# Emscripten: An LLVM-to-JavaScript Compiler

Alon Zakai
Mozilla
azakai@mozilla.com

## Abstract

JavaScript is the standard language of the web, supported on essentially all web browsers. Despite efforts to allow other languages to be run as well, none have come close to being universally available on all browsers, which severely limits their usefulness on the web. However, there are reasonable reasons why allowing other languages would be beneficial, including reusing existing code and allowing developers to use their languages of choice.

We present Emscripten, an LLVM-to-JavaScript compiler. Emscripten compiles LLVM assembly code into standard JavaScript, which opens up two avenues for running code written in other languages on the web: (1) Compile a language directly into LLVM, and then compile that into JavaScript using Emscripten, or (2) Compiling a language's entire runtime (typically written in C or C++) into JavaScript using Emscripten, and using the compiled runtime to run code written in that language. Examples of languages that can use the first approach are C and C++, as compilers exist for them into LLVM. An example of a language that can use the second approach is Python, and Emscripten has been used to compile CPython (the standard C implementation of Python) into JavaScript, allowing standard Python code to be run on the web, which was not previously possible.

Emscripten itself is written in JavaScript (to enable various dynamic compilation techniques), and is available under the MIT license (a permissive open source license), at `http://www.emscripten.org`. As an LLVM-to-JavaScript compiler, the challenges in designing Emscripten are somewhat the reverse of the norm – one must go from a low-level assembly into a high-level language, and recreate parts of the original high-level structure of the code that were lost in the compilation to low-level LLVM. We detail the algorithms used in Emscripten to deal with those challenges.



## 1 Introduction

Since the mid 1990's, JavaScript has been present in most web browsers (sometimes with minor variations and under slightly different names, e.g., JScript in Internet Explorer), and today it is well-supported on essentially all web browsers, from desktop browsers like Internet Explorer, Firefox, Chrome and Safari, to mobile browsers on smartphones and tablets. Together with HTML and CSS, JavaScript is the standards-based foundation of the web.

Running other programming languages on the web has been suggested many times, and browser plugins have allowed doing so, e.g., via the Java and Flash plugins. However, plugins must be manually installed and do not integrate in a perfect way with the outside HTML. Perhaps more problematic is that they cannot run at all on some platforms, for example, Java and Flash cannot run on iOS devices such as the iPhone and

iPad. For those reasons, JavaScript remains the primary programming language of the web.

There are, however, justifiable motivations for running code from other programming languages on the web, for example, if one has a large amount of existing code already written in another language, or if one simply has a strong preference for another language (and perhaps is more productive in it).

As a consequence, there have been some efforts to compile languages **into** JavaScript. Since JavaScript is present in essentially all web browsers, by compiling one's language of choice into JavaScript, one can still generate content that will run practically everywhere. Examples of this approach include the Google Web Toolkit, which compiles Java into JavaScript; Pyjamas, which compiles Python into JavaScript; and rumors have it that projects in Oracle and Microsoft, respectively, allow running JVM and CLR bytecode in JavaScript (which would allow running the languages of the JVM and CLR, respectively).

In this paper we present another project along those lines: **Emscripten**, which compiles LLVM assembly into JavaScript. LLVM (Low Level Virtual Machine) is a compiler project primarily focused on C, C++ and Objective-C. It compiles those languages through a *frontend* (the main ones of which are Clang and LLVM-GCC) into the LLVM intermediary representation (which can be machine-readable bitcode, or human-readable assembly), and then passes it through a *backend* which generates actual machine code for a particular architecure. Emscripten plays the role of a backend which targets JavaScript.

By using Emscripten, potentially many languages can be run on the web, using one of the following methods:

- Compile **code** in a language recognized by one of the existing LLVM frontends into LLVM, and then compile that into JavaScript using Emscripten. Frontends for various languages exist, including many of the most popular programming languages such as C and C++, and also various new and emerging languages (e.g., Rust).
- Compile the **runtime** used to parse and execute code in a particular language into LLVM, then compile that into JavaScript using Emscripten. It is then possible to run code in that runtime on the web. This is a useful approach if a language's runtime is written in a language for which an LLVM frontend exists, but the language iself has no such frontend. For example, no currently-supported frontend exists for Python, however it is possible to compile CPython – the standard implementation of Python, written in C – into JavaScript, and run Python code on that (see Subsection X.Y).

