Abstract

Dynamic languages such as JavaScript have been widely used in web applications, including cryptographic-based application. Due to the dynamics of JavaScript, writing efficient programs and generating efficient execution code is not easy. The aim of this project is to investigate the dynamic behaviours of JavaScript and analyse these behaviours, research the JavaScript engines and their foremost optimisations. Based on the results of investigation, we offers proposals for efficient programming and designing high-peformance JavaScript engine. The work presents here has profound implications for future studies. The main contributions of this project are described as follows:

- Took experiments on JavaScript benchmark, concludes the behaviours of practical JavaScript program, which can be a guidance to design modern JavaScript engines and adopt compiler optimisation techniques.
- Performed a full-scale type analyses on JavaScript both at compiler-time and run-time.
- Investigated how inline caching and traditional compiler optimisations applied to JavaScript JIT compiler.
- Proposed a programming guideline for programmers writing more efficient JavaScript programs based on the investigations on JavaScript engines.
- Designed an architecture for modern high-performance JavaScript engine, especially its optimising JIT compiler.

Acknowledgements

I would like to express my thanks to my supervisor Dr. Dan page, for his invaluable advice, continuous help and assistance throughout this project.

I would like to greatly thank my family for their constant support and encouragement.

Contents

Chapter 1

Introduction

1.1 Motivation

With the development of the Internet, scripting languages (e.g. JavaScript, Perl, Python) and some other dynamic languages are favored by many developers. As the outstanding representative of these dynamic languages, JavaScript has been widely used. The TIOBE programming community index (which is an indicator of the popularity of programming languages) for April 2011 shows JavaScript ranks top 10 among all the programming languages, JavaScript were adopted in more than 1.5% software systems. By JavaScript program, client side (e.g. web browser) can response to the requests of user (e.g. form input) without uploading the data to the server side. This makes some applications are available without the need of installing software or plug-ins, which is simpler than desktop application for data exchange and management. In order to protect these data, client-side JavaScript encryption programs are adopted in some situations. When client-side needs encryption, it calls the cryptographic JavaScript programs in the client. There are some developed libraries to support JavaScript encryption, which suggests that there is an considerable interest in executing various cryptographic primitives in JavaScript. Client-side encryption definitely needs a high performance, but JavaScript and its interpreters have many features which make programming easier but writing high-performance programs harder. Therefore, improving the performance of cryptographic JavaScript programs plays a significant role in client-side encryption and decryption. On the other hand, as a dynamically, weakly typed and prototype-based languages, JavaScript has cought the eyes of many researchers in recent years.

1.2 Aims and Objectives

The initial goal of this project is try to develop a JavaScript virtual machine extension based on the existing modern JavaScript engines, which improves existing cryptographic libraries without any alteration to the library source code. However, during the first attempt to develop it, we found that developing an extension based on the existing engine needs large workload, even larger than developing a JavaScript engine from scratch. Because the original code is tightly coupled and has low cohesion. In addition, recent research (e.g.[3]) presents a spraying attack for the client-side JIT compiler and modern engines have few defense against this attack. A engine has no powerful security definitely cannot be regarded as a platform for cryptographic programs. Based on these, we change the focus of this project. The new aim of this project is to conduct a full-scale study of JavaScript and its platform from programming level to compilation level. Based on the investigation, we raise a guideline for programmers to more efficient programming in JavaScript and design an architecture for modern JavaScript engine which properly combines the state-of-the-art optimisation techniques together.

1.3 Thesis Organisation

This thesis is organised as follows: Firstly, we describe the JavaScript as a language and its features, then explain why we need to optimise JavaScript. Since we aim at the JavaScript for cryptography, next we illustrate the primitives – AES and RSA, explain the process of their encryption and decryption. Secondly, a number of tricks and reduction algorithms that used to effective implement AES and RSA are outlined. Afterwards we use examples to show existed JavaScript cryptographic libraries how to optimise the AES and RSA algorithms. Above is the first part of this project – language level optimisation. The other part is to investigate how to speed up the JavaScript cryptographic programs by improve its engines. We first describe the basic concept of virtual machine and just-in-time compiler, in addition of how interpreter and just-in-time compiler associated with each other. Several state-of-the-art JavaScript engines are analysed. Chapter 3 contains an analysis of JavaScript execution behaviours by experiments. To deal with the side effect of JavaScript dynamic type system, we perform type analysis for JavaScript, including run time type specialisation and static type inference algorithm. Inline caching is described in detail to accelerates the property search of JavaScript. Chapter 4 is a discussion of the results found which includes a guideline for JavaScript programming and a modern JavaScript engine design.

Chapter 2

Background

2.1 JavaScript

2.1.1 Features of JavaScript

JavaScript enables a real time, dynamic and interactive techniques, making information and user is no longer a simple relationship between display and browsing. It can response the user directly, does not need the web server. The method it used to response the user is event-driven. Clicking the mouse, moving the window or selecting a menu can be regarded as events. Event-driven means once these events happened, it will incur a response. This method reduces the pressure on the server side and improve the user's web browsing experience.

JavaScript is a cross-platform language. It depends on web browser rather than platform. Executing JavaScript programs needs only a computer that can run JavaScript-supported web browser. In fact, the most attractive point is programs written in JavaScript can be significant smaller than equivalent programs in other languages, but these small programs can do a lot. Backbone of JavaScript includes a core set of objects like Array, Data and so forth. The core part of JavaScript can be extended to client-side JavaScript and server-side JavaScript by adding different new objects. Client-side JavaScript provides a set of objects to control the web browser and its supported Document Object Model (DOM). In server-side JavaScript, added objects make JavaScript program can be run in the server side. The following piece of code shows a small example of JavaScript program.

```
var names = new Array();
names[0] = "Tom";
names[1] = "Jerry";
names[2] = "Jack";
for(var i = 0; I < names.length; i++) {
    alert(names[i]);
}
```

This program creates an array to store the names. We can see that “`var`” represents the type of object, types of JavaScript variable are associated with values. In this program, variable `‘‘names’’` bounds to an array of string. Array in JavaScript is dynamic, which likes a combination of `array`, `ArrayList` and `Hashtable` in Java. It does not need to specify the size of an array.

The main features of JavaScript include:

- A structured language. JavaScript supports C language style (e.g. `if`, `while`, `switch` syntax) except the scope. Usually, names of variables in a program are not always valid, the scope limits the valid range of names. In the C-style programming languages, block-based scope is not supported. But JavaScript owns this feature. The same as C, JavaScript treats expression and syntax differently. The slightly different from C is JavaScript can omit the ending symbol (semicolon) in many cases.

- Dynamic type system. In JavaScript, types are bound with values instead of variables like in traditional static language, which leads that a variable can store a value of any type. For example, a variable initially stores an integer can be assigned a string later. Another impact of dynamic type system is a high degree of operator overloading. In static programming languages, the types of operands that an operator can accept are fixed and limited, like multiplication can only be performed between numbers. But because of the dynamic type system of JavaScript, any operator can be regarded as operands of any type. During the process to do arithmetic, it needs to determine the type of operand dynamically and trigger proper type conversion. The object also perform its dynamics. JavaScript is an object-based language, its object actually is an associative array. Accessing the properties of an object is essentially a mapping from the names of properties (as indices) to the values of associative array. Based on this design, the structure of objects can be changed on the fly. In other words, a property can be removed or added at any time. Besides, the function of `eval()` contributes to the dynamical of JavaScript programs.

- A functional language. The functions in JavaScript can be passed as arguments to other functions because they are treated as first-class objects. Functions can be defined in the internal of another functions, which result in closure. With closure, the variables in the scope of outer function can still be accessed if inner function is invoked, even the outer function has finished execution.

- Prototype-based. The prototype system of JavaScript simulates the inheritance of object-oriented programming languages in a sense. When it attempts to access a property of an object but finds the property does not exist, the prototype object of this object will be accessed in order to find the property with the same name. If it still does not exist, it needs to search along with the prototype chain until reaching that property or the end of the prototype chain. Any functions in JavaScript can be invoked as a constructor. On the one hand, constructor has the responsibility to initialise the properties of objects. On the other hand, it sets the prototype object of the constructed object as a property of this constructor itself (with the name of “prototype”). Except being invoked as a constructor or an ordinary function, a function can also be assigned as a property of an object thereby invoked as its member variable.

