分类号：TP391　　学校代码：10697

密　级：公开　　学　　号：201620959

西北大学
Northwest University

硕 士 学 位 论 文

MASTER' S DISSERTATION

基于 WebAssembly 的 JavaScript 性能优化方案研究与实现

学科名称：计算机应用技术

作　　者：薛　超　　指导老师：汤战勇　副教授

西北大学学位评定委员会

二〇一九年

Research and Implementation of JavaScript Performance Optimization Based on WebAssembly

A thesis submitted to

Northwest University

in partial fulfillment of the requirements

for the degree of Master

in Computer Applications Technology

By

Xue Chao

Supervisor: Tang Zhanyong Associate Professor

2019

西北大学学位论文知识产权声明书

本人完全了解西北大学关于收集、保存、使用学位论文的规定。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。本人授权西北大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时授权中国科学技术信息研究所等机构将本学位论文收录到《中国学位论文全文数据库》或其它相关数据库。

保密论文待解密后适用本声明。

学位论文作者签名：薛超　　指导教师签名：汤成勇

2019 年 6 月 5 日　　2019 年 6 月 5 日

西北大学学位论文独创性声明

本人声明：所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，本论文不包含其他人已经发表或撰写过的研究成果，也不包含为获得西北大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：薛超

2019 年 6 月 5 日

摘要

随着互联网技术的发展，Web 程序的应用也日趋广泛，JavaScript 所承担的作用也不只是设计之初那样处理简单的 DOM 交互，更多的是需要为图形处理，物理引擎和虚拟现实等计算密集型操作提供支持。然而 JavaScript 以易用性为主的设计理念导致其性能上无法高效的应对繁重的处理需求，特别是在计算资源资源有限的平台。因此如何能够有效的提升 JavaScript 代码的执行效率，这对于网络应用程序的发展至关重要。

以往采用的在 Web 端利用插件实现繁杂密集运算功能的方式由于安全性和代码移植限制等问题逐渐被舍弃。在 Web 端引入即时编译的技术使得 JavaScript 的执行性能有了大幅提升，但是优化及退优化操作和额外的内存开销也使其无法高效的应付功能繁杂的应用，在配置资源有限的移动端平台更为显著。为了解决以上方法存在的缺陷，使得 Web 应用程序拥有更高效的执行效率。我们利用最新的前端字节码技术，实现了一种基于 WebAssembly 的 JavaScript 代码性能优化方案。一方面利用 WebAssembly 对与性能相关的计算密集型代码段进行优化；另一方面利用代码段合并方法减少数据交互过程所带来的的性能开销。本文针对以下几个方面进行研究：

（1）对 JavaScript 性能问题进行深入研究，分析执行效率低下的原因。并且对当前 JavaScript 代码优化方案进行分析，包括编译技术，变量类型模拟和部分求解器等多种主流的优化方案。详细分析了这些优化方案的基本原理，并且结合产生 JavaScript 性能缺陷的原因分析优化方案的不足。

（2）分析利用 WebAssembly 对 JavaScript 进行性能优化的可行性。通过分析 WebAssembly 的适用范围和设计特点，提出了一个 JavaScript 代码性能优化方案，讨论了该方案的实现原理。详细说明了代码段转换和代码段合并的设计方法。

（3）本文设计并实现原型优化系统 JSOPW（JavaScript optimization by WebAssembly）。选择多个实际 Web 应用程序作为测试用例，同时和现有技术进行对比，收集测试用例在优化前后的时空开销。并且在多浏览器平台上针对优化方案的通用性进行了验证，通过对比评估实验验证了优化方案的有效性。

关键词：JavaScript 性能优化，代码段合并规则，代码转换，WebAssembly

ABSTRACT

With the development of Internet technology, the application of Web applications is becoming more and more extensive, so the role of JavaScript is not to deal with simple DOM interactions as at the beginning of design, but also is needed to support computationally intensive operations such as graphics processing, physics engines and virtual reality. However, JavaScript's design philosophy based on ease of use has made it impossible to efficiently cope with heavy processing demands, especially on platforms with limited resources. How to effectively improve the efficiency of JavaScript code execution is critical to the development of web applications.

Due to the increasing complexity of Web applications, the use of plug-ins in the past has implemented complex and intensive computing functions on the Web side, but the usage rate has gradually declined due to security and performance issues. Although the use of just-in-time compilation technology on the Web has greatly improved the performance of JavaScript, optimization and de-optimization and extra memory overhead make it impossible to cope with complex applications, and the mobile platform with limited configuration resources is more to be significant. In order to solve the shortcomings of the above methods, the web application has more efficient execution performance. We use the latest WebAssembly technology to propose a JavaScript code performance optimization solution. On the one hand, WebAssembly is used to optimize performance-related computationally intensive code segments; on the other hand, code segment merge methods are used to reduce the performance overhead of data interaction. This article studies the following aspects:

(1) Investigate JavaScript performance issues and analyze the causes of performance issues. And analyze the current JavaScript code optimization scheme, including Compiler technology, variable type simulation and partial solver and other major optimization schemes. Detailed analysis of the basic principles of these optimization program, and combined with the cause of JavaScript performance defects to analyze the shortcomings of the optimization scheme.

(2) Discuss the feasibility of using WebAssembly to optimize the performance of JavaScript. By analyzing the applicable scope and design features of WebAssembly, describe the design ideas and implementation principles of the JavaScript code performance optimization solution. Detailed description of the design of the code segment conversion module and code segment merge rules.

(3) For the method proposed in this paper, Design and implement the prototype optimization system JSOPW (JavaScript optimization by WebAssembly) , introduce the main modules of the system, and give the key algorithm design description in each module. Select multiple real-world web applications as test cases, and compare them with the existing technology to collect the space-time overhead of the test cases before and after optimization. And for the versatility of multi-browser platform verification optimization scheme, the effectiveness of the optimization scheme is verified by comparative evaluation experiments.

Keywords: JavaScript performance optimization, code segment merge rules, transcoding, WebAssembly

目录

第一章 引言

1.1 研究背景和意义

JavaScript 设计之初被作为实现 HTML DOM 交互的小型脚本语言，其典型应用之一就是验证用户表单输入等。但是随着互联网的发展，Web 应用程序的功能日趋丰富，并且应用平台也逐渐多元化。由于 JavaScript 在 Web 应用开发上的绝对垄断地位，它不得不为基于计算密集型操作的应用程序提供服务，例如图像处理，虚拟现实和游戏引擎等。虽然这些年间 JavaScript 也在不断发展，但是仍然很难应对繁杂庞大的计算量，特别是在计算资源有限的平台上。

JavaScript 目前已经成为最流行的设计语言之一，它不仅被广泛应用于客户端 Web 应用程序，而且还用于移动端应用程序和桌面应用程序。目前功能齐全的 JavaScript 框架即使在传统页面上也能十分方便地实现炫目的功能，但是在计算资源有限的移动端执行这些功能需要比桌面更多的运行时间。图 1 展示了 2018 年可用硬件解析运行 JavaScript 应用程序所需的时间。由图中的描述可以看出在高配置的设备中 JavaScript 脚本具有更高效的执行性能。但是随着设备配置的下降 JavaScript 的运行时间逐步增长，特别是以 Alcatel 1X 为代表的 100 美元以下的低配置移动设备。经过统计，骁龙 617 作为一种最典型配置，其脚本运行时间是苹果 A11 的 6.36 倍，而最慢的安卓 Go 则达到了 21.4 倍。在 Web 应用程序功能日趋复杂的今天，这样巨大的性能差异会严重影响用户的使用体验。所以提高 JavaScript 的执行性能以适应不同配置的平台就显得十分重要。

然而，JavaScript 的性能缺陷是语言本身设计固有的，因为其最初的设计目的在于易用性而不是性能。但是随着 Web 应用程序处理过程的复杂化，性能不足的问题便显现出来。为解决 JavaScript 执行性能的问题，许多 JavaScript 引擎引入了即时编译技术（Just-in-time compilation），它是将部分高频率 JavaScript 代码编译成机器码，避免了对代码的重复解释执行，进而提升程序的执行性能。但是执行过程中优化及去优化操作的性能开销和对各种记录信息的内存开销使得 JavaScript 代码优化仍然存在很大的提升空间。由于 JavaScript 无类型的特性，导致其在复杂的类型推导上面损耗大量的执行性能。于是出现将静态类型语言编译生成 JavaScript 的优化策略，通过在 JavaScript 中模拟静态数据类型来实现性能优化，但是由于其本身仍然是 JavaScript，

第二章 JavaScript 优化方法及 WebAssembly 优化可行性分析

本章的主要内容是详细介绍目前工业界针对 JavaScript 代码性能问题所提出的解决方案，并且从原理上分析 JavaScript 性能缺陷和目前优化方案的不足。然后分析 WebAssembly 相比 JavaScript 在处理计算密集类型代码时的优势。最后研究利用 WebAssembly 进行 JavaScript 性能优化方案的可行性。

2.1 JavaScript 性能优化

现阶段 Web 平台的功能日益广泛，并且越来越受到各种应用程序的欢迎。而 JavaScript 是目前 Web 上唯一支持的高级语言，其应用场景也日趋多元化。尽管 JavaScript 引擎有了巨大并且持续的改进，但是 JavaScript 的性能对于密码学，图形处理或者物理引擎等计算密集型的应用来说仍然不足[33]。造成性能缺陷的并不是因为运行引擎设计不充分，而是因为 JavaScript 语言本身的缺陷[34]。JavaScript 作为一种动态类型语言，只有在运行状态时才会做类型检查，这就意味着编译器不能轻易的将操作转换成针对某种特定类型的操作指令。如果没有确切的类型信息，编译器必须发出较慢的通用机器代码来处理各种数据类型的组合，而这就带来了不可避免的性能损失。并且 JavaScript 解释执行的过程需要将纯文本代码解析生成相应的抽象语法树，然后通过遍历抽象语法树才能获取相应的机器码，进而得到编译完成的 Web 程序，这个过程也产生了性能开销。针对 JavaScript 的性能缺陷，目前也提出多种解决方案，本节主要分析目前常用的优化方案及其局限性。

2.1.1 即时编译技术

JavaScript 作为一种解释性语言，执行过程中需要对代码进行逐行的解释执行，但是每执行一次就需要解释执行一遍，从而导致运行效率低下[35]。为了提高 JavaScript 的执行效率，大部分的 JavaScript 引擎都采用了即时编译技术，直接将部分 JavaScript 代码编译成本地的字节码[36]。编译为字节码的优势在于执行时不需要进行重复的编译，并且可以在编译的过程中对代码进行优化，提高代码的执行效率。JavaScript 动态类型的开销主要来自运行时类型解析和运行时属性查找。目前开发者已经研究出多种技术对动态类型推导过程进行优化，这些技术主要包括类型推断[37]，类型特化和内联缓存[38]。类型推断基于程序分析或从语言汇总的类型系统来识别变量类型。类型特

化试图根据运行时类型分析推测变量类型。内联缓存记录调用站点上先前属性查找的结果，假设对象类型不经常更改。这些技术也应用于即时编译技术中，以获得更高效的执行效率。

不同的浏览器对即使编译存在不同的实现方法，但是基本都采取了相同的思想。主要是在 JavaScript 引擎汇中添加监视器模块，对程序运行过程中代码的运行次数和代码段所使用的数据类型进行记录。监视器会针对代码的执行频率对代码段进行标记，如果代码段执行频率较高就称该代码段为暖代码，执行频率非常高就称为热代码。并且针对不同频率的代码会利用不同的编译器进行处理。基准编译器会对暖代码进行编译，并且将编译后的结果保存。如果在程序之后的运行过程中发现了相同的代码段，JavaScript 引擎会直接调用编译后的结果，而不用重复性的解释执行[39,40]。优化编译器会对热代码段进行优化编译，因为代码段运行频率非常高，那么有效的优化方案就会使得程序的执行效率更为高效。为了使得热代码段拥有高效的运行效率，优化编译器会对代码段进行一些假设，并且按照假设进行优化来生成更为高效的机器码。在执行之前对获取的信息和假设信息进行比较判断，如果为真则对直接调用优化后的代码，如果不成立则进行退优化，通过抽象语法树编译成一般形式的机器码。JavaScript 作为动态类型的解释型语言[41,42]，它的数据类型会在运行时进行确认，这也导致对数据类型进行假设时结果不稳定。优化编译一方面会提升程序的执行效果，但是 JavaScript 中类型假设错误等导致退优化同样会导致性能开销。综上所述，虽然即时编译可以带来性能改进，但是优化及退优化，监视器记录等性能开销也给运行时带来了开销，这就使得 JavaScript 性能难以预测。

2.1.2 Prepack

Prepack 是 Facebook 设计实现的一款 JavaScript 优化工具，它可以在编译时完成原本在运行时完成的操作。Prepack 利用完全等效的精简代码替换原 JavaScript 脚本中的待优化代码段。这可以减少程序运行中大多数的计算和对象分配性能开销。

Prepack 是在抽象语法树级别运行的，并且利用具体执行的方式对原代码进行预编译。Prepack 的核心是一个 JavaScript 解释器，解释器执行的过程中可以跟踪程序的执行过程，并且对部分代码段的结果进行优化。Prepack 还实现了一套符号执行引擎，这一方面使得程序在处理分支结构时可以探索所有的可能性，提高代码的覆盖程度，为代码段优化可供可靠的依据。并且因为符号执行的运用可以追踪程序的部分抽象值和类型域信息。当程序执行结束，Prepack 会获取到程序最终的堆栈信息，并且

排列这些堆栈数据生成更为直观的 JavaScript 代码段，创建并且初始化堆栈中可以访问到的所有对象。堆中的数据很可能是一些代码段的运算结果，Prepack 将会利用这些值替换程序中原本的运算多项式，进而达到优化的目的。

Prepack 的应用场景存在严格的限制，我们以表 1 的两个测试用例说明，测试用例 1 和测试用例 2 同样实现了斐波那契数列的计算功能，但是可以看出测试用例 2 中直接对函数进行调用，所以在优化后两组测试用例结果完全不同。测试用例 1 代码仅格式存在变化，但是计算逻辑并没有改变。测试用例 2 则直接利用运算结果替换原本复杂的运算过程。从例子可以看出，Prepack 的优化只适用于可以获得局部处理结果的脚本，而对于无法通过预处理进行结果替换的代码并没有有效的优化。

表 1　Prepack 优化用例

	测试用例 1	测试用例 2

优化前	function fibonacci(x) {
	  return x <= 1 ? x : fibonacci(x - 1) + fibonacci(x - 2);
	}
		(function () {
		  function fibonacci(x) {
		  return x <= 1 ? x : fibonacci(x - 1) + fibonacci(x - 2);
		  }
		  global.x = fibonacci(10);
		})();

优化后	var fibonacci;
	(function () {
	  var _$0 = this;
	  var _1 = function (x) {
	  return x <= 1 ? x : fibonacci(x - 1) + fibonacci(x - 2);
	  };
	  _$0.fibonacci = _1;
	}).call(this);
		(function () {
		  var _$0 = this;
		  _$0.x = 55;
		}).call(this);

对于 Prepack 的优化策略，本质上是对程序可运行部分的预先处理，通过预处理结果来替换程序原本繁琐的执行过程。但是这也会带来一些问题，首先对于循环体的预处理 Prepack 采用了循环体展开的方法，利用大量的重复功能代替原本的循环结构。这一方面会使得程序的体积膨胀，另一方面可能会影响程序执行过程中原本即时编译

的编译优化效果。所以说 Prepack 只能针对脚本进行预处理优化，而导致 JavaScript 性能开销的类型推导问题该优化方案并没有提供有效的解决方案。

2.1.3 Asm.js

Asm.js 是 JavaScript 的一个严格子集，它严格的限制了 JavaScript 的功能和运行方式。这样做的目的就是为了使编译好的 Asm.js 代码尽可能快的运行，在优化过程中尽可能少的作出假设。从本质上讲 Asm.js 仍然属于 JavaScript，所以不需要任何的插件就可以兼容的运行在各种浏览器上，即使浏览器不支持 Asm.js 的技术。

Asm.js 设计的主要目的仍然是解决严重影响 JavaScript 运行性能的动态数据类型问题。Mozilla 的 Asm.js 解决动态类型的方式和 WebAssembly 一样，都是利用其他静态类型语言代替 JavaScript。从使用方式上讲，Asm.js 也是作为其他语言的编译目标，可以通过 Emscripten 编译生成。Emscripten 框架将 C/C++代码传入到 LLVM[43]中，并且将 LLVM 生成的字节码转换成 Asm.js。Asm.js 具有高效运行性能的主要因素在于很好的利用了 CPU 特性。Ams.js 利用“与”操作等简化 JavaScript 中数据的计算过程，提高运行效率。并且利用位操作，注解等方式，在 JavaScript 中模拟静态数据类型的操作，减少类型推导。Asm.js 所采用的优化方式并没有摆脱 JavaScript 语法规则本身，所以它仍然属于 JavaScript，只是针对性能进行了更加严格的限定。

虽然 Asm.js 属于 JavaScript 语言，并且能够在其他浏览器上面稳定执行，但是如果没有 Firefox 浏览器 JavaScript 引擎汇总的 OdinMonkey 模块它的运行性能会十分糟糕。严格的功能限定只会让可执行代码变得更加冗余，如果没有特定的解析方式性能必然受到影响。并且 Asm.js 仍然没有摆脱 JavaScript 本身，虽然通过模拟实现静态数据类型，然是 JavaScript 设计上的性能缺陷依然被保留，所以无法成为一种可靠的性能解决方案。这也从一方面说明 JavaScript 的优化需要浏览器厂商共同制定标准。

2.1.4 WebAssembly

WebAssembly 是一种由多家浏览器厂商共同推出的新的可移植二进制代码形式，它可以为 Web 端应用程序提供接近本地层代码的执行速度，并且可以作为 C/C++等语言的编译目标。正因为这样，它为以前难以移植到 Web 端的应用提供了解决方案，并且提供了高效的运行环境[44]。

WebAssembly 允许使用者采用 C，C++或 Rust 等代码，并将其编译生成相应的 WebAssembly 模块。用户可以将其加载到 Web 应用程序中并通过 JavaScript 调用它，它不是 JavaScript 的替代品，它需要与 JavaScript 一起使用。目前，Emscripten 提供

了一个框架，用于将 C/C++等编译生成 WebAssembly，并且生成在 JavaScript 中嵌入 WebAssembly 模块所需的执行环境。通过这个框架，就可以在 JavaScript 运行过程中调用通过 C/C++等实现的函数。