From a technical standpoint, the main challenges in designing and implementing Emscripten are that it compiles a low-level language – LLVM assembly – into a high-level one – JavaScript. This is somethat the reverse of the usual situation one is in when building a compiler, and leads to some unique difficulties. For example, to get good performance in JavaScript one must use natural JavaScript code flow structures, like loops and ifs, but those structures do not exist in LLVM assembly (instead, what is present there is essentially 'flat' code with *goto* commands). Emscripten must therefore reconstruct a high-level representation from the low-level data it receives.

In theory that issue could have been avoided by compiling a higher-level language into JavaScript. For example, if compiling Java into JavaScript (as the Google Web Toolkit does), then one can benefit from the fact that Java's loops, ifs and so forth generally have a very direct parallel in JavaScript (of course the downside is that this approach yields a compiler only for Java). Compiling LLVM into JavaScript is less straightforward, but wee will see later that it is possible to reconstruct a substantial part of the high-level structure of the original code.

We conclude this introduction with a list of this paper's main contributions:

- We describe Emscripten itself, during which we detail its approach in compiling LLVM into JavaScript.

- We give details of Emscripten's 'Relooper' algorithm, which generates high-level loop structures from low-level branching data. We are unaware of related results in the literature.

In addition, the following are the main contributions of Emscripten itself, that to our knowledge were not previously possible:

- It allows compiling a very large subset of C and C++ code into JavaScript, which can then be run on the web.
- By compiling their runtimes, it allows running languages such as Python on the web.

The remainder of this paper is structured as follows. In Section 2 we describe, from a high level, the approach taken to compiling LLVM assembly into JavaScript. In Section 3 we describe the workings of Emscripten on a lower, more concrete level. In Section 4 we give an overview of some uses of Emscripten. In Section 5 we summarize and give directions for future work on Emscripten and uses of it.

## 2 Compilation Approach

Let us begin by considering what the challenge is, when we want to compile something into JavaScript. Assume we are given the following simple example of a C program, which we want to compile into JavaScript:

```
#include <stdio.h>
int main()
{
  int sum = 0;
  for (int i = 1; i < 100; i++)
    sum += i;
  printf("1+...+100=%d\n", sum);
  return 0;
}
```

This program calculates the sum of the integers from 1 to 100. When compiled by Clang, the generated LLVM assembly code includes the following:

```
@.str = private constant [14 x i8]
        c"1+...+100=%d\0A\00"

define i32 @main() {
  %1 = alloca i32, align 4
  %sum = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %sum, align 4
  store i32 1, i32* %i, align 4
  br label %2

; <label>:2
  %3 = load i32* %i, align 4
  %4 = icmp slt i32 %3, 100
  br i1 %4, label %5, label %12

; <label>:5
  %6 = load i32* %i, align 4
  %7 = load i32* %sum, align 4
  %8 = add nsw i32 %7, %6
  store i32 %8, i32* %sum, align 4
  br label %9

; <label>:9
  %10 = load i32* %i, align 4
  %11 = add nsw i32 %10, 1
  store i32 %11, i32* %i, align 4
  br label %2

; <label>:12
  %13 = load i32* %sum, align 4
  %14 = call i32 (i8*, ...)*
        @printf(i8* getelementptr inbounds
          ([14 x i8]* @.str, i32 0, i32 0),
          i32 %13)
  ret i32 0
}
```

At first glance, this may look more difficult to translate into JavaScript than the original C++. However, compiling C++ in general would require writing code to handle preprocessing, classes, templates, and all the idiosyncrasies and complexities of C++. LLVM assembly, while more verbose in this example, is lower-level and simpler to work on. It also has the benefit we mentioned earlier, which is one of the

main goals of Emscripten, that many languages can be compiled into LLVM.

A detailed overview of LLVM assembly is beyond our scope here. Briefly, though, the example assembly above can easily be seen to define a function main(), then allocate some values on the stack (alloca), then load and store various values (load and store). We do not have the high-level code structure as we had in C++, with a loop, instead we have code 'fragments', each with a label, and code flow moves from one to another by branch (br) instructions. (Label 2 is the condition check in the loop; label 5 is the body, label 9 is the increment, and label 12 is the final part of the function, outside of the loop). Conditional branches can depend on calculations, for example the results of comparing two values (icmp). Other numerical operations include addition (add). Finally, printf is called (call). The challenge, then, is to convert this and things like it into JavaScript.