Take a small example to look at how JavaScript performs its features. The following piece of code defines a “class”(actually JavaScript has no concept of class) of `Person`, then it can define new objects. Since `Info` is the added function of objects, each new instance can invoke this function.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.info = function() {
        var str = 'name:  '  + this.name + ', age:  '  + this.age;
        alert(str);
    }
}
var person1 = new Person('Jack', '20');
var person2 = new Person('Tom', '25');
person1.info();
person2.info();
```

2.1.2 Why Optimisation Is Necessary

The syntax of JavaScript is similar to C, but its source code does not need to be compiled before sending it to the client browser from server. Instead, a JavaScript interpreter embedded in the Web browsers translates the source code into bytecode and executes it. Early JavaScript engine adopted such interpretive method which is ineffective. In fact, JavaScript program has gradually become bottleneck among interactive web applications. It is generally acknowledged that most of this bottleneck because of its dynamic type.

Compiler needs to know the types or a collection of types of variables during compilation in order to execute or generate corresponding code. The variables in JavaScript program have dynamic type, the type of operand in the computing is not restricted to static. The type checking during the process of interpreting or dynamic executing will make it too complex and inefficient.

Interpretation is slower than compilation because it has an extra translation process, thus we need to optimise the code and improve the interpreter to offset this penalty. In addition, compilers involve many optimisation techniques to generate fast machine code by lexical analysis while interpreters have fewer such optimisations [39]. But in fact, for the majority of JavaScript programs, the types are stable though they are interchangeable silently.

2.1.3 Run-time and Virtual Machines

JavaScript needs objects and functions that provided by run-time environment to communicate with world. For example, JavaScript itself has no syntax for input and output, handles the objects embedded in Web browser can expose the result of program. Therefore, the run-time environment is the platform for running JavaScript programs. Generically, run-time environment is a software collection that supports to run programs. It often supports various services, like type checking, debugging, code generation and code optimisation. For many advanced programming languages, run-time system is the bridge between programs and hardware. In the limit, run-time system actually is a virtual machine that aid program running.

The virtual machine we will consider here is process virtual machine which differs from system virtual machines [34]. Virtual machine techniques are applied in the field of high level programming language, such as Java's "Write once, run anywhere". The success of Java profits from its run-time system platform, which is a layer built by virtual machine between operating system and program. In the traditional platform, the front end of compiler is used to do lexical analysis, syntax analysis, semantic analysis and generate intermediate code. A code generator in the back end will translate intermediate code into binary code, which includes the machine code targets on the specific instruction set and operating system. In order to execute the programs in different platforms, the program must be recompiled. But the appearance of high-level language virtual machine (HLL VM) changes this model. The steps of HLL VM is similar to traditional model expect that the program will be dispatched in a higher level. The machine code that generated by compiler in traditional model similar to a stack-based (which most mainstream HLL chose) or register-based instruction set in HLL VM [34]. These portable virtual instruction sets will be dispatched and executed in the platform which can run these instruction sets. In the stack-based VM, the stacks are used to provide operands, return the results, pass parameters for the execution and so on. Invocation and return in the execution process are showed as pop and push in the stack. The operands of VM instructions come from the top of stack, or on the neck of operator. The result will be returned to the top of stack after computing. For example, `ADD` presents addition of two numbers. The two numbers should be at the top of the stacks, which were pushed by previous instruction. Then numbers are popped from stack and added. The result are pushed back to the stack.

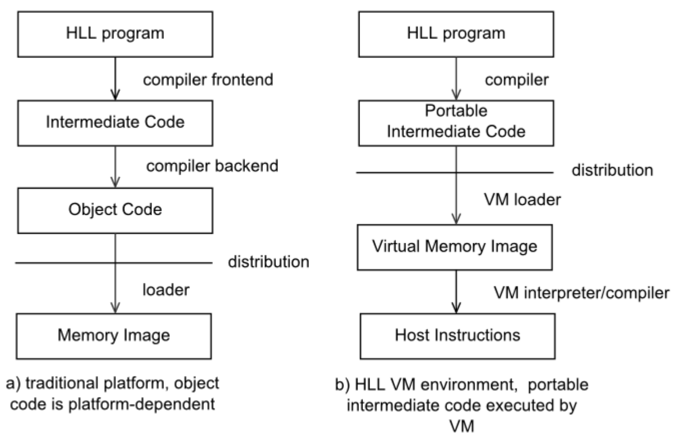

Figure 2.1.1: High-Level language virtual machine environments [34]

With the development of HLL VM, it is found that simply interpret the intermediate code has a much lower performance. Because of this, just-in-time (JIT) compiler was emerged. The JIT compiler chooses the code that called most often, compiles them into native code on the fly (This is why called just-in-time) and stores them in the memory for further use. This process runs concurrently with the execution of the VM. Next subsection will discuss the JIT compilation in detail.

2.1.4 Just-in-time Compilation

Traditionally, there are two approaches to execute programs: static compilation and interpretation. As we know, for complied language like C and Pascal, the compiler translates the source code into assembly language or machine code. Interpreted language like Basic directly interpret and execute the source code while Java Virtual Machine (JVM) of Java interpret bytecode. To combine advantages of static interpretation and static compilation, JIT compilation systems have emerged. This method can speedup the execution of intermediate code (e.g. bytecode in JavaScript). They compile bytecode into native machine code before executing it. Many modern environments like JVM and .Net Framework of Microsoft adopt JIT techniques to reduce running time and save space.

By looking at JavaScript engines with high performance (which we will discuss in detail following), it can be found that the optimisation techniques they used definitely include JIT system. Chrome V8 engine adopts JIT compiler to compile JavaScript source code into machine code and store them in memory for executing. TraceMonkey uses trace-based technique which performs 22 times improvement. Thus it is clear that although they adopt different approaches to realise JIT system, they show a significant improvement on performance if using JIT technique. For example, a web browser needs to use different subroutines to address different requests, but only some of them may used at a time. JIT compiler avoids web browser compiling these unused code to enhance the performance.

JIT system became commonly used from Java [2], its related development and application also based on Java platform. JIT system is simple in terms of its primary principle, which dynamically compiles the bytecode generated into machine code during run time in order to efficiently execute on the hardware. Compiling all the source code to machine code definitely affects the efficiency of the compiler. In addition, there is a delay loading the JavaScript engine. The more bytecode compiles, the more memory space would be occupied. Therefore we need a trade-off between JIT compilation and interpretation. In practice, 20% of source code implies 80% running time in most programs. So we need techniques aim to find those 20% source code, then compile it to native code and adopt efficient optimisation techniques to enhance the execution efficiency of program. In this thesis, we call these frequent executed code as “hot”. Usually, only “hot” code is complied by JIT while the other part of code is only interpreted. Next we will describe the techniques used to determine what is the “hot” code and when should JIT apply on “hot” code.

2.1.4.1 Method-based JIT

This technique firstly monitors the every block (or method) of code being invoked during execution, comparing the time of invoking with threshold. If the counter exceeds the threshold, then the function becomes a “hot region”. We use “hot region” in the method-based JIT to denote the code segment that JIT system selects for optimisation [20]. When JIT technique introduced, the whole method was treated as an alternative unit of region. JVM used to employed this approach to select the region. During the period of generating native machine code, a traditional control-flow graph will be employed to record the methods. Since some optimising compilers use Static Single Assignment (SSA) form [7] as an intermediate representation, the execution time is increasing. With the development of trace technique, the trace-based JIT was emerged.

2.1.4.2 Trace-based JIT

The main idea of trace-based JIT is only to optimise code segments which most worth optimising. In other words, these code segments will be executed most frequently. It needs time and space to translate bytecode to native machine code. Therefore if the code execution frequency is not high enough, the extra time it takes to compile will not be offset by saved execution time. Statistical data shows the time spent on executing code with linear structure occupies small proportion of the total running time, while loop structures accounts for the majority of time. Therefore, the trace-based JIT focus on the loop. Even in a loop, only frequent executed path will be compiled.

2.1.4.3 Trace Tree

According to Andreal [16], a trace tree can be represented as a direct graph, each trace tree consists of a set of traces. Following figure is a sample trace tree. The node labeled as anchor in the figure is a node that a cycle of instructions starts with and ends with, and these cycles of instructions are called traces. Since a loop may includes more than one trace, we call the first trace in the loop trunk trace. A trace is a straightway stream of instructions without branches, so it needs to ensure the encountered condition is the same with that trace recorder is recording. Guard instructions are used to complete this work. If guard instructions find it is different, JIT would exit and back to the interpretation. To make sure the consistency of path, guard instructions need to verify the type and value of variables. Due to the JavaScript's dynamic type, the type of variables may be changed during the running process. Since different types of variables have different processing flow, so the type of variables needs to be cared to ensure the current processing flow is the same with the recording trace. Besides, processing flow also relates to the value of variables. For example, conditional branch instructions will jump to different paths according to different values of variables.

A single trace can deal with the simple program which does not include nested loops, branch conditions and method calls. But if this trace tree model can not address more complex but common used program structure, it would obviously have limitations. Reconstruct the trace tree would be more expansive. Therefore, effective use of constructed trace tree and extend the current trace tree at a appropriate time would improve the model. Extending the trace tree actually means making the side exit (which is the exception exit) point to a fragment. This fragment stores a new trace. Before extending, the side exit points to the loop header of this trace because this means a finish of a trace. After extending, side exit points to a fragment thereby the trace will not stop immediately when encounters exception but transfer to a new trace to continue. Therefore, it will not exit the trace tree no matter whether satisfy the condition.

Extending the trace tree needs to set the root of fragment which corresponding to the new trace as current trace tree, and side exit points the new created fragment. And then start to record the new trace, the loop exit instruction which points to the header of loop will be generated once finished. This process is similar to create the trace initially. During the extending, it is crucial to manage the global variables. Because new added branches share the global variables with original trace. The global variables of original trace modified in the new path are treated as global variables of new added path. Combining the global variables of new branch and original trace is necessary when finish recording the new branch.

As mentioned before, JIT would exit and side exit would back to the interpretation if guard instruction is triggered. If the number of a side exit exceeds a specified threshold, it means the loop contains another hot path, trace tree would be extended from this side exit which points to the trunk tree [17]. This another hot path is called branch trace. In order to keep real time interaction with native code, the trace tree needs to be recompiled for each successful extension,. Since every expansion of trace tree will increase its size, so a counter needs to be defined to control the size of trace tree. Once the counter beyond the threshold, the expansion would be stopped.

Overall, the basic process to extend the trace tree can be concluded as:

1. If the size of trace tree is beyond the threshold, then end;
2. If current side exit is first encountered, then create a fragment whose root is current side exit, otherwise use the fragment which created last time;
3. If execution times of current side exit do not reach the threshold, then end;
4. Record current trace from current side exit;
5. Combine the global variables of branch trace and those of trunk trace.

Once the tree tree is formed, the compiler translates it into the machine code.

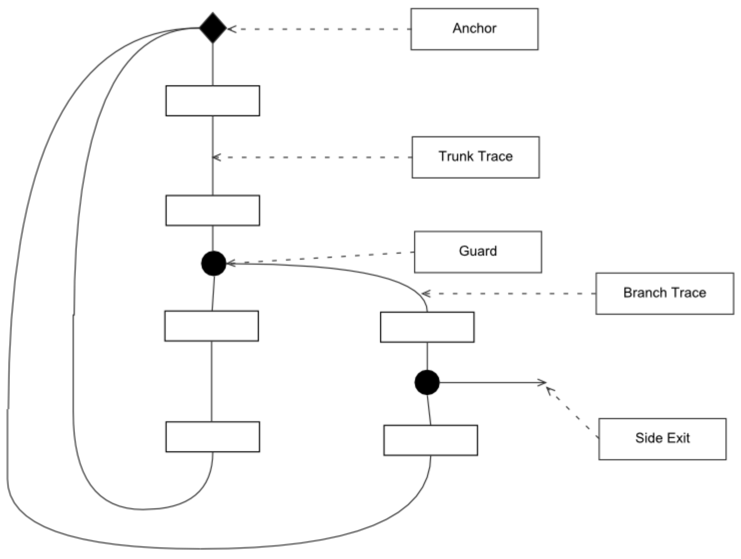

Figure 2.1.2: An example of trace tree

2.1.4.4 Improved Trace Tree

Jungwoo [17] presented a concurrent trace-based JIT compiler for JavaScript which reduces 97% pause time on average compared to the single threaded JIT compiler. As explained before, in trace-based JIT, tracing often carried on before compilation. This concurrent JIT makes tracing and compilation happened at the same time. Usually, concurrent system uses locks to deal with synchronization problem. Instead, this system uses Compiled State Variable (CSV) as a synchronization variable on each trace. Different states have different values, increasing or decreasing the values makes pause time almost equals to zero. As described before, trace recorder will record the hot path with type of variables and check the assumption (which uses guard instruction). But if a loop has more than one hot path, the interpret restart to trace the branch path which triggers guard. To improve the code quality of trace tree with more branch traces, trace stitching technique is proposed in this paper in order to only compiles new branch trace instead of recompiling the whole trace. Their testing results shows their approach eliminates most the pause time and improves the performance throughout. In addition, more aggressive optimisations can be performed without wasting pause time because of concurrency.

[?] forwards an trace-based approach that can cover all the hot paths of entire programs rather than only consider the hot paths in loops. Unlike the JIT compilers which decide when to trigger compiler depends on the execution frequency, this JIT compilation scheme based on trace intervals. Trace intervals are a set of interpreted instructions with specified length. This method can make a trade-off between compilation and execution since the compilation units are more uniform. This paper also uses an approach to compile more than one trace in parallel. Similar to [17], this JIT compiler uses a state variable to deal with the problem of synchronization. False loop filtering technique is proposed by [19] to filter the false loop which means not repeating cyclic execution paths in order to improve the quality of traces. Call-stack-comparison is the algorithm to detect false loops.

Compared to method-based JIT, trace-based JIT has a smaller compilation unit, so the generated native code also occupies smaller memory space. In addition, the startup time of trace-based JIT is less than method-based JIT since the former one does not to construct the static control flow graph. On the contrary, the trace tree needs to recompiled once traces added to the trace tree, which costs extra compiling time. If trace is too large, the JIT needs to abandon this trace out of consideration of performance. Besides, since the code that compiled is less, trace-based JIT has less opportunities to do code optimisation which may cause the generated code is not good enough. Therefore, trace-based JIT still has some challenges to overcome. [22] shows that trace-base JIT could be generate code with high quality if long traces could be selected by extending the compilation scope. It also states that taking advantages of both method-base JIT and trace-based JIT may be a better way.

2.1.4.5 Other Optimisation Techniques for JIT

Besides method-based and trace-based approaches, there are still several influential techniques to optimise JIT compiler. Double-dispatch type inference is an idea that came up with by [6]. The aim of the technique is to generate codes which can virtually closer to the quality of code of statically-typed language. By double dispatch which invokes the operations from both directions (e.g. `a.add(b), b.add(a)`) to select precise semantics, it can translate the untyped code into intermediate representation with static type. Differs from the traditional type inference technique which tries to infer the the type of variables beforehand, double dispatch mechanism does this during the run-time. Many other techniques like polymorphic inline caching and escape analysis are considerable ways to enhance the performance of JIT compiler.

2.1.5 Examples of JavaScript VM and JIT Compiler

Some browsers have developed fast and efficient JavaScript Virtual Machines (or JavaScript engines). The features of well-known JavaScript engines are as follows:

2.1.5.1 Rhino

Rhino [13] is an open source JavaScript engine which implemented entirely in Java. Therefore, it is run on the Java Virtual Machine (JVM). Rhino is often embedded into other Java applications. Rhino actually includes both JavaScript interpreter and compiler (which compiles JavaScript source code to JVM bytecode). Rhino provides different modes of optimisation: interpretive mode, compiler mode, compiler with optimisation mode. Interpretive mode minimises compilation time at a cost of performance at run time. The source code will not translate into byte codes to save the memory. No optimisation will be performed on generated byte code in the compiler mode so it has low efficiency. Many optimisation techniques like data and type flow analysis are used in the third mode. Compared to other JavaScript engines, the advantage of Rhino is it has a map between script language object and Java object, which reduce the limitation of interpretation environment. But Rhino has no built-in support for browser so it is suitable for server-side usage.

2.1.5.2 TraceMonkey

TraceMonkey [11] is developed by Mozilla as an extension of SpiderMonkey [14]. Unlike the optimisation techniques adopted in traditional static language, as its name suggests, TraceMonkey is based on trace tree which based on the hypothesis that the most running time is consumed in the hot loops.

In TraceMonkey, the program will be interpreted firstly. At that time, the engine will find the hot paths and compile them into executable code. Each complied path corresponds to mapped type of a value and its execution path. In practice, the program may not run in the same path every time, the type of variable might be different next time. Therefore, TraceMonkey will record the path and mapped type during execution and try to check them during compiling. In this way, type checking will be inserted into the complied code. If a check failed, the path will abandoned by the engine. If the engine finds that it leaves a path frequently, it will instead recording a branch path. In other words, the set of paths that engine records covers all the hot paths of loops. Since compiling branch paths and jumping between them will cost time, code with many branches and frequently changing types runs slower [4]. In addition, TraceMonkey adopts property cache data structure to speed up communications between JIT compiler and interpreter [12].

Here is an example [15] to explain how TraceMonkey executes a program which computes primes. The first step is translate them into the bytecode. When $i = 2$, the TraceMonkey changes to recording mode from line 4 since it is the start of an inner loop (which will be executed many times), we label first trace as T_{45}. TraceMonkey will record the trace as LIR which is a low-level intermediate representation. When finishing executing the inner loop, the trace recorder will stop recoding. Then the compiler of TraceMonkey compile LIR into native code. The recoded information of variables can be employed to optimise the code. When $i = 3$, the code from outer loop has become hot since it starts to execute at second time. The trace recorder will find that inner loop has been already complied into trace, the TraceMonkey combines the compiled trace and current recording trace by treating the existed trace as a subroutine. We call this trace T_{16}. When $i = 4$, the value meets the condition of line 2. Since it is the first time to take this branch, the guard instruction will fail and incur a side exit. At this time, TraceMonkey will back to the interpretation mode. After $i = 6$, the trace $T_{23,1}$ which covers line 2,3 and 1, T_{16} and T_{45} are the enough traces to cover all the hot paths of program. From that moment, the entire problem will be run as machine code [15].