WebAssembly 被创造的主要目的是在 Web 端获取更好的执行性能，它的二进制文件形式比 JavaScript 文件拥有更小的体积，因此加载和执行速度也更快。JavaScript 是一种无类型语言，在编写代码时不需要进行变量类型的定义，也不需要提前编译。这使得代码的编写变得简单快捷，但是这也意味着 JavaScript 引擎需要承担许多工作。它必须在浏览器端对代码进行解释执行。解释执行 JavaScript 的过程需要将纯文本的代码解析生成抽象语法树的数据结构，并且将其转换成二进制格式[45]。而 WebAssembly 以二进制的形式提供，解析速度更快[46]。并且 WebAssembly 是一种静态类型语言，它与 JavaScript 不同，不需要在执行的过程中进行复杂的数据类型的推导，而且大多数的优化都是在编译源代码期间，甚至在进入浏览器之前发生的。这就使得在浏览器端可以减少额外的性能开销。并且 WebAssembly 和 C/C++这样的语言一样，内存都是手动管理的，并不提供垃圾回收机制。所有的这些机制都是为了提供了更好，更可靠的性能。所以通过 WebAssembly 可以将大量与性能相关的计算密集型代码移植到 Web 端运行。

2.2 优化方案可行性分析

WebAssembly 标准是一种高效的二进制编码格式，能够以接近本地代码的执行效率在 Web 端运行。由于 WebAssembly 是由微软，火狐，谷歌和苹果一起推出的，所有主流浏览器的最新版本都支持 WebAssembly。此外，现有语言的各种编译器，如 C，C ++和 Rust，以及专为 WebAssembly 设计的新语言，都支持它作为编译目标。该生态系统为程序员提供了针对不同应用选择最合适特征编程语言的自由，例如，一种有利于性能而不是易于使用的计算密集型应用程序。目前许多开发者都积极的在各种应用场景下使用 WebAssembly。

例如，Egret Engine [47]是白鹭科技开发的知名游戏引擎，通过引入 WebAssembly 技术，可以将 HTML5 代码编译成机器码运行，进一步使得游戏引擎的运行效率提升 300%。Unity3D[48]是一款为玩家提供三维视频游戏，实时三维动画等互动内容的游戏开发工具，并且支持包括 Windows，Mac 和 Android 等多个平台。从 2018 年开始 Unity3D 正式开始支持 WebAssembly，并且依靠 WebAssembly 的强大性能为玩家提供

稳定的服务。WebAssembly 在各浏览器环境下广泛应用，而许多黑客也青睐于这种性能接近本地层代码的语言，越来越多的恶意挖矿脚本被编译成 WebAssembly 形式，增加脚本的运行效率并且使得用户不易发现。这也说明了 WebAssembly 在运算性能上具有很大的优势。Magnum[49]是一款轻量级和模块化的游戏、数据可视化 OpenGL 图形处理引擎，为了保证运行的效率，在网页环境下必须将代码转换为 WebAssembly 的二进制形式。

综上所述，WebAssembly 技术已经可以在现有的应用场景中发挥稳定的作用。对于现有的 JavaScript 优化方案，由于无法从根本上解决 JavaScript 设计上的问题，导致无法提供很稳定的优化效果。除此之外，由于 Web 应用程序的应用的广泛性和运行平台的多样性，导致只有浏览器厂家共同制定统一标准才能有效改善 JavaScript 运行性能的问题。而 WebAssembly 正是由浏览器厂家一同制定的标准。虽然 WebAssembly 的主要目的是作为其他低级别语言的编译目标，进而将与性能相关的功能代码高效的移植到 Web 端。但是我们可以利用 WebAssembly 对现有 JavaScript 代码中与性能相关的计算密集型代码进行性能优化，通过 JavaScript 与 WebAssembly 相互嵌套的形式对代码进行重构。而且由于以 C/C++为例的 WebAssembly 编译的源语言都是静态类型语言，所以用 WebAssembly 代替部分 JavaScript 代码可以避免运行过程中复杂类型推导的性能开销，并且结合即时编译的优化可以获得更加稳定的优化效果。虽然目前 WebAssembly 从功能上无法覆盖 JavaScript 上的全部语法规则，但是利用 WebAssembly 与 JavaScript 相结合的方式，将注重性能的计算密集类型通过 WebAssembly 实现依然可以应对目前的应用场景。我们通过动态符号执行的方式检测程序运行过程中的数据类型，获取数据类型稳定的代码段，并且以 WebAssembly 语法可模拟为限制条件筛选可优化代码段。通过这样的方式将部分动态类型语言转换成静态类型语言，从而避免 JavaScript 运行过程中类型推导导致的性能开销。由于数据类型的限制和 WebAssembly 功能的限制，我们可以将筛选的代码段通过转换生成等效的 C 代码。再利用 Emscripten 将 C 代码编译生成 WebAssembly 的二进制形式，使其可以在 Web 端运行。由于 WebAssembly 的运行效率接近于本地执行，但是为了保证原本程序执行的稳定性，需要在进入和退出 WebAssembly 模块时进行数据流转移，这就导致了额外的性能开销。为了解决额外数据交互性能开销的问题，我们又以数据依赖关系为基础，设计了 WebAssembly 模块合并方法，通过模块合并减少数据传递的频率，进而提升程序整体的执行性能。综上所属，无论是从可行性还是优化效果，

本文提出的基于 WebAssembly 实现的 JavaScript 性能优化方案都是有效的。

2.3 本章小结

本章的主要内容，一方面探究了 JavaScript 产生性能问题的原因以及针对性能缺陷各界采取的解决方案，例如即时编译技术，Asm.js 技术和 Prepack 等，并且分析当前解决方案存在的局限性。另一方面介绍了 WebAssembly 相比 JavaScript 的性能优势和主要应用范围。最后对基于 WebAssembly 对 JavaScript 进行性能优化方案的可行性进行研究。

第三章　基于 WebAssembly 的 JavaScript 性能优化方法

通过上一章的分析，对于 JavaScript 的性能缺陷和现有解决方案有了初步的了解。本文结合 WebAssembly 技术，提出了一种基于 WebAssembly 的 JavaScript 性能优化方法。WebAssembly 是一种二进制格式代码，作为 C/C++等源代码的高效编译目标可以运行在 Web 端。

3.1 JSOPW 优化方法概述

JSOPW 的核心在于通过筛选规则获取待优化代码中的数字计算类型的代码段，并且编译生成 WebAssembly 模块。再针对 WebAssembly 模块调用过程中产生的性能开销设计代码段合并规则以减少数据交互次数。图 2 描述了 JSOPW 优化方案的执行过程。

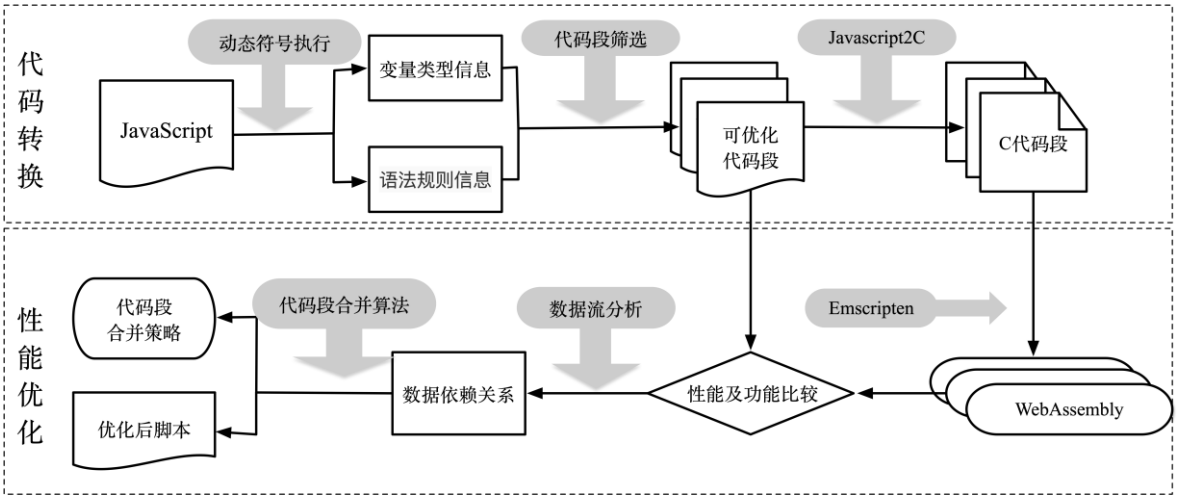


图 2　JSOPW 优化流程图

由于 WebAssembly 是作为 C/C++等低级别源代码的编译目标，所以它的功能无法完全覆盖 JavaScript 的语法规则，所以我们的优化目标代码为 JavaScript 的一个严格子集。根据框架的功能，我们将整个优化框架分为两部分：

（1）JavaScript 代码转换方法

目前 WebAssembly 的官方实现只支持数据计算类型的代码，所以我们将优化目标锁定为 JavaScript 脚本中的计算密集型代码。这部分我们首先通过动态符号执行对目标脚本的变量类型进行统计分析。将分析后的变量类型信息映射到所在的语句生成语句的类型信息，进一步的生成符合 WebAssembly 功能的目标 JavaScript 代码段。经

过 JavaScript2C 方法将目标代码段转换为 C 代码，再利用 Emscripten 将 C 代码编译为相应的 WebAssembly 模块。

（2）代码段性能优化方法

首先利用单元测试的思想将之前筛选获取的 JavaScript 代码段集合和相应的 WebAssembly 模块集合生成可运行单元。再利用动态执行的方式比较 JavaScript 代码段单元和相应的 WebAssembly 模块单元的执行效率并记录。根据各模块执行效率对比信息利用代码段合并方法对模块进行分析并生成相应的代码段合并策略，从而减少 WebAssembly 模块调用的性能开销，提升程序的执行效率。

3.2 JavaScript 代码转换方法

JavaScript 作为一种动态类型语言，它的数据类型只有在运行时才会通过推导获得，并且它还是一种弱类型语言，导致执行过程中变量类型可能随时发生变化。所以频繁且复杂的变量类型推导导致了运行过程中的性能开销。为了解决这个问题，我们利用静态类型语言对可替换部分功能重新实现，进而提升执行效率。所以我们设计利用 C 语言替换部分 JavaScript 代码，并且利用 WebAssembly 作为 C 的编译目标运行在 Web 端。

为了保证优化的目标代码可以通过静态类型语言模拟，需要通过对 JavaScript 代码段进行筛选，一方面保证目标代码段可以被 WebAssembly 功能实现，另一方面保证静态类型语言模拟下程序的正确性和稳定性。

3.2.1 变量类型获取

为了获取程序执行过程中的变量类型信息，我们利用动态符号执行[50,51]的方式进行分析，并且为了保证变量类型在各种执行路径下保持统一，我们利用动态符号执行的方式对待优化代码进行动态分析。符号执行是一种程序自动生成符号模型的技术。它已经成功用于包括对 C，C++，Java，JavaScript，x86-二进制文件等的高覆盖率测试中。顾名思义，符号执行技术主要就是利用符号值来推断程序执行路径的方法。具体来说就是利用符号代替程序中的数据值，并且根据程序的执行路径和判断谓词生成表示不同路径的符号表达式，进而在控制流图的基础上生成符号执行图。结合具体执行的方式，利用符号表达式生成具体的测试用例，进而动态的执行程序的所有路径。

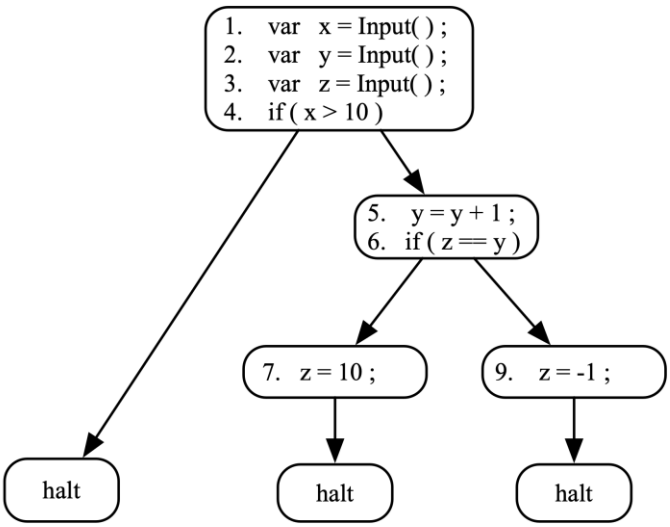
在符号执行的运行过程中，如果遇到判断谓词，符号执行工具就会将当前执行路径的路径约束添加到这条路径的约束集合中。路径约束是指程序的分支条件中与程序

输入符号相关谓词条件的取值，就是含有输入符号的布尔类型表达式。路径约束集合是用来收集每一条路径上收集到的路径约束，用“与”操作进行连接，并且通过使用约束求解器对约束集合进行求解，就可以得到这条路径是否可达[52]。如果约束求解器有解则说明路径可达，否则则表示路径不可达。在资源充足的情况下符号执行可以对程序的执行路径全覆盖。为了防止动态执行过程中路径爆炸的问题，可以采用状态合并等方式进行处理。

因为程序的主要目的在于优化而非代码分析，则可以通过提供测试用例的方式来增加执行的覆盖路径，提高优化框架的执行效率。我们使用图 3（a） 中的 JavaScript 程序作为一个运行示例，其中 var a=b 表示利用表达式 b 的值来初始化变量 v。语句 var x=Input()表示从输入流中获取值并且赋值给变量 x。动态符号执行通过探索图 3（b）的中执行树来选择程序的执行策略。我们在动态符号执行的过程中添加监测模块，记录程序执行过程中堆栈信息和控制流信息，并且排列这些堆栈数据获得更为直观的 JavaScript 变量数据值，最后通过树型结构对节点属性和嵌套结构进行保存。

```
1   var  x = Input( ) ;
2   var  y = Input( ) ;
3   var  z = Input( ) ;
4   if ( x > 10 ) {
5      y = y + 1 ;
6      if ( z == y ) {
7         z = 10 ;
8      } else {
9         z = -1 ;
10     }
11  }
```

（ a ）



（ b ）

图 3　符号执行生成节点信息

3.2.2 变量类型解析

因为 WebAssembly 的设计目是为了保证一些计算相关的代码被移植到 Web 端仍然可以保证高效的运行效率，所以 WebAssembly 本身无法完全覆盖 JavaScript 的语法规则。为了保证程序执行的稳定性和正确性，我们只对可以有效的映射到本地代码的语法规则进行优化，例如变量声明，赋值，控制逻辑，数组和函数等。而对于可能会产生错误的例如原型继承，闭包和反射等不进行处理。因此，我们需要针对上述条件

筛选符合优化条件的语句生成可优化代码段，而不符合这些条件的代码段称为不可优化代码段。

可优化代码段的生成主要根据动态符号执行过程中监测模块生成的变量类型结构（VariableInfo）进行分析处理，其中根据抽象语法树中节点类型不同生成如图 4 中所描述的嵌套结构以保留代码结构信息。并且因为采用动态符号执行对程序路径进行全覆盖的分析，所以根据不同执行路径会生成多个 VariableInfo 嵌套的树结构，我们称为 VariablesTree。而可优化代码段的生成第一步就是解析 VariableInfo 结构，并且映射到待优化代码中。但是在符号执行过程生成的 VariablesTree 结构存在两个问题：重复节点信息和无控制流结构。产生这两个问题的主要原因在于 VariablesTree 结构是程序动态执行的过程获取的，和程序的执行路径相关。这就导致每个 VariablesTree 只包含一条执行路径，并且由于动态执行过程中循环结构被展开导致存在大量重复的 VariableInfo 节点。而原本待优化的 JavaScript 代码是静态结构，保留完整的循环结构和谓词分支结构，所以我们需要将动态生成的多个 VariableTree 结构与静态代码产生映射。JSOPW 主要通过合并映射节点方法和代码块重构方法来解决这两个问题。

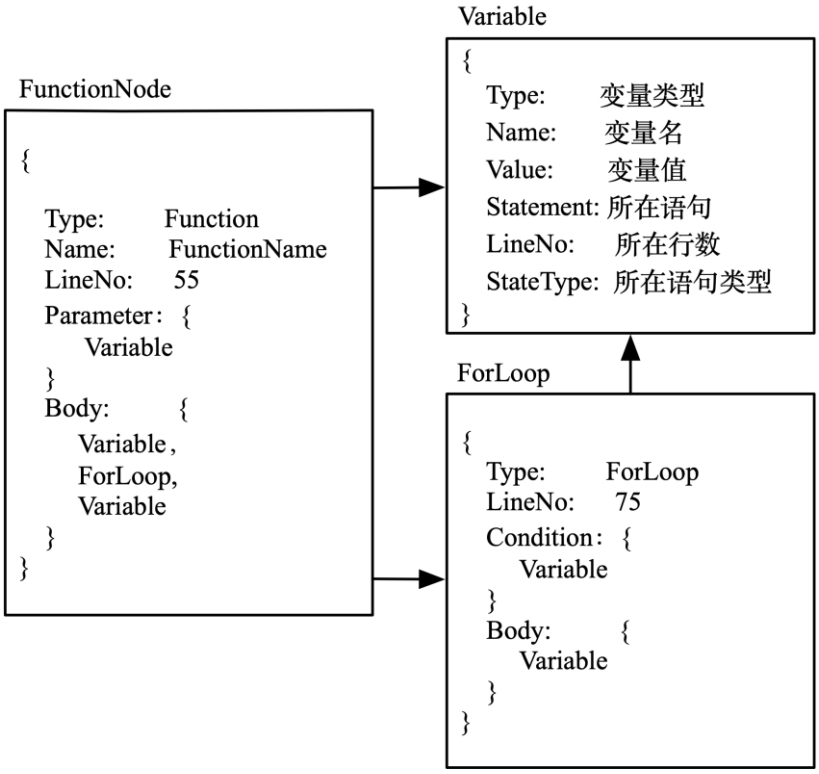


图 4　变量类型结构-VariableInfo

合并节点映射方法：针对重复 VariableInfo 节点的问题，我们利用抽象语法树信

息进行映射。由于程序执行过程中抽象语法树信息是保持一致的，所以我们通过在 VariableInfo 节点中添加抽象语法树信息和待优化代码建立联系。通过对 VariableTree 进行解析，获取每个变量的 VariableInfo 节点，再根据节点中保存抽象语法树中变量所在语句和行数信息作为 key 值对节点进行管理。通过行数信息和所在语句据可依保证每一个 key 值有且只有一条代码语句和它映射。所以通过每个变量映射到所在语句，就可以根据将每个变量类型是否符合优化条件相“与”的结果判断该语句是否符合优化条件。最后根据对所有 VariableTree 进行解析，保证目标语句在任何执行路径下都符合优化条件。