In general, Emscripten's approach is to translate each line of LLVM assembly into JavaScript, 1 for 1, into 'normal' JavaScript as much as possible. So, for example, an *add* operation becomes a normal JavaScript addition, a function call becomes a JavaScript function call, etc. This 1 to 1 translation generates JavaScript that resembles assembly code, for example, the LLVM assembly shown before for main() would be compiled into the following:

```
function _main() {
  var __stackBase__  = STACKTOP;
  STACKTOP += 12;
  var __label__ = -1;
  while(1) switch(__label__) {
    case -1:
      var $1 = __stackBase__;
      var $sum = __stackBase__+4;
      var $i = __stackBase__+8;
      HEAP[$1] = 0;
      HEAP[$sum] = 0;
      HEAP[$i] = 0;
      __label__ = 0; break;
    case 0:
      var $3 = HEAP[$i];
      var $4 = $3 < 100;
      if ($4) { __label__ = 1; break; }
      else    { __label__ = 2; break; }
    case 1:
      var $6 = HEAP[$i];
      var $7 = HEAP[$sum];
      var $8 = $7 + $6;
      HEAP[$sum] = $8;
      __label__ = 3; break;
    case 3:
      var $10 = HEAP[$i];
      var $11 = $10 + 1;
      HEAP[$i] = $11;
      __label__ = 0; break;
    case 2:
      var $13 = HEAP[$sum];
      var $14 = _printf(__str, $13);
      STACKTOP = __stackBase__;
      return 0;
  }
}
```

Some things to take notice of:

- A switch-in-a-loop construction is used in order to let the flow of execution move between fragments of code in an arbitrary manner: We set *__label__* to the (numerical representation of the) label of the fragment we want to reach, and do a break, which leads to the proper fragment being reached. Inside each fragment, every line of code corresponds to a line of LLVM assembly, generally in a very straightforward manner.

- Memory is implemented by *HEAP*, a JavaScript array. Reading from memory is a read from that array, and writing to memory is a write. *STACKTOP* is the current position of the stack. (Note that we allocate 4 memory locations for 32-bit integers on the stack, but only write to 1 of them. See the Load-Store Consistency subsection below for why.)

- LLVM assembly functions become JavaScript functions, and function calls are normal JavaScript function calls. In general, we attempt to generate as 'normal' JavaScript as possible.

## 2.1 Load-Store Consistency (LSC)

We saw before that Emscripten's memory usage allocates the usual number of bytes on the stack for variables (4 bytes for a 32-bit integer, etc.). However, we only wrote values into the first location, which appeared odd. We will now see the reason for that.

To get there, we must first step back, and note that Emscripten does not aim to achieve perfect compatibility with all possible LLVM assembly (and correspondingly, with all possible C or C++ code, etc.); instead, Emscripten targets a subset of LLVM assembly code, which is portable and does not make crucial assumptions about the underlying CPU architecture on which the code is meant to run. That subset is meant to encompass the vast majority of real-world code that would be compiled into LLVM, while also being compilable into very performant JavaScript.

More specifically, Emscripten assumes that the LLVM assembly code it is compiling has **Load-Store Consistency** (LSC), which is the requirement that loads from and stores to a specific memory address will use the same type. Normal C and C++ code generally does so: If $x$ is a variable containing a 32-bit floating point number, then both loads and stores of $x$ will be of 32-bit floating point values, and not 16-bit unsigned integers or anything else. (Note that even if we write something like

```
float x = 5
```

then the compiler will assign a 32-bit float with the value of 5 to $x$, and not an integer.)

To see why this is important for performance, consider the following C code fragment, which does *not* have LSC:

```
int x = 12345;
[...]
printf("first byte: %d\n", *((char*)&x));
```

Assuming an architecture with more than 8 bits, this code will read the first byte of $x$. (It might, for example, be used to detect the endianness of the CPU.) To compile this into JavaScript in a way that will run properly, we must do more than a single operation for either the read or the write, for example we could do this:

```
var x_value = 12345;
var x_addr = stackAlloc(4);
HEAP[x_addr]   = (x_value >> 0) & 255;
HEAP[x_addr+1] = (x_value >> 8) & 255;
HEAP[x_addr+2] = (x_value >> 16) & 255;
HEAP[x_addr+3] = (x_value >> 24) & 255;
[...]
printf("first byte: %d\n", HEAP[x_addr]);
```