```
1  for (var i = 2; i < 100; ++i) {
2      if (!primes[i])
3          continue;
4      for (var k = i + i; i < 100; k += i)
5          primes[k] = false;
6  }
```

JägerMonkey is the faster JavaScript engine in the recent released Firefox 4. The new engine combines the advantages of both method-based and trace-based approach. JägerMonkey compiles all the intermediate representation which generated by SpiderMonkey without tracing and recording the execution path. Though trace-based TraceMonkey is fast enough, it still has programs that do not trace, like programs with many branches. And it takes time to switch between interpretation and compilation. To deal with this problem, JägerMonkey takes Nitro from WebKit JavaScript engine as an assembler to build a method-based JIT in order to eliminate the overhead of switching. The following figure shows the components of JavaScript engine of Firefox.

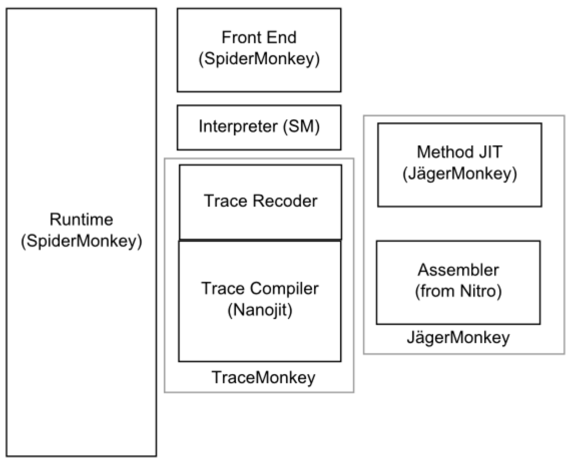

Figure 2.1.3: Major components of Firefox JavaScript engine [26]

2.1.5.3 V8

V8 is the JavaScript engine of Google Chrome. V8 uses "hidden class" to implement the objects of JavaScript in order to access the properties faster. At the head of each object, there is a pointer points to its class which defines properties of objects. Once the objects are changed (e.g. add or delete properties), its class would be updated dynamically. In V8, each property accessing point stores the class of objects which executed last time. If the class of current object is the same as the class of last accessed object, it means that their storage location in class is same, so the property can be used directly. Otherwise, property lookup algorithm will be employed to search this property and this class with storage location of property will be stored. This is inline caching approach that V8 uses. Besides these, V8 implements a highly optimised memory management and garbage collection system. In V8, JavaScript source code is translated into machine code directly, without intermediate representation. There is no intermediate bytecode because V8 has no interpreter. Overall, the design of V8 is based on the features of programs during execution, like consistency of property accessing address. However, with the expansion of scope of JavaScript application, the features that V8 based become narrower. For example, in the computing oriented programs, a large number of floating-point numbers and their arithmetic incur plenty of small objects store in the heap. And inline caching of V8 does not work in some programming models.

Crankshaft was introduced at the end of 2010, aims to further improve the performance based on V8. It uses a base compiler and an optimising compiler instead of only one compiler doing all the work in V8. Similar to TraceMonkey, Crankshaft compiles hot code into machine code which monitored by a run-time profiler. The base compiler compiles all the code at first, but does not involve much optimisation to save the time. The optimising compiler recompiles the hot code and dose heavy optimisations to improve the code quality. On the other hand, the profiler gathers the information of variables more than once which differs from one tracing in the TraceMonkey and granularity of hot code is the whole methods rather than traces.

2.2 Cryptographic Primitives

2.2.1 AES

2.2.1.1 Overall Design

Symmetric key encryption was the only type of encryption in use prior to the development of asymmetric key (public-key) encryption in the late 1970s. In symmetric key cryptography, a single cryptographic key is shared during communication for both encryption and decryption. According to the different means of encrypting plaintext, symmetric key algorithms can be divided into block ciphers and stream ciphers. Block ciphers split the message into blocks with fixed length, the output ciphertext are the same length as input plaintext. Advanced Encryption Standard (AES), which has been widely used, belongs to block ciphers.

The size of its input blocks, output blocks and intermediate blocks during encryption and decryption are all 128-bits; key size can be 128, 192 or 256 bits, the corresponding number of rounds are 10, 12 and 14. The plaintext blocks and intermediate blocks are called state, which is a two-dimensional array of bytes [8]. Cipher key presents the initial key of AES algorithm, which is expanded into other keys the algorithm needs. The round functions of AES algorithm includes four invertible transformations, i.e. `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. We take 10 rounds algorithm as an example to illustrate the construction of AES algorithm. For 128-bits algorithm, the first nine rounds consists `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`, but the final found has only `SubBytes`, `ShiftRows` and `AddRoundKey`. Before the first round, it is required to do `AddRoundKey` operation.

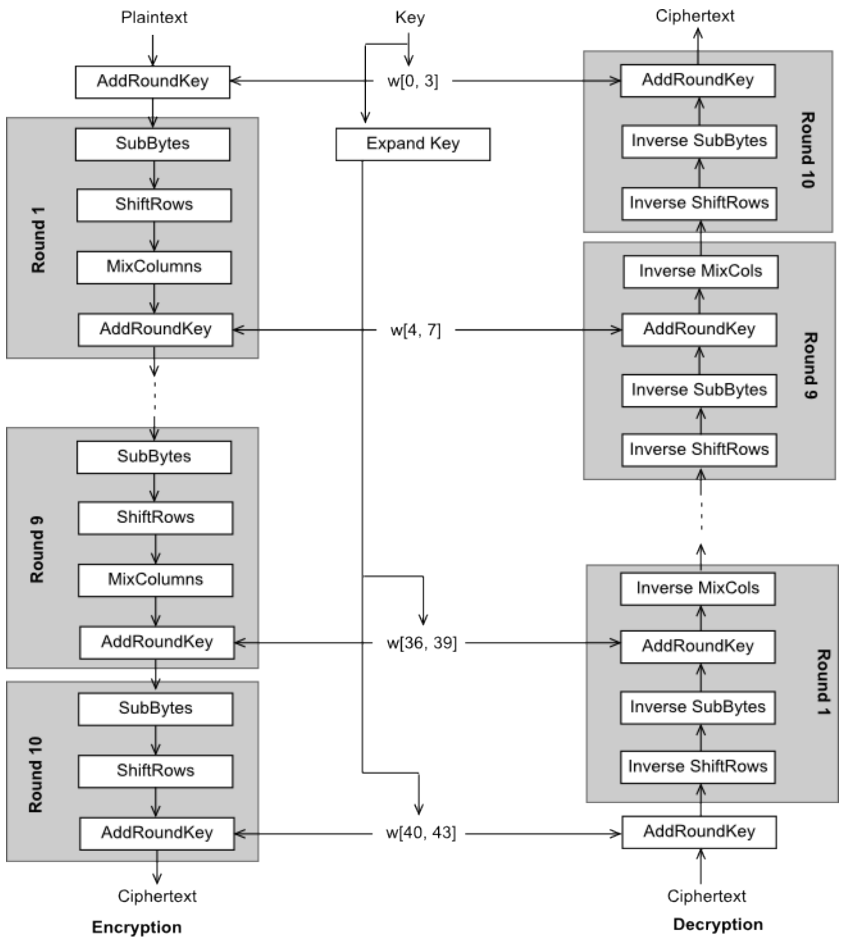

Figure 2.2.1: AES encryption and decryption [8]

2.2.1.2 Round Functions

- `SubBytes` is a transformation on each byte of the states using a simple substitution table which often called S-box. AES defines a S-box, which is a 256-bits matrix represents 256 possible transformations. The security of S-box is the determinant of `SubBytes` function. In fact, the S-box directly determines the success or failure of AES in terms of security since it is the only non-linear part of the whole algorithm.

- `ShiftRows` is a row's transformation of states. It acts on the each row of state, row 0, 1, 2, 3 shift to left 0, 1, 2, 3 bytes. Because `ShiftRows` operation is defined before, itself can not enhance the security of algorithm. But it is much more useful than it looks. Since in the encryption process, the first four bytes of plaintext are directly copied to the first column of input state, and next four bytes copied to the second column of input state, and in turn. Further, the round keys are applied to the state column by column. Therefore, the row shift moves one byte from one column to another column with a linear distance of a multiple of 4 bytes. Also, the transformation ensures that 4 bytes in a column is expanded to 4 different columns.

- `MixColumns` operates on each column independently. Each byte in each column maps to a new value, which can be get from 4 bytes in this column through function transformation.

$$\begin{bmatrix} S_0' \\ S_1' \\ S_2' \\ S_3' \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S_0 \\ S_1 \\ S_2 \\ S_3 \end{bmatrix}$$

  The aim of this coefficient of matrix is try to make all the bytes of each column in good confusion. Each output bit is influenced by all input bits after `MixColumns` and `ShiftRows` transformation, but it is impossible to compute the input by simple analysis of output. In addition, the coefficient of `MixColumns` $\{01\}$, $\{02\}$, $\{03\}$ are based on the efficiency of the algorithm: these multiplications involve at most one shift operation or one XOR operation.

- `AddRoundKey` is just a XOR operation of 128-bits input state and 128-bits round key. It is a simple transformation and easy to implement, it can affect each bits of input state. The complexity of `AddRoundKey` and the other transformations ensure the security of the algorithm.

Since the transformations in the encryption is invertible, decryption algorithm is similar to encryption algorithm but only change the transformation used in encryption to corresponding inverse transformations.

2.2.1.3 Block Cipher Modes of Operation

AES algorithm provides a construction for encipherment, the size of its blocks stays 128-bits. But in practice, the plaintext often much longer than the length of the blocks. Thus we need split the plaintext to satisfy the specified length. The problem of how to split plaintext is what modes of operation mainly concern. Currently, the frequently used modes of operation for confidentiality includes ECB, CBC, OFB, CTR [10].

- The Electronic Code Book (ECB) mode is the simplest mode of operation. It deals with one plaintext block at a time and uses the same cipher key every time. Under this mode, each block of plaintext is encrypted independently to generate independent block of ciphertext, so that each result will not be influenced by other results. Because of this, multiple encrypt operations and decrypt operations can be executed in parallel. During the transmission, error occurs in any block will not influence other blocks. On the contrary, ECB mode of operation is easy to reveal the information of plaintext. Because if the sent plaintext includes fixed and repeated data, and the size of this plaintext equals to the size of cipher block, it may generate the same ciphertext. Once the same ciphertext accumulated to a certain number, the plaintext can be guessed from ciphertext.

- The Cipher Block Chaining (CBC) mode combines the previous ciphertext block with the current plaintext block before encrypting the current plaintext block. This combination is a `XOR` operation. In this way, a chaining of ciphertext will be formed. The result of `XOR` operation of the first plantext block and initialisation vector is the first ciphertext block, the following plaintext block combines the previous ciphertext block before encryption till the last block cipher is generated. Because each ciphertext block affects the previous ciphertext block, so the fixed and repeated data will be performed differently. In this mode, encryption process must be sequential, not parallel.

- The Counter (CTR) mode uses a string with the same length of plaintext block as a counter. For given counter sequence $T_1, \cdots, T_{t-1}, T_t$, plaintext $P_1, \cdots, P_{t-1}, P_t$ (where $|P_1| = \cdots = |P_{t-1}| = n, |P_t| = u$), then the CTR mode can be defined as follows [10]:

  - CTR encryption:

$$\begin{aligned}
O_i &= E_k\left(T_i\right), \quad i = 1, 2, \cdots, t; \\
C_i &= P_i \oplus O_i, \quad i = 1, 2, \cdots, t-1; \\
C_t &= P_t \oplus MSB_u\left(O_t\right).
\end{aligned}$$

  - CTR decryption:

$$\begin{aligned}
O_i &= E_k\left(T_i\right), \quad i = 1, 2, \cdots, t; \\
P_i &= C_i \oplus O_i, \quad i = 1, 2, \cdots, t-1; \\
P_t &= C_t \oplus MSB_u\left(O_t\right).
\end{aligned}$$

Where MSB_u denotes the most significant u bits of the last block. Since CTR mode has many advantages like high efficiency, simplicity and provable security, the latter designed mode of operation was affected by it. For example, CCM mode is a combination of CTR mode and CBC-MAC mode.

2.2.2 RSA

Unlike symmetric key cryptography, public key cryptography uses two separate keys. It is based on mathematical functions rather than simple patterns on bits which is used on symmetric key cryptography. RSA algorithm [31] was proposed by Riverst, Shamir and Adleman in 1978. It is a widely deployed representative public key encryption algorithm with proved safety and is relatively easy to understand and implement. A high level description follows:

2.2.2.1 Key Generation

Choose two large primes p and q at random,

1. Compute $n = pq$, $\varphi(n) = (p-1)(q-1)$, which is Euler's totient function.
2. Choose an integer e which satisfy $\gcd(e, \varphi(n)) = 1$, then public key is (e, n).
3. Compute d such that $ed = 1 \,(\text{mod}\ \varphi(n))$, then private key is (d, n).

The prime numbers p and q should be large enough, otherwise n will be attacked by factorisation. According to the ability of modern computers, n should be chosen as 1024-bit or 2048 bit, so p and q are 512 bit or 1024 bit. In addition, it requires that p and q should not be too close and the greatest common divisor between them should be small enough. Another chosen parameter is public key exponent e, the value of e that RSA lab suggests is 65537 which has high security and fast computing speed.

2.2.2.2 Encryption and Decryption

Firstly, split the message M into blocks and translate them into integers, the length of each block $0 < m < n - 1$. Then encrypt or decrypt each block sequentially:

1. Encryption: Computing $c = m^e \,(\text{mod} n)$ to obtain the ciphertext by public key exponent e and plaintext m,
2. Decryption: Plaintext m can be recovered by computing $m = c^d \,(\text{mod} n)$ using private key exponent d.

2.2.3 Implementation of Primitives in JavaScript

2.2.3.1 General Implementation Techniques for AES

2.2.3.1.1 S-box Generation As mentioned before, S-box is a substitution table used in the `SubBytes` round function which is a non linear transformation. It is generated from multiplicative inverse in the field $GF\left(2^8\right)$. $GF\left(2^8\right)$ represents a characteristic 2 finite field, which is a field contains finite number of elements in abstract algebra, with 8 terms. AES uses a irreducible polynomial of degree 8 $x^8 + x^4 + x^3 + x + 1$ which corresponses to multiplication in $GF\left(2^8\right)$ to do arithmetic like multiplication. If it needs to compute these in each encryption process, the speed would be definitely slow. All the elements in the finite field $GF\left(2^8\right)$ mapping the elements after multiplicative inverse and affine transformation constitute the S-box. The S-box and inverse S-box which used in the decryption process with 256 entries are widely used in cryptographic since it is described in the AES standard submission. Several methods were emerged to optimise the S-box. The research shows that the difference between S-box and inverse S-box is their linear arithmetic part [18]. Multiplicative inverse which is the most complexity part is the same. So if we use only lookup table in the multiplicative inverse part, one look-up table is enough for both S-box and inverse S-box. This method would reduce one lookup table but add inexpensive linear operation part, which can reduce the code size. [38] shows that use combinational logic can efficient do the arithmetic operations in the $GF\left(2^8\right)$.

2.2.3.1.2 Round Functions Implementation Round functions are the core of the AES algorithm. Its efficiency direct determines the speed of encryption and decryption. It can use following method to construct lookup tables, and can use these lookup tables to implement round functions efficiently.

Let S-box lookup of byte $S_{r,c}$ is $S\left[s_{r,c}\right]$, then `SubBytes` and `ShiftRows` operations can be represented as:

$$\begin{bmatrix} b_{0,c} \\ b_{1,c} \\ b_{2,c} \\ b_{3,c} \end{bmatrix} = \begin{bmatrix} S\left[s_{0,c}\right] \\ S\left[s_{1,c}\right] \\ S\left[s_{2,c}\right] \\ S\left[s_{3,c}\right] \end{bmatrix}, \quad \begin{bmatrix} d_{0,c} \\ d_{1,c} \\ d_{2,c} \\ d_{3,c} \end{bmatrix} = \begin{bmatrix} S\left[s_{0,c(0)}\right] \\ S\left[s_{1,c(1)}\right] \\ S\left[s_{2,c(2)}\right] \\ S\left[s_{3,c(3)}\right] \end{bmatrix}$$

`MixColumns` and `AddRoundKey` can be represented as:

$$\begin{bmatrix} e_{0,c} \\ e_{1,c} \\ e_{2,c} \\ e_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S\left[s_{0,c(0)}\right] \\ S\left[s_{1,c(1)}\right] \\ S\left[s_{2,c(2)}\right] \\ S\left[s_{3,c(3)}\right] \end{bmatrix} \oplus \begin{bmatrix} k_{0,c} \\ k_{1,c} \\ k_{2,c} \\ k_{3,c} \end{bmatrix} \tag{2.1}$$

The expression above is a combination of `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. The $c(r) = [c + h(r, Nb)] \bmod Nb$, where $h(r, Nb)$ is the bytes that row r needs to shift. It can be transferred as follows:

$$\begin{bmatrix} e_{0,c} \\ e_{1,c} \\ e_{2,c} \\ e_{3,c} \end{bmatrix} = S\left[s_{0,c(0)}\right] \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S\left[s_{1,c(1)}\right] \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \tag{2.2}$$

From the above expression we can construct four tables:

$$T_0[x] = \begin{bmatrix} S[x] \cdot 02 \\ S[x] \\ S[x] \\ S[x] \cdot 03 \end{bmatrix} \quad T_1[x] = \begin{bmatrix} S[x] \cdot 03 \\ S[x] \cdot 02 \\ S[x] \\ S[x] \end{bmatrix} \quad T_2[x] = \begin{bmatrix} S[x] \\ S[x] \cdot 03 \\ S[x] \cdot 02 \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ S[x] \cdot 03 \\ S[x] \cdot 02 \end{bmatrix}$$

Therefore, round functions transformation can simplified as:

$$\left[d_{0,c}, d_{1,c}, d_{2,c}, d_{3,c}\right]^T = T_0\left[s_{0,c(0)}\right] \oplus T_1\left[s_{1,c(1)}\right] \oplus T_2\left[s_{2,c(2)}\right] \oplus T_3\left[s_{3,c(3)}\right] \oplus k_{round,c} \tag{2.3}$$

where $k_{round,c}$ denotes the round key of column c.

These four tables occupy 4 KBytes since each one occupies 1 KBytes. For round functions transformation of each column, it only needs 4 table lookups and 4 `XOR` operations, which greatly improves the efficiency of the algorithm. From the above analysis, it is not difficult to get $T_i[x] = RotByte(T_{i-1}[x])$, where $RotByte$ rotates the bytes of a column. Therefore,

$$\begin{aligned}[d_{0,c}, d_{1,c}, d_{2,c}, d_{3,c}]^T = k_{round,c} \oplus T_0\left[s_{0,c(0)}\right] \oplus RotByte \\ \left(T_0\left[s_{1,c(1)}\right] \oplus RotByte\left(T_0\left[s_{2,c(2)}\right] \oplus RotByte\left(T_0\left[s_{3,c(3)}\right]\right)\right)\right)\end{aligned} \tag{2.4}$$

If T_0 is constructed, the other three tables are easily got. [8]

2.2.3.2 General Implementation Techniques for RSA

2.2.3.2.1 Prime Number Generation The first step to process the RSA algorithm is to choose two large and safe prime numbers. To avoid the elliptic curve factoring algorithm, the generated two prime numbers also should be the similar length. The random number generator `Math.random()` in JavaScript is not feasible to apply in the cryptographic programs since the generated numbers are not restrictly unpredictable. Therefore, we needs to design the method to do this work independently. If it has generated two numbers, we need to test the primality of them. In practice, test method for primality based on Remann's hypothesis [27] which emerged by Miller and Rabbin has been used widely. In theory, it seems that this method can fast generate a pseudoprime number. But in fact, a random generated number can hardly pass the Miller-Rabin test. So, it needs to do preprocess before test. For example, use little prime numbers like 3, 5, 7, 11 to divide the random generated number before Miller-Rabin test.

2.2.3.2.2 Modular Multiplication Because the numbers involved in the RSA algorithm are usually very large, the speed of their arithmetic plays a crucial role. Montgomery algorithm [28] is one of the most famous methods to speed modular multiplication. [9] improved the Montgomery algorithm to faster the speed. Let N be a positive multi-precision integer, R and T be multi-precision integers, which satisfy $R > N$, $\gcd(N, R) = 1$, $0 \leq T \leq NR$. Montgomery algorithm can compute $TR^{-1} \bmod N$ without using classical multiplication. If represents N as b based n-bit integer, assumes $R = b^n$. This assumption satisfies $R > N$, but satisfies $\gcd(N, R) = 1$ only when $\gcd(b, N) = 1$. For RSA algorithm, N is odd, so b can be the power of 2. It often use $b = 2^{32}$ in the 32-bit computers. The improved algorithm shows as follows:

Algorithm 2.1 Improved Montgomery algorithm [9]

```
**Input**: A multi-precision integer $T$, $\omega = -m^{-1} \bmod b$.
**Output**: A multi-precision integer $a = TR^{-1} \bmod N$.
$a \leftarrow T$
**for** $i = 0$ **to** $n - 1$, **do**
    $u_i \leftarrow a_i \omega \,(\text{mod} b)$
    $a \leftarrow a + u_i \cdot N \cdot b^i$
**end**
$a \leftarrow a/b^n$
**if** $a \geq N$, **then**
    $a \leftarrow a - N$
**end**
**return** $a$
```

The operation inside the recursion needs $n + 1$ single precision multiplications for each recursion, so it requires $n\,(n + 1)$single precision multiplications totally since it has n recursions. Besides, some trivial addition and shift operations would be involved, but no need to do division.

2.2.3.2.3 Modular Exponentiation From RSA encryption and decryption processes, we know the process is to do modular exponentiation actually. Modular exponentiation can be divided into modular multiplication and exponentiation. Efficient modular multiplication has been illustrated, this section will discuss the method to do exponentiation. A naive way to compute m^e is to do $e - 1$ multiplication, but usually the choices of e is large enough so that it is not a efficient way. In general, there are two more efficient ways to do exponentiation. One is decrease the time to do multiplication, the other way is reduce the numbers of multiplications [36]. These several algorithms developed from these two aspects. Right-to-left binary exponentiation, left-to-right exponentiation, left-to-right k-ary exponentiation, and sliding-window exponentiation are the common algorithms to do exponentiation. The following algorithm combines left-to-right exponentiation and Montgomery algorithm to reduce the time to do modular exponentiation.

Algorithm 2.2 Montgomery exponentiation [36]

```
**Input**: A multi-precision integer $1 \leq x < m$, modular $m = (m_{l-1} \cdots m_0)_b$, $R = b^l$,
$e = (e_t \cdots e_0)_2$ with $e_t = 1$
**Output**: $A = x^e \bmod m$.
$\tilde{x} = Mont\left(x, R^2 \bmod m\right)$, $A \leftarrow R \bmod m$
**for** $i$ **from** $t$ **downto** $0$, **do**
    $A \leftarrow Mont\,(A, A)$
    **if**$e_i = 1$ **then** $A \leftarrow Mont\,(A, \tilde{x})$
**end**
$A \leftarrow Mont\,(A, 1)$
**return** $A$
```

Since in RSA encryption and decryption, the exponent e is fixed, so the algorithms target on the fixed-exponent can further reduce the arithmetic time by reducing he number of multiplications. Addition chain exponentiation and vector addition chain exponentiation are the examples of them.

2.2.3.3 Examples from Existing JavaScript Libraries

There are some existed JavaScript cryptography libraries like sjcl (Stanford JavaScript Crypto Library), Clipperz, jCryption and Forge.

sjcl is a JavaScript library which currently focus on the symmetric cryptography. It implements 128-bit, 192-bit and 256-bit AES algorithm and adopts some strategies to improve the code for the JavaScript interpreter. Their results show that they have reduced both the code size and running time successfully. The tests also tell that the performance of their library has a great improvement compared to other similar libraries. For AES, AES-CMAC mode is adopted for ensure the data integrity while OCB mode and CCM mode are employed to provide both data integrity and confidentiality. Instead of precomputing the lookup tables in the code and computing the lookup tables during the encryption, sjcl builds the tables on the web browser before starting to do encryption to avoid increasing the code size or running time [35]. The following code shows sjcl how to expand the S-box tables. As mentioned before, these tables can be computed on the client so we do not need to send them via Internet. `encTable` and `decTable` stores four tables for encryption and decryption respectively.

