# Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code

## 面临的问题

目前，javascript是一门非常流行的语言，但是它存在一些性能上的瓶颈问题，为了弥补JavaScript在一些应用上效率低下的问题，工业界推出了webassembly，以此来弥补JavaScript性能上的问题。

在webassemly出现之前，也有类似于asm.js这种跨平台且通用的方案，这是个编译目标的方案，只不过是编译成了带强类型的js，其运行机制还是JavaScript的过程，只不过由于其强类型的特点，节省了运行时的类型检查，所以比原生JavaScript快了34%，实际上也没有快很多。

webassembly也是一个编译目标，可以让C，C++等语言编写的程序编译成webassembly的编码，之后就可以通过js加载webassembly代码，然后在浏览器上运行。但是以往的研究都说webassembly的运行速度只比机器码慢10%到15%，但是论文作者认为以往的测试并不合理，因为以往的基于PolyBenchC的测试，测试的都是webassembly和其它语言的科学计算的性能，类似于矩阵相乘这种，但是这种测试有很大的局限性，对于需要调用系统接口的程序，是无法测试的，因为JavaScript运行在浏览器上的时候是不能调用系统接口的，因此无法进行相关性能上的测试。

由于PolyBenchC测试上的局限性，本文决定采用SPEC Benchmark作为测试套件来测试webassembly的性能，SPEC中包含了对于视频处理，posix应用等多方面的测试。但是之前有提到，chrome不为JavaScript提供系统接口调用，在之前的工作中，已经有了Browsix这个工具，这是一个能为JavaScript程序提供POSIX接口的框架，该框架运行在浏览器中，是一个类-UNIX内核。但是仍然面临的问题是，Browsix这个工具只能为JavaScript程序提供系统接口的调用，但是对于webassembly代码是无法提供系统接口的，原因在于Browsix和JavaScript代码是通过SharedArrayBuffer来进行通讯的，但是由于webassembly的进程是独立的，无法通过ShareArrayBuffer和Browsix的内核进行交互，所以作者实现了一个能够让webassembly代码调用系统接口的Browsix-WASM, 该方案是对于Browsix的扩展，使其能够支持webassembly通信。

这篇论文的相关研究不多，因为他是直接进行的一种性能测试，然后为了进行这个测试补充完善了部分工具，之后分析了一些影响性能的原因。

## 作者介绍

这篇文章的作者是就读于马萨诸塞大学阿默斯特分校的phd学生Abhinav Jangda，主要研究领域是操作系统和编程语言。

## 主要贡献

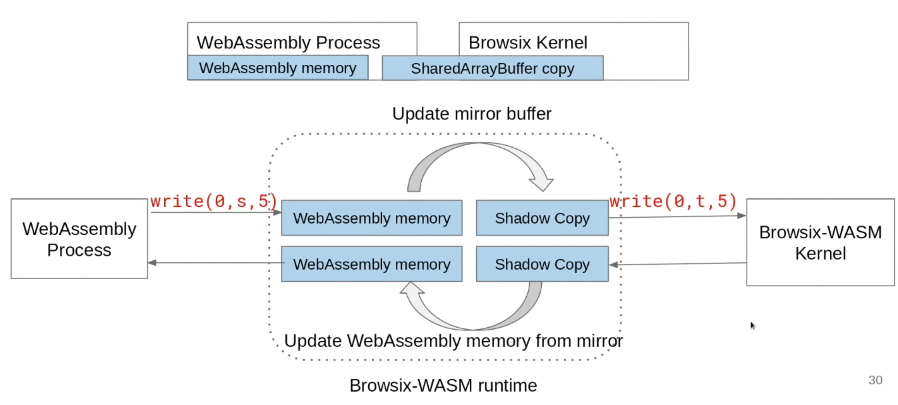
1. 为了能够准确的测试webassembly的性能，作者主要做了以下工作：
2. 改进Browsix到Browsix-WASM，使其支持对于webassembly的测试
3. 构建Browsix-SPEC，用于收集运行中的SPEC的测试数据
4. 对于造成webassembly的性能不如机器码的原因进行了分析