Here we allocate space for the value of $x$ on the stack, and store that address in *x_addr*. The stack itself is part of the 'memory space', which is the array *HEAP*. In order for the read on the final line to give the proper value, we must go to the effort of doing 4 store operations, each of the value of a particular byte. In other words, *HEAP* is an array of bytes, and for each store into memory, we must deconstruct the value into bytes.[1]

Alternatively, we can store the value in a single operation, and deconstruct into bytes as we load. This will be faster in some cases and slower in others, but is still more overhead than we would like, generally speaking – for if the code **does** have LSC, then we can translate that code fragment into the far more optimal

```
var x_value = 12345;
var x_addr = stackAlloc(4);
HEAP[x_addr] = x_value;
[...]
printf("first byte: %d\n", HEAP[x_addr]);
```

(Note that even this can be optimized even more – we can store $x$ in a normal JavaScript variable. For now though we are just clarifying why it is useful to assume we are compiling code that has LSC – doing so lets us generate shorter and more natural JavaScript.)

In practice the vast majority of C and C++ code does have LSC. Exceptions do exist, however, for example:

[1] Note that we can use JavaScript typed arrays with a shared memory buffer, which would work as expected, assuming (1) we are running in a JavaScript engine which supports typed arrays, and (2) we are running on a CPU with the same architecture as we expect. This is therefore dangerous as the generated code may run differently on different JavaScript engines and different CPUs. Emscripten currently has optional experimental support for typed arrays.

- Code that detects CPU features like endianness, the behavior of floats, etc. In general such code can be disabled before running it through Emscripten, as it is not needed.

- *memset* and related functions typically work on values of one kind, regardless of the underlying values. For example, memset may write 64-bit values on a 64-bit CPU since that is usually faster than writing individual bytes. This tends to not be a problem, as with *memset* the most common case is setting to 0, and with *memcpy*, the values end up copied properly anyhow.

- Even LSC-obeying C or C++ code may turn into LLVM assembly that does not, after being optimized. For example, when storing two 32-bit integers constants into adjoining locations in a structure, the optimizer may generate a single 64-bit store of an appropriate constant. In other words, optimization can generate nonportable code, which runs faster on the current CPU, but nowhere else. Emscripten currently assumes that optimizations of this form are not being used.

In practice it may be hard to know if code has LSC or not, and requiring a time-consuming code audit is obviously impractical. Emscripten therefore has automatic tools to detect violations of LSC, see SAFE_HEAP in Subsection X.Y.

Note that it is somewhat wasteful to allocation 4 memory locations for a 32-bit integer, and use only one of them. It is possible to change that behavior with the QUANTUM_SIZE parameter to Emscripten, however, the difficulty is that LLVM assembly has hardcoded values that depend on the usual memory sizes being used. For example, *memcpy* can be called with the normal size each the value would have, and not a single memory address each as Emscripten would prefer. We are looking into modifications to LLVM itself to remedy that.

## 2.2 Performance

When comparing the example program from before (that counts the integers from 1 to 100), the generated code was fairly complicated and cumbersome, and unsurprisingly it performs quite poorly. There are two main reasons for that: First, that the code is simply unoptimized – there are many variables declared when fewer could suffice, for example, and second, that the code does not use 'normal' JavaScript, which JavaScript engines are optimized for – it stores all variables in an array (not normal JavaScript variables), and it controls the flow of execution using a switch-in-a-loop, not normal JavaScript loops and ifs.

Emscripten's approach to generating fast-performing code is as follows. Emscripten doesn't do any optimization that can otherwise be done by additional tools: LLVM can be used to perform optimizations before Emscripten, and the Closure Compiler can perform optimizations on the generated JavaScript afterwards. Those tools will perform standard useful optimizations like removing unneeded variables, dead code, etc. That leaves two major optimizations that are left for Emscripten to perform:

- **Variable nativization**: Convert variables that are on the stack into native JavaScript variables. In general, a variable will be nativized unless it is used outside that function, e.g., if its address is taken and stored somewhere or passed to another function. When optimizing, Emscripten tries to nativize as many variables as possible.

- '**Relooping**': Recreate high-level loop and if structures from the low-level labels and branches that appear in LLVM assembly. We detail the algorithm Emscripten uses for this purpose in Section X.