```
for (x = xInv = 0; !sbox[x]; x ^= x2 || 1, xInv = th[xInv] || 1) {
    // Compute sbox
    s = xInv ^ xInv<<1 ^ xInv<<2 ^ xInv<<3 ^ xInv<<4;
    s = s>>8 ^ s&255 ^ 99;
    sbox[x] = s;
    sboxInv[s] = x;
    // Compute MixColumns x8 = d[x4 = d[x2 = d[x]]];
    tDec = x8*0x1010101 ^ x4*0x10001 ^ x2*0x101 ^ x*0x1010100;
    tEnc = d[s]*0x101 ^ s*0x1010100;
    for (i = 0; i < 4; i++) {
        encTable[i][x] = tEnc = tEnc<<24 ^ tEnc>>>8;
        decTable[i][s] = tDec = tDec<<24 ^ tDec>>>8;
    }
}
```

The optimisation strategies that sjcl employed includes loop unrolling, bitslicing and unify encryption and decryption functions. Loop unrolling is an optimisation technique which decrease the loop cost by increasing the instructions. If loop body is simple and large part of instructions is used to check conditions of loop, it can unroll the loop to linearise the object. Then the program has less iterations, the cost decreases but the size of code increases. For the local use, loop unrolling is often adopted. But for JavaScript in the client side, code size is an important factor to be considered. So in sjcl, the developers determine whether use loop unrolling or not for AES depends on different situations: unrolling short loops, unrolling main round functions loops and leave other loops rolled. The current most efficient AES implementation adopts bitslicing technology. Bitslicing breaks the AES algorithm into bit operations so that several encryptions or decryptions can be computed in parallel [35]. But sjcl developers pointed out that bitslicing does not perform well in JavaScript. The poor performance of loop unrolling and bitslicing in JavaScript seems blame JIT compiler (which we will discuss in detail latter) in JavaScript engines. Another strategy used in sjcl to reduce the code size is using a flag to distinguish the encryption and decryption functions which have been combined by different look up tables.

[33] shows a RSA implementation in JavaScript, it adopts Barrett algorithm to reduce the modular exponentiations. Montgomery reduction requires $2 \cdot n \cdot n$ digit-multiplications for an n-digit N while Barrett reduction needs $2 \cdot (n+1) \cdot (n+1)$ multiplications. But [5] argues that though Montgomery reduction has better performance for general modular exponentiations, Barrett reduction performs better for small arguments.

Chapter 3

Investigation

3.1 JavaScript Execution

3.1.1 JavaScript Execution Behaviour

Compared to the statically typed language, the dynamics of JavaScript type system can be performed as follows.

- Variables do not need to be declared statically. The type of variables can be changed according to the operations at run-time.
- The type of objects and parameters in functions can be changed dynamically.
- The type of properties of objects can be dynamically added, deleted or changed.
- It can generates executable program via `eval()` function and constructor on the fly. In JavaScript, `eval()` function is a combination of a statement executor and an expression calculator. It executes the argument it receives and returns a result.
- Arrays are associative arrays, which can be used as hashmap. Therefore the index of array does not necessarily starts from 0, or continuous.

```
1  var a = 0;
2  a = “hello”;
3  a += null;
4  b = {};
5  b.c = x;
6  delete b.c;
7  eval “b.c = x”;
8  c = [];
9  c[1000] = b;
```

Examples above show some features of JavaScript dynamic type system. From line 1 to 3, the type of variable `a` changed during the execution. It was defined and initialised as a number, and then it was reassigned as a string. Variable `b` does not need to be declared before being assigned. Line 3 means a `string` added with `null` is still a `string`. Line 6 shows an instance of dynamically deleting property of an object. A simple `eval()`

function in line 7 executes a statement. Empty array `c` in line 8 becomes a sparse array by assignment in line 9.

An empirical study [30] promotes some assumptions about the dynamic behaviours of JavaScript which summarised from literature and implementations. Though a small modification to the prototype hierarchy would influence the control flow graph of the program and the type of affected objects, many type systems for JavaScript and static analyses chose not to consider the prototype since the vast majority of the prototype hierarchy is invariant. Some researchers believe that most of the dynamic behaviours in JavaScript occur at the "initialisation phase" of the program. The rest behaviours of programs are often static. Therefore, the programmers can assign as a complete type as possible to the objects at the initialisation phase, and only a small number of properties and objects would be changed after creation. Some JIT compiler optimisations are based on the similarity of JavaScript and other object-oriented programming languages like Java. For example, these languages have a low call-site dynamism. Furthermore, the assumptions like most execution time is spent on hot loops and JavaScript programs are often has modest size have been highlighted previously. These assumptions are important for analysing the dynamic behaviours of JavaScript and would be the foundation of several ideas to speedup JavaScript.

3.1.1.1 Prototype and Object Property

JavaScript has no concept of class, the objects are essentially maps from names to values. The mapping can be infinitely and dynamically changed. Because of the dynamics of JavaScript, layout of object properties in memory is dynamic. Hence the relative position or offset cannot be determined ahead of time. The efficiency of property accessing inferior to static programming languages since it needs to determine the position of property on the fly at run-time. Since JavaScript has no class, it also has no class-based inheritance. The property inheritance of prototype in JavaScript is similar to class inheritance.

Unlike static programming language, there are significant differences between getting and assigning values of object properties in JavaScript. In the process to get the value of properties, the prototype object will be checked if the specific property cannot be found. The corresponding value will be returned once the property is found in the prototype object. On the contrary, the assignment operation only affects the object itself. A new property will be added to the object if the specific property cannot be found. In other words, getting the value of a property may involve its prototype while assigning value to a property only operates on its object itself. Due to the difference, the experiment separates these two operations. We do some experiments on a set of JavaScript programs. The test programs are from SunSpider JavaScript benchmark. It is a widely used benchmark for testing JavaScript performance. There are several reasons why we chose this benchmark rather than others. The code of SunSpider is extracted from real applications so that it has better representativeness. It balances the features of core JavaScript to have a more general testing. The result of repeatedly testing has little fluctuation, which declares the benchmark is statistically reliable.

The figure 3.1.1 shows that almost all the value getting operations of properties hit on the object itself or prototype object. Between them, the object itself has vast majority proportion. On average, about 96% and 4% value getting operations hit on the object itself and the prototype object respectively. Further looking at the value getting operations hit on the object itself, figure 3.1.2 displays that 99% properties has the same storage location with last access. For the operations hit on the prototype object, it has the similar result which shows in figure 3.1.3. The value assignment operations of properties only assign the value to the object itself but not affect the prototype object. The experiments also expose that the storage position of property and the prototype object almost do not change at the same call site if the expected property exists.

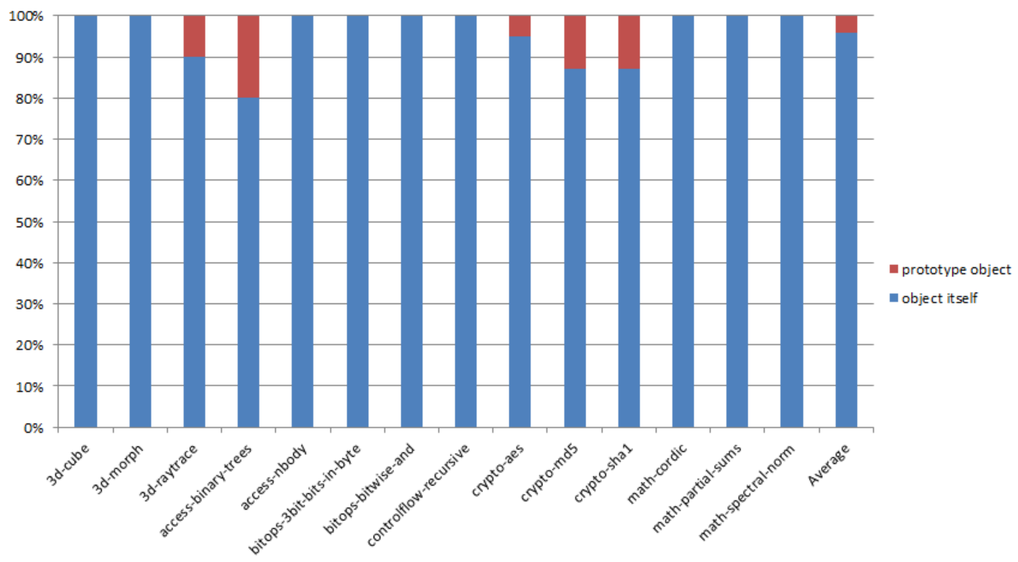

Figure 3.1.1: The comparison of properties location

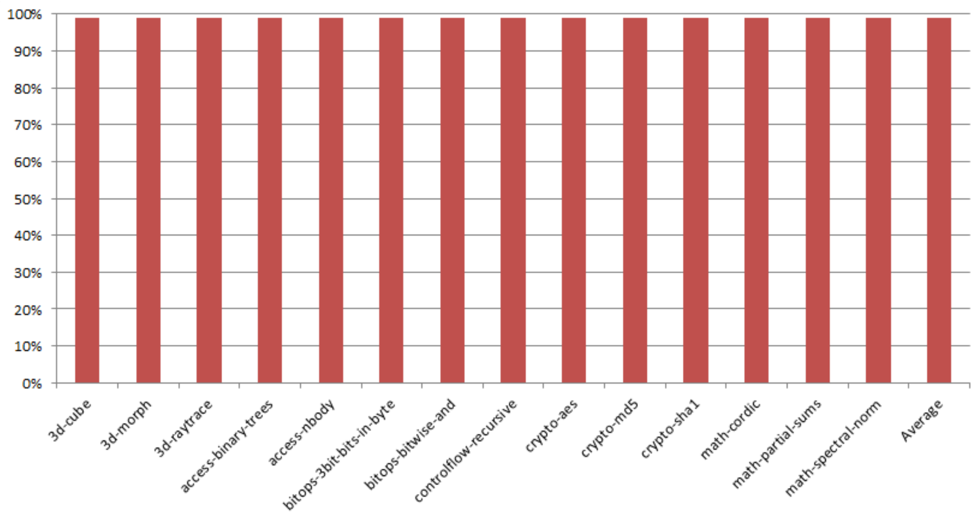

Figure 3.1.2: The proportion of same properties location (in object itself)

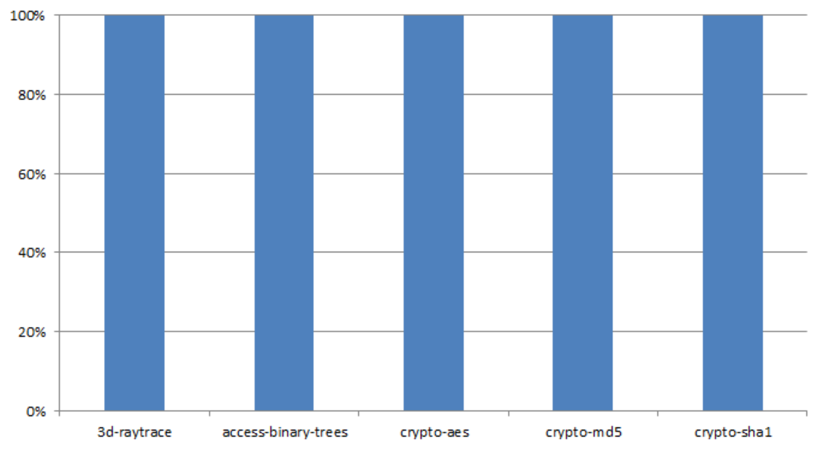

Figure 3.1.3: The proportion of same properties location (in prototype object)

3.1.1.2 Array

Just like array in other dynamic languages, array in JavaScript is associative array. It can be represented by hash table which takes index of array as key. The advantage of hash table is its values do not need to be continuous, but this could have unfavourable impact on caching and accessing arrays. In real application, arrays are more likely to be handled continuously. If it makes sure its continuity, the operation on array will be more efficiency.

Based on this, the array in JavaScript can be designed as: a linear array with index from 0 and a hash table. If keys are non-negative integers and within the length of linear array, value can be stored in the linear array directly. Otherwise, if the array can accept the key by proper expansion, then expands the linear array and stores the value (but make sure the elements that did not accessed do not occupy a large proportion). If the linear array needs a great expansion, then put the value in the hash table. For the values with negative indices, also puts them into the hash table.