代码块重构方法：代码块重构的主要目的在于判断块结构的代码整体是否符合转换条件。块结构代码主要是针对循环语句结构，函数体和判断语句结构等。我们针对一个 VariableTree 结构是否能覆盖块结构中的所有语句将块结构分为完整控制流结构和不完整控制流结构。由于每个 VariableTree 结构只能保留一条执行路径上的变量信息，我们称在执行路径中只包含块结构代码部分子节点的模块为不完整控制流模块，例如循环判断条件不满足的循环体和包含 ElsePart 的条件语句等。而函数体等属于完整控制流结构。为了保证不完整控制流模块分析的完整性，需要针对所有执行路径上的 VariableInfo 信息进行处理，并且通过合并产生块结构代码的类型信息。例如图 5-a 所示，每个 VariablesTree 结构保存一条执行路径上的变量类型信息，通过对 n 个 VariablesTree 进行合并实现所有代码路径的全覆盖。通过将不完整控制流模块每部分的判断结果相“与”就可以获得整个模块是否符合优化规则，如图 5-b 中所示。我们以 for 循环语句结构和 If 条件语句结构为例进行分析。for 循环语句结构的不完整性在于执行过程中判断条件为否不执行循环体的情况，所以首先判断循环结构必然会执行的初始化语句，循环判断语句和循环递增语句是否符合可优化规则。除此之外，针对循环体需要对循环体内的语句和嵌套的块结构代码进行分析判断，并且将结果相“与”，如果结果为真则说明 for 循环结构可以整体进行优化，反之不行。If 条件语句同理，不仅需要对条件判断语句进行筛选，还需要对条件语句的 ThenPart 和 ElsePart 两部分进行分析，只有所有的子结构都符合优化规则，才能将条件判断结构整体设置为可优化代码段。

通过合并节点映射方法和代码块重构方法后，可以获得符合优化条件的代码语句和块结构代码集合，通过合并相邻代码语句和块结构代码就可以生成可优化的目标代码段。

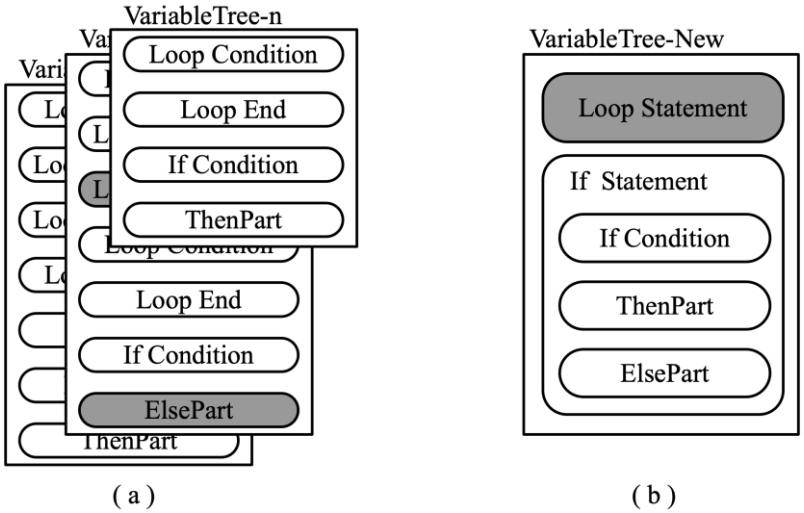


图 5　代码块重构

3.2.3 JavaScript2C

图 6 展示了我们实现的从 JavaScript 脚本到 WebAssembly 模块的全过程，其中将 C 代码编译生成 WebAssembly 模块的过程通过 Emscripten 编译器实现。而 JavaScript 脚本转换生成 C 代码的过程通过本文设计的 JavaScript2C 方法实现，该方法的主要实现是在抽象语法树级别对代码语法规则做转换。

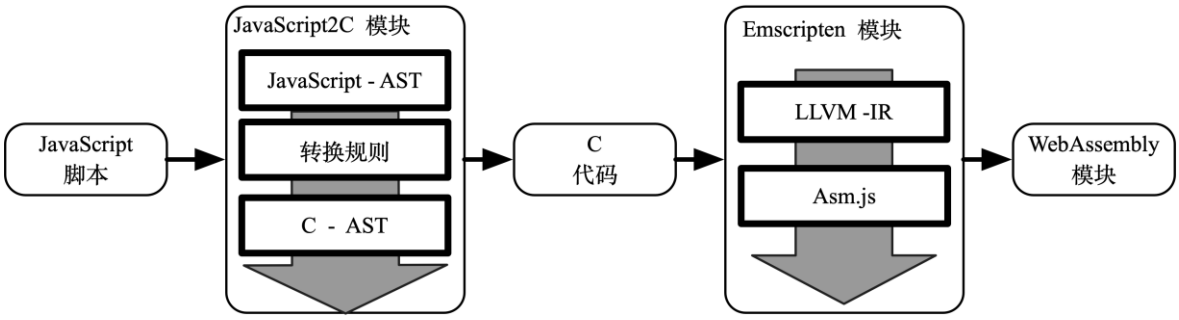


图 6　WebAssembly 模块编译流程图

编译器前端的主要功能是通过词法分析器和语法分析器的协同工作对输入的目标代码段进行解析。词法分析器的主要功能是将目标程序中的“token”提取出来，而语法分析器将这些零散的“token”按照定义好的语法规则组成有意义的表达式和函数等，例如“a=b+c;”，前端词法分析器看到的是“a , = , b , + , c”这些单独的符号,语法分析器按照定义的语法，先把他们组成表达式“b+c，再组成“a=b+c”的语句。我们在实现时利用 Moliza 的 Rhino 引擎生成 JavaScript 抽象语法树[53]，再通过转换规则生成 C 抽象语法树，进一步生成目标 C 代码。因为我们的优化目标是 JavaScript

语言的子集，主要针对计算密集型的代码段，所以转换规则以计算相关的变量类型声明和变量作用域规则为主，其他代码语法规则相关的转换按照抽象语法树的属性重构。而不支持的复杂语法规则在之前进行的可优化代码段筛选过程中被过滤。

A. 变量类型声明修改

JavaScript 代码被称为无类型语言，所谓无类型就是指在变量声明阶段不需要对变量类型进行限定，并且变量可以保存任何类型的数据值。并且 JavaScript 还是一种动态类型的编程语言，只有在运行过程中才会知道变量的数据类型。这样的规则设计使得 JavaScript 引擎必须承担复杂的类型推导，导致了性能开销。而我们的目标代码 C 语言作为一种静态数据类型，在使用变量前需要针对变量的数据类型进行相应的声明。所以我们需要对 JavaScript 代码数据流进行分析，获取变量的数据类型，并且获取数据类型稳定的代码段。以动态符号执行的分析结果为基础，可以获取每个变量在不同执行路径下的数据流信息，就可以分析获得该变量从声明到结束过程中数据类型的变化情况。我们通过动态符号执行过程获取的 VariableInfo 结构中的数据类型和数据值信息，利用抽象语法树中获取的变量作用域信息可以推断出变量的声明类型。针对 JavaScript 代码动态变量类型的特点，分析动态符号执行中各执行路径下变量类型的变化情况，如果存在某一执行路径下变量类型为 WebAssembly 不支持的数据类型，则该变量所在语句就无法被优化。否则为变量传递过程中合法的数据类型转换语句添加强制类型转换以保证程序编译过程的正确性。

通过这种方式我们可以将部分 JavaScript 代码的动态数据类型变成稳定的静态数据类型，这样就可以避免在与性能相关的计算过程中进行大量的类型推导，并且可以避免优化编译过程中退优化导致的性能开销，使得优化过程变得稳定。

B. 作用域规则修改

在作用域规则方面，JavaScript 使用的是函数级作用域，而 C 代码使用的是块级别作用域。虽然在 JavaScript 的 ES6 标准中添加了块级作用域，但是依然保留了函数级作用域的语法规则。作用域规则决定了变量的可见性和生存时间，作用域规则不同会导致变量调用逻辑的误差，进而导致程序运行错误。我们通过在抽象语法树级别进行作用域分析，对变量声明节点和调用节点在函数结构中的相对位置进行修改。由于 JavaScript 本身采用的是函数级别作用域规则，在作用域中没有块级别的界限。所以我们首先在抽象语法树中以循环体，If 语句，For 循环体等块结构节点为分界线，对块结构代码中变量的生存周期进行分析，并且对块结构外变量调用过程中变量定义的

可见性进行分析。例如获取循环结构内变量的声明情况，并且获取相应变量在循环结构外的调用情况。其次，我们针对 JavaScript 中变量声明和调用顺序进行分析。通过以上的分析，我们在抽象语法树中修改变量的声明范围，根据变量声明在块结构代码内外的可见性对声明和调用节点进行调整，进而将函数级作用域规则转换成块级作用域规则。

通过变量声明修改和作用域规则修改可以将筛选的 JavaScript 转换成静态数据类型代码，之后通过抽象语法树上语法规则的转换生成相应的 C 代码。最后，将生成的 C 代码通过 Emscripten 编译器编译生成相应的 WebAssembly 模块。

3.3 代码段性能优化方法

我们通过代码转换的方式生成目标代码集合和相应的 WebAssembly 模块集合。虽然 WebAssembly 本身拥有高效的运行效率，但是为了保证程序运行的稳定性和正确性，需要对 WebAssembly 模块添加额外的调用和数据传输功能。代码段性能优化方法针对这些额外功能的性能开销和 WebAssembly 的性能提升进行权衡。

3.3.1 动态性能分析

转换后的 WebAssembly 模块具有三部分，wasm 代码解析及实例化部分，数据交互部分和 wasm 代码执行部分。WebAssembly 二进制代码格式的解析效率和运行效率相比 JavaScript 都存在优势，但是额外产生的数据交互部分会对模块的整体执行效率造成性能开销。由于数据交互部分产生的性能开销大小无法估量，所以我们通过动态执行的方式对 WebAssembly 模块和相应 JavaScript 代码的执行效率进行对比，并且对 WebAssembly 模块运行过程中数据交互部分进行分析，我们以保证程序数据流正确性的状态转移和 WebAssembly 内存访问两部分进行分析。

A. 状态转移

为了保证 WebAssembly 模块与所在上下文的 JavaScript 代码间数据依赖关系的正确性和程序的稳定运行，我们以虚拟机保护方法[54,55]中状态保存的设计思路为基础，设计在进入 WebAssembly 之前实现状态转移，并且在 WebAssembly 模块返回后进行状态恢复操作，如图 7 所示。状态转移的主要目的是使 WebAssembly 模块可以正确的处理 JavaScript 环境中的数据，所以需要将 WebAssembly 模块中所依赖的数据信息依次从 JavaScript 传递到 WebAssembly 中。状态恢复的主要目的是使的 WebAssembly 模块返回后可以更新后续 JavaScript 代码所依赖的数据，将在 WebAssembly 环境中修

改的变量和新声明的变量返回到 JavaScript 运行环境中，实现数据流的更新。在这种处理方式下，虽然 WebAssembly 环境和 JavaScript 环境相互独立，但是通过数据依赖关系依然可以保证程序数据流的正确性和程序执行的稳定性。

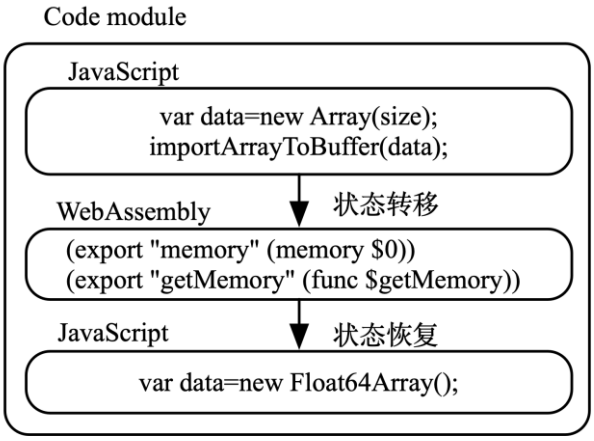


图 7　WebAssembly 模块数据交互图

B. 内存访问

WebAssembly 的设计者为了保证内存访问的安全性，使的 WebAssembly 环境和 JavaScript 环境数据的访问不能直接相互访问，而只能通过读写线性内存来实现，如图 8 所示。虽然这样的实现方式保证了 WebAssembly 模块内存读写的安全性，但是这也导致 WebAssembly 模块和 JavaScript 代码的数据交互只能通过频繁的内存读写来实现。然而为了保证数据流正确性所采用的状态转移和状态恢复两个步骤就会产生不可避免的性能开销。

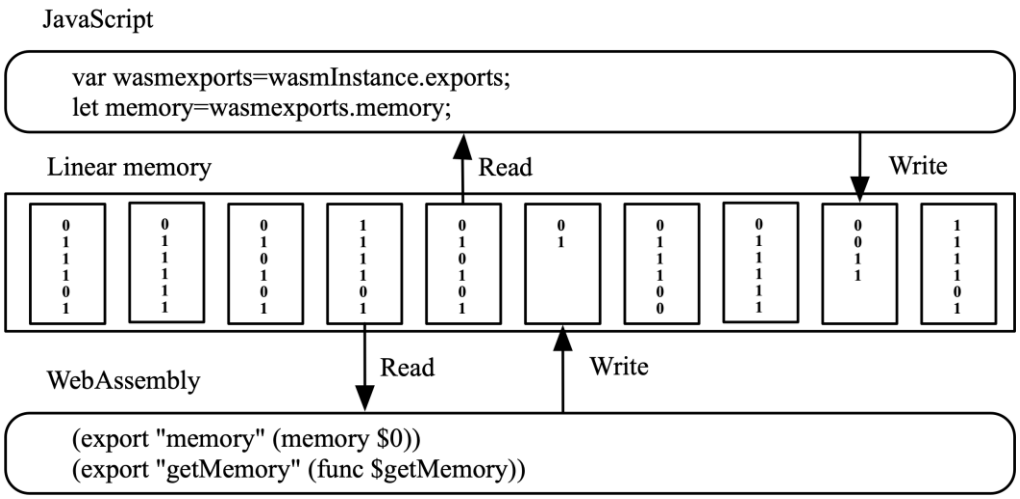


图 8　WebAssembly 内存交互图

由于存在数据交互导致的性能开销，生成的 WebAssembly 模块无法像官方描述那样拥有高效的执行性能，还可能存在因为大量数据读写导致模块整体性能下降的状况。所以我们需要对 JavaScript 代码段和相应的 WebAssembly 模块进行动态的性能比

较，并且进行标记。如图 9 所示，我们采用单元测试的思想，将 JavaScript 代码段和 WebAssembly 模块分别封装成单元模块，并且利用动态符号执行过程统计的变量数据值作为测试单元的驱动参数，使得每个单元模块都可以稳定的动态运行。然后利用监视器获取 JavaScript 代码段和相应 WebAssembly 模块的运行时间，利用运行时间的长短判断 WebAssembly 模块性能增加或是减少，并且利用标记对性能信息进行标记。并且如果转换后性能下降则保留原本的 JavaScript 代码段。其次通过比较单元测试输出值确定功能的一致性，由于参数集是通过动态符号执行的约束求解器获取，所以可以保证输出值比较的全面性。如果功能一致则调用 WebAssembly 代码段，否则保留原 JavaScript 代码段。

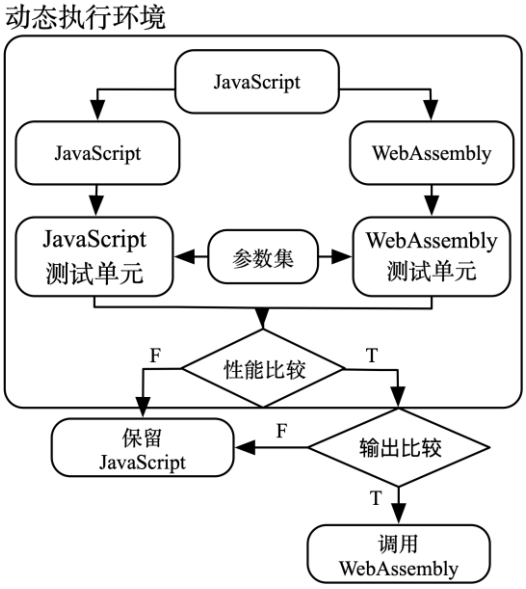


图 9　动态性能比较

通过标记各个模块的性能信息，我们可以筛选性能提升的WebAssembly模块替换相应的JavaScript代码段，通过这样的方式提升整体的运行效率。并且以性能对比的结果作为WebAssembly代码段合并的判断条件。

3.3.2 代码合并方法

通过上一小节分析我们可以知道引发 WebAssembly 模块性能下降的主要原因是 JavaScript 和 WebAssembly 之间的额外的数据交互，所以我们尝试利用合并模块的方式减少数据交互的次数，从而减少内存读写的性能开销。但是因为 WebAssembly 模块的分割是由于存在 WebAssembly 无法模拟的语法规则或功能，所以代码合并规则最终以优化策略的方式供用户参考，方便用户对代码结构进行重构。

虽然说通过 WebAssembly 模块间的合并可以减少数据交互导致的性能开销，但

是由于各个模块间交互的数据流大小不同造成的性能开销也不同。由于无法通过定量的方式评估数据交互的性能损失，我们就以定性的方案描述合并后的性能变化。我们以上一小节中获得的模块性能对比标记为基础。针对模块合并后性能产生的变化提出两种合并规则：可合并规则和推荐合并规则。可合并规则是以代码结构为判定标准，因为 JavaScript 中存在例如原型属性，方法等 WebAssembly 无法模拟的语法规则，但是这些语法规则可以通过例如函数定义等其他方式进行重构，如果模块可以通过语法规则的重构进行合并则称模块为可合并。由于每个 WebAssembly 模块运行性能不同，通过可合并规则进行合并后可能很难确定性能相比 JavaScript 代码是否更高效，所以我们提出了推荐合并规则。推荐合并规则顾名思义通过模块合并后整体性能一定会提升，并且推荐合并规则一定是可合并的。综上所述，可合并规则针对代码结构进行判断，而推荐合并规则以性能提升作为判断依据。

首先我们针对推荐合并规则进行描述，所以假定所有节点都符合可合并规则。我们在两种场景下描述模块合并规则：函数体内模块合并和函数体外模块合并。函数体内的模块分割主要是因为存在 JavaScript 代码功能模块，而函数体外的模块分割主要是存在 WebAssembly 无法模拟的例如类，方法等语法规则导致的。

A. 函数体外模块合并

由于 WebAssembly 功能的支持和为了保证优化程序执行的稳定性，我们优化目标代码以 JavaScript 语法规则的一个严格子集为基础，所以例如 JavaScript 面向对象相关的语法规则都无法模拟。主要是因为 JavaScript 是一种以原型为基础的语言，其面向对象的实现和 C++等存在较大区别，所以目前无法稳定的模拟这部分的语法规则。而这就导致了 WebAssembly 模块被不可模拟的 JavaScript 语法规则分割的情况，例如对象中每个方法体都是以 WebAssembly 代码实现。在这种情况下程序的运行时不得不在 JavaScript 和 WebAssembly 模块间频繁切换，并且进行数据交互以保证程序数据流正确性。这个过程导致了大量额外的性能开销，函数体外模块合并主要解决这种情况的性能问题。