When run with Emscripten's optimizations, the above code looks like this:

```
function _main() {
  var __label__;
  var $1;
  var $sum;
  var $i;
  $1 = 0;
  $sum = 0;
```

```
  $i = 0;
  $2$2: while(1) { // $2
    var $3 = $i;
    var $4 = $3 < 100;
    if (!($4)) { __label__ = 2; break $2$2; }
    var $6 = $i;
    var $7 = $sum;
    var $8 = $7 + $6;
    $sum = $8;
    var $10 = $i;
    var $11 = $10 + 1;
    $i = $11;
    __label__ = 0; continue $2$2;
  }
  var $13 = $sum;
  var $14 = _printf(__str, $13);
  return 0;
}
```

If in addition the Closure Compiler is run on that output, we get

```
function K() {
  var a, b;
  b = a = 0;
  a:for(;;) {
    if(!(b < 100)) {
      break a
    }
    a += b;
    b += 1;
  }
  _printf(J, a);
  return 0;
}
```

which is fairly close to the original C++ (the differences, of having the loop's condition inside the loop instead of inside the for() expression at the top of the original loop, are not important to performance). Thus, it is possible to recreate the original high-level structure of the code that was compiled into LLVM assembly, despite that structure not being explicitly available to Emscripten.

We will see performance measurements in Section 4.

### 2.3 Dealing with Real-World Code

As mentioned above, in practice it may be hard to know if code has LSC or not, and other difficulties may arise as well. Accordingly, Emscripten has a set of tools to help detect if compiled code has certain problems. These are mainly compile-time options that lead to additional runtime checks. One then runs the code and sees the warnings or errors generated. The debugging tools include:

- **SAFE_HEAP**: This option adds runtime checks for LSC. It will raise an exception if LSC does not hold, as well as related issues like reading from memory before a value was written (somewhat similarly to tools like Valgrind). If such a problem occurs, possible solutions are to ignore the issue (if it has no actual consqeuences), or altering the source code. In some cases, such nonportable code can be avoided in a simple manner by changing the configutation under which the code is built (see the examples in Section X).

- **CHECK_OVERFLOWS**: This option adds runtime checks for overflows in integer operations, which are an issue since all JavaScript numbers are doubles, hence simple translation of LLVM numerical operations (addition, multiplication, etc.) into JavaScript ones may lead to different results than expected. **CHECK_OVERFLOWS** will add runtime checks for actual overflows of this sort, that is, whether the result of an operation is a value too large for the type it defined as. Code that has such overflows can enable **CORRECT_OVERFLOWS**, which fixes overflows at runtime. The cost, however, is decreased speed due to the correction operations (which perform a bitwise AND to force the value into the proper range). Since such overflows are rare, it is possible to use this option to find where they occur, and modify the original code in a suitable way (see the examples in Section).

# 3 Emscripten's Architecture

In the previous section we saw a general overview of Emscripten's approach to compiling LLVM assembly into JavaScript. We will now get into more detail into how Emscripten implements that approach and its architecture.

Emscripten is written in JavaScript. A main reason for that decision was to simplify sharing code between the compiler and the runtime, and to enable various dynamic compilation techniques. Two simple examples: (1) The compiler can create JavaScript objects that represent constants in the assembly code, and convert them to a string using JSON.stringify() in a convenient manner, and (2) The compiler can simplify numerical operations by simply eval()ing the code (so "1+2" would become "3", etc.). In both examples, writing Emscripten is made simpler by having the exact same environment during compilation as the executing code will have.

Emscripten has three main phases:

- The **intertyper**, which converts from LLVM assembly into Emscripten's internal representation.
- The **analyzer**, which inspects the internal representation and generates various useful information for the final phase, including type and variable information, stack usage analysis, optional data for optimizations (variable nativization and relooping), etc.
- The **jsifier**, which does the final conversion of the internal representation plus additional analyzed data into JavaScript.

## 3.1 The Runtime Environment

Code generated from Emscripten is meant to run in a JavaScript engine, typically in a web browser. This has implications for the kind of runtime environment we can generate for it, for example, there is no direct access to the local filesystem.