```
a[0] = 1; a[1] = 7;
a[2] = 9; a[5] = 10;
a[-2] = 5; a[10000] = 2;
a[“bc”] = 1000;
```

For above example, the current length of an array is 6, so the storage of the value is as follows.

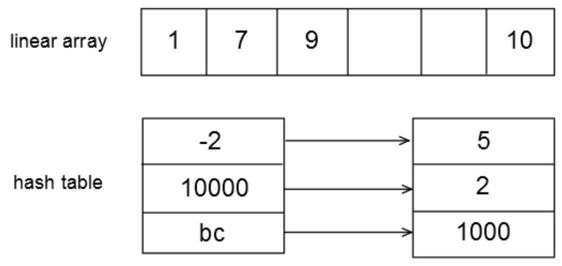

Figure 3.1.4: The storage of elements for an array

Based on this, we do experiment on the features of array access. We mainly focus on the indices, continuity of key-value. Figure 3.1.5 presents the status of arrays when accessing elements of arrays. From figure 3.1.5, we conclude that in majority cases, arrays are full or almost full and the index starts with 0. It indicates that arrays access and storage are continuous in most situations. In three cryptographic programs, not all the arrays are full, but the empty rate of each array is smaller than 5%. In all the array access operations, the index of about 91% arrays starts with 0, 94% arrays are full on average. In more than 99% arrays, the empty rate of each array is smaller than 5%.

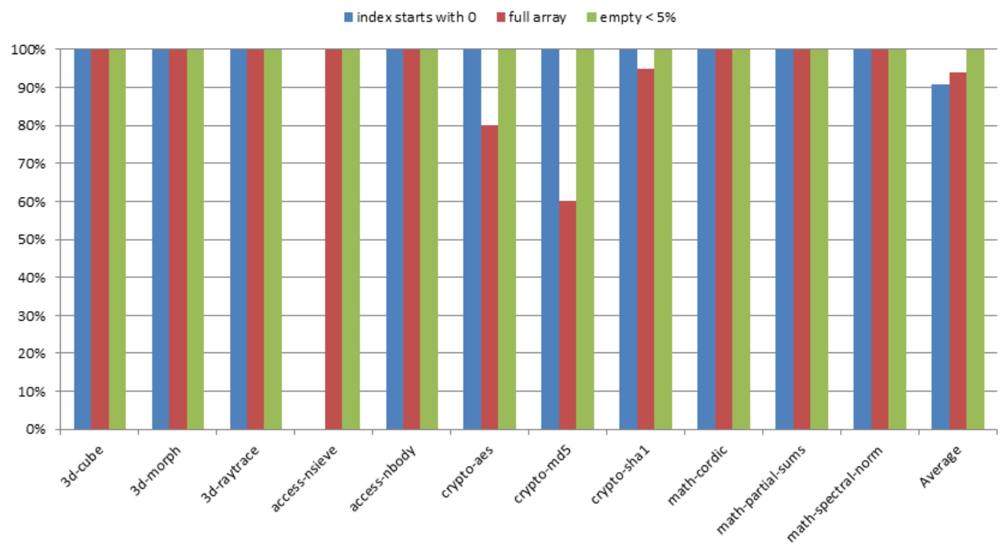

Figure 3.1.5: The status of arrays

3.1.1.3 Number

The experiments exam the operations on the type of `Number`. Since JavaScript is weakly typed language, it has not specific integer or floating point types. The numbers in JavaScript are 64-bit floating point numbers, which are represented as IEEE-754 Doubles.

The experiment gathers data of operations on 32-bit integers and the others. The arithmetic in the experiments includes addition, subtraction, multiplication, division, modular arithmetic, bitwise operation, increment operation and decrement operation. From figure 3.1.6, integer operations account for the vast proportion of some test programs (especially in the bitwise operation test program) , while integer operations and floating point number operations both give some weight. In general, there are 64% arithmetic operations of all is between integers. From that we can see, more accurately distinguishing integer number and floating point number will make programs get better performance.

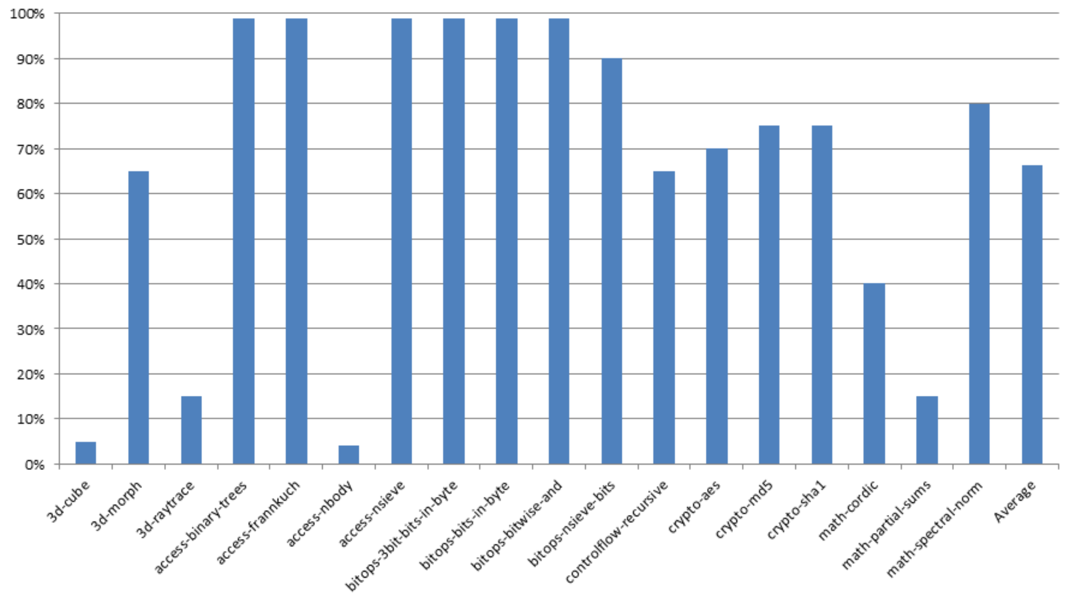

Figure 3.1.6: The proportion of integer operations

From the experiments, we can conclude as follows.

- For the practical JavaScript programs, separating 32-bit integers and 64-bit floating point numbers will help to improve the performance of program execution.
- Almost in all the call sites, no matter expected property is in an object itself or in the prototype, its position is almost permanent.
- The actual behaviours of an array in JavaScript is similar to it in static programming languages. In majority cases, the index starts with 0 and continuous.

Though JavaScript provides abundant dynamics, the practical JavaScript programs use it very limited. The verdict not only is an important guidance to design efficient JavaScript engine, but also provides a theoretical basis for applying the traditional compiler optimisation techniques to JavaScript.

3.1.2 Intermediate Representation

During the process of translating JavaScript source code into the object code, the engine may construct one or more intermediate representations. The advantage of using multiple intermediate representations is it can apply efficient optimisation at low-level as well as keep the code information collected from the front and the middle end. Abstract syntax tree is typical high-level intermediate representation which used for syntax analysis. After that, the interpreter often converts it to bytecode, which is a kind of middle-level intermediate representation. Many optimisations are based on the intermediate representations.

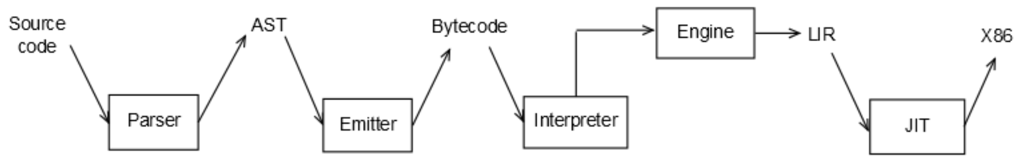

Figure 3.1.7: A typical intermediate representation for JavaScript engine

3.1.2.1 Three-address Code

Three-address code is a popular format for intermediate representation. It is a sequence of basic program steps like an addition of variables. This form allows a significant code optimisation, since we can decompose the long three-address sequence into basic blocks. As the name suggests, the universal form `x = y op z` of these instructions have three addresses: two operands `y` and `z`, and one result variable `x`.

Example: $x + y * z \rightarrow t1 = y * z, t2 = x + t1$

Since the right hand of an instruction can only have one operator in three-address code, it decomposes arithmetic expression with multiple operators and nested control flow, so it suitable for generating and optimising the object code.

3.1.2.2 Static Single-Assignment Form

Def-use chain is a data structure that can obtain information about uses of variables or all the definitions of a variable. The information is important for many data flow analysis. With def-use chain, compiler can store two lists consists of pointers: pointers in one list point to all the uses of variables in syntax, pointers in the other list point to all the definitions of variables in that syntax. By means of that, compiler can efficiently jump from use to definition or from definition to use. Static single-assignment (SSA) form is an improvement of def-use chain, it efficiently splits the values used in the program and its storage location. In SSA, each variable only has one definition value. Because of this, data flow analysis and optimisation algorithm can be simpler. In some sense, the aim of SSA is to get the same accuracy using flow-insensitive analysis instead of using flow-sensitive analysis. Because though flow-sensitive analysis is often more complex than flow-insensitive analysis, it can get more accuracy result.

By using SSA form, it can significantly enhance many compiler optimisations, including constant propagation, sparse conditional constant propagation. The other feature of SSA is it only has one definition reachable in each reference. The implementation of these two features relies on the introduction of φ function and rename the variables. φ node is a virtual symbol, it merges the different definitions for a variable into a definition. We can rename the different definitions of variables x as $a_1, a_2, \ldots$. In a linear program, we can rename all the definitions of a, then rename the use of a as last definition. For the programs have branches, each use of a renames as the definition that most near a. Hence, this technique splits multiple references and multiple definitions.

SSA form also has its disadvantages. Firstly, programs need to be converted to SSA form before analysis and optimisation. The programs need to be converted to other forms again since hardware architecture does not support SSA form. The two transformations have extra overhead. Secondly, changes of the programs after the optimisation lead to the exchange of φ node positions. Therefore, maintaining the φ nodes need to be taken into account while optimisation the programs, which complicates program optimisation and results in poor portability. Furthermore, the method of φ node only considers multiple definitions of a variable which is caused by control flow, but overlooks the reason of may-define. As a result, SSA form seems not very efficient when it exists a large mount of may-define. May-define may be caused by arrays, which is common used in modern programming language.

TraceMonkey is typically implemented in a low-level intermediate representation (LIR), which is three-address SSA code. LIR is almost a one-to-one map to machine instructions. To ease the implementation of the subsequent optimisation, traces in TraceMonkey use the SSA form without join points or φ node, which is called linear SSA code. As the name suggests, a piece of linear SSA code is a linear sequence of instructions. As mentioned above, to deal with the branches of control flow and loop header, φ function and joins are often introduced. But the tracers in TraceMonkey have not supported to form these branches. For example, if we want to compute the maximum of two numbers, we could inline an alternate of a conditional since a number can be an integer coerced to a double. The LIR distinct integer and double, like the instruction `ldi` represents load integer number while `ldd` means load double number, and so as in instructions for store and arithmetic.

```
function(x) {
    if(x % 2 == 0)
        print(“even”)
    else
        print(“odd”)
    print(“done”)
}
```

```
i0:  get "x"
i1:  mod i0, $2
i2:  guard i1
i3:  const "even"
i4:  call "print" i3
i5:  const "done"
i6:  call "print" i5
```

```
i0:  get “x”
i1:  mod i0, $2
i2:  branch i1
      ↙ then         ↘ else
i3:  const “even”     i5:  const “old”
i4:  call “print” i3  i6:  call “print” i5
      ↘      join       ↙
i7:  const “done”
i8:  call “print” i7
```

Figure 3.1.8: The comparison of linear IR and non-linear IR (with joins)

3.2 JavaScript Optimisation

3.2.1 Type Analysis for JavaScript

For the dynamic language like JavaScript, it may need to determine the type of each operand. Then the code can be generated or interpreted according to the specific type. The example below shows the type checking for a simple arithmetic operation, from which we can seen that even for a simple arithmetic operation, its complete type checking also can be complicated. Accurate type analysis can decrease the complexity and increase the efficiency. More type information gets, less type checking needs to be done. Many techniques are focused on ways to obtain accurate type information and try to get that type information as fast as possible. From the view of compiler, the means to get type information can be divided into run-time type analysis and static type analysis. As the name suggests, the former technique aims to gather as much type information at the run-time, while the latter one do the work ahead of compilation.

```
a + b → ToNumber(a) + ToNumber(b)
function ToNumber(x) {
    switch(typeof x) {
        case number:  return x;
        case string:  return Str2Num(x);
        case true:  return 1;
        case false:  return 0;
        case object:  return Obj2Num(x);
        case null:  return 0;
        case undefined:  return NaN;
    }
}
```

3.2.1.1 Run-time Analysis

In JavaScript, the tests of types in the run-time is needed to ensure the meaningfulness of arithmetic. However, the tests can be repeatedly for the same variables since the type information is dropped after performing operations. Therefore, dynamic type checking is a source of performance loss in dynamic typed languages. Run-time type removal aggravate the run-time cost. One way to solve this problem is to develop a type system for the language and perform the static analysis. However, the static analyses may not perform well because of some features. For example, updating on selective code in the run-time may invalidate the previous analyses.

3.2.1.1.1 Type Specialisation As we know, the raw JavaScript interpreter has a poor performance. The source code is converted into bytecode or abstract syntax tree by the front end of a engine, and then directly interpreted by interpreter. However, for example, the expression `a = b + c` can have many explanations. If `b` and `c` both are numbers, it is a numeric addition. If `b` and `c` both are strings, it is a string concatenation. `b` and `c` also can represent different types. Because of this dynamic typing, the JavaScript interpreter uses its internal finer-grained types. Usually, the `Number` is separated into 32-bit integers and 64-bit floating-point numbers. Because arithmetics on floating-point numbers need more instructions than integers on a x86 architecture. To store type information, the bytecode

JavaScript interpreters “boxed” values. The “boxed” means represent the values with a type tag. The data structure to store this box format could be hash tables. For example, the integer 10 can be represented as `(INT, 10)`. The variables can only store boxed values, which must be “unboxed” (which means to extract the tag) before an operation. Therefore, if we execute the syntax `a = b + c`, the first step is to read the operation from cache, box `b` and `c`, and then check the type of `b` and `c` to determine the action. To compute it, it needs to unbox the `b` and `c`, and then do the central work: execute the syntax. After execution, the output `a` needs to be boxed and put into cache. Thus it can be seen that executing one syntax of JavaScript needs pretty much work before or after that. The technique of “boxed” and “unboxed” the value is a kind of dynamic type checking. Usually, a type tag is a machine word. In JavaScript interpreter, it often consists of up to the 3 of the least significant bits and remaining bits of data. For example, the integers in SpiderMonkey are the combination of the value of the integer and the tag `xx1`. The “boxed” and “unboxed” actions are actually the execution of a sequence of instructions. Before a value can be accessed, the instructions firstly separate the value from the tag. If the tags matches types, the type-appropriate arithmetic will be done. If the comparison fails, an error will be occurred. Though the type tag technique can distinguish objects from different types, it has extra space and time overhead.

The JIT compiler we mentioned before compiles the bytecode into native object code and stores the native code into the run-time repository. When executes the above syntax, the run-time will call the native code of syntax. By the aid of the JIT compiler, the action of reading and writing the operation or result to the memory can be omitted, which is much faster than just interpreting the bytecode. But we still needs to take “boxed” and “unboxed” actions which hurts the performance.

If the JIT compiler gets exact type information of each local variable thereby knows exact the action will be taken, it does not needs to check the type in the run-time and the variables do not need to be boxed or unboxed before or after executing. The extra actions taken in the type tagging will be omitted. In other words, in statically typed languages like Java, the types of variables are declared in programs so that the compiler knows the exact type of each value. Instead of using generic format to represent all the values and performing generic operations in JavaScript, the compiler can use specialised format and operation to generate more efficient code in Java. The assumption that some code is type stable which can be compiled like statically typed language is called type specialisation. The idea behind type specialisation is to let interpreter runs a bit of program for monitoring the types of values. Then the JIT compiler can compile the type-specialised code. The JIT compiler in TraceMonkey is using type specialisation. The trace we described before is the typed trace in which each local variable has a type. Type specialisation aims to eliminate the overhead of “unboxing” and “boxing” values. Instead of boxing technique used in the interpreter, the trace records the variables to its native record, thereby the reading and writing actions are equivalent to simple loading and storing.

All the numbers in JavaScript are 64-bit floating-point numbers, but some operations in JavaScript programs are the arithmetic between integers (e.g. bitwise operation or array access). In this case, it sets the numbers as integers at first and then converts the result back to doubles if necessary. For arrays, the process will be a bit more complex. It must firstly convert integer indices into strings for the convenient of property accessing, and then convert to the integers [15]. If the JavaScript engine eliminates these conversions, the performance will be improved. Type specialisation for numbers exactly does this job: separating the representation of integers and doubles.

The design of type specialisation is based on the assumption that hot loops are mostly type-stable since the hot loops are the target for generating traces. In other words, type specialisation relies on tail of the loop can jump back to the loop top directly. But if the types of variables in the loop change after the first running, the generated traces cannot be used since the types are not compatible. This can be solved by creating a new trace and jumping between traces. However, the solution is not perfect when the types are highly unstable. For the case with deeply nested type-unstable trees, the process of recording and compiling will take a long time which even longer than basic interpretation. In this situation, it has to give up specialised types and back to the basic JIT compiler. It still exists other limitations of type-specialised JIT. The elements of arrays, the properties of objects still often need to be boxed and unboxed, still need to be checked the type. The arithmetic of integers still requires overflow checking.

3.2.1.1.2 Type Inference

To eliminate the remaining cases that still need boxing mechanism, type inference is considered. The idea of type inference is try to prove the types of values by static analysis. The core difference between type specialisation and type inference is the former one uses types has been seen before while type inference uses types that are provable true. Both of them are based on speculation, but in different ways. Type specialisation needs to use type guards to check the consistent of the types which has been seen before. Type inference speculation based on the types which are more prevalent. We consider an addition arithmetic of two integers: `a = b + c`. The result has two situations which we cannot be use: `a` is still an integer or `a` becomes a double because of overflow. In type specialisation based JIT, the type guard is used before or after the arithmetic to make sure `a` is an integer. In type inference based JIT, the fast addition or the slow addition will be done depends on whether `b` and c are integers or not. If we know enough information about the range of operands, the type check and the overflow check can be omitted by type inference algorithm.

Static type inference is a complement of run-time type analysis. It can eliminate the guards which we used in the run-time to validate speculation. With type inference, it can produce a set of types for a variable through each point in the program. We take an example:

```
for (var i = 0; i < max; i++) {
    a[i] = i * i;
}
for (var i = 0; i < max; i++) {
    total += a[i];
}
```

With type inference, it can tell that `i` is always `a` number, therefore `a[i]` and `total` are always numbers. Furthermore, if you know `a` is an array of integers and `i` is an integer, you do not need to check that `a[i]` is an integer. To implement the type inference, a worked type inference algorithm for JavaScript and the ability to integrate the algorithm with the JIT compiler to generate high quality code are two key things. Before developing the type inference algorithm for JavaScript, researchers are usually define a formalism for it at first. Based on the formalisation, one of the major challenges of type inference is to track the objects to determine the structure of them. Static analyses are often tools for developing type inference algorithms.

Combining static analysis with type specialisation is reasonable to be thought of. It means detect the types of values by static analysis ahead of time and then compile them with type specialisation since both of them are effective for JIT optimisation.

3.2.1.2 Static Analysis

Static program analysis is a method to analyse the program without actually executing programs. Static analysis can be performed either in the source code or the object code. Static analysis is a tool that can improve reliability, security, performance of programs. In this section, we only consider its impact on program performance. The statical information computed by static analysis may be used for optimising run-time execution. Different approaches of static analysis include abstract interpretation, points-to analysis, shape analysis and so forth.

3.2.1.2.1 Abstract Interpretation The abstract interpretation technique interprets a program's semantics with abstract values. It usually based on lattices which is a particular partially ordered set. Static analysis by abstract interpretation used in JavaScript programs is checking the type of variables.

By simple type inference algorithm, it can tell a variable in a loop is a number, but it cannot distinct integers and floating-point numbers since it cannot infer the value ranges of variable. If the analyses can tell a variable will never outside of the range $[-2^{31}, 2^{31}-1]$ and never assigned a fractional value, an infinity or a `NaN`, it can be safely specialised a 32-bit integer. The first condition can be decided by interval analysis which used to determine the range of variables. A numerical abstract domain in abstraction interpretation is often used to check the bound of values. The choice of numerical abstract domain is typically a balance of precise and cost. These abstract domains (e.g. Octagons) with non-linear is not suitable for JavaScript. The interval abstract domain is a better choice. Since the interval analysis needs to consider `NaN`, `Int32`, infinity and others, so the interval can be divided into these four types and empty interval. Then the interval can be represented as:

$Interval = \perp_i \mid NaN \mid Normal\,(a,b) \mid OpenLeft\,(b) \mid OpenRight\,(a) \mid \top_i, \quad a,b \in Int32$ [25]

In abstract interpretation, the concretisation function γ is used to formalise. So, the abstract function can be formalised as:

$$\alpha_i\,(r) = \begin{cases} NaN & r\,is\,NaN \\ OpenRight\,(2^{31}-1) & 2^{31}-1 < r \leq +\infty \\ OpenLeft\,(-2^{31}) & -\infty \leq r < -2^{31} \quad [25] \\ Normal\,(floor\,(r)\,, ceiling\,(r)) & -2^{31} \leq r \leq 2^{31}-1 \\ \top_i & otherwise \end{cases}$$

For example, $\alpha_i\,(5, +\infty, NaN) = OpenRight\,(5) = \{5 \ldots 6 \cdots + \infty\} \cup \{NaN\}$. Now, the abstract domain *Interval* is powerful enough to do the interval analysis to determine whether the value is within the range of `Int32`. Next, we continue solving the problem that whether a variable will be never assigned a fractional value, an infinity or a `NaN`. This can be done by kind analysis. The same as *Interval*, we use concretisation function to formalise the kinds.

$$\alpha_k(r) = \begin{cases} Int32 & r\, is\, a\, Int32 \\ Float64 & r\, is\, a\, Float64[25] \\ \top_k & otherwise \end{cases}$$

For example, $\alpha_k(5.78) = Float64, \alpha_k(3) = Int32$. After applying interval analysis and kind analysis on the variables, the variation analysis is also necessary. Because it can refine the loop invariant which discovered by interval analysis and kind analysis. The aim of the analysis is to infer the increment of variables in the loop, which is similar to the interval analysis in some sense. But one of the differences is it assigns the local variables to the interval $[0, 0]$so that it can compute increments of variables.

During the execution, when the JIT compiler meets a function with dynamically typed parameters, it invokes the analyses mentioned above to infer the numerical type for variables, then compiles this function and stores it in memory. The next invocation for the same function can save time by reading them from cache directly.

3.2.1.2.2 Points-to Analysis Points-to analysis establishes which a reference variable or an object can point to which set of objects. Points-to information is the basis of optimisation of compiler. If point-to analysis can infer precise type information of a variable x, then $x.msg()$ can be executed by static calling instead of relative inefficiency dynamic dispatch. However, the design of point-to analysis could be very challenging. It has been proved that it is impossible to get complete points-to information via static analysis. Furthermore, it is also not feasible to describe and analysis the structures and features of whole programs. Therefore, the result of points-to analysis is approximate in some sense. The more accuracy the algorithm describes the structures of programs, the more accuracy points-to information will be obtained, but the more time complexity will be suffered. The accuracy of flow sensitivity, context sensitivity, object sensitivity, field sensitivity determine the overall accuracy. In principle, the insensitivity of some aspects definitely sacrifices part of accuracy of algorithm. However, the sensitivity part must increase the complexity of algorithm and decrease the efficiency of algorithm. As a result, the main problem of points-to analysis is making sure the accuracy of algorithm as well as decrease the overhead of space and time. Anderson [1] came up with a constraint-based points-to analysis algorithm for C, which is believed the most accuracy flow insensitive, context insensitive algorithm. He describes the direct point-to relationship as a map from variables to objects, and then computes indirect points-to relations by solving the transitive closure of constraint based set, to get the entire set of points-to information. This algorithm was widely used in the present points-to analysis. The points-to analysis for JavaScript [23]is based on Andersen's algorithm.

```
1    var v = start();
2    var o = new Object(); //01
3    o.x = new Object(); //02
4    o.y = new Object(); //03
5    o[v] = new Object(); //04
```

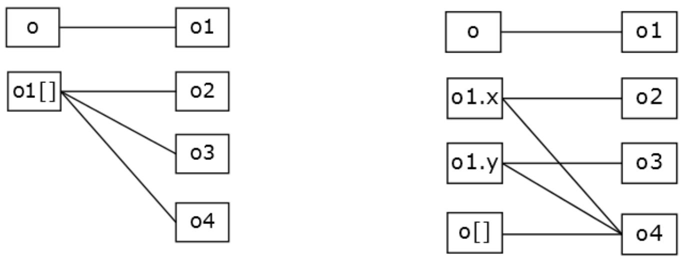

Figure 3.2.1: Example of JavaScript program and its points-to graph with or without properties

In the example above, line 2 creates a new object `o1` without properties. In lines 3 and 4, two non-existing properties `x` and `y` is created and referenced by `v`. In line 5, the elements of array `o[v]` may be `x`, `y` or a non-existing property. Therefore, the property name may be statically unknown when it is accessed. In this case, the result of traditional analysis may be inaccurate because the Anderson's algorithm regards the elements of an array as an aggregate. But in fact, the object properties in JavaScript could have very complex hierarchical data structures due to its prototype and constructors. Hence, maintaining the information for each property as the graph above shows can increase the accuracy of analysis. For example, `o1.x` and `o1.y` represent the points-to information for properties `x` and `y`, `o[v]` deals with the situation that property cannot be statically determined. This approach considers the points-to information for each property which differs from conventional Andersen's algorithm. Based on this, [23] defines set constraints for SimpleScript which is a restricted JavaScript. A set constraints is "the expression evaluates to an object in a points-to set of object including those of set expression at run-time". It can be represented as $x_e \supseteq se$, where $x_e$obtains the set variable for the points-to set. The constraint generation rules are designed for representing the semantics of the SimpleScript expression. With this points-to analysis, it can eliminate partial redundancy references by detecting whether a property points to the same value at different points in a program. However, the analysis only applies to a small first-order language, and was not validated in a large set of JavaScript programs with complicated hierarchical structures.

3.2.2 Inline Caching

3.2.2.1 Why Inline Caching

In C language, accessing a member of a structure can be done by simple instruction. From the example, the C compiler can see the definition of `struct` so that it knows the memory address of the members of `struct Person`. Therefore, the member `person->isMale` can be compiled into machine instructions from the address of `person` with the addition of offset of `isMale` in `Person`.

C code:
```
struct Person {
    int age;
    int isMale;
};
int isAMale(
    struct Person *person) {
    return person->isMale;
}
```

Assembly code:
```
isAMale:
.LFB2:
  pushq  %rbp
.LCFI0:
  movq  %rsp, %rbp
.LCFI1:
  movq  %rdi, -8(%rbp)
  movq  -8(%rbp), %rax
  movl  4(%rax), %eax
  leave
  ret
.LFE2:
  .size  isAMale, .-isAMale
```

However, because of the dynamics of JavaScript, accessing the property of objects can be not easy. No support for specifying the types by an identifier results in compiler does not know the memory address of the specific property.

```
function foo(obj) {
    return obj.p;
}
```

In order to find the property `p` of `obj`, it needs to search the property name `p` with its associated prototype, and then get its value. It is super slow to execute this kind of search. Furthermore, it is sometimes hard to determine the function to invoke at a call site at compiler-time because of polymorphic and inheritance. Therefore dynamic dispatch techniques like inline caching is a necessity at run-time.

3.2.2.2 Monomorphic Inline Caching

Inline caching records the previous steps which how to find the property to avoid performing a lookup at each time. In other words, inline caching improves the performance of object type lookups. By using a method cache which is a hash table of function address, the method lookup can be accelerated [21]. It is observed that the objects often has the same type at most call sites (where calls a function). In other words, the types of operands are likely to remain the same in the different interpretations, given a specific instruction in the code. In this case, the next function lookup can be omitted if caches the function address of the point of send. It needs to be checked whether the two addresses matches since the function of the receiver may have changed.

There are two states of a call site during the process of realising this mechanism. At first, a call site is “uninitialised”. The program run-time will perform the method address lookup if it reaches an uninitialised call site. And the result of lookup will be cached at the call site whose state will become “monomorphic” at that time. As mentioned before, if the run-time reaches the same call site, it invokes the function directly instead of performing address lookup.

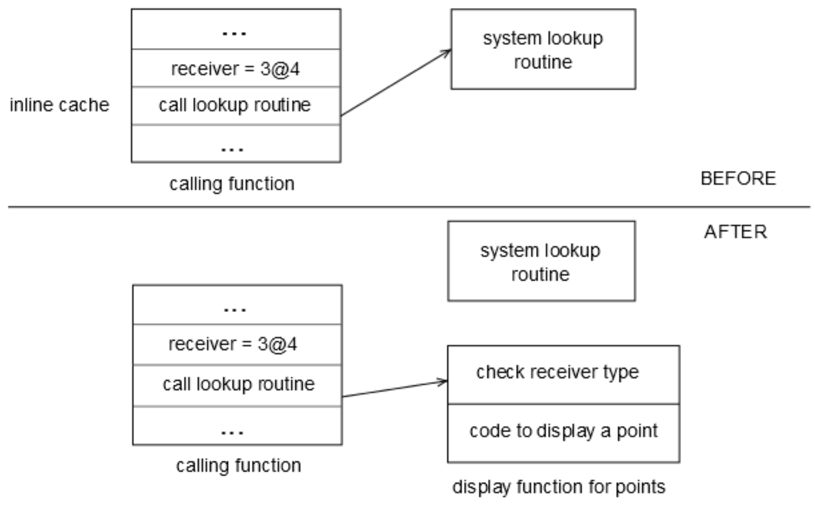

Figure 3.2.2: Example of inline caching [21]

The sends can be divided into three types according to the degree of the polymorphism: "monomorphic" (receiver has only one type in particular call site), "polymorphic" (receiver has a limited number of types), and "megamorphic" (receiver has very many types) [21]. Therefore, it makes sense that caching can be more flexible according to different situations.

Previous inline caching is efficient only the type of receiver remains same at a particular call site. In the case that receiver has several equally likely types, inline caching will not perform well because the call target needs to switchover between different methods. The fail of checking which mentioned before will incur cache misses. Inline cache misses may slow down the inline caching because it has extra overhead. In this case, successive technique called polymorphic inline caching is utilised.

3.2.2.3 Polymorphic Inline Caching

This technique is applied to sends with a few types (typically less than ten). A polymorphic inline caching (PIC) caches several lookup results instead of only caching last lookup result. A small stub routine is created for each site, which grows as more receiver types are seen.

PICs do not work well for call sites with a larger number of receiver types (megamorphic). For this situation, it is better to use the original property search algorithm again.

3.2.2.4 Implementation of Inline Caching

Algorithm 3.1 Inline Caching [40]

```
send(name, args, cache)
    receiver = args[0]
    if cache.class == receiver.class and cache.state == state
        function = cache.function
    else
        function = lookup(receiver.class, name)
        cache.state = state
        cache.class = receiver.class
        cache.function = function
    end
    function(args)
end
```

Algorithm 3.2 Polymorphic Inline Caching [40]

```
send(name, args, cache)
    receiver = args[0]
    if cache.class == receiver.class and cache.state == cache.psate.value
        function = cache.function
    else
        if cahce is polymorphic
            for entry in cache.array
                if entry.class == receiver.class and
                      entry.state == cache.pstate.value
                    function = entry.function
                    break
            end
        else
            convert cache to polymorphic
        end
        if function not found
            lookup function and store result in cache
    end
    function(args)
end
```

These are the basic algorithms to implement inline caching and polymorphic inline caching. The main problem is how to decide the receiver type and jump to the corresponding function. It is often implemented by a jump table. The jump table contains a stub to infer the type of the receiver and jump to the target function. When a particular call site encounters a different type, the jump table will be allocated. The jump table grows as number of types increases. Once this number reaches the limitation of the types, it will turn to the megamorphic case.

3.2.3 Compiler Optimisation

Between HIR and LIR, we can apply a set of optimisations. Transferring a set of instructions to faster instructions without losing the function often refers to code optimisation. In a JavaScript program, it often exists much redundancy. One source of the redundancy is the source code, programmer may re-compute for convenience. But in more situations, redundancy is the side effect of high-level programming languages themselves. When a program was compiled, each accessing to data structure will be expanded into multiple low-level arithmetics. Accessing to the same data structure often shares many common low-level arithmetics. But the programmers do not know these arithmetics so they cannot eliminate the redundancy by themselves. Hence, the techniques apply to the compilers are used to eliminate the redundancy. However, the heavy optimisations would hurt the startup performance of JavaScript engine, so a limited set of optimisation is appropriate.

3.2.3.1 Constant Propagation

Constant propagation is an important data-flow analysis method. It spots the variables with constant value in all possible executed paths, computes the result of expressions whose operands are all constant values in the compiler time, then propagates spotted constants as far as possible. It is a static optimisation technique in the compiler-time. Its implementation makes other compiler optimisations (including dead code elimination, algebraic simplifications and so on) have better performance.

Simple Constant

```
A = 1 + 2;
B = A * 3;
if(cond)
    C = 5;
else
    C = B + 5;
```

Conditional Constant

```
A = 1 + 2;
B = A * 3;
if(A > 5)
    C = 5;
else
    C = B + 5;
```

Usually the variables with constant value are from:

- Uncareful coding;
- In order to ensure the readability of the program.

Kildall [24] first raised the problem of constant propagation. He came up the simple constant algorithm to solve the problem. Simple constant algorithm detects and propagates constants with the aid of the program flow graph. Wegman [37] divides constant propagation algorithm into four types according to simple constant, conditional constant and represented graphs. These four algorithms aim at detecting simple constant, conditional constant, sparse simple constant and sparse conditional constant respectively.

Since the constant propagation problem has been shown to be undecidable in general, there is no effective method to detect all the possible constants in programs so far. Existed algorithms more focus on detecting and propagating constants with simple expressions. But for the hidden constants in the complex form, conservation estimation is often used.

3.2.3.2 Loop Optimisation

Since loop is a basic structure in many cryptographic algorithms and it is the region where has largest payoff, so it is reasonable to optimise them to get better performance.

3.2.3.2.1 Loop-invariant Code Motion Loop invariant code is the code that can be hoisted outside the loop without affecting the result of the program. If the value of a variable is the same during each iteration, it will have a significant improvement if applies loop invariant code motion.

```
for (var i = 0; i < N; i++) {
    x = y * z; a[i] = x * x;
}
```

$\rightarrow x = y*z$ and $x*x$ can be hoisted outside the loop since they will not affect the semantics of the program.