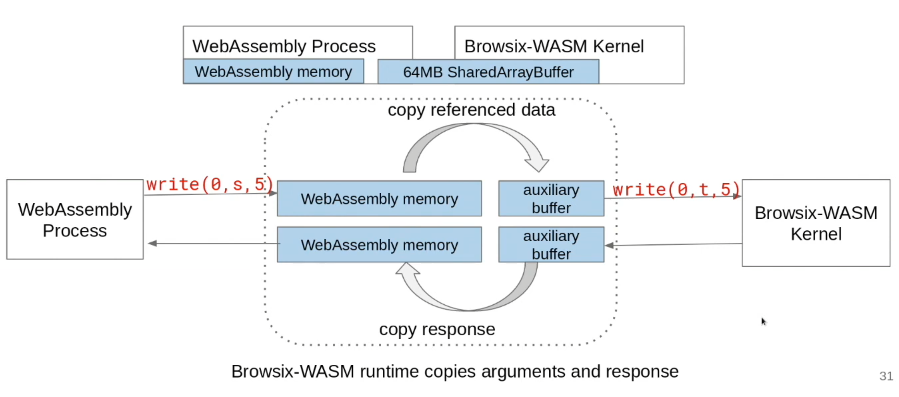
## 研究所采用的方法

主要是分为两个部分，首先是对于原本的Browsix框架进行改进，使其能够支持webassembly程序调用系统接口。另一部分就是

### Browsix-WASM

对于Browsxi不支持Webassembly的测试，原因在于webassembly代码的内存和Browsix的内存是不共享的，无法用SharedArrayBuffer进行通信，所以作者首先采用拷贝Webassembly的内存的方式，当要通信的时候拷贝一下webaseembly代码的内存，通信完毕以后再拷贝回去。

但是这样有一些问题，首先拷贝全部的内存会导致拷贝的开销很大。其次，拷贝全部内存就相当于需要有两倍的内存占用，非常不合理。所以作者提出每次运行时候为程序预留64MB的空闲空间，每当需要进行系统接口调用的时候，就拷贝需要通信的部分，通信完毕之后再覆盖回去，对于64MB内存不够的情况，就分多批次进行拷贝和通信调用。

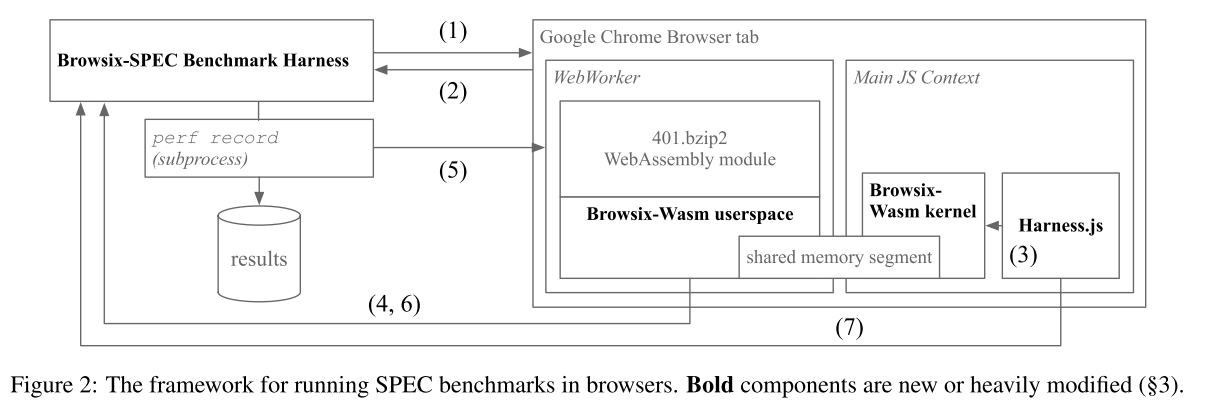


但是这样处理之后，仍然面临着一些问题，进行某几个文件操作测试的时候，仍然有很高的开销。原因在于Browsix-WASM的文件系统，如果对于文件的操作是每次增加一点点，那么每次都要申请内存，然后拷贝过去之后再扩容，这样的效率很低，作者们修改了文件系统的一些操作，将每次申请改为了一次申请4KB，这样对小的增量处理就变得非常的友好。