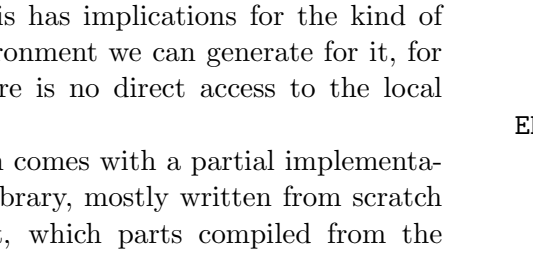
Emscripten comes with a partial implementation of a C library, mostly written from scratch in JavaScript, which parts compiled from the newlib C library. Some aspects of the runtime environment, as implemented in that C library, are:

- Files to be read must be 'preloaded' in JavaScript. They can then be accessed using the usual C library methods (fopen, fread, etc.). Files that are written are cached, and can be read by JavaScript later. While limiting, this approach can often be sufficient for many purposes.
- Emscripten allows writing pixel data to an HTML5 canvas element, using a subset of the SDL API. That is, one can write an application in C or C++ using SDL, and that same application can be compiled normally and run locally, or compiled using Emscripten and run on the web. See the raytracing demo at `http://syntensity.com/static/raytrace.html`.
- *sbrk* is implemented using the **HEAP** array which was mentioned previously. This allows a normal *malloc* implementation written in C to be compiled to JavaScript.

## 3.2 The Relooper: Recreating high-level loop structures

The Relooper is among the most complicated components in Emscripten. It receives a 'soup of labels', which is a set of labeled fragments of code – for brevity we call such fragments simply 'labels' – each ending with a branch operation (either a simple branch, a conditional branch, or a switch), and the goal is to generate normal high-level JavaScript code flow structures such as loops and ifs.

For example, the LLVM assembly on page X has the following label structure:



In this simple example, it is fairly straightforward to see that a natural way to implement it using normal loop structures is

```
ENTRY
while (true) do
  2
  if (condition) break
  5
  9
12
```

In general though, this is not always easy or even possible – there may not be a reasonable high-level loop structure corresponding to the low-level one, if for example the original C code relied heavily on *goto* instructions. In practice, however, almost all real-world C and C++ code tends to be amenable to loop recreation.

Emscripten's Relooper takes as input a 'soup of labels' as described above, and generates a structured set of code 'blocks', which are each a set of labels, with some logical structure, of one of the following types:

- **Simple block**: A block with one internal label and a Next block, which the internal label branches to. The block is later translated simply into the code for that label, and the Next block appears right after it.
- **Loop**: A block that represents an infinite loop, comprised of two internal sub-blocks:
  - **Inner**: A block of labels that will appear inside the loop, i.e., when execution reaches the end of that block, flow will return to the beginning. Typically a loop will contain a conditional *break* defining where it is exited. When we exit, we reach the Next block, below.
  - **Next**: A block of labels that will appear just outside the loop, in other words, that will be reached when the loop is exited.
- **Multiple**: A block that represents an divergence into several possible branches, that eventually rejoin. A Multiple block can implement an 'if', an 'if-else', a 'switch', etc. It is comprised of
  - **Handled blocks**: A set of blocks to which execution can enter. When we reach the multiple block, we check which of them should execute, and go there. When execution of that block is complete, or if none of the handled blocks was selected for execution, we proceed to the Next block, below.
  - **Next**: A block of labels that will appear just outside this one, in other words, that will be reached after code flow exits the Handled blocks, above.

Remember that we have a *__label__* variable that helps control the flow of execution: Whenever we enter a block with more than one entry, we set *__label__* before we branch into it, and we check its value when we enter that block. So, for example, when we create a Loop block, it's Next block can have multiple entries – any label to which we branch out from the loop. By creating a Multiple block after the loop, we can enter the proper label when the loop is exited. Having a *__label__* variable does add some overhead, but it greatly simplifies the problem that the Relooper needs to solve. Of course, it is possible to optimize away writes and reads to *__label__* in many cases.

Emscripten uses the following recursive algorithm for generating blocks from the soup of labels:

- Receive a set of labels and which of them are entry points. We wish to create a block comprised of all those labels.
- Calculate, for each label, which other labels it *can* reach, i.e., which labels we are able to reach if we start at the current label and follow one of the possible paths of branching.
- If we have a single entry, and cannot return to it from any other label, then create a Simple block, with the entry as its internal label, and the Next block comprised of all the other labels. The entries for the Next block are the entry to which the internal label can branch.