我们以图的形式表示程序中各个模块的数据依赖关系，图中的节点表示对象的方法由 WebAssembly 实现，并且由于语法规则的限制相互独立。节点中连线表示模块间存在数据依赖关系并且需要进行状态转移和状态恢复操作保证数据流的正确性。我们以节点合并规则的适用范围可以将推荐合并规则分为整体合并和局部合并，具体如下：

整体合并：整体合并主要针对程序中只包含 WebAssembly 模块，即目标代码整体可以利用 WebAssembly 重新实现。这种情况下，无论各个模块的性能如何，合并后程序的运行效率一定会提升。如图 10 中所描述，我们利用 T 表示数据交互的时间消耗，D 表示转换为 WebAssembly 后程序执行相比原代码减少的时间，为负数。“+”号表示模块运行转换成 WebAssembly 模块后运行时间增长。为了保证规则的稳定性，我们都以时间开销最大的情况进行分析。在最坏的情况下，生成的每个模块运行时间都比 JavaScript 代码长，这说明模块间的数据交互开销严重的影响了每一个模块的运行性能。我们对具体节点进行分析，在图 10-a 中，A 节点在转换为 WebAssembly 模块后运行时间增加，而 A 节点的运行时间由两部分组成：WebAssembly 程序运行时间和状态恢复时间 T1。B 点的时间组成和 A 相同，同样在转换为 WebAssembly 形式后运行时间增加。对于 C 点，C 的两个前驱节点为 A 点和 B 点，说明 C 点会使用 A 和 B 两个节点的输出数据，为保证规则的稳定性，我们假设 C 点会传入和传出 A 点和 B 点的所有数据。同理 D 点会传入 A 点和 B 点产生的所有数据。所以程序执行过程中的数据交互就是频繁读写 A 点和 B 点产生的数据流。为了避免这些模块间的数据传递开销，我们将模块进行整体的合并，如图 10-b 所示，程序的整体运行之间只有四个节点 WebAssembly 代码解析和运行的时间。如果程序中所有模块都可以转换成 WebAssembly 形式，并且模块间的分割是因为存在 WebAssembly 无法模拟的 JavaScript 结构相关的语法规则，那么可以利用推荐合并规则中的整体合并对性能进行优化。

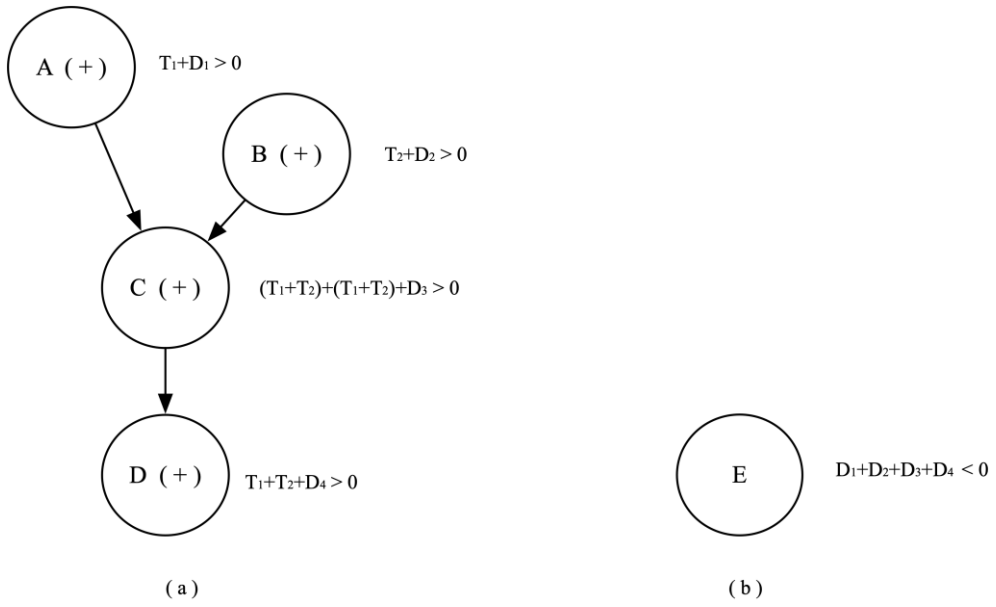


图 10　模块整体合并

局部合并：当存在不可优化模块时，如果存在某节点的所有前驱节点都为运行时间开销减少节点，那么该节点和前驱合并后的新节点也为运行时间开销减少节点，并且该规则具有传递性。图 11 描述局部合并过程，首先以图 11-a 中节点 C 为例进行分析，C 的两个前驱节点 A 点和 B 点都是时间减少节点，而 C 节点为时间增加节点，则合并后就可以减少 A 点和 B 点到 C 点的数据交互开销。每个节点的性能开销为三部分，WebAssembly 程序减少时间为负数 D，状态转移时间和状态恢复时间。为了保证规则的稳定性，将转改转移时间和状态恢复时间都设定为两者中的较大值 T。原本 A 点的时间开销为 $T_1+T_1+D_1$，B 点的时间开销为 $T_2+T_2+D_2$，为了保证规则的稳定性我们认为 C 点会接受 A 点和 B 点的所有输出数据，那么 C 点的时间开销为（T_1+T_2）+（T_1+T_2）+D_3，通过节点的合并，我们可以节省 A 节点和 B 节点到 C 的数据传递时间，图 11-b 中显示合并后的新节点 C’的时间开销为（T_1+T_2）+（T_1+T_2）+$D_1+D_2+D_3$，A 节点的时间性能提升则 $T_1+T_1+D_1$ 小于 0，同理 B 节点 $T_2+T_2+D_2$ 也小于 0，那么在 D_3 小于 0 的情况下我们可以推断 C’节点（$T_1+T_1+D_1$）+（$T_2+T_2+D_2$）+D_3 小于 0。因为假设 C 点数据交互为最大数据量，所以在数据量小的情况下合并后的节点 C’执行性能一定会提升。合并后更新 A，B 和 C 三个节点的性能标记，并且利用同样的规则对 C’点的后继节点进行分析。由于 C’点性能标记的更新，该合并规则也适用于 C’的后继节点，所以该规则具有传递性。

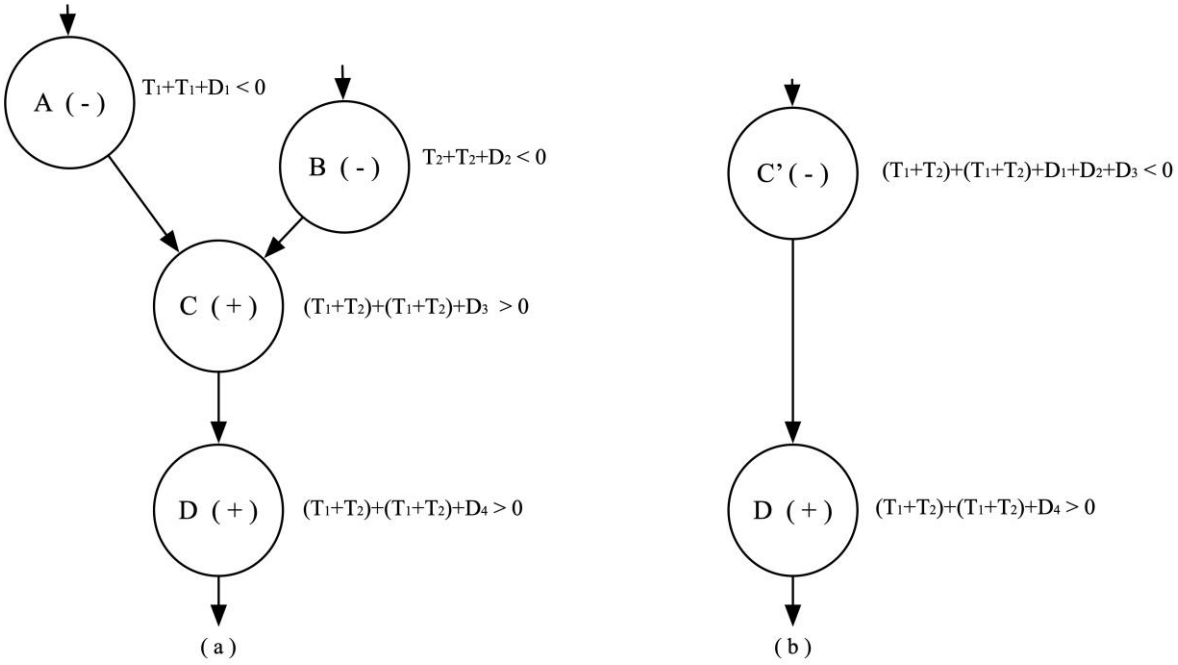


图 11　模块局部合并

B. 函数体内的模块合并

首先我们对函数体内数据依赖进行定义，如果语句 S_1 定义或者改变了一个变量，而语句 S_2 引用了这个变量，并且程序中从 S_1 到 S_2 的路径上这个变量没有被再次定义，那么语句 S_1 和 S_2 之间存在数据依赖关系。我们根据是否可生成 WebAssembly 将函数体内代码分为 WebAssembly 模块和 JavaScript 模块，例如图 12 所示。我们以代码模块为单位，在函数体内对代码模块进行数据依赖分析。如果两个 WebAssembly 模块间存在的 JavaScript 模块与它们不都存在数据依赖关系，则可以调整代码结构，合并两个 WebAssembly 模块。如图 12-a 所示，WebAssembly-a 模块和之后的 JavaScript 模块不存在数据依赖关系，并且由于 WebAssembly 模块处理与性能相关并与功能无关的数据操作，所以和 JavaScript 模块不存在功能上的先后顺序，所以可以对两个模块的顺序进行交换。通过模块交换后两个 WebAssembly 模块可以进行合并（如图 12-b 所示），这就使得原本需要从 WebAssembly-a 模块返回数据到 JavaScript 模块和 JavaScript 模块传递数据到 WebAssembly-b 模块的操作被省略，进而减少模块间的数据交互次数，从而减少运行开销。在合并模块后需要对模块的性能标记信息进行更新，性能变化规则与函数体外模块合并规则情况相同。

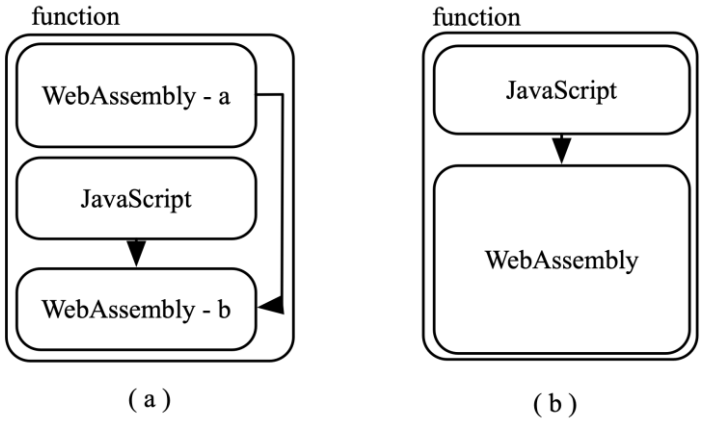


图 12　函数体内模块合并

3.3.3 含不可优化代码合并规则

由于 JavaScript 功能的多样性，并非所有模块都可以生成 WebAssembly 模块，如图 13 所示。在图 13 中，黑色节点表示包含不可优化代码段的模块，白色节点表示 WebAssembly 模块，白色节点中的符号表示该方法转换为 WebAssembly 形式后运行时间增加或是减少。图 13-a 中的实线指向表示节点间存在大量的数据传递，虚线表示极少量的数据传递，例如非数组的数据读写。首先我们以抽象语法树为基础，对代

码转换方法生成的 WebAssembly 模块进行分析，以模块间的分割条件生成独立节点。然后通过数据依赖关系获得各个节点间的数据依赖图，因为极少量的数据传递对性能的影响较少所以我们忽略虚线部分的依赖关系获得图 13-b 所描述的数据依赖，如果对性能有着极其严格的要求可以将虚线以实线的合并规则进行处理。在删除虚线后我们获得存在大量数据传递的数据依赖图，针对黑色节点我们可以利用函数体内合并规则进行判断处理，而针对白色节点间的数据依赖我们利用函数体外合并规则进行合并，但是在处理过程中会忽略黑色节点和黑色节点的后继节点，这是因为黑色节点无法生成相应的 WebAssembly 模块，进而无法实现合并。通过合并后我们可以获得如图 13-c 的数据依赖图，图中包含了无法处理的黑色节点，通过推荐合并规则合并的新节点和符合可合并规则的节点 A 和 B。通过对动态性能分析后的 WebAssembly 模块进行合并处理，可以有效的减少数据交互带来的性能开销，提升程序整体的执行效率。

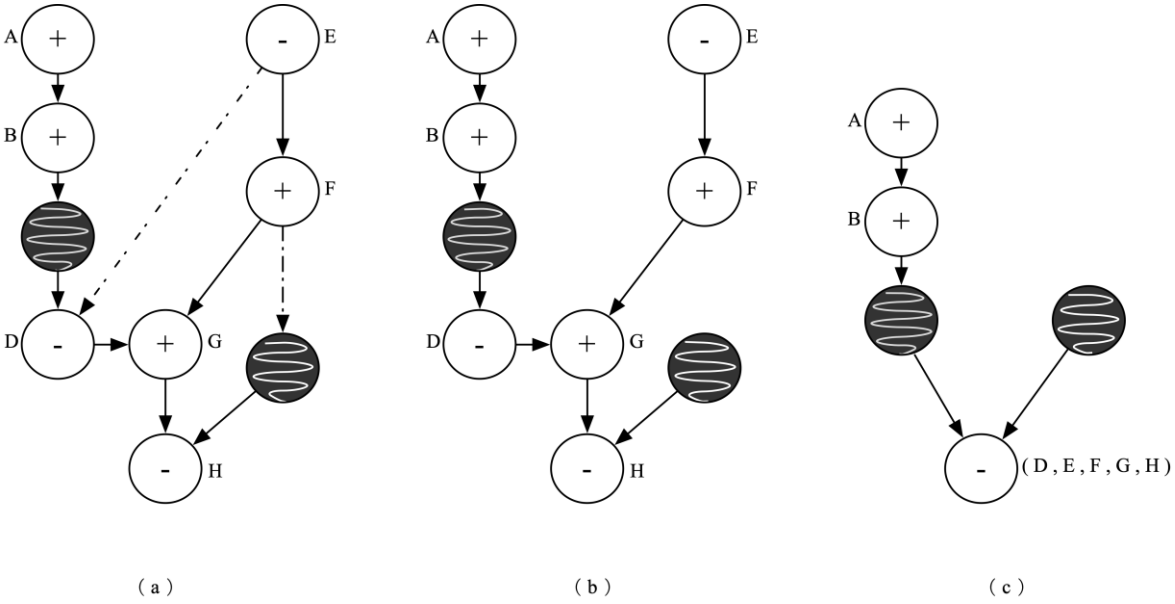


图 13　包含不可优化代码段合并过程

3.4 本章小结

本章阐述了利用 WebAssembly 对 JavaScript 代码优化方法的研究思路。从 JavaScript 代码转换方法和代码段性能优化方法两方面描述了优化策略的设计与实现。在 JavaScript 代码转换方法中利用动态符号执行的方式进行变量类型获取，并且通过 WebAssembly 功能支持和数据类型匹配两方面筛选出可优化代码段，最后利用 JavaScript2C 方法和 Emscripten 编译器生成相应的 WebAssembly 模块。代码段性能优

化方法对 JavaScript 源代码和 WebAssembly 模块进行性能比较，并且针对 JavaScript 与 WebAssembly 数据交互带来的性能开销提出解决方案，利用代码段合并方法生成对应的代码段合并策略，进而通过模块的合并减少额外的数据交互，进一步提升程序的执行性能。

第四章　JSOPW 原型系统的设计与实现

通过对 JavaScript 代码优化技术的分析，本文引入了基于 WebAssembly 的 JavaScript 性能优化方法。上一章节中描述了 JavaScript 代码转换方法和代码段性能优化方法的研究思路，本章在此基础上设计实现了 JSOPW 原型系统。

4.1 系统模块设计

图 14 描述了 JSOPW 原型系统的模块划分图，整个系统根据功能分别为任务管理模块，动态分析模块，类型解析模块，代码转换模块和性能优化模块。动态分析模块，类型解析模块和代码转换模块对 JavaScript 代码进行分析并且生成 WebAssembly 模块集合，性能优化模块进行性能比较和代码段合并策略的生成。

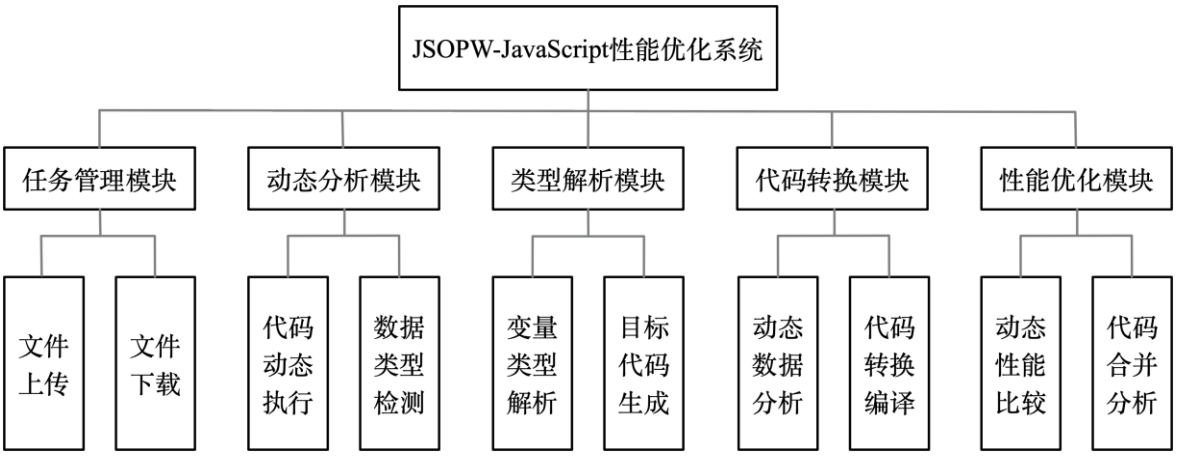


图 14　JSOPW 性能优化系统模块划分图

4.2 动态分析模块

动态分析模块的主要功能是利用动态符号执行的方式动态遍历程序的所有执行路径，并且对路径中变量类型信息进行收集。对于 JavaScript 的动态分析方式存在许多成熟的技术方案，例如采用动态符号执行的 ExpoSE[56]和采用多路径执行的 MultiSE[57]等方式。我们在实现的过程中利用了 ExpoSE 实现的动态符号执行方法，通过添加监测模块的方式获取变量数据类型信息，并且以上一章节中所描述的 VariableTree 结构进行输出。