### Browsix-SPEC

Browsix-SPEC可以看作是一个辅助测试的工具，用来执行WebAssembly的测试，并且获取性能计数器的数据。

**运行过程**



1. BROWSIX-SPEC基准测试工具使用WebBrowser自动化工具Selenium.2启动了一个新的浏览器实例。

2. 浏览器通过基准测试工具采用HTTP方式加载页面的HTML，harness和BROWSIX-WASM内核。都是js格式

3. harness.js初始化BROWSIX-WASM内核，并启动一个新的BROWSIX-WASM进程，该进程执行runspec shell脚本（图2中未显示）。runspec依次生成从SPEC 2006中提供的C源代码编译的标准specinvoke（未显示）。specinvoke从BROWSIX-WASM文件系统读取speccmds.cmd文件，并使用适当的参数启动401.bzip2。为了运行基准测试，runspec在运行目录中的文件speccmds.cmd中写入了一系列命令。然后，这些命令由specinvoke解释。 这个runsepec最后会用适当的参数启动401.bzip2

4. 在实例化WebAssembly模块之后，但在调用基准测试的主要功能之前，BROWSIX-WASM用户空间runtime向BROWSIX-SPEC发出XHR请求，以开始记录性能计数器统计信息。

5. 基准测试工具找到与Web Worker 401.bzip2进程相对应的Chrome线程，并将perf附加到该进程。

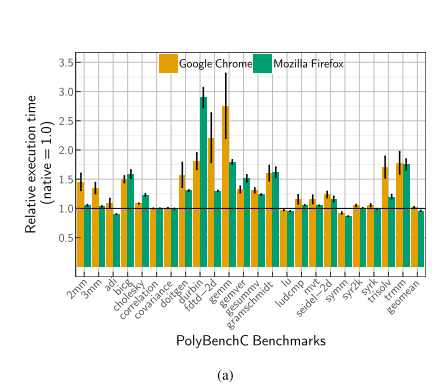
6. 在基准测试结束时，BROWSIX-WASM用户空间运行时对基准测试工具执行了最终的XHR，以结束性能记录过程。当runspec程序退出时（可能多次调用了测试二进制文件后），harness JS POST

7. SPEC结果目录的tar存档到BROWSIX-SPEC。BROWSIX-SPEC收到完整的结果存档后，将结果解压缩到一个临时目录中，并使用SPEC 2006随附的cmp工具验证输出。最后，BROWSIX-SPEC终止了浏览器进程并记录了基准测试结果。

## 实验结果及性能评估

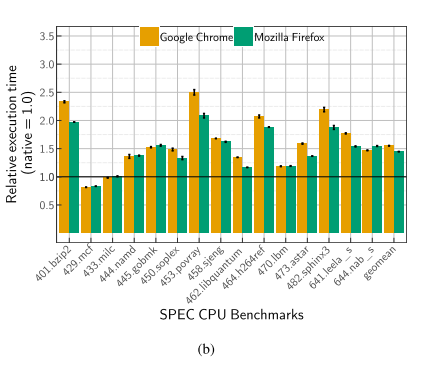
### PolyBenchC 测试

首先还是采用PolyBenchC进行测试，虽然PolyBenchC的测试不会产生系统调用。但是可以证明用于系统调用的内核部分不会产生额外的开销，下图为Browsix-WASM运行PolybenchC测试的时间，然后跟以往的测试进行对比，就知道Browsix-WASM本身的运行开销非常的低，平均为0.2% 最大值为1.2%。