```
x = y * z;
t = x * x;
for(var i = 0; i < N; i++) {
    a[i] = t1;
}
```

Now, $x = y * z$ and $x * x$ only needs to compute once before enter the loop, instead of repeating them N times.

Loop invariant code is often hoisted outside in the middle end of the compiler. The intermediate representation (IR) in the middle end is object machine independent, while IR in the front end is object machine dependent. Though loop invariant code motion is effective in the the middle end, much loop invariant code will be introduced when IR transfers from the middle end to the back end. Hence, it is also necessary to perform the loop invariant code again in the back end of the compiler.

Suppose there is a statement $t \leftarrow a \oplus b$ in a loop, if each operand a_i and b_i satisfy the specific requirement, then definition $d : t \leftarrow a \oplus b$ is loop invariant code. It requires that a_i is constant, or all the definitions reach d of a_i are outside of the loop, or only one definition of a_i reaches d and this definition is loop invariant [29]. We can use iteration algorithm to seek the loop invariant code on the basis of these conditions. First it needs to find out the constant operand or definitions from outside of the loop, then seeks definition whose operands are all loop invariant.

However, loop invariant code motion increases the pressure of registers and increases the size of the code outside of the loop. In fact, not all the loop invariant code is worth hoisting.

3.2.3.2.2 Loop Unrolling Loop unrolling is an optimisation technique which decreases the loop cost by increasing instructions. If loop body is simple and large part of instructions is used to check conditions of loop, it can unroll the loop to linearise the object. Then the program has less iterations, the cost decreases but the size of code increases. Loop unrolling also can improve register locality and memory hierarchy locality [32].

Loop unrolling has its own drawbacks. It increases the size of the object code, since the loop body is vast in its source code form. It is generally believed that the higher the degree of loop unrolling is, the smaller overhead the loop occupies. However, the efficiency also closely relates to the operations in the loops. Not all the computation will be performed as expected, since it depends on the functional unit of CPU. An objective law is loop unrolling will perform better if the array is long, and the performance improves with the increase of the degree of unrolling. On one hand, it needs to consider the length of array, and a reasonable degree of unrolling. On the other hand, the overhead of space and time need to be taken into account, try to get the best profit under the premise that space consumption will not sharply increase.

Initial code:
```
var i, j;
for(i = 0;i < 4;i++) {
    for(j = 0; j < 4; j++) {
        a[i][j] ^= rk[i][j];
    }
}
```

Modified code:
```
var i;
for(i = 0;i < 4;i++) {
    a[i][0] ^= rk[i][0];
    a[i][1] ^= rk[i][1];
    a[i][2] ^= rk[i][2];
    a[i][3] ^= rk[i][3];
}
```

For local use, loop unrolling is often adopted. But for JavaScript in the client side, code size is an important factor to be considered. So in sjcj, the developers determine whether use loop unrolling or not for AES depends on different situations: unrolling short loops, unrolling the main round functions loops and leave other loops rolled.

3.2.4 Other Tricks

3.2.4.1 Escape Analysis

Escape analysis is the cutting-edge optimisation technology for now. It is not a mean of optimise the code directly, but is an analysis method to provide the basis for other optimisation methods. The foundation of escape analysis is to analyse the dynamic scope of objects. An object was defined in a function may referrenced by another function. For example, it may be passed to external functions as a parameter. This behaviour is called method escape. If we can prove an object will not escape outside the function, which means the other functions cannot access to this function by any means, then efficient optimisation like scalar replacement can be applied.

- Scalar Replacement. Scalar is a variable that can no longer be broken down into smaller variable to represent, which can only store one value at a time. Oppositely, if a variable can still be broken down, it is called aggregate. The object in JavaScript is a typical aggregate. On the basis of program accessing, disassembling a JavaScript object and translating its used member variables into primitive data types is referred as scalar replacement. If it is proved that an object will not be accessed by external functions via escape analysis, and this object can be decomposed. It may create the member variables which used by this function instead of creating this object when the program is executed. Decomposing the object not only enables allocate member variables of objects in the stack, but also eases the further optimisations.

Escape analysis is realised in JVM of Sun JDK 1.6, but it is not mature enough; there is still huge space for improvement. Because escape analysis cannot guarantee the performance gain definite higher than the consumption. If we want to determine whether an object will escape accurately, complex sensitive data flow analysis is needed to analyse the impact of all branches in programs on the object. This process has a high time consuming. Therefore, the VM has to adopt lower accurate, but lower time consuming algorithm to execute escape analysis. In the result of testing, the programs often have a good performance in the benchmark after executing escape analysis. However, in practical applications especially in large-scale applications, the effects of escape analysis may be instable or it cannot determine the un-escape objects effectively because of time consumption which leads poor performance (i.e. gain of JIT compiler). Though escape analysis has not fully matured, it is an important direction for JIT compiler optimisation. For the later VM, escape analysis will definite become the foundation of a series effective optimisation technologies.

3.2.4.2 Garbage Collection (GC)

During the execution of JavaScript programs, they create objects, strings, arrays and so on. These actions take memory, so it needs to apply the memory from operating systems. When the objects will no longer be used, the JavaScript engine must be able to free up the memory automatically to prevent crashing. The module discovers these unused memory and frees up them is called garbage collector. The garbage collection typically used in JavaScript engine is a non-generational collector based on mark-sweep algorithm.

3.2.4.2.1 Mark-Sweep algorithm Mark-Sweep algorithm is the basic garbage collection algorithm and the first one solved recursive reference. As its name suggests, the process to do garbage collection can be divided into two phases: "mark" and "sweep". The first step is to mark all the objects which need to be collected. Then it sweeps all the marked objects. The objects that are not reachable will not be recycled immediately until the entire memory space is run out. At that time, JavaScript programs will be paused and the garbage collection mechanism will be activated by the memory management system to recycle the memory block that will no longer be referred. Only after all the memory blocks are recycled, the program gets to resume. Furthermore, a large number of discontinuous memory fragments will be generated after sweeping. Too many memory fragments may lead that it cannot find enough large continuous memory blocks to allocate large objects in the following running process, and it has to trigger another garbage collection in advance. The time wasted in each program pause may be more than 100ms, which has a negative impact on general performance.

3.2.4.2.2 Generational Collection Generational garbage collector is based on that most objects have very short survival time. The research shows that the vast majority of objects (more than 90%) are transient objects which have little time to live. Generational GC separates the memory into blocks according to the different life cycles of objects, namely nursery generation and tenured generation. In the nursery area, the amount of objects died and only few of objects survived. Thus the objects allocated in the nursery area are frequently collected. If the nursery objects are still alive after several collections, then it is referred as tenured generation and copied to the tenured area. It rarely collects garbage in the tenured area. For the objects in the tenured area, mark-sweep algorithm can be used. Generational garbage collection adopts the most suitable method to deal with different objects, which enhances the effective and reduces program pause time.

3.2.4.2.3 Incremental GC The other approach to reduce the pause is to split a long pause into a number of short pauses. In this way, users have no awareness of the

the program pauses. Incremental GC detects and sweeps a small part of unreachable objects at each time instead of seeking and recycling all the unreachable objects at a time. Since only a part of memory executes garbage collection, each time lasts shorter time. Generational GC has its difficulty. When a collector tries to traverse to get a reachable graph of data structure, the graph may be changed by running program without being seen by the collector. Hence the strategy to do incremental garbage collection must contain a method to trace the changes of the reachable graph. In other words, implementing incremental GC needs to ensure the continuity of collector if the program modifier the objects when the collecting process has not yet completed.

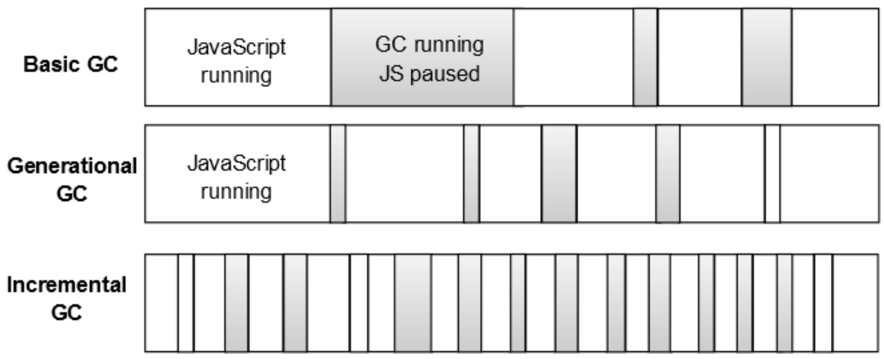

Figure 3.2.3: Comparison of garbage collectors

Chapter 4

Conclusion

4.1 Critical Evaluation

4.1.1 Programming Guideline

After looking at the reasons that make JavaScript slow, we can speedup the JavaScript programs first from programming.

The tests executed below are in the environment as follows:

- CPU: Intel Core i5 M480 2.67GHz
- RAM: 4.00GB
- Operating system: 64-bit win7 professional
- Browsers:
  - Chrome 13.0
  - Firefox 5.0
  - Safari 5.0.5
  - Opera 11.51

4.1.1.1 Global Variables

It is reasonable to use global variables in programs for convenience. But global variables will make JavaScript engine run slowly because of several reasons. The first one is it needs more time to search the global variables than local variables when the variables are referenced due to the difference of scope. Secondly, a global variable has a longer life. The local variables will be destroyed by garbage collector if it will no longer be used while the global variables keep alive along with the program.

The improvement can be done using local variables in function. Local function variables have better performance is reasonable. Because few local function variables exist in most functions, and references to local function variables can be converted to efficient executable instructions by the JIT compiler. Figure 4.1.1 shows a test of iterations with global variables or local variables. The results present the version with local variables performs observable faster. In the mainstream browsers, the version with local variables is about 40% faster on average faster than the version with global variables.

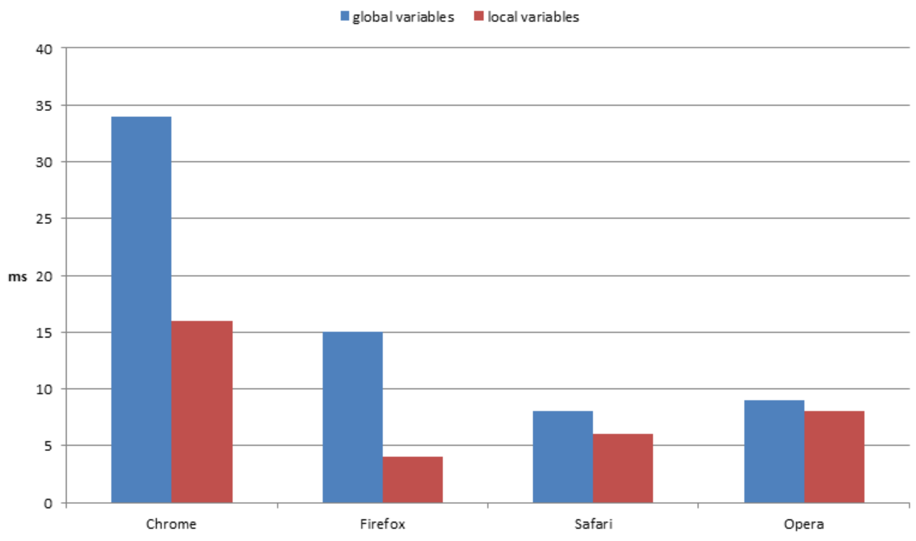

Figure 4.1.1: The comparison of using global variables and local variables

4.1.1.2 Loops

There are various styles of iteration in JavaScript: `for`, `for-in`, `do-while` and `while`. The test shows there is no difference between `for`, `do-while` and `while` (except in Opera). But `for-in` loop is extraordinary slower than the others in all the tested browsers. The `for-in` loop needs the JavaScript engine to create a list of all the properties and check the redundancy, and then call a C++ function to search through the property list. But the normal `for` loop is the target of JIT compiler. The hot code of loop is compiled into native code which is much faster.