动态符号执行的设计结构如图 15 所示，主要由分配器，执行器，约束求解器和监测模块组成。分配器的主要功能在于管理路径探索的状态，获取程序动态执行时的

数据信息，通过约束求解器获取的具体输入参数传递给执行器驱动程序运行。执行器的主要作用在于动态执行目标代码，根据当前路径分支语句中的谓词结构获取所有的符号约束，并且通过修改符号约束获得新路径的约束信息。监测模块的主要功能是监测程序执行过程中的堆栈信息，进而得到变量数据信息，作为之后变量类型解析的基础。而约束求解器的主要作用在于根据描述路径信息的约束表达式求解出满足约束条件的符号表达式，并且通过符号表达式生成具体的执行参数值，使得执行器可以拓展新的执行路径。约束求解器将参数信息返回分配器。通过这样的迭代运行，动态符号执行框架可以覆盖程序的所有执行路径，并且获得所有路径中变量的类型信息。

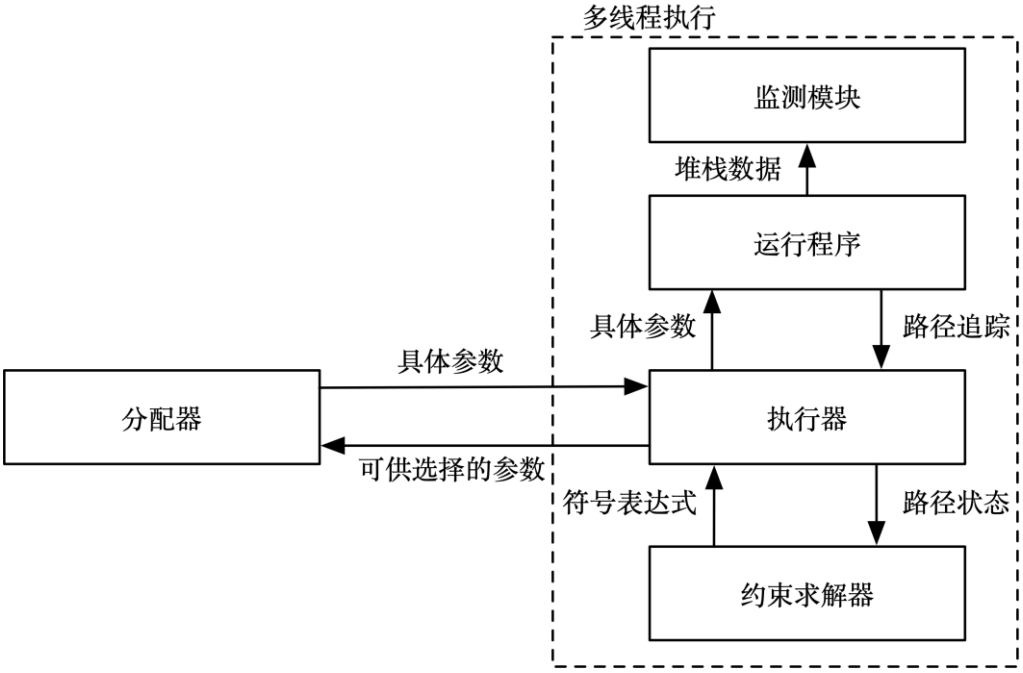


图 15　动态符号执行设计结构图

4.3 类型解析模块

类型解析模块由两部分组成，变量类型解析模块和可优化代码段生成模块。动态数据分析模块主要解析变量类型信息。可优化代码段生成模块通过合并节点映射和代码块重构方法生成可优化代码段。

4.3.1 变量类型解析

类型解析模块的主要功能是对动态符号执行过程生成的 VariableTree 结构进行解析，将解析的结果映射到具体的代码语句中，并且利用 TreeNode 结构对控制流结构和数据类型信息进行管理，实现主要功能的变量类型解析算法描述如下：

算法 1　变量类型解析算法

获取的变量类型信息字符串:　source

语句类型管理结构:　TreeNode

块类型语句映射集合:　BlankList

普通语句映射集合:　StatList

```
1:    TreeNode node=NULL
2:    for  i = 0; i < source.length() ; i++   do
3:        if   source.charAt(i) == ' { '   then
4:             node = new TreeNode( )
5:        end if
6:        if   source.charAt(i) == ' } ' then
7:            get Type,Name,LineNo,Value,StateType,Stats
8:            String key = Stats + LineNo
9             if   StatList.contains( key ) == True || BlockList.contains( key ) == True do
10:               TreeNode tmp = getValue( key )
11:               if CheckValid(Type, Name , value) == False then
12:                  tmp.setIsValid( False )
13:               end if
14:           else if CheckValid( Type , Name , value ) == True do
15:                  node = InitNode(key , LineNode , Stats , Name )
16:                  if IsBlock( node ) == True do
17:                       BlockList.put( key , node )
18:                  else
19:                       StatList.put( key , node )
20:                  end if
21:              end if
22:   end for
```

代码实现解析如下：

3-5：根据节点信息字符串的分割字符‘{‘ 作为单独节点的开始，并且初始化保存语句信息的管理结构 TreeNode。

7-8：根据节点信息字符串 source 解析获得节点相应的 Node（节点类型），Name

（变量名称），LineNo（节点行号），Value（变量值），StateType（节点所在语句类型），Stats（节点所在语句）。 为解决重复代码导致的映射误差，利用变量所在语句和行号作为 key 值指向唯一一条语句。

9-13：如果 key 值已经在映射关系中，说明该语句中有其他变量被解析过，首先根据 key 值获取相应的语句管理节点。利用 CheckValid()函数对当前变量的数据类型合法性进行分析，主要依据当前变量值是否为数据类型，或者数字数组类型。如果结果为 False，则更新语句 TreeNode 节点为非法，之后不会再访问。

14-21:如果当前变量所在的语句首次被解析，根据解析获得的节点信息初始化新的 TreeNode 节点，并且利用 CheckValid()函数对变量进行分析，如果分析符合可优化条件则针对节点类型将 TreeNode 节点添加到块类型语句映射集合或普通语句映射集合中。

4.3.2 可优化代码段生成

可优化代码段生成模块的主要功能分为两部分。一部分利用代码段重构方法对块节点进行重构，生成符合优化条件的块结构语句。另一部分对可优化代码语句进行合并，生成目标可优化代码段。实现具体功能的代码段生成算法描述如下：

算法 2　代码段生成算法

```
符合要求的块类型代码列表: ValidBlockList
符合要求的语句代码列表:     ValidList
相邻语句代码段:             AdjacentList
代码段集合:                 ValidStatementList
1:    for   i = ValidBlockList.size()-1; i >= 0; i--   do
2:    AstNode node = ValidBlockList.get(i)
3:     if   IsChildNodeInBlock(node , ValidList ) == TRUE   then
4:              RemoveChildNodeInValidList()
5:              AddNodeToValidList()
6:         end if
7:    end for
8:    AstNode Next = NULL
9:    for   i = 0 ; i < ValidList.size()-1 ; i++   do
10:       if   Next != NULL && Next == ValidList.get(i)   then
```

```
11:             AdjacentList.put( Next )
12:       else
13:             ValidStatementList.put( AdjacentList )
14:             AdjacentList = ValidList.get(i)
15:       end if
16:       Next=ValidList.get(i).getNext()
17:   end for
18:   return ValidStatementList
```

ValllidBlockList 保存经过初步筛选的块结构节点，由于动态执行每次只能解析一条执行路径，所以 ValidBlockList 集合保存不完整控制流块结构语句，例如 if 条件语句中的判断语句，for 循环语句中的初始化语句等。ValidList 集合则保存变量类型符合 WebAssembly 执行要求的独立语句。ValidStatementList 集合保存最终符合优化条件的代码段。详细的算法描述如下：

1-7：依次访问保存块语句的 ValidBlockList 集合，并且判断块结构中所有语句是否都保存在 ValidList 集合中。如果所有语句都保存在 ValidList 集合中则说明块语句结构整体都符合优化规则，则将原本的语句从 ValidList 集合中删除防止重复添加，并且将该块语句添加到 ValidList 集合中，用来处理块结构嵌套的情况。

8-17：这部分代码实现代码段合并。依次访问保存符合优化规则语句的 ValidList 集合，并且通过 Next 指针指向抽象语法树中代码段的下一条语句。如果 Next 指针指向语句保存在 ValidList 集合中，则将 Next 指向语句添加到代码段中，否则将代码段语句的 AdjacentList 集合添加到保存合法代码段集合的 ValidStatementList 集合中。然后初始化新的 AdjacentList 集合，并且添加当前节点作为新代码段的头节点。

18：返回合法的代码段集合。

4.4 代码转化模块

代码转换模块由两部分组成，动态数据分析模块和代码转换编译模块。动态数据分析主要对代码段依赖的数据流进行分析[29]，用于构造状态转移和状态恢复功能。代码转换编译主要利用 JavaScript2C 模块将可优化 JavaScript 代码段转换成 C 语句，并且生成相应的 WebAssembly 模块。

4.4.1 动态数据分析模块

动态数据分析模块主要对目标代码段中的数据依赖[58]进行分析，具体需要获取代码段传入变量集合，代码段传出变量集合。获取数据集合后将访问方式修改为索引访问的形式。具体数据交互监测算法描述如下：

算法 3　数据交互监测算法

```
声明变量名集合:          VarNameList
声明变量节点集合：       VarNodeList
声明变量作用域集合:      VarScopeList
传入 C/C++环境变量集合:  VarNodeArr
记录节点和节点索引映射:  InputArrIndex
1:  if node instanceof Name then
2:      parent = node.getParent()
3:      for i = VarNameList.size() – 1 ; i >= 0 ; i--  do
4:          if node == VarsNameList.get(i) then
5:              if  checkScope(node, VarScopeList.get(i)) == True then
6:                  index=i
7:                  break
8:              end if
9:          end if
10: end for
11: VarDef = VarNodeList.get(index)
12: if VarDef !=NULL && VarDef.getline() < startline && node.getline() > startline
    then
13:     if VarNodeArr.contains(node) == False then
14:         VarNodeArr.add(node)
15:     end if
16:     InputArrIndex.put( node , VarNodeArr.indexOf( node ) )
17: end if
```

VarNameList，VarNodeList 和 VarScopeList 三个集合描述了在可优化代码段中声明变量的变量名信息，抽象语法树节点信息和所在作用域信息，保证了分析过程中变量作用域信息的正确性。InputArrIndex 保存了传入数据节点和线性内存索引的映射关系，VarNodeArr 保存了传出数据在线性内存的索引信息。具体的算法描述如下：

1 ：算法主要针对变量节点，首先对节点类型进行筛选，获取抽象语法树中的变量节点。

2-10 :通过遍历声明变量的名称集合 VarNameList 判断当前访问变量是否在可优化代码段中声明，如果是在代码段中声明则获取该变量声明节点。利用作用域信息确认当前访问节点的变量声明节点，如果确认则记录声明节点在数据传出数组中的索引号。

11: 通过索引获取当前变量的声明节点信息。

12-17:判断当前节点的声明位置是否属于可优化代码段，如果在范围内则判断是否已经将当前节点添加到传入数据中，如果没有则添加到传入数组中。最终将当前节点信息和传入数组索引的映射保存。

4.4.2 代码转换编译模块

代码转换编译模块主要将可优化代码段编译生成相应的 WebAssembly 模块，具体过程如图 16 所示，详细实现描述如下：

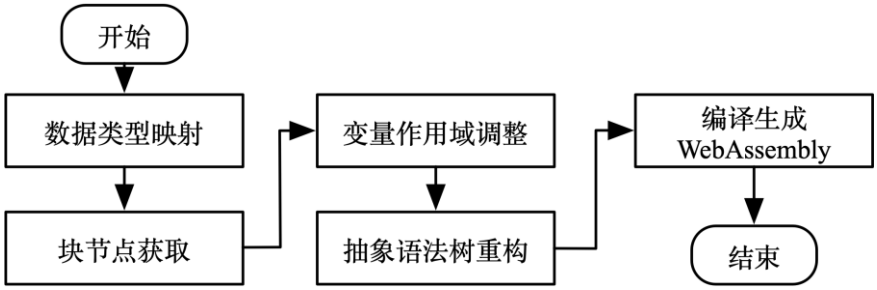


图 16　代码转换编译流程图

（1）利用变量类型解析模块获得的变量类型信息修改变量声明节点抽象语法树信息，在抽象语法树节点类中添加静态数据类型的描述属性。

（2）针对 JavaScript 函数级作用域和 C 块级作用域规则的区别，在抽象语法树中获取块级别的节点，例如 If 语句节点，For 循环语句节点等。

（3）通过在抽象语法树中获取的块结构节点，首先对各块结构中变量的作用域进行分析，并且按照先声明再调用的规则对变量声明节点的语法树位置进行调整。最后按照块级别作用域规则的可见性和生命周期对所有变量的声明和调用节点在抽象语法树中进行调整。

（4）按照 C 语言的语法规则重构抽象语法树，并且遍历重构后的抽象语法树输出转换成的 C 语言代码。

（5）利用 Emscripten 编译器编译生成 wasm 二进制格式文件。根据动态数据分

析模块获取的数据流传递信息在 JavaScript 文件中构建状态转移和状态恢复语句，并且添加 wasm 代码解析调用语句。

4.5 性能优化模块

性能优化模块主要分为两部分，动态性能比较模块和代码段合并模块。动态性能比较主要通过动态执行的方式比较 WebAssembly 模块和 JavaScript 代码的性能，并记录生成相应的性能标记。代码段合并主要以性能标记为基础，利用代码段合并算法生成 WebAssembly 模块合并策略，减少数据交互导致的性能开销。

4.5.1 动态性能比较模块

动态性能比较模块流程图如图 17 所示，具体实现步骤如下：

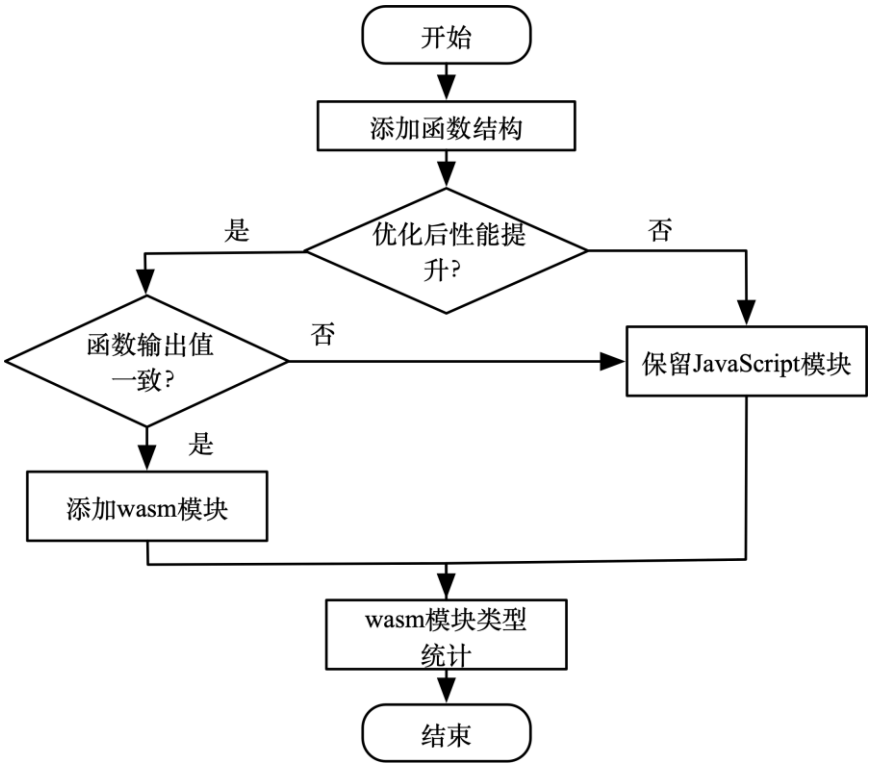


图 17　动态性能比较流程图

（1）利用单元测试的思想[59]，分别给 JavaScript 代码段和相应的 WebAssembly 模块添加函数体结构生成可运行测试单元。并且根据动态分析模块获取的变量信息设置单元模块运行所需的实参。

（2）利用动态执行的方式进行性能对比。我们利用 google 的 V8 引擎动态调用单元模块，并且在单元模块执行前后添加计时器。通过比较模块运行时间来评估性能。如果 WebAssembly 模块运行时间更短则在待优化脚本中利用 WebAssembly 模块替换

原 JavaScript 代码段，否则不进行处理。除此之外，根据性能比较结果生成模块性能标记，作为代码段合并依据。最后针对 WebAssembly 模块和 JavaScript 代码段输出值是否一致判断是否进行转换。

（3）对 WebAssembly 模块类型进行分析，判断 WebAssembly 模块属于函数体内模块还是函数体外模块，并且根据动态数据分析的结果生成数据依赖图，根据性能标记对数据依赖图中节点进行标记。

4.5.2 代码段合并算法

代码段合并算法以数据依赖图为基础，通过代码段合并更新性能标记，进而生成可靠的性能优化策略。具体的代码段合并算法如算法 4-1 和算法 4-2 描述：

算法 4-1　代码段合并调用算法

```
节点访问记录集合:          Visit
代码段分类标记变量：       Level
节点邻接表：               List
1:    Level=1
2:    for i = 0 ; i < e ; i++ do
3:        if Visit[i]==0 then
4:            Level=CodeSegmentMerge( List , i , Level , Visit )
5:        end if
6:    end for
```

Level 变量的主要功能是作为代码段分类标记。相同的 Level 值表示模块之间复合可合并规则，不同的 Level 值表示无法进行合并。并且利用 Level 值的奇偶性区分不同的合并规则。偶数值 Level 表示采用推荐合并规则，而奇数值 Level 表示可合并规则。代码段合并调用函数的功能在于依次访问未访问节点，更新 Level 值获取不同代码段的合并信息。

2-6:根据邻接表依次访问节点，如果节点未被访问过则调用 CodeSegmentMerge() 函数以当前节点为根节点访问数据流图，搜索可以进行合并的节点，并且更新 Level 变量。

代码段合并算法主要根据每个节点的性能标记，根据第三章提出的推荐合并规则对数据流图进行访问，并且更新性能标记和 Level 值，实现代码段合并策略的生成。具体算法实现由算法 4-2 描述：