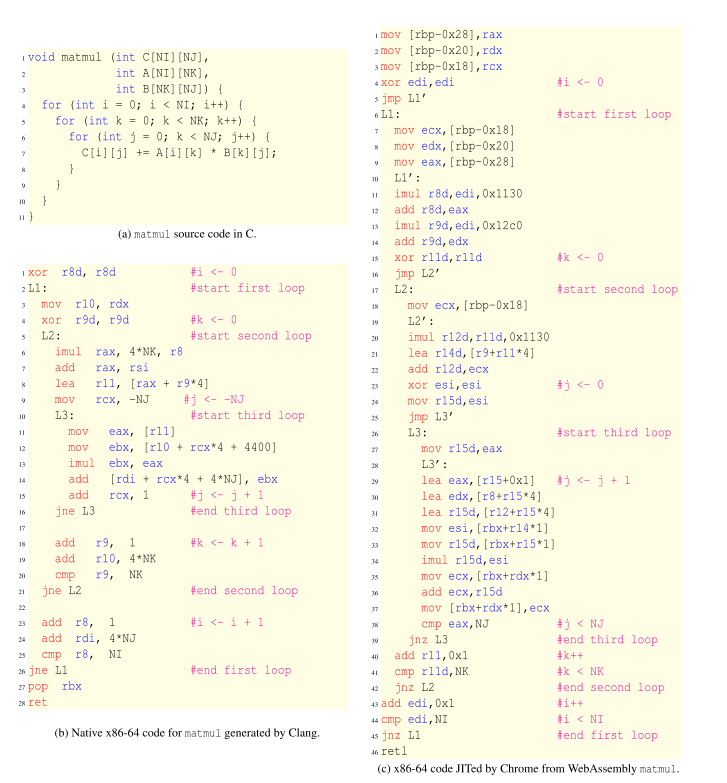
### SPEC 测试

之后用Browsix-WASM运行SPEC测试套件对Webassembly的性能进行测试，最终的测试结果显示，除了少数的测试项，WebAseembly的运行速度基本都比机器码要慢。再Chrome浏览器中，最慢的速度是机器码的2.5倍，有7项测试指标慢了1.5倍以内。在Firefox浏览器中，最慢的是机器码的2.08倍，其中7项的速度在1.5倍以内，所以最后平均下来，在chrome中，WebAssembly运行时间是机器码的1.55倍。在Firefox中机运行时间是1.45倍，其中各项测试的结果如下图所示。



## 原因分析

文章中进行了一系列的测试，得出的结果是WebAssembly比机器码的运行速度慢了不少，所以需要分析其中的原因，首先举一个矩阵相乘的例子。



如上图所示，其中左上角是C代码源码，左下角是Clang编译后的机器码，右下角是JIT编译后的代码，可以很明显的发现以下三点不同

1. 代码体积明显增长，其中JIT编译后的代码是Clang编译后的代码的两倍左右的长度。
2. 可以很明显的发现，对于同样的功能实现，chrome JIT产生的代码所用的寄存器的数量明显要比Clang编译产生的多
3. JIT产生的代码有额外的跳转分支，造成了代码的冗余。

### 详细分析影响性能的因素

论文中详细分析了webassembly性能不如机器码的原因，主要分为寄存器压力增加、代码分支增加、代码量增加三个点，之后分别对这三个点的原因进行了详细的探讨。

**寄存器压力增加**

1. 保留寄存器

对于Chrome浏览器来说，它会保存一些寄存器用来进垃圾回收等操作，保留的寄存器增加了寄存器开销。

1. 寄存器分配策略问题

可以看出clang编译后的代码使用了10个寄存器，而Chrome编译后的代码使用了12个寄存器，原因在于Chrome采用的是线性扫描的寄存器分配方法，这种方法虽然会造成寄存器的压力，但是编译速度很快。

1. X86寻址模式

可以看到clang编译后的机器码采用了x86的寻址模式，这种寻址模式可以很好的减少寄存器的压力。

**额外的分支命令**

1. 额外的循环跳转命令

可以看到chrome生成的代码有额外的分支

1. 调用函数时的栈溢出检查

由于webassembly本身运行环境的原因，每次进行函数调用的时候都要先进行栈溢出的检查，这些检查也会生成额外的跳转。

1. 函数表下标检查

Webassembly函数的每次直接调用，需要进行合法性检查，这样同样会造成额外的分支命令

**增加的代码量**

可以很明显的看到Chrome JIT编译后的代码体积很大，体积大的代码明显有更多的指令，更多的指令会导致L1缓存的miss增加，最终导致代码的运行速度下降

## 总结

这篇文章指出了以往对于webassembly的测试不全面，并改进了原本针对JavaScript的Browsix框架，以便进行对Webassembly的测试，最终的测试结果表明Webassembly并不像之前他们说的那么快，通过全面的测试后发现webassembly在全场景下比机器码慢了55%左右。