```
// for loop
for (var i=0; i < array.length; i++) {
    sum += array[i];
}
// do-while loop
var j=0;
do {
    sum += array[j++];
} while (j < array.length)
// while loop
var k = 0;
while (k < array.length) {
    sum += array[k++];
}
// for-in loop
for (var i in array) {
    sum += array[i];
}
```

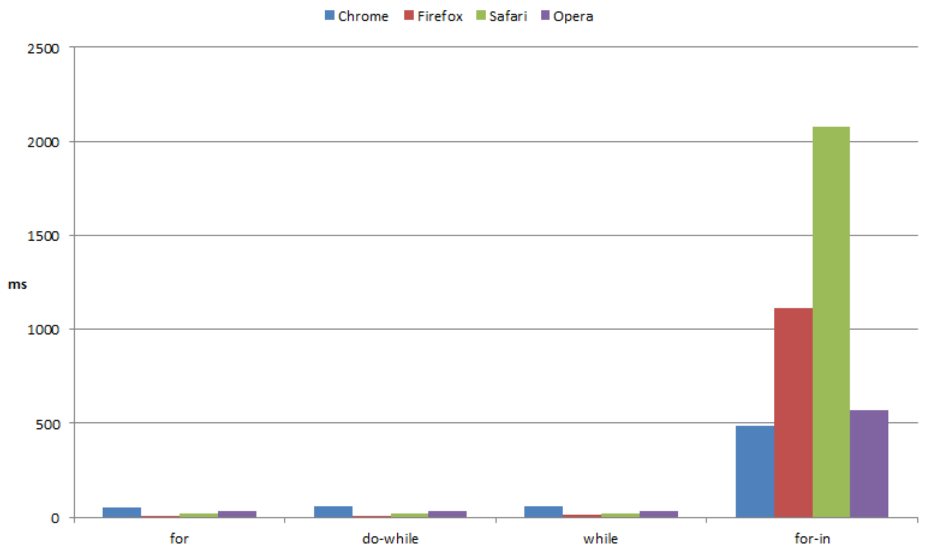

Figure 4.1.2: The comparison of various iteration styles

4.1.1.3 `eval()` and `with()`

`eval()` and `with()` functions are rarely used, but it will kill performance if used, especially in loops. In JavaScript, `eval()` function is a combination of a statement executor and an expression calculator. It executes the argument it receives and returns a result. Each time `eval()` is called, the JavaScript engine converts the source code to the executable code. If the `eval()` is in a loop, executing hundred times would substantially hurt the performance. Since `eval()` is dynamically executed, the engine cannot know the contents it is passed. Thus the `eval()` function exactly among the cases that can only be interpreted. It means the JIT compiler cannot optimise the surrounding context of the call to `eval()`, the interpreter needs to interpret the code at run-time. At the most time, the use of `eval()` is unnecessary. It can be replaced by a function call which can be optimised via inline caching.

Figure 4.1.3 shows the comparison of the iterative computations with `eval()` and without `eval()` in various Web browsers. It presents that the time spent on them respectively is not at a magnitude. Especially in Firefox, the browser sometimes crashed when the iterations are large.

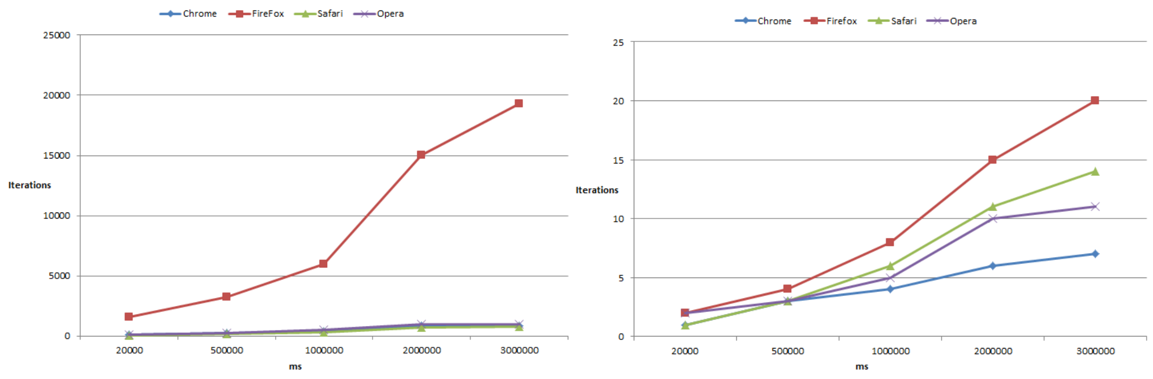

Figure 4.1.3: The comparison of iterative computations with `eval()` or without `eval()` function in various browsers

`with()` function in JavaScript is intended to provide a way to reduce object references when it needs to access objects recursively. The same as `eval()`, `with()` statement is also not often used. But once used, it is harmful to the performance. Because `with()` statement expands the scope so that the JavaScript needs to search through it each time a variable is referenced. In this case, the JIT compiler in the engine cannot determine the content of the scope at compiler time. Then the JIT compiler cannot optimise it while it can if the scope is created by normal functions. The figure below shows the time used to access objects iteratively with `with()` statement or without `with()` statement. Similar to `eval()`, the difference is huge no matter in which browser.

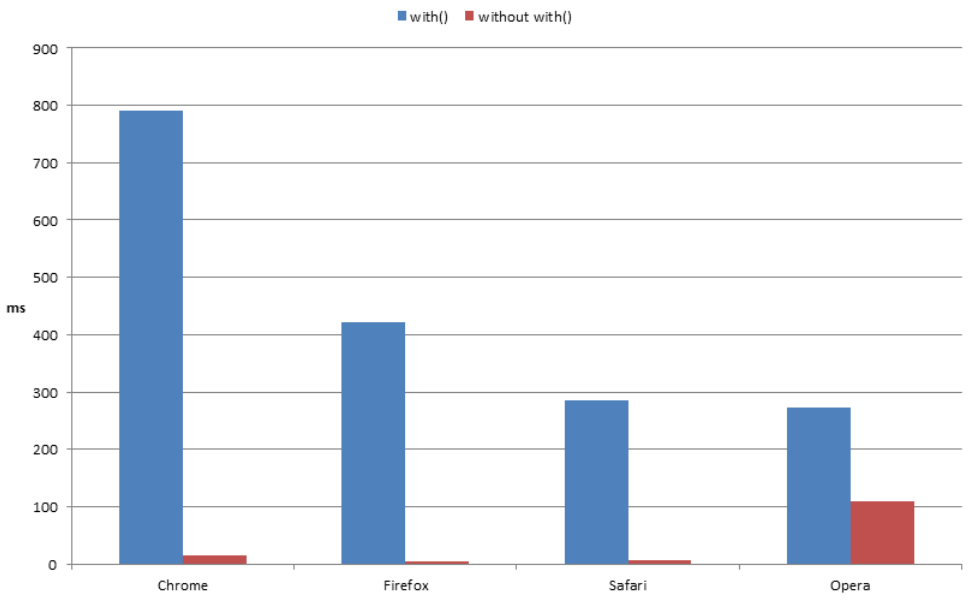

Figure 4.1.4: The comparison of `with()` and without `with()`

Overall, it is better not to use `with()` and `eval()` function, especially in loops which would be executed repeatedly. In the situations that they must be used, separating them from the other code and keeping them as little as possible.

4.1.1.4 Property Depth

Though property access become faster because of inline caching which we described before. But for the first time a property is accessed, the property address has not been cached yet. It needs to search the property chain even up to several levels. The test results also match the theoretical explanation. The figure 4.1.5 shows the deeper a property is, the longer it takes to search. For example, `obj.p1.p2` takes longer time to retrieve than `obj.p1`.

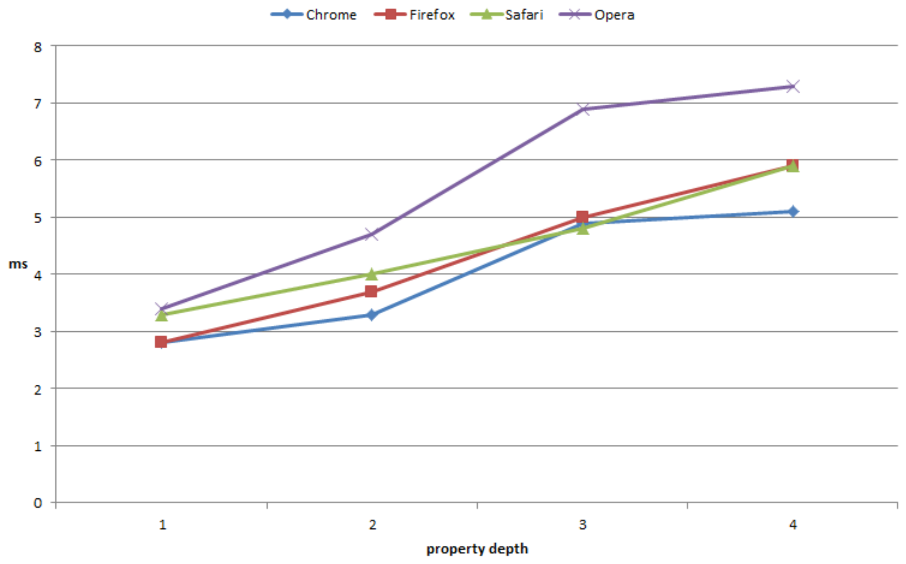

Figure 4.1.5: The impact of property depth

4.1.1.5 Code Size

Since JavaScript cryptographic libraries are called on the client side when it needs client-side encryption, so minify the code size would definitely speedup code loading. Removing comments and white-spaces is the first idea come into mind. Replacing local variables with smallest possible variable names also can reduce the code size. We use a tool called YUI compressor which developed by Yahoo! to compress the code of JavaScript cryptographic libraries. The results shows that it saves about 50% compared to original code.

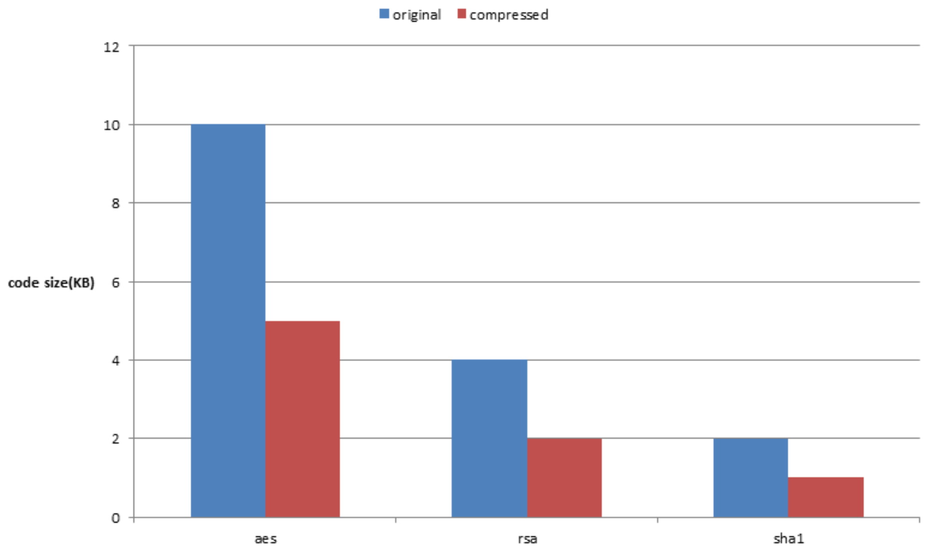

Figure 4.1.6: The comparison of code size

The aim of these tips is JIT compiles as much code as possible instead of slowly interpreted.

To conclude, the tips for programmers writing fast JavaScript programs can be summarised as:

- Use local function variables instead of global variables.
- Avoid using `for-in` loop.
- Avoid using `eval()` and `with()` function.
- Compress the code.
- Minimise property depth.
- The last one and also the most important one is keeping the code type-stable.

4.1.2 Architecture

The investigation presents that the performance of JavaScript programs largely relies on the type analysis and inference. More complex and accurate type analyses are definitely of great benefit to improve the performance. However, the overhead increases as the complexity of analysis increases. In the environment like Web browser, it needs a short startup time. If the result of type analysis has a high accuracy, it inevitably can generate high quality code. Synthesising different type analysis techniques properly can balance the complexity and performance. Based on the investigation, we design an architecture for efficient JavaScript engine.

From the macroscopic view, the JavaScript engine has two components: the front end and the back end. The front end takes charges of lexical analysis and syntax analysis. The outcome of the front end is bytecode or abstract syntax tree. In our design, we choose bytecode as the output of the front end. Then there are three modes to execute the bytecode: interpretation, compilation by basic JIT compiler and compilation by optimising JIT compiler. The code that is not worth compiling or cannot be compiled directly interpreted by interpreter. The hot code is compiled by optimising JIT compiler. TraceMonkey in Firefox is a kind of optimising JIT compiler. The code given up by optimising JIT compiler because of performance backs to the basic JIT compiler. The crucial part and also the target of this thesis is optimising JIT compiler. It is worth noticing that garbage collector in our design is generational GC which has a much shorter paused time than garbage collector based on mark-sweep algorithm.

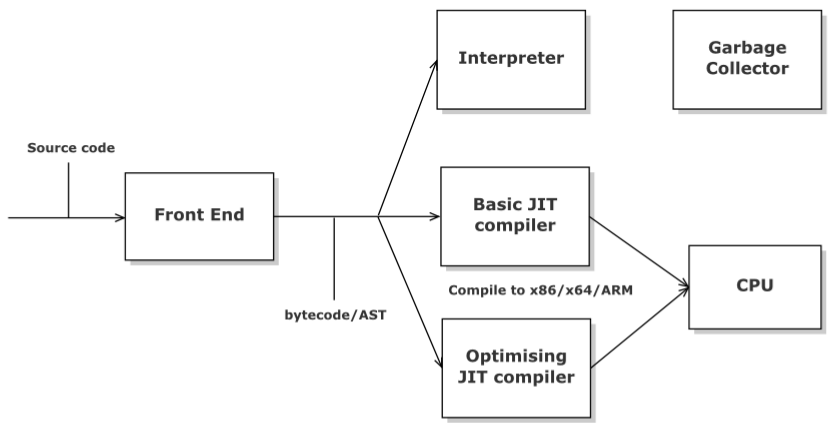

Figure 4.1.7: Architecture of JavaScript engine

The optimising JIT compiler in our design more focuses on local optimisations and gives up many global optimisations which are relative expensive. The whole process can be divided into three phases. In the first phase, compiler converts the bytecode to the HIR. Before that, compiler finishes parts of optimisations based on bytecode. For example, method inlining, constant propagation and so forth will be applied before HIR. In the second phase, type inference integrates with type specialisation is used to generated type specialised LIR. The final phase uses linear scan register allocation algorithm to allocate register on LIR, uses various inline caching to boost property access, and also applies peephole optimisation (which replaces original small sets of instructions with faster sets of instructions) on LIR. Finally the platform-dependent native code is generated. Our design supports to enable or disable the specific sets of optimisations depends on the specific demands. Because a JIT compiler enables heavy optimisations can generate higher quality native code which decreases the execution time. But the overhead of compilation increases. The JIT compiler disables heavy optimisations is opposite. Therefore, whether enable the optimisation and enable how many optimisations depend on the specific environment.

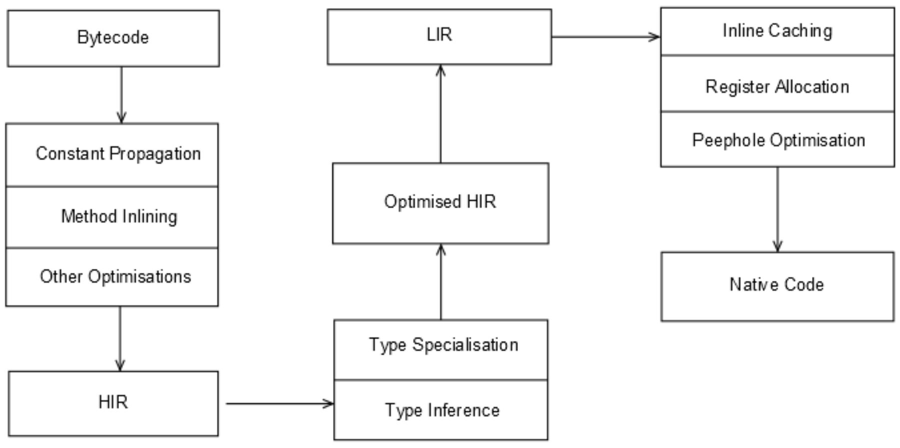

Figure 4.1.8: Architecture of optimising JIT compiler

4.2 Conclusion and Future Work

In short, the thesis is an attempt to make full-scale study on JavaScript behaviours and its engines. Firstly, we took experiments on JavaScript benchmark, analysed the reasons that make JavaScript programs especially cryptographic programs not so fast. To address the problem, we performed a full-scale type analysis on JavaScript both on its compiler-time and run-time. We investigated how inline caching and traditional compiler optimisations applied to JavaScript JIT compiler. Based on the investigation, we proposed a programming guideline for programmers writing more efficient JavaScript programs, and designed an architecture for modern high-performance JavaScript engine, especially its optimising JIT compiler.

The investigation of JavaScript shows that it may need a large amount of dynamic type checking during the execution. The type checking can be eliminated by run-time type specialisation and static type inference algorithm. If it can apply hardware-supported speculation algorithm to the platform, it likely could get a morse efficient result. Therefore, it is worth researching the features of hardware platform for future work. It can improve the performance of JavaScript program by integrating the features of hardware and the existed machine independent optimisations.

Bibliography

[1] L.O. Andersen. Program analysis and specialization for the c programming language. Technical report, Technical Report 94-19, University of Copenhagen, 1994.

[2] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[3] D. Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *Blackhat, USA*, 2010.

[4] C. Blizzard. an overview of TraceMonkey. 2009.

[5] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In *Advances in Cryptology CRYPTO 93*, pages 175–186. Springer, 1994.

[6] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. *Donald Bren School of Information and Computer Science, University of California, Irvine, Irvine, CA Report*, 2007.

[7] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[8] J. Daemen and V. Rijmen. *The design of Rijndael: AES–the advanced encryption standard*. Springer Verlag, 2002.

[9] S. Dusse and B. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptologyï¿œï¿œEUROCRYPTï¿œï¿œ90*, pages 230–244. Springer, 2006.

[10] M. Dworkin. Recommendation for Block Cipher Modes of Operation. Methods and Techniques. Technical report, NATIONAL INST OF STANDARDS AND TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV, 2001.

[11] Mozilla Foundation. JavaScript:TraceMonkey.

[12] Mozilla Foundation. Property cache.

[13] Mozilla Foundation. Rhino: JavaScript for Java.

[14] Mozilla Foundation. SpiderMonkey Internals.

[15] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M.R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM SIGPLAN Notices*, volume 44, pages 465–478. ACM, 2009.

[16] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. In *Technical Report*. Citeseer, 2006.

[17] J. Ha, M.R. Haghighat, S. Cong, and K.S. McKinley. A concurrent trace-based just-in-time compiler for JavaScript. *Dept. of Computer Sciences, The University of Texas at Austin, Tr-09-06*, 2009.

[18] S. Han and X. Li. Efficient Methods to Implement S-BOX and INV-S-BOX in AES Algorithm. *Weidianzixue(Microelectronics)*, 40(1):103–107, 2010.

[19] H. Hayashizaki, P. Wu, H. Inoue, M.J. Serrano, and T. Nakatani. Improving the performance of trace-based systems by false loop filtering. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 405–418. ACM, 2011.

[20] D. Hiniker, K. Hazelwood, and M.D. Smith. Improving region selection in dynamic optimization systems. 2005.

[21] U. Holzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[22] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the International Symposium on Code Generation and Optimization (to be published), CGO*, volume 11, 2011.

[23] D. Jang and K.M. Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1930–1937. ACM, 2009.

[24] G.A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[25] F. Logozzo and H. Venter. Rata: Rapid atomic type analysis by abstract interpretation–application to javascript optimization. In *Compiler Construction*, pages 66–83. Springer, 2010.

[26] D. Mandelin. JagerMonkey and Nitro Components. 2010.

[27] G.L. Miller. Riemann's hypothesis and tests for primality*. *Journal of computer and system sciences*, 13(3):300–317, 1976.

[28] P.L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[29] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.

[30] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM SIGPLAN Notices*, volume 45, pages 1–12. ACM, 2010.

[31] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[32] V. Sarkar. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*, pages 153–166. ACM, 2000.

[33] D. Shapiro. RSA in JavaScript. 2005.

[34] J.E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes.* Morgan Kaufmann Pub, 2005.

[35] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *2009 Annual Computer Security Applications Conference*, pages 373–381. IEEE, 2009.

[36] P.C. Van Oorschot, A.J. Menezes, and S.A. Vanstone. *Handbook of applied cryptography.* Crc Press, 1996.

[37] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[38] J. Wolkerstorfer, E. Oswald, and M. Lamberger. An ASIC implementation of the AES SBoxes. *Topics in Cryptology CT-RSA 2002*, pages 29–52, 2002.

[39] N.C. Zakas. *High Performance JavaScript.* Yahoo Press, 2010.

[40] S.S. Zakirov, S. Chiba, and E. Shibayama. Optimizing dynamic dispatch with fine-grained state tracking. In *Proceedings of the 6th symposium on Dynamic languages*, pages 15–26. ACM, 2010.