算法 4-2 代码段合并算法

```
节点访问记录集合:          Visit
节点邻接表：               List
数据访问队列：             Queue
访问节点索引：             x
1:    Visit[ x ] = 1
2:    Queue.push( x )
3:    List[x].level = level
4:    If List[ x ].IsCaculate == False then
5:        return level+2
6:    end if
7:    While Queue.IsEmpty() == False do
8:        index = Queue.pop()
9:        HasParent = False
10:       for p = List[ index ].head ; p != NULL ; p = p->next do
11:   if   Visit[ p->index ] == 0 &&
      List[ p->index ].IsCaculate == True && p->input == 0   Then
12:                   HasParent=True
13:                   getAcient( visit , Queue , List , p->index , &tail )
14:                   Visit[ index ] = 2
15:         end if
16:         if HasParent == False do
17:             for p = List[ index ].head ; p != NULL ; p = p->next   do
18:                   if   visit[ p->index ] != 1 && p->input == 1
      && List[ p->index ].IsCaculate == True   then
19:                       ParentCondition = UpdateParentCondition( List , p->index )
20:                       if ParentCondition == 1 then
21:                           List[ index ].level = level + 1
22:                           List[ p->index ].level = level + 1
23:                           List[ p->index ].time = -1
24:                       else
25:                           List[ p->index ].level = level
```

```
26:                     end if
27:                     Visit[ p->index ] = 1
28:                     Queue.push( p->index )
29:                 end for
30:             end if
31:     end Loop
32:     return level+2
```

函数功能为利用 Level 变量对存在数据传递的代码段进行合并操作，具体功能表述如下：

1-3: 标记当前索引节点被访问，并且将节点加入到访问队列中。以初始化 Level 值更新节点的 Level 属性。

4-6: 如果节点的 IsCaculate 属性值不等于 1，则说明该节点属于不可优化代码段，所以无法进行合并操作。更新 Level 值，通过加 2 表示当前可合并代码段访问完毕，之后访问节点都不能合并入当前代码段中。

10-14：通过条件语句判断，如果存在其他 WebAssembly 模块有数据流传递给当前节点，则对当前节点进行处理。首先标记 HasParent 为 True，说明当前节点存在其他未访问前驱节点。并且将当前节点在 Visit 集合中设置为 2，主要是为了防止重复访问和忽略访问。getAcient()函数的主要功能是寻找节点的根流入节点，根流入节点就是存在数据传入到当前节点，但是不存在输入流或前驱节点为不可优化代码段的节点。将根流入节点加入到访问队列 Queue 中便于之后访问。

18-26:利用条件语句判断当前节点和后继节点能否进行合并，如果可以合并则判断两节点合并后性能是否一定会提升。首先利用 UpdateParentCondition()函数更新当前节点的前驱节点性能标记信息，如果该节点的所有前驱节点都是性能提升节点则当前节点也是性能提升节点，否则则为合并后不确定节点。对于性能提升节点，首先利用 Level+1 的奇数值更新 Level 属性，表示合并后性能一定会提升，并且更新当前节点的 time 属性表示该节点合并后为性能提升节点。

27-28: 标记后继节点为以访问，并且加入到访问队列中便于后续访问。

4.6 本章小结

本章主要讨论了基于 WebAssembly 的 JavaScript 性能优化研究方法的系统设计和

实现。首先介绍了系统主要模块，具体功能分别为任务管理模块，动态分析模块，类型解析模块，代码转换模块和性能优化模块。详细的讨论了类型解析模块中数据类型分析算法和代码段生成算法的设计和实现。代码转换模块中分析了数据交互监测算法的设计和代码段编译的执行流程。最后详细的描述了性能优化模块中代码段合并算法的设计与实现。

第五章 系统实验评估与分析

在原型系统 JSOPW 的基础上，本节选取多个测试用例，分别通过 JSOPW 和 Facebook 实现的 Prepack 优化工具进行优化处理。通过和现有优化工具的性能对比说明 JSOPW 优化的有效性。并且在不同浏览器和不同平台下进行性能评估，说明在不同场景下优化方案的有效性。

5.1 实验设计

本章描述实验环境和具体实验步骤，并且对测试用例及对比工具进行说明。

5.1.1 实验环境和实验步骤

本文的实验环境如下：

安卓平台：vivo Xplay6；系统 Android 6.0；处理器： 高通骁龙 820；内存 6GB。

IOS 平台：IPhone 8；系统：IOS11；处理器：苹果 A11；内存：3GB。

操作系统： macOS Mojava（版本 10.14.3）；处理器：Intel Core i7 3.3GHz；内存：16 GB; 实验过程所用到的浏览器：Chrome（64 位，版本 72.0.3626.119）、Safari（版本 12.0.3）和 Firefox Developer Edition（64 位 版本 66.0b8）。

具体的实验步骤设计如下：

（1）测试用例保护：选择 5 种实际场景的 JavaScript 应用程序，利用本文设计的 JSOPW 优化方法和 Prepack 优化工具对 5 种测试用例分别进行优化。

（2）时间性能评估：在对 5 种测试用例进行优化后，首先通过比较 JSOPW 优化方法和 Prepack 优化工具优化后测试用例运行时间来说明优化方法的有效性。除此之外，通过对比代码段合并算法前后数据交互时间和程序整体的执行时间说明代码段合并算法的有效性。最后针对不同平台和不同浏览器下测试用例的性能对比说明不同平台优化方法的有效性和可行性[60,61]。

（3）文件体积评估：通过比较测试用例优化前后体积的变化说明 JSOPW 方法的可应用性。

5.1.2 测试用例和对比工具介绍

在实验中，我们选取了五种不同的 JavaScript 实现的 Web 应用程序作为测试用例，主要分为三维图形处理，数字信号处理和计算应用三种类型，测试用例的具体信息在

表 2 种进行描述：

表 2　测试用例的基本信息

测试用例	简介
webgl_geometry	一个 3D 图形特效应用，渲染出旋转的染色体。优化代码主要针对该 Web 应用调用的 three.js 框架中的方法。
FFT	一个数字信号处理应用，利用 JavaScript 实现的 FFT 算法，来自 JavaScript Benchmark[62]。优化代码主要针对 web-dsp 数字信号处理框架中 FFT 相关算法。
DFT	一个数字信号处理应用，利用 JavaScript 实现的 DFT 算法，来自 JavaScript Benchmark。优化代码主要针对 web-dsp 数字信号处理框架中 DFT 相关算法。
N- Queen	n-皇后问题。计算矩阵大小从 8 到 12 的皇后问题的解法个数。针对 Prepack 的优化特点，优化代码为立即执行代码和非立即执行代码。
Quick-Sort	快速排序问题。统计一百万次对包含 100 个数字的数组进行快速排序所用的时间。针对 Prepack 的优化特点将代码分为立即执行代码和非立即执行代码。

我们利用 Facebook 设计的 Prepack 优化工具进行对比，Prepack 的优化方式主要利用预处理对目标代码中可简化代码段进行处理，进而达到优化程序性能的目的。

5.2 性能评估

本章通过和对比工具性能评测，代码段合并前后性能比较、多浏览器运行性能比较、多平台性能比较和测试用例优化前后空间体积变化说明优化方案的可行性。

5.2.1 运行时间对比

图 18 描述了七种测试用例在 JSOPW 和 Prepack 两种优化工具后相比源码的性能

提升率。其中 N-Queen_e 和 Quick-Sort_e 两个测试用例和 N-Queen 和 Quick-Sort 实现相同的功能，但是 N-Queen_e 和 Quick-Sort_e 优化的代码段为立即执行函数。立即执行函数可以起到隔离作用域的目的，并且在声明后可以立即执行，不需要调用过程。

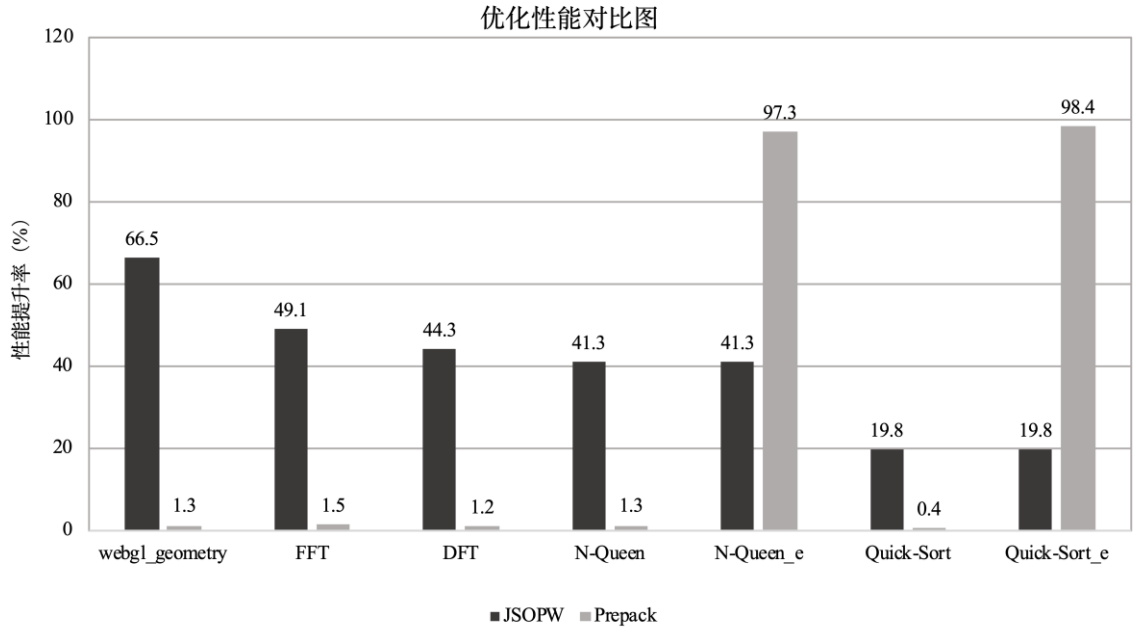


图 18　JSOPW 和 Prepack 性能对比图

从图中可以看出，通过 JSOPW 优化方法优化的测试用例都有性能提升，并且以 webgl_geometry，FFT，DFT，N-Queen 和 N-Queen_e 的性能提升较大。而与之相比的 Prepack 的优化效果只有针对立即执行函数类型的测试用例时才有非常明显的效果，但是针对无法直接运行的测试用例其优化效果并不明显。这主要是因为 Prepack 采用的优化方案类似于预处理，通过强制执行的方式在运行过程中利用部分运行结果替换原本的代码表达式，进而达到优化的目的，但是这就使得优化规则的应用范围较小，针对无法预处理的测试用例就无法起到优化的目的，在 Prepack 的官方描述中称其目前只能应用于特定场景，优化立即执行的代码。并且 Prepack 只能优化那些执行后仍然可以通过全局对象访问的数值，但是在实际的开发中并不提倡使用全局变量，因为会导致占用更多内存单元，破坏函数封装性等问题，所以 Prepack 目前在实际测试用例中的应用范围很受限。而 JSOPW 的优化方式是利用 WebAssembly 高效的执行效率重新实现 JavaScript 中的部分代码。所以在与性能相关的计算密集型的测试用例中 JSOPW 具有更广泛的应用场景。并且 JSOPW 的优化规则和 Prepack 的优化策略并不冲突，在运行性能要求较高的应用程序中，利用将 Prepack 优化后的应用程序利用 JSOPW 进行再次优化，进而达到更高效的执行性能。

5.2.2 代码段合并性能比较

图 19 中描述了测试用例进行代码段合并前后运行时间和数据交互时间的对比情况。我们主要针对 webgl_geometry，FFT 和 DFT 这三种代码结构较为复杂的应用程序进行性能比较，其中测试用例名称后的 B 表示代码段合并前，A 表示代码段合并后。

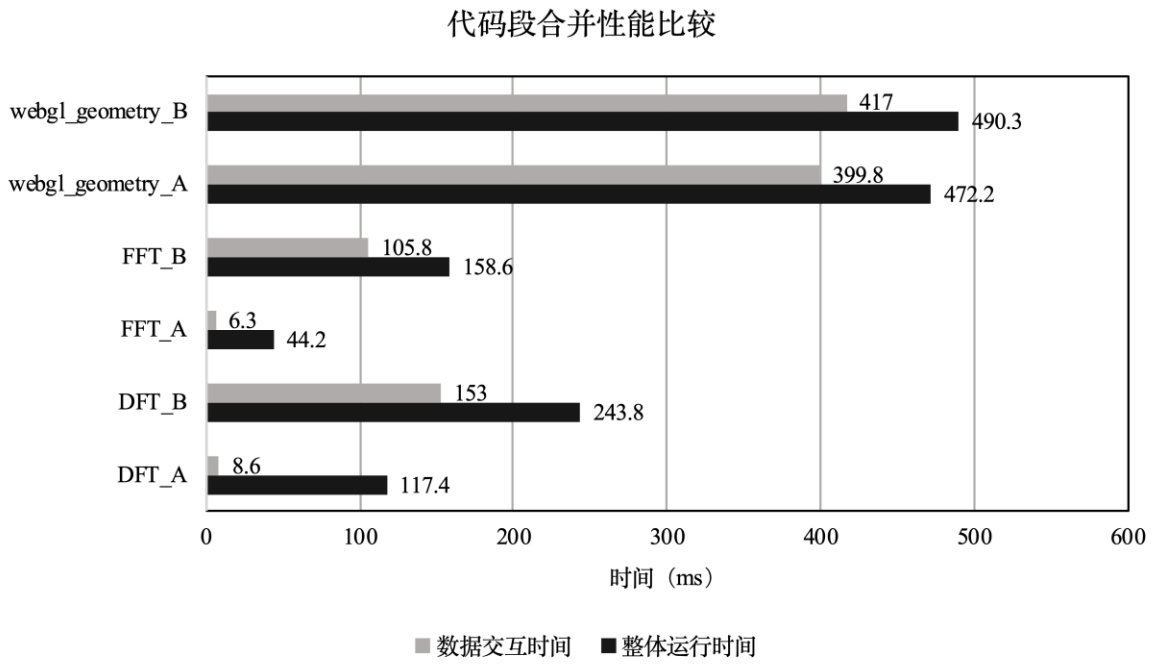


图 19　代码段合并算法效果图

通过图 19 的表述，经过代码段合并后数据交互时间都存在下降的情况，并且程序的整体运行时间也有缩短。这说明了代码段合并算法对于性能开销的减小具有有效性。但是由于合并算法仅针对可合并模块，所以部分测试用例通过合并后性能提升有限。以 webgl_geometry 为例，测试用例在代码段合并后无论是程序的整体运行时间还是数据交互时间的减少量都非常有限，说明程序中可合并模块少，无法通过合并减少数据交互导致的性能开销。而 FFT 算法通过合并后程序整体以 WebAssembly 形式运行，消除了 WebAssembly 模块与 JavaScript 代码的大部分数据交互，提升了程序整体的运行性能。所以通过实验表明，通过代码段合并算法可以有效的减少数据交互的性能开销，提升程序的执行效率。

5.2.3 浏览器性能比较

图 20 描述了五种测试用例在不同浏览器下的优化率。我们主要针对 Chrome，Firefox 和 Safari 三种浏览器进行对比。我们将测试用例在不同浏览器中执行，统计各

测试用例优化前后的性能提升率。这主要是因为各浏览器厂商针对 WebAssembly 性能优化策略不同，导致 WebAssembly 应用程序的执行性能不同。

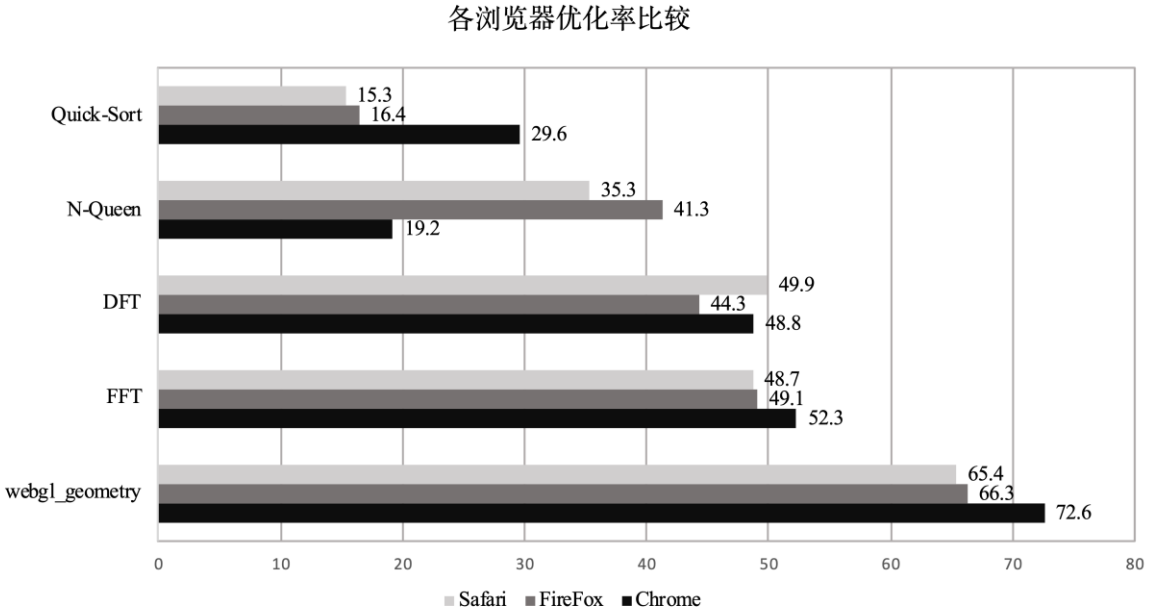


图 20　各浏览器性能提升率比较图

通过图中的描述，优化后的测试用例在三种浏览器中执行性能都有所增强，并且优化率基本一致。Webgl_geometry，FFT 和 DFT 这种以大量数据计算为核心的测试用例来说性能提升比较明显，而 N-Queen 和 Quick-Sort 这些以数据交换为核心的测试用例优化率较小。整体上 Chrome 的性能提升率较高，这说明 Chrome 浏览器上 WebAssembly 和 JavaScript 代码执行性能差距较大。通过实验可知，虽然各浏览器对 WebAssembly 代码采用的优化规则不同，但是通过 JSOPW 对 JavaScript 代码进行优化在各浏览器都有较为可观的效果。

5.2.4 移动端性能比较

图 1 中所描述 2018 年 JavaScript 代码在可运行硬件中执行性能比较，可以知道采用 A11 处理器的 IPhone 手机拥有最高效的执行效率。所以我们在性能资源较高的移动端平台进行性能比较，图 21 描述了在 IOS 平台下测试用例优化前后的运行时间比较。通过 JSOPW 对测试用例优化后测试用例的整体执行时间减少，经过计算后得知测试用例的平均优化率在 20.7%。所以通过实验可以得知，优化方法在 IOS 这样性能资源充沛的移动端平台也具有有效性。