- If we can return to all of the entries, return a Loop block, whose Inner block is comprised of all labels that can reach one of the entries, and whose Next block is comprised of all the others. The entry labels for the current block become entry labels for the Inner block (note that they must be in the Inner block, as each one can reach itself). The Next block's entry labels are all the labels in the Next block that can be reached by the Inner block.

- If we have more than one entry, try to create a Multiple block: For each entry, find all the labels it reaches that cannot be reached by any other entry. If at least one entry has such labels, return a Multiple block, whose Handled blocks are blocks for those labels, and whose Next block is all the rest. Entry labels for those two blocks become entries of the new block they are now part of. We may have additional entry labels in the Next block, for each entry in the Next block that can be reached from the Handled ones.

- We must be able to return to at least one of the entries (see proof below), so create a Loop block as described above.

Note that we first create a loop only if we must, then try to create a multiple, then create a loop if we can. We could have slightly simplified this in various ways. The algorithm as presented above has given overall better results in practice, in terms of the 'niceness' of the shape of the generated code, both subjectively and in some benchmarks (however, the benchmarks are limited, and we cannot be sure the result will remain true for all possible inputs to the Relooper).

Additional details of the algorithm include 'fixing' branch instructions accordingly. For example, when we create a Loop block, then all branch instructions outside of the loop are converted into *break* commands, and all branch instructions to the beginning of the loop are converted into *continue* commands. Those commands are then ignored when called recursively to generate the Inner block (that is, the *break* and *continue* commands are guaranteed, by the semantics of JavaScript, to get us to where we need to go – they do not need any further consideration for them to work properly).

Emscripten also does an additional pass after running the Relooper algorithm which has been described. The Relooper is guaranteed to produce valid output (see below). The second pass takes that valid output and optimizes it, by making minor changes such as removing *continue* commands that occur at the very end of loops (where they are not needed), etc. In other words, the first pass focuses on generating high-level code flow structures that are correct, while the second pass simplifies and optimizes that structure.

We now turn to an analysis of the Relooper algorithm. It is straightforward to see that the output of the algorithm, assuming it completes successfully – that is, that if finishes in finite time, and does not run into an error in the last part (where it is claimed that if we reach it we can return to at least one of the entry labels) – is correct in the sense of code execution being carried out as in the original data. We will now prove that the algorithm must in fact complete successfully.

First, note that if we successfully create a block, then we simplify the remaining problem, where the 'complexity' of the problem for our purposes now is the sum of labels plus the some of branching operations:

- This is trivial for Simple blocks (since we now have a Next block which is strictly smaller).

- It is true for loop blocks simply by removing branching operations (there must be a branching back to an entry, which becomes a *continue*).

- For multiple blocks, if the Next block is non-empty then we have split into strictly smaller blocks (in number of labels) than before. If the next block is empty, then since we built the Multiple block from a set of labels with more than one entry, then the Handled blocks are strictly smaller than the

current one. The remaining case is when we have a single entry.

The remaining issue is whether we can reach a situation where we fail to create a block due to an error, that is, that the claim in the last part does not hold. For that to occur, we must not be able to return to any of the entries (or else we would create a Loop block). But since that is so, we can, at minimum, create a Multiple block with entries for all the current entries, since the entry labels themselves cannot be reached, contradicting the assumption that we cannot create a block, and concluding the proof.

We have not, of course, proven that the shape of the blocks is optimal in any sense. However, even if it is possible to optimize them, the Relooper already gives a very substantial speedup due to the move from the switch-in-a-loop construction to more natural JavaScript code flow structures. TODO: A little data here.

# 4 Example Uses

Emscripten has been run successfully on several real-world codebases, including the following:

TODO: Performance data

## 4.1 CPython

Other ways to run Python on web: pyjamas/pyjs, old pypy-js backend, ?? also IronPython on Silverlight and Jython in Java. Limitations etc.

CPython is the standard implementation of the Python programming language written in C. Compiling it in Emscripten is straightforward, except for needing to change some **#defines**, without which CPython creates platform-specific assembly code.

Compilation using llvm-gcc generates approximately 27.9MB of LLVM assembly. After running Emscripten and the Closure Compiler, the generated JavaScript code is approximately 2.76MB in size. For comparison, a native binary version of CPython is approxiately 2.28MB in size, so the two differ by only 21%.

The **CORRECT_OVERFLOWS** option in Emscripten is necessary for proper operation of the generated code, as Python hash code relies on integer overflows to work normally.

