ArgList	
	|	Arg	
	|	Nil	*empty production, end of list*
Arg	::=	Register	*a name, for example "r3"*
	|	Block	*index to a block*
	|	Constant (Annotation)	*for example, c0(\`Hello world\`)*

Instructions

As previously mentioned, an instruction consists of an *opcode* as well as zero or more *arguments*. An argument is either a register, block index, or constant. What follows is a complete list of instructions and their arguments.

Arithmetic

`ADD`
Add

Sum register, Addend register, Addend register

`SUB`
Subtract

Difference register, Minuend register, Subtrahend register

`MUL`
Multiply

Product register, Factor register, Factor register

`DIV`
Divide

Quotient register, Dividend register, Divisor register

`IDIV`
Integer divide instruction for GML div operator

Quotient register, Dividend register, Divisor register

`MOD`
Modulo

Remainder register, Dividend register, Divisor register

Logic

`AND`
Conjunction

Return register, Operand register, Operand register

`OR`
Disjunction

Return register, Operand register, Operand register

`XOR`
Exclusive disjunction

Return register, Operand register, Operand register

`NEG`
Unary negation

Return register, Operand register

`PLUS`
Unary plus (will coerce to double)

Return register, Operand register

`LNOT`
Logical negation

Return register, Operand register

`BNOT`
Bitwise negation

Return register, Operand register

`SHL`
Logical left shift

Return register, Operand register

`SHR`
Logical right shift

Return register, Operand register

`SAR`
Arithmetic right shift

Return register, Operand register

`RET`
Return with register value

Input register

`RETV`
Return with void

-

Control flow

`B`
Branch unconditionally

Block block index

`BT`
Branch if `CC` register is true

Block block index

`BF`
Branch if `CC` register is false

Block block index

`BJ`
Branch through jump table

Input register,

Block_1 block index, ..., Block_n block index

`UNREACHABLE`
An instruction that should never be reached

-

Comparison

`LT`
Less than (result stored in cc register)

Left register, Right register

`LE`
Less than or equal (result stored in cc register)

Left register, Right register

`GT`
Greater than (result stored in cc register)

Left register, Right register

`GE`
Greater than or equal (result stored in cc register)

Left register, Right register

`EQ`
Equal (result stored in `cc` register)

Left register, Right register

`NE`
Not equal (result stored in `cc` register)

Left register, Right register

`IN`
JavaScript `in` operator (result stored in `cc` register)

Property register, Object register

`TYPE_EQ`
JavaScript `===` operator (result stored in `cc` register)

Left register, Right register

`TYPE_NE`
JavaScript `!==` operator (result stored in `cc` register)

Left register, Right register

Registers

`LDC`
Load constant address into register

Return register, Input constant

`MOV`
Copy value from one register to another

Return register, Input register

`MOVCC`
Copy the current condition code (`cc` register) to another register

Return register

`TOCC`
Copy the register value to the `cc` register (converting to boolean)

Input register

Properties

`PGET_A`
Assert that a property exists, and get its value (or throw ReferenceError)

Return register, Object register, Property name constant

`PGET`
Get the value of a property on an object

Return register, Object register, Property name constant

`PSET_A`
Assert that a property exists, and set its value (or throw ReferenceError)

Input register, Object register, Property name constant

`PSET`
Set the value of a property on an object

Input register, Object register, Property name constant

`IGETG`
Get the value at the given index in a GML object or array

Return register, Object register, Index register

`ISETG`
Set the value at the given index in a GML object or array

Input register, Object register, Index register

`IGETJ`
Get the value at the given index in a JavaScript object or array

Return register, Object register, Index register

`ISETJ`
Set the value at the given index in a JavaScript object or array

Input register, Object register, Index register

`PARRAY`
Force named property to be an array (create new property with empty array if needed)

Return register, Object register, Property name constant

`IARRAY`
Force indexed property to be an array, (create new property with empty array if needed)

Return register, Object register, Index register

`RARRAY`
Coerce value into an array (create an empty array in the register if needed)

Return register, Input register

Functions

`CALL`
Invoke a function value

Self register, Function register, Arg_1 register, ..., Arg_n register

`STATIC_CALL`
Invoke a statically-linked function

Self register, Function constant, Arg_1 register, ..., Arg_n register

`CALLEE`
Get a reference to the callee object (the current function)

Return register

`OTHER`
Get the `other` value from the current execution context

Return register

`CLOSURE_OBJ`
Get a reference to the closure object

Return register

`CLOSURE_NEW`
Turns a function into a closure

Return register, Function object register, closure object register

`ARGINIT`
Get the argument at the given index

Return register, index constant

`ARGC`
Get the argument count

Return register

`ARGGET`
Get the argument at the given index

Return register, index register

`ARGSET`
Set the argument at the given index

Return register, index register

`MAPPED_ARGS`
Create a JavaScript mapped arguments object

Return register, Locals Object register, Argname_1 constant, ..., Argname_n constant