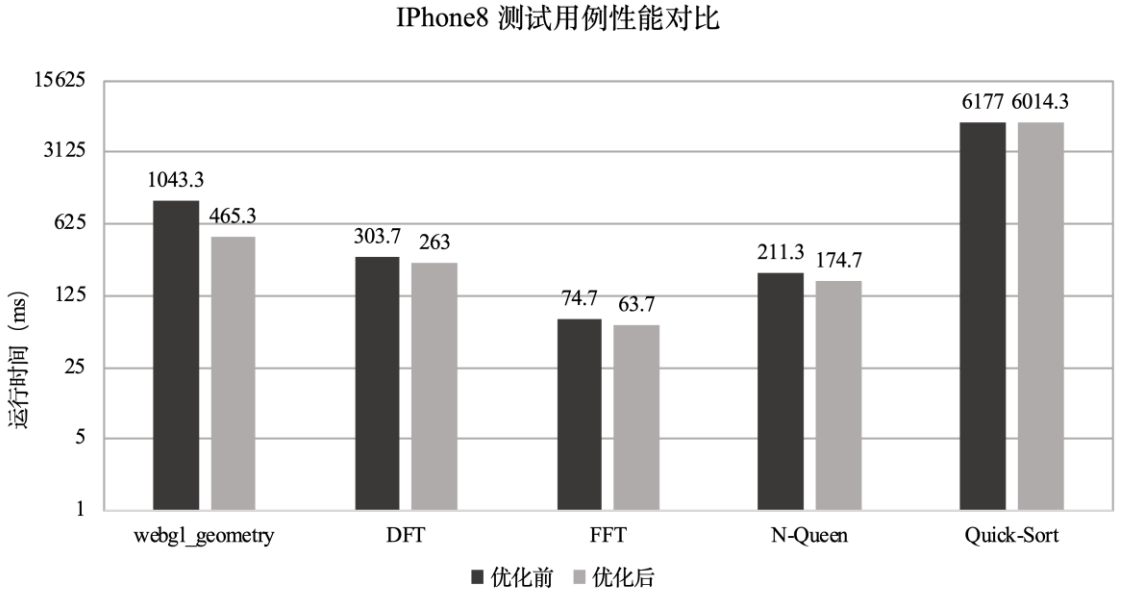


图 21　IOS 平台性能比较图

图 22 描述了在安卓平台下测试用例在优化前后运行时间的比较。经过统计可知所有测试用例在 IPhone8 中的运行时间比 VIVO 手机中快 72.6%。所以手机配置的差别对于 JavaScript 的运行效率具有很大影响。通过实验可以看出通过优化后所有测试用例的运行时间都有减少，并且平均优化率达到 31.46%，相比 IPhone 中性能提升更多，这也说明对低配置平台来说性能优化更加具有意义。综上所述，JSOPW 优化方法在低配置手机下也可以提供有效的性能优化。

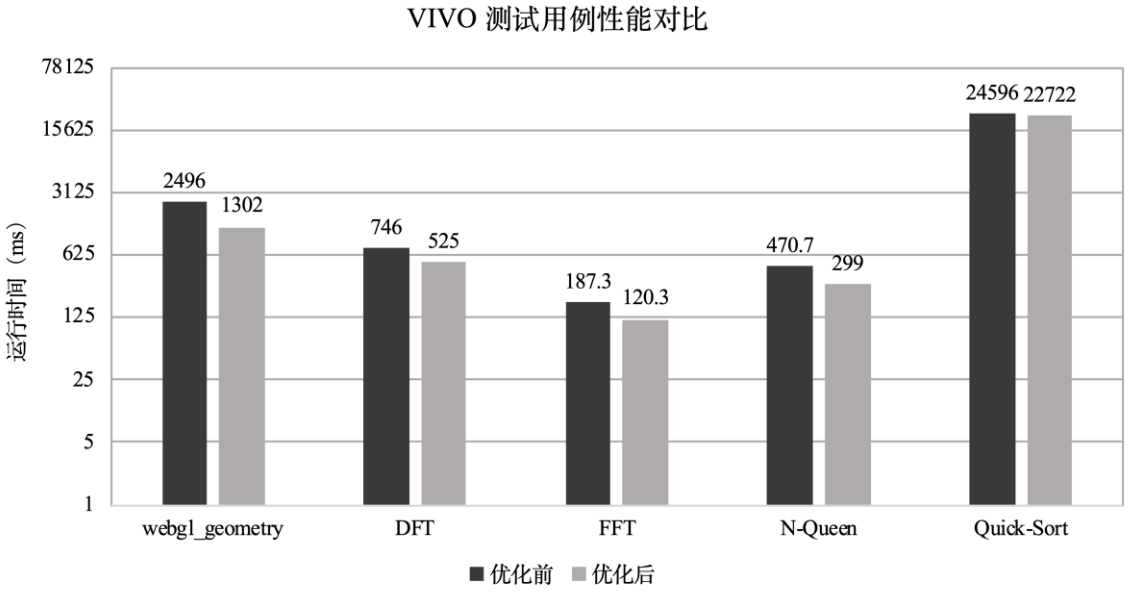


图 22　Android 平台性能比较图

5.2.5 空间大小比较

由于 JavaScript 代码多用于网络环境中，所以体积的变化也是关注的重点之一。我们对测试用例优化前后的体积变化进行统计。实验中仅针对优化代码段所在的文件进行统计，具体实验数据如表 3 所示。

表 3 优化后体积变化（KB）

测试用例	优化前	优化后	增长率
webgl_geometry	1065.39	1080.61	1.42%
FFT	222.32	224.72	1.07%
DFT	237.74	240.09	0.99%
N-Queen	0.64	1.72	168.75%
Quick-Sort	0.95	2.13	124.21%

通过实验可知，测试用例在经过优化后都存在体积膨胀的情况，这主要是因为在转换为 WebAssembly 之后，需要添加 WebAssembly 文件的解析代码，实例化代码，调用代码等，并且还要根据数据交互添加不定量的内存读写代码等。所以在进行优化后会存在体积膨胀的情况。但是通过实验可以看出，随着测试用例体积的增大，优化后程序的体积膨胀率减小，这主要是因为 WebAssembly 二进制格式相比 JavaScript 具有更小的体积，而且相关调用代码等体积十分有限，所以随着待优化代码段体积的增加所占比例会逐渐减少。通过 N-Queen 和 Quick-Sort 两个测试用例可以看出虽然体积增长率很高，但是增加的代码量仅仅在 1K 左右。所以在实际场景下 JSOPW 优化方案造成的体积膨胀在可接受范围内。

5.3 本章小结

本章主要通过运行时间和空间体积两方面对 JSOPW 优化方案进行评估。首先利用 JSOPW 和现有优化工具 Prepack 对实际 Web 应用程序进行优化，针对运行时间进行评估。然后利用测试用例在代码段合并前后的运行时间和数据交互时间的对比说明代码段合并规则的有效性。通过在不同浏览器和不同平台下测试用例的性能对比，说明多场景下 JSOPW 优化方案的有效性。最后利用体积膨胀率说明优化方案的可用性。通过上述实验可以说明 JSOPW 优化方案在各场景下具有有效的优化效果。

总结与展望

总结

随着 Web 应用程序的多元化，JavaScript 所承担的作用也不限于简单的 DOM 交互，更需要为例如图像处理，虚拟现实和游戏引擎等计算密集型的应用程序提供服务。然而 JavaScript 代码的设计以易用性为主，所以面对功能日益复杂的 Web 应用程序其性能上的缺陷逐渐显现出来。特别是对于配置资源有限的平台，高性能的执行效率显得更加重要。本文通过研究当前主流的 JavaScript 性能优化方法的特征和所存在的局限性，将 WebAssembly 这样的二进制代码格式应用于 JavaScript 代码性能优化中。并且为了保证语法规则和语言功能的契合，设计了相应的代码转换模块。与此同时为减少 WebAssembly 嵌入 JavaScript 中所带来的额外数据交互开销，设计了代码段合并规方法。除此之外，在研究的基础上实现了原型系统 JSOPW，并且通过设计性能分析实验来评估本文设计方法在各应用场景的有效性和实用性。

本文主要的研究工作总结如下：

（1）JavaScript 代码性能优化方法的研究

对 JavaScript 性能问题进行深入研究，分析产生性能问题的原因。并且对当前 JavaScript 代码优化方案进行研究分析，包括编译技术和代码类型模拟等多种主要的优化方案。详细分析了这些优化方案的基本实现原理，并且结合产生 JavaScript 性能缺陷的原因分析优化方案的缺陷。

（2）基于 WebAssembly 优化方案的研究。

分析利用 WebAssembly 对 JavaScript 进行性能优化方案的可行性。通过分析 WebAssembly 的适用范围和设计特点，提出了一个 JavaScript 代码性能优化方案，并且讨论了该方案的实现原理。首先根据 WebAssembly 的适用范围和语言特性，设计 JavaScript 代码段筛选规则以保证 WebAssembly 功能和语法可覆盖目标代码段。设计代码转换方法将目标代码段转换生成相应的 WebAssembly 模块。通过对 WebAssembly 模块和 JavaScript 代码数据交互性能开销的研究，设计并提出代码段合并方法。

（3）设计并实现基于 WebAssembly 的 JavaScript 性能优化系统。

针对本文提出的方法设计并实现原型优化系统 JSOPW，主要介绍了类型解析，代码转换和性能优化等主要模块，并且描述了各个模块中关键算法及主要流程的设计

实现。选择多个实际 Web 应用程序作为测试用例，同时利用 Facebook 设计的 Prepack 优化方案进行对比，收集测试用例在优化前后的时空开销。并且针对多浏览器多平台验证优化方案的通用性，通过对比评估实验验证优化方案的有效性。

展望

本文提出的基于 WebAssembly 的 JavaScript 代码性能优化方案在一定程度上可以提高 JavaScript 的执行性能，但是目前仍然存在一些缺陷，具体有以下几个方面：

（1）目前本文中采用的方法是将 JavaScript 中部分代码转换成 C 语言的形式，所以对于 JavaScript 代码规则无法完全模拟实现，这也使得可转换代码类型受限并且筛选出的目标代码段较为零散。为了解决这样的问题可以利用 C/C++或 Rust 设计 WebAssembly 支持的可以模拟更多 JavaScript 语法规则和功能的接口，使得可优化代码更具有普遍性。

（2）在 WebAssembly 模块的调用过程中，为了保证程序运行的稳定性和数据流的正确性，在 JavaScript 中添加了状态转移模块和状态恢复模块。这两部分对于线性内存频繁的读写造成了额外的性能开销。在当前 WebAssembly 技术的发展中，其应用更有利于处理大量的计算类型并且进行统一输入输出的场景。虽然我们针对这部分采用了代码段合并的方式来统一部分代码段的数据交互，但是由于语法规则和数据依赖关系的限制导致代码段合并规则无法解决所有的情况。所以针对这部分计划对代码模块进行更细致的分析，进而设计出可以更为普遍的代码段合并方法。

（3）虽然 WebAssembly 由浏览器厂商合作设计推出，但是通过实验可以看出各浏览器厂商针对 WebAssembly 的优化规则也不同，并且不同平台也有性能的差异。所以后期尝试针对不同浏览器和平台选择不同的优化策略，并且可以引入其他的优化方案，设计出可以根据不同场景采用不同优化策略的整体优化方案，使得优化方法更加具有针对性。

参考文献

[1] Haas A, Rossberg A, Schuff D L, et al. Bringing the web up to speed with WebAssembly[C] //ACM SIGPLAN Notices. ACM, 2017, 52(6): 185-200.

[2] ammo.js, Direct port of the Bullet physics engine to JavaScript using Emscripten [EB/OL]. https://github.com/kripken/ammo.js/

[3] Zakai A . Emscripten: an LLVM-to-JavaScript compiler.[C]// Acm International Conference Companion on Object Oriented Programming Systems Languages & Applications Companion. ACM, 2011.

[4] Zakai A. Fast Physics on the Web Using C++, JavaScript, and Emscripten[J]. Computing in Science & Engineering, 2018, 20(1): 11-19.

[5] three.js, Javascript 3D library [EB/OL]. https://threejs.org

[6] DSP.js, JSter Javascript Catalog [EB/OL]. http://jster.net/library/dsp-js

[7] A-Frame – Make WebVR [EB/OL]. https://aframe.io

[8] Hackett B, Guo S. Fast and precise hybrid type inference for JavaScript[J]. ACM SIGPLAN Notices, 2012, 47(6): 239-250.

[9] Grimmer M, Seaton C, Schatz R, et al. High-performance cross-language interoperability in a multi-language runtime[C]//ACM SIGPLAN Notices. ACM, 2015, 51(2): 78-90.

[10] Roman Zhuykov, Eugene Sharygin, “Ahead of time optimization for JavaScript programs”, Proceedings of ISP RAS, 27:6 (2015), 67–86

[11] Park H, Kim S, Park J G, et al. Reusing the optimized code for JavaScript ahead-of-time compilation[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2018, 15(4): 54.

[12] Van Es N, Stievenart Q, Nicolay J, et al. Implementing a performant scheme interpreter for the web in asm. js[J]. Computer Languages, Systems & Structures, 2017, 49: 62-81.

[13] Martinsen J K, Grahn H, Isberg A. Using Thread-Level Speculation to Enhance JavaScript Performance in Web Applications[J]. HPI Future SOC Lab: Proceedings 2013, 2015, 88: 111.

[14] Martinsen J K , Grahn H , Isberg A . Using speculation to enhance javascript performance in web applications[J]. IEEE Internet Computing, 2013, 17(2):10-19.

[15] Li S , Cheng B , Li X F . TypeCastor: Demystify dynamic typing of JavaScript applications[C]// High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings. DBLP, 2011.

[16] Park J T, Kim H G, Moon I Y. WebAssembly Performance Analysis for Multimedia Processing in Web Environment[C]// International conference on future information&communication engineering. 2018, 10(1): 288-290.

[17] Watt C. Mechanising and verifying the WebAssembly specification[C]//Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM, 2018: 53-65.

[18] Matsakis N D, Klock II F S. The rust language[C]//ACM SIGAda Ada Letters. ACM, 2014, 34(3): 103-104.

[19] Møller A. Technical perspective: WebAssembly: a quiet revolution of the web[J]. Communications of the ACM, 2018, 61(12): 106-106.

[20] Jangda A, Powers B, Guha A, et al. Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code[J]. arXiv preprint arXiv:1901.09056, 2019.

[21] Letz S, Orlarey Y, Fober D. FAUST Domain Specific Audio DSP Language Compiled to WebAssembly[C]//Companion of the The Web Conference 2018 on The Web Conference 2018. International World Wide Web Conferences Steering Committee, 2018: 701-709.

[22] Finney R, Meerzaman D. Chromatic: WebAssembly-Based Cancer Genome Viewer[J]. Cancer informatics, 2018, 17: 1176935118771972.

[23] Herrera D, Chen H, Lavoie E, et al. WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices[R]. Technical Report. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University, 2018.

[24] Jensen S H , Anders Møller, Thiemann P . Type Analysis for JavaScript[C]// Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. Springer-Verlag, 2009.

[25] Bierman G, Abadi M, Torgersen M. Understanding typescript[C]//European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 2014: 257-281.

[26] Neamtiu I, Foster J S, Hicks M. Understanding source code evolution using abstract syntax tree matching[J]. ACM SIGSOFT Software Engineering Notes, 2005, 30(4): 1-5.

[27] Zhuykov R, Vardanyan V, Melnik D, et al. Augmenting JavaScript JIT with ahead-of-time compilation[C]//2015 Computer Science and Information Technologies (CSIT). IEEE, 2015: 116-120.

[28] Auler R, Borin E, de Halleux P, et al. Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler[C]//International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 2014: 218-237.

[29] Jeon S, Choi J. Reuse of JIT compiled code in JavaScript engine[C]//Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, 2012: 1840-1842.

[30] Beckman L, Haraldson A, Oskarsson Ö, et al. A partial evaluator, and its use as a programming tool[J]. Artificial Intelligence, 1976, 7(4): 319-357.

[31] Sümeyye Süslü, Csallner C . SPEjs: a symbolic partial evaluator for JavaScript[C]// the 1st International Workshop. 2018.

[32] Prepack, Partial evaluator for JavaScript[EB/OL]. https://prepack.io

[33] Reiser M, Bläser L. Accelerate JavaScript applications by cross-compiling to WebAssembly[C]//Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages. ACM, 2017: 10-17.

[34] Selakovic M, Pradel M. Performance issues and optimizations in JavaScript: an empirical study[C]//Proceedings of the 38th International Conference on Software Engineering. ACM, 2016: 61-72.

[35] Schroepfer A, Kerschbaum F. secure computation in JavaScript[C]//Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011: 849-852.

[36] Lee S W , Moon S M , Jung W K , et al. Code size and performance optimization for mobile JavaScript just-in-time compiler[C]// Workshop on Interaction Between Compilers & Computer Architecture. ACM, 2010.

[37] Ahn W , Choi J , Shull T , et al. Improving JavaScript performance by deconstructing the type system[J]. ACM SIGPLAN Notices, 2014, 49(6):496-507.

[38] Heo J, Woo S, Jang H, et al. Improving JavaScript performance via efficient in-memory bytecode caching[C]//2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia). IEEE, 2016: 1-4.

[39] Suganuma T, Yasue T, Kawahito M, et al. Design and evaluation of dynamic optimizations for a Java just-in-time compiler[J]. Acm Transactions on Programming Languages & Systems, 2005, 27(4):732-785.

[40] Krall A . Efficient JavaVM Just-in-Time Compilation[C]// International Conference on Parallel Architectures & Compilation Techniques. IEEE, 1998.

[41] Misek J , Zavoral F . Semantic analysis of ambiguous types in dynamic languages[J]. Journal of Ambient Intelligence & Humanized Computing, 2018(3):1-8.

[42] Arceri V, Preda M D, Giacobazzi R, et al. SEA: String Executability Analysis by Abstract Interpretation[J]. arXiv preprint arXiv:1702.02406, 2017.

[43] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004: 75.

[44] Heil S, Siegert V, Gaedke M. ReWaMP: Rapid Web Migration Prototyping Leveraging WebAssembly[C]//International Conference on Web Engineering. Springer, Cham, 2018: 84-92.

[45] Guha A, Saftoiu C, Krishnamurthi S. The essence of JavaScript[C]//European conference on Object-oriented programming. Springer, Berlin, Heidelberg, 2010: 126-150.

[46] Rossberg A. WebAssembly: high speed at low cost for everyone[C]//ML16: Proceedings of the 2016 ACM SIGPLAN Workshop on ML. 2016.