The demo can be seen live at `http://www.syntensity.com/static/python.html`. Potential uses include ... etc.

## 4.2 Bullet Physics Engine

Mention other bullet-¿js manual port, via Java

## 4.3 Lua

Mention other lua-¿js compilers

# 5 Summary

We presented Emscripten, an LLVM-to-JavaScript compiler, which opens up numerous opportunities for running code written in languages other than JavaScript on the web, including some not previously possible. Emscripten can be used to, among other things, compile real-world C and C++ code and run that on the web. In turn, by compiling the runtimes of languages implemented in C and C++, we can run them on the web as well. Examples were shown for Python and Lua.

One of the main tasks for future work in Emscripten is to broaden it's standard library. Emscripten currently includes portions of libc and other standard C libraries, implemented in JavaScript. Portions of existing libc implementations written themselves in C can also be compiled into JavaScript using Emscripten, but in general the difficulty is in creating a suitable runtime environment on the web. For example, there is no filesystem accessible, nor normal system calls and so forth. Some of those features can be implemented in JavaScript, in particular new HTML features like the File API should help.

Another important task is to support multithreading. Emscripten currently assumes the code being compiled is single-threaded, since JavaScript does not have support for multithreading (Web Workers allow multiprocessing,

but they do not have shared state, so implementing threads with them is not trivial). However, it would be possible to emulate multithreading in a single thread. One approach could be to not generate native JavaScript control flow structures, and instead to use a switch-in-a-loop for the entire program. Code can then be added in the loop to switch every so often between 'threads', while maintaining their state and so forth.

A third important task is to improve the speed of generated code. An optimal situation would be for code compiled by Emscripten to run at or near the speed of native code. In that respect it is worth comparing Emscripten to Portable Native Client (PNaCl), a project in development which aims to allow an LLVM-like format to be distributed and run securely on the web, with speed comparable to native code.

Both Emscripten and PNaCl allow running compiled native code on the web, Emscripten by compiling that code into JavaScript, and PNaCl by compiling it into an LLVM-like format, which is then run in a special PNaCl runtime. The major differences are that Emscripten's generated code can run on all web browsers, since it is standard JavaScript, while PNaCl's generated code requires the PNaCl runtime to be installed; another major difference is that JavaScript engines do not yet run code at near-native speeds, while PNaCl does. To summarize this comparison, Emscripten's approach allows the code to be run in more places, while PNaCl's allows the code to run faster.

However, improvements in JavaScript engines may narrow the speed gap. In particular, for purposes of Emscripten we do not need to care about *all* JavaScript, but only the kind generated by Emscripten. Such code is **implicitly statically typed**, that is, types are not mixed, despite JavaScript in general allowing assigning, e.g., an integer to a variable and later a floating point value or even an object to that same variable. Implicitly statically typed code can be statically analyzed and converted into machine code that has no runtime type checks at all. While such static analysis can be time-consuming, there are practical ways for achieving similar results quickly, such as tracing and local type inference, which would help on such code very significantly, and are already in use or being worked on in mainstream JavaScript engines (e.g., SpiderMonkey).

The limit of such an approach is to perform static analysis on an entire program compiled by Emscripten, generating highly-optimized machine code from that. As evidence of the potential in such an approach, the PyPy project can compile RPython – something very close to implicitly statically typed Python – into C, which can then be compiled and run at native speed. We may in the future see JavaScript engines perform such static compilation, when the code they run is implicitly statically typed, which would allow Emscripten's generated code to run at native speeds as well. While not trivial, such an approach is possible, and if accomplished, would mean that the combination of Emscripten and suitable JavaScript engines will let people write code in their languages of choice, and run them at native speeds on the web.

Finally, we conclude with another another avenue for optimization. Assume that we are compiling a C or C++ runtime of a language into JavaScript, and that that runtime uses JIT compilation to generate machine code. Typically code generators for JITs are written for the main CPU architectures, today x86, x86_64 and ARM. However, it would be possible for a JIT to generate JavaScript instead. Thus, the runtime would be compiled using Emscripten, and it would pass the generated JavaScript to *eval*. In this scenario, JavaScript is used as a low-level intermediate representation in the runtime, and the final conversion to machine code is left to the underlying JavaScript engine. This approach can potentially allow languages that greatly benefit from a JIT (such as Java, Lua, etc.) to be run on the web efficiently.