HIGH-PERFORMANCE WEB SIMULATION

Jan Voracek[(a)]

[(a)] Faculty of Electrical Engineering and Informatics, University of Pardubice

[(a)]jan.voracek@student.upce.cz

ABSTRACT

This paper demonstrates a possibility of creating high-performance web simulations. It utilizes Emscripten and asm.js to create a performance-optimized simulation that runs in a web browser. The paper provides a short introduction into used technologies and compares the performance of optimized simulation of fluid dynamics with usual approach of creating web simulations.

Keywords: computer simulation, web-based simulations, javascript, compilation.

1. INTRODUCTION

Computer simulation is an effective approach to study the behaviour of various systems over time. Simulated systems are often quite complex, and therefore have usually high demands on performance of the environment where they are executed. When creating a simulation model, it is necessary to take this into account and use corresponding level of abstraction. For example, we can replace complex input systems (systems from which data is entered into our model) with random number generators with corresponding probability distribution.

Sometimes, however, even abstraction from everything unessential may not be enough. Some simulation systems are computationally intensive themselves. The selection of appropriate technologies for the implementation of the simulator can help here. Generally, we can follow rules saying that compiled languages provide higher performance than interpreted and that languages translated into native code provide higher performance than languages compiled into bytecode and run in a virtual machine (Java VM, CLR).

This paper, however, focuses on the web simulation and introduces possibilities of creating performance-optimized web simulations to the reader.

2. WEB SIMULATION

Web technologies are already used for several years in the field of simulation and are still gaining popularity. They are often utilized thanks to their portability. All you need to run a simulation is a modern web browser. You can run the simulation on a desktop computer, laptop, tablet, smartphone or smart TV (Voráček, 2015).

Lower performance is often referred as the biggest disadvantage of web simulation (Karták, 2015). We will see how it is possible to mitigate this disadvantage in the following chapters.

3. JAVASCRIPT

JavaScript is the only language that can run in the web browser[1] (on the client side). It is an interpreted language with a dynamic type system. This means that the source code is not compiled but runs directly.

All variables in JavaScript are dynamic. It means that their type can be changed by

[1] In Dartium web browser you can run scripts written in the Dart language. However, Dartium is not intended for common use. See www.dartlang.org/tools/dartium/.

assigning another value and their true type is determined on run time. When you assign a one to a variable, it will have some numeric type. Whether it will be integer or floating point number depends on JIT (Hanenberg, 2010).

JIT (just-in-time optimization) is a process when the interpreter executing the source code tries to optimize the code during the execution (Sanghoon, 2012).

4. ASM.JS

Asm.js is a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers. This sublanguage effectively describes a sandboxed virtual machine for memory-unsafe languages like C or C++. A combination of static and dynamic validation allows JavaScript engines to employ an ahead-of-time[2] (AOT) optimizing compilation strategy for valid asm.js code (Herman, 2014).

As JavaScript is a language with a dynamic type system, asm.js uses type annotations to indicate the type of variable. For example:

- `var x = f()|0;` tells that x is an integer,
- `var y = +f();` tells that y is a double,
- `var z = f()>>>0;` uses output as unsigned int.

Another very important part of asm.js specification is typed arrays. Software written in JavaScript usually uses simple, generic arrays (in fact they are maps). However, JavaScript contains also typed, fixed arrays which have better both memory and time complexity, for example:

- `Uint8Array` – array of 8-bit unsigned integers,
- `Int32Array` – array of 32-bit unsigned integers,
- `Float64Array` – array of 64-bit floating point numbers.

See the JavaScript reference for more.

5. EMSCRIPTEN

Emscripten is a source-to-source compiler that runs as a back end to the LLVM (Low Level Virtual Machine) compiler and produces a subset of JavaScript known as asm.js described in the previous chapter (Zakai, 2011).

Emscripten also do some optimization of the code, for example:

- **Variable nativization:** Converts variables that are on the stack – which is implemented using addresses in the HEAP array (see Zakai, 2011 for more) – into native JavaScript variables.
- **Relooping:** Recreate high-level loop and if structures from the low-level code block data that appears in LLVM assembly.

Emscripten's compilation approach is to generate "natural" JavaScript, as close as possible to normal JavaScript on the web, so that modern JavaScript engines perform well on it (Zakai, 2011).

However, there are some limitations of Emscripten:

- **64-bit Integers:** JavaScript numbers are all 64-bit doubles, with engines typically implementing them as 32-bit integers where possible for speed. A consequence of this is that it is impossible to directly implement 64-bit integers in JavaScript, as integer values larger than 32 bits will become doubles, with only 53 significant bits (Zakai, 2011).
- **Multithreading:** JavaScript has Web Workers, which are additional threads (or processes) that communicate via message passing. There is no shared state in this model, which means that it is not directly possible to compile multithreaded code in C++ into JavaScript (Zakai, 2011).

6. LLVM

An LLVM is a compiler or a compiler infrastructure which compilers can be implemented in.

A simple diagram illustrating the LLVM workflow is shown in Figure 4. An LLVM

[2] Ahead-of-time optimization is an approach used by compiled languages where the optimization runs within the compilation step.

consists of a front-end and a back-end. The front-end translates high-level programming languages, e.g. C, C++, Java, Python, into a LLVM-IR (intermediate representation), which is a low-level programming language similar to assembly languages and is a language independent code. The back-end then translates the LLVM-IR into the architecture- and hardware-specific code (Lattner, 2004).