`UNMAPPED_ARGS`
Create a JavaScript unmapped arguments object

Return register

Exceptions

`THROW`
Raise an exception

Input register

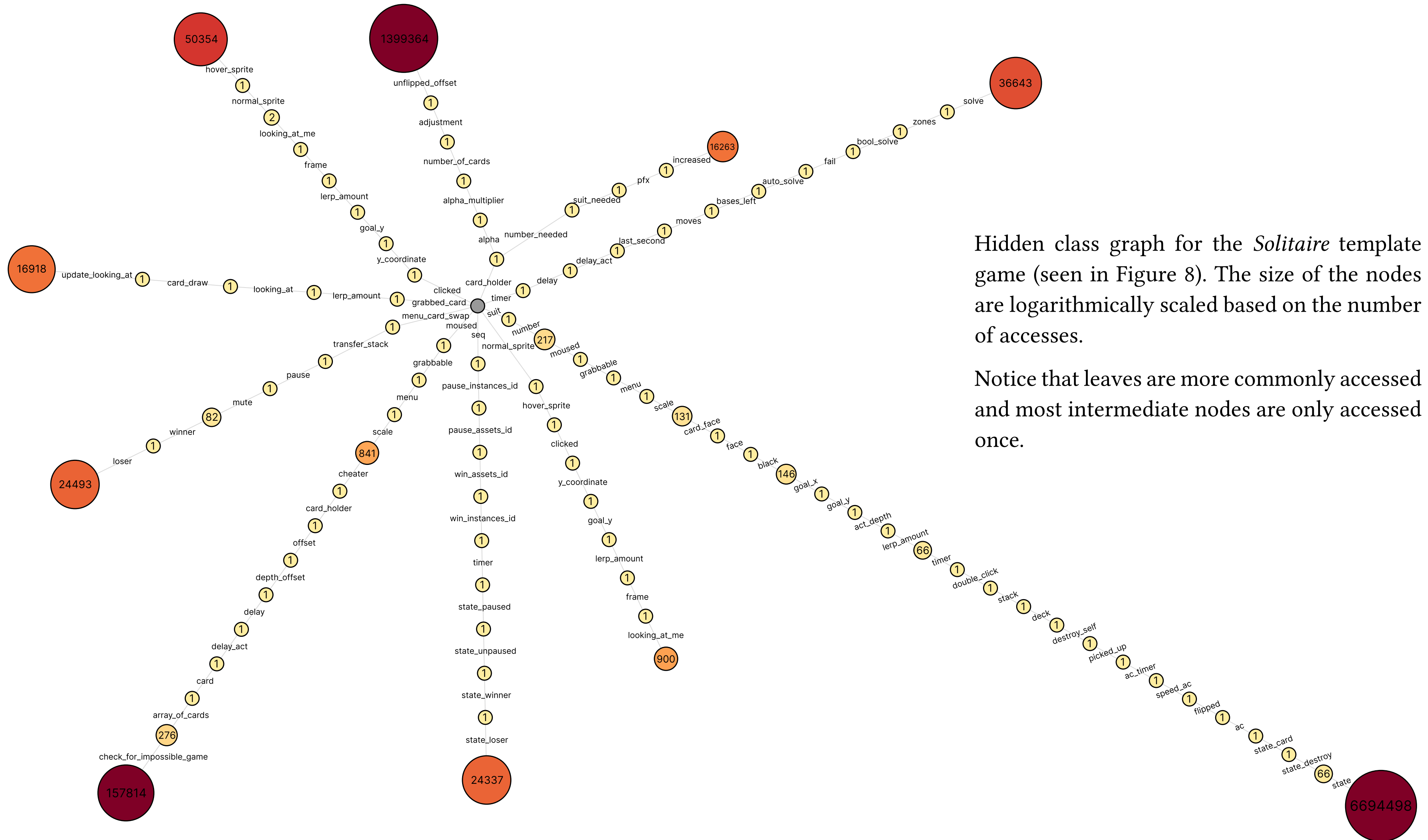
`TRY_ENTER`
Cache `other` and increment try counter before try block (with an ID)

Id register

`TRY_RESTORE`
Restore `other` and try counter to their previous state after try block

Id register

Appendix B
Hidden Class Graph



Hidden class graph for the *Solitaire* template game (seen in Figure 8). The size of the nodes are logarithmically scaled based on the number of accesses.

Notice that leaves are more commonly accessed and most intermediate nodes are only accessed once.