[47] Egret Engine[EB/OL].https://github.com/egret-labs/egret-core/

[48] Unity, the world's leading real-time engine[EB/OL]. https://unity3d.com

[49] Magnum[EB/OL].https://github.com/mosra/magnum

[50] Guo S , Kusano M , Wang C . Conc-iSE: Incremental Symbolic Execution of Concurrent Software[C]// IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016.

[51] Loring B, Mitchell D, Kinder J. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript[J]. arXiv preprint arXiv:1810.05661, 2018.

[52] Bucur S , Kinder J , Candea G . Prototyping symbolic execution engines for interpreted languages[J]. Acm Sigplan Notices, 2014, 42(1):239-254.

[53] Rhino: JavaScript in Java[EB/OL].https://github.com/mozilla/rhino

[54] 汤战勇, 王怀军, 房鼎益,等. 基于精简指令集的软件保护虚拟机技术研究与实现[J]. 微电子学与计算机, 2011, 28(8):1-3.

[55] 汤战勇, 李光辉, 房鼎益,等. 一种具有指令集随机化的代码虚拟化保护系统[J]. 华中科技大学学报(自然科学版), 2016, 44(3):28-33.

[56] Loring B, Mitchell D, Kinder J. ExpoSE: practical symbolic execution of standalone JavaScript[C]//Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. ACM, 2017: 196-199.

[57] Sen K , Necula G , Gong L , et al. MultiSE: Multi-path Symbolic Execution using Value Summaries[C]// Esec/fse Joint Meeting on Foundations of Software Engineering. 2015.

[58] Mohaqeqi M , Abdullah J , Yi W . Modeling and Analysis of Data Flow Graphs Using the Digraph Real-Time Task Model[C]// Ada-Europe International Conference on Reliable Software Technologies. Springer International Publishing, 2016.

[59] Mackinnon T, Freeman S, Craig P. Endo-testing: unit testing with mock objects[J]. Extreme programming examined, 2000: 287-301.

[60] Jacobsson M, Wåhslén J. Virtual machine execution for wearables based on WebAssembly[C]//13th EAI International Conference on Body Area Networks (BodyNets 2018). 2018.

[61] Herrera D, Chen H, Lavoie E, et al. WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices[R]. Technical Report. Technical

report SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University, 2018.

[62] Kraken JavaScript Benchmark [EB/OL]. https://krakenbenchmark.mozilla.org

致谢

岁月如歌，光阴似箭。回首三年研究生生活，感慨万千。在毕业论文即将完成的时候，我要对那些曾经帮助过我的人道一声感谢。

首先要感谢我的导师汤战勇。感谢汤老师让我的研究生三年变得充实，从研零刚进入实验室就可以接触到软件安全的相关实验，并且参加相关会议了解最新的网络攻防技术，这使得我可以对安全领域有全面的认知。研一入学很幸运的投入到和腾讯的前端对抗项目中，这是研究生三年中压力最大的一年，每月的三方会议，每周无数次的需求讨论和验收。在这个过程中使我感受到在学校里感受不到的工作节奏和技术需求，这也使我不断提高自己的抗压能力和解决问题能力。到了研三，重心从项目转移到科研，感谢汤老师耐心的指导我如何用科研的思路去解决问题，并且更深层次的挖掘解决方案的科学意义。虽然在科学研究方面道行尚浅，但是这研究的过程使得我能够更加深刻全面的看待问题，这是我三年来最大的收获。感谢房鼎益老师，房老师严谨的态度使我十分钦佩，在毕业论文的研究过程中，感谢房老师对我研究方向和研究思路的指正，并且结合实际应用的分析使我对研究内容有更加深入的理解。感谢陈晓江老师，很幸运在研究生三年里遇到陈老师，通过陈老师让我见识到学术界的大牛如何发现问题和解决问题，并且可以了解到最前沿的研究内容。感谢王薇老师，感谢王老师耐心的指导我明确论文的研究思路，并且提出细致的修改意见。

其次感谢家人对我的支持。感谢父母对我无微不至的照顾，并且作为我出去闯荡的坚强后盾。感谢我的女朋友，这么多年对我的支持，并且督促我减肥。

感谢我的学弟王帅，曹帅和常原海。感谢你们在腾讯项目时对我工作的支持，任劳任怨的完成枯燥的实验，并且耐心细致的完成项目的测试和后续工程。希望你们在科研和工作方面都有好的发展。感谢我的舍友冯超，廉英浩、张晓阳、李振、田超雄，为我分担压力。

感谢国家自然科学基金项目（61672427）和陕西省国际科技合作与交流计划（2017KW-008，2019KW-009）对本文的资助，并且感谢爱迪德公司这三年对我科研上的支持。

感谢这些年的经历使我成长，祝福所有人一切顺利。

攻读硕士学位期间取得的科研成果

1. 发表学术论文

[1] Xue C, Tang Z, Ye G, et al. Exploiting Code Diversity to Enhance Code Virtualization Protection[C]//2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2018: 620-627.

2. 申请（授权）专利

[1] 汤战勇，薛超，王帅等. 一种基于前端字节码技术的 JavaScript 虚拟化保护方法: 中国，201810446970.2[P]. 2018 年 5 月 11 日

3. 参与科研项目

[1] 国家自然科学基金，基于时频域指纹映射的中国书法运笔姿态与动作的无线感知与识别方法（61672427）。

[2] 陕西省重点研发计划-国际合作项目，大规模跨场景条件下无线深度感知与行为识别关键技术研究 （2019KW-009）。

[3] 陕西省重点研发计划-国际合作项目，基于攻击建模的软件动态保护方法及其有效性评测研究（2017KW-008）。

并且需要浏览器设计相应的优化模块，所以其性能优化的普遍性存在局限。

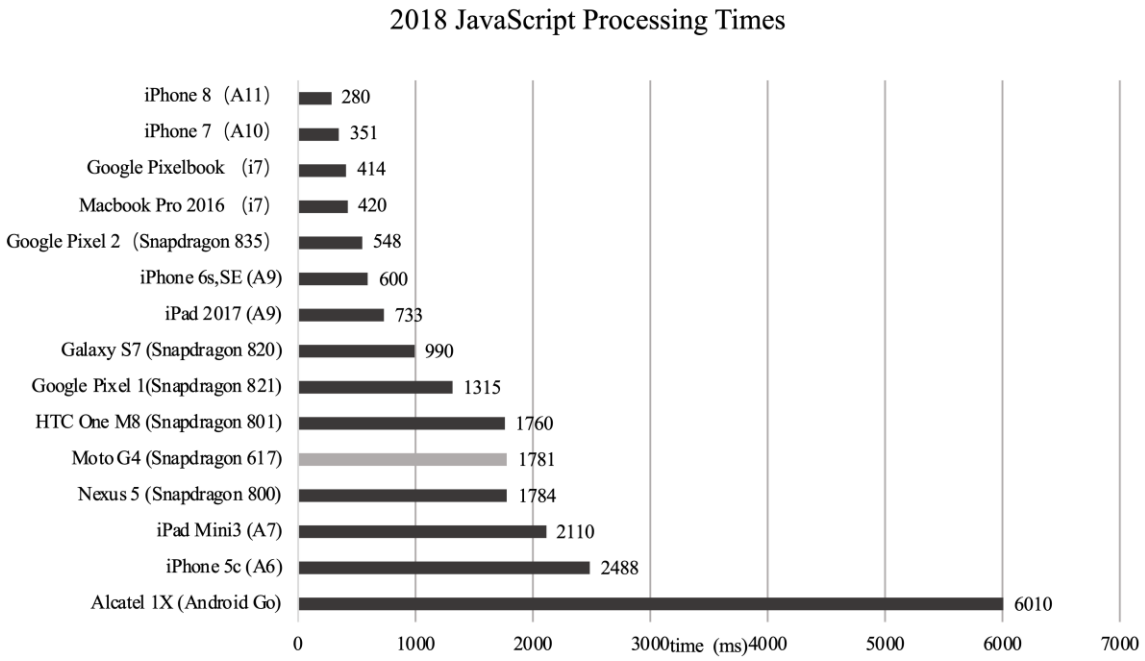


图 1　2018 可用硬件 JavaScript 程序运行时间统计

针对目前 JavaScript 代码优化方案的缺陷，我们提出了一种基于 WebAssembly[1] 的 JavaScript 代码性能优化方案。该方案基于 WebAssembly 这样一种通用且高效的二进制代码格式，将 JavaScript 中与性能相关的计算密集型代码提取并转换成 WebAssembly 模块的形式，并且设计调用模块保证程序运行的稳定性和数据流的正确性。然后针对 JavaScript 代码和 WebAssembly 模块数据交互性能开销的问题设计代码段合并方法，通过将 WebAssembly 代码段合并来减少数据交互次数，进而减少运行开销。通过这种方式对 JavaScript 中与性能相关的计算密集型代码进行优化。

1.2 国内外研究现状

1.2.1 JavaScript 代码优化

在 Web 程序应用日趋广泛的今天，JavaScript 也扮演越来越重要的角色。从之前实现简单交互的验证到如今实现三维图形渲染，游戏引擎等更为复杂的处理工作。虽然在这个过程中人们尝试利用插件等方式在 Web 端引入更高效的开发语言，但是至今 JavaScript 仍然保持着非凡的垄断地位，而基于 JavaScript 实现的应用程序功能也更加复杂。例如，Ammo.js[2]是一个功能强大的 Web 端物理引擎。它拥有连续碰撞检测，自定义图形和多方式场景调节等功能。它是以 Bullet 物理引擎库通过 Emscripten[3,4] 移植到 Web 端实现的，这使得 Ammo.js 可以作为 Web 端最完整的物理引擎之一。

Three.js[5]是以 webgl 为基础的渲染库，封装了 3D 渲染中重要的方法。通过 Three.js 可以在 Web 端创建和展示炫彩的三维计算机图形。DSP.js[6]是基于 JavaScript 的综合数字信号处理库。它包括振荡器，FFT 和 DFT 变换等大量的的数字信号分析和生成功能。A-Frame[7]是一个在 Web 端实现虚拟现实应用的框架，利用 HTML 和实体组件创建多平台的 VR 应用程序。

由于 JavaScript 承担功能的日趋复杂，也越来越多的人将研究目标转向 JavaScript 性能的提升。Hackett[8]等人提出快速准确的类型推导算法来解决性能上动态类型的约束。该算法通过增加运行时类型限定的静态分析方法。生成能够紧密反应程序实际行为的信息，进而保证类型推断的准确性。通过这种方式提升 JavaScript 程序运行的效率。Grimmer[9]等人提出了高性能的跨语言优化方案。这是一种运行时允许多种语言无缝切换的优化方案。它允许程序员使用 JavaScript 代码的表示方法合法的访问外部函数或者对象，从而将所需要的样板代码减少到最少，并且可以在不同的语言中使用最符合要求的数据结构，进而达到高效的运行效率。Zhuykov [10]等人针对 JavaScript 引擎使用即时编译器（JIT compiler，just-in-time compiler）优化相比静态编译复杂性有限[11]，提出了具有 AOT（Ahead of time）编译功能的多层 JIT 编译器方法，并且在两个 JavaScript 引擎（JavaScriptCore 和 V8）中实现了一些提前编译思想。Es[12]等人设计实现的 Asm.js 规范定义了一个可优化的 JavaScript 子集，它已经很好地作为与性能相关的 Web 应用程序的编译目标。Martinsen[13]等人在 JavaScript 引擎中利用线程级推测[14]实现利用多核处理器提高并行处理性能。Li[15]等人设计提出了 TypeCastor，这是一个新的 JavaScript 引擎，使用了分阶段编译和优化模型。在第一阶段，使用静态编译执行高级和复杂的优化。在第二阶段，低级别和平台特定的优化在运行时在执行引擎中进行。通过这种改进提升 JavaScript 的执行性能。

1.2.2 WebAssembly

WebAssembly 是一种新型可运行在 Web 端的二进制代码格式，它不仅能保证代码执行的安全性，更重要的是为 Web 提供了接近本地层代码的执行速度[16,17]。并且可以作为 C/C++，Rust[18]等代码的编译目标，使得不同的代码运行在 Web 端，提升了执行性能，而且使得原本难以移植到 Web 上的应用程序稳定的在浏览器中运行。由于 WebAssembly 是谷歌，苹果，火狐和微软为解决性能问题共同提出的解决方案，这保证 WebAssembly 在各浏览器中的通用性[19]。Emscripten 框架也提供了编译的功能，使得用户可以直接将 C/C++等代码编译成 WebAssembly 模块，提高了开发的便

捷性。

虽然 WebAssembly 仍然处于发展阶段，但是针对 WebAssembly 的研究和应用在学术界也引起广泛的关注。例如，Jangda[20]等人利用 WebAssembly 构建了 BROWSIX-WASM 拓展，在 Web 端实现了 Unix 的标准 API。通过 BROWSIX-WASM 可以直接在浏览器中运行没有经过修改的通过 WebAssembly 编译的 Unix 应用程序。然后和本地层的性能进行大规模的评估，并且根据评估结果分析产生性能差距的原因。Letz[21]等人展示了 Faust 音频 DSP 语言如何用于生成基于 WebAssembly 的高效 Web 音频节点。Finney[22]等人通过将 WebAssembly 技术应用于癌症生物信息学，设计出可以直观检查癌症数据集的新型网络浏览器工具。Herrera[23]等人针对 WebAssembly 的性能在多平台多浏览器中进行评估，分析 WebAssembly 实现的计算密集型代码段在不同浏览器和不同平台的性能对比。

通过对 JavaScript 代码性能优化研究现状的分析可以发现，目前的研究方向主要集中在编译器和类型推导[24,25]两个方面。首先 JavaScript 的执行过程需要将纯文本的代码解析生成抽象语法树[26]，然后通过遍历抽象语法树获得相应的机器码，进而得到编译好的程序，这个过程产生了性能开销，虽然引进了即时编译技术将部分使用频率高的代码段编译存储，避免重复解释执行，但是监测程序运行过程也会导致性能开销[27]。并且 JavaScript 的运行性能缺陷是从设计之初就存在的，其本身是作为便捷的开发语言实现 Web 端简单的交互，所以 JavaScript 设计时采用无类型的形式，这使得 JavaScript 引擎执行时需要细粒度的类型描述，而在程序执行的过程中还需要进行“unbox”和“box”等繁琐的类型检测。为了改善 JavaScript 的执行效率，后来引入了即时编译技术，并且更进一步的引入了类型特化的即时编译技术[28,29]。即时编译技术使得频繁操作的程序可以预先编译，类型特化更可以通过类型推断对代码进行优化，进而实现性能的提升。但是 JavaScript 无类型的特点会引发类型推断错误，执行退优化进而产生性能开销。目前也存在对 JavaScript 功能进行限定并且模拟数据类型的优化方式，但是由于需要浏览器特殊的优化策略导致通用性不强。而 WebAssembly 是由浏览器厂家共同设计实现，所以可以应用于各家浏览器。并且由于 WebAssembly 静态数据类型的特点，所以免去了 JavaScript 中复杂类型推导造成的性能开销。更重要的是，WebAssembly 的二进制形式不需要解析生成抽象语法树再解释执行，这就使得 WebAssembly 拥有更高的执行效率。而 WebAssembly 的设计目的就是将 C/C++等语言实现的与性能相关的计算密集类型操作移植到 Web 端，使得可以在 Web 端进行

高效的数据处理。这也是我们研究利用 WebAssembly 进行 JavaScript 性能优化的理论基础。

1.3 研究内容

本文以在 Web 端具有高效运行效率的 WebAssembly 为基础，设计实现了一种 JavaScript 代码性能优化方案，通过筛选获取 JavaScript 代码中与性能相关并且 WebAssembly 可模拟的计算密集型代码段，将代码段转换生成 WebAssembly 模块并且与原代码段进行性能比较。最后以性能比较结果和数据流依赖关系为基础生成代码段合并策略。

本文主要针对以下几个方面进行研究：

（1）研究当前 JavaScript 性能优化方案

对 JavaScript 性能问题进行深入研究，分析产生性能问题的原因。并且对当前 JavaScript 代码优化方案进行研究，包括编译技术，变量类型模拟和部分求解器等多种主要的优化方案[30,31]。详细分析了这些优化方案的基本原理，并且结合产生 JavaScript 性能缺陷的原因分析当前优化方案的局限性。

（2）研究基于 WebAssembly 的性能优化方案

分析利用 WebAssembly 对 JavaScript 进行性能优化方案的可行性。通过分析 WebAssembly 的适用范围和设计特点，提出了一个 JavaScript 代码性能优化方案，并且讨论了该方案的实现原理。首先根据 WebAssembly 的适用范围和语言特性，设计 JavaScript 代码段筛选规则以保证 WebAssembly 功能和语法可覆盖目标代码段。设计代码转换方法将目标代码段转换生成相应的 WebAssembly 模块。最后，通过对 JavaScript 代码和 WebAssembly 模块之间数据交互性能开销的研究，设计并提出代码段合并方法。

（3）设计并实现基于 WebAssembly 的 JavaScript 性能优化系统。

针对本文提出的方法设计并实现原型优化系统 JSOPW，主要介绍了类型解析，代码转换和性能优化等主要模块，通过讨论各个模块中关键算法的实现和部分功能的执行流程来阐述系统的设计逻辑。选择多个实际 Web 应用程序作为测试用例，同时利用 Facebook 设计的 Prepack[32]优化方案进行对比，收集测试用例在优化前后的时空开销。并且针对多浏览器多平台验证优化方案的通用性，通过对比评估实验验证优化方案的有效性。

1.4 论文组织结构与章节安排

本文的研究内容主要分为六个章节，详细组织结构如下所示：

第一章为引言。主要针对本文的研究背景和意义进行讨论，介绍了国内外关于JavaScript 性能优化和 WebAssembly 技术应用的研究现状。并且阐述本文的研究内容和文章结构。

第二章对当前 JavaScript 代码优化方案进行介绍，包括即时编译，Prepack 和 Asm.js 等方案的研究现状。然后介绍 WebAssembly 在 Web 端的性能优势，最后给出本文设计方法的可行性分析。

第三章对本文设计的基于 WebAssembly 的 JavaScript 性能优化方案的进行阐述。以 JavaScript 代码转换方法和代码段合并方法为核心说明整个优化方案的设计思路。

第四章对实现的 JSOPW 原型系统进行说明。详细阐述了系统实现过程中的主要算法和核心执行流程。

第五章通过原型系统和现有优化方案对 Web 应用程序优化效果的对比进行评估实验。并且针对代码段合并方法的有效性进行实验验证。最后在不同浏览器和不同平台的场景下对测试用例进行性能评估，论证在不同场景下优化方案的有效性。

第六章对本文的研究进行总结，并且针对当前研究的缺陷展望下一阶段的研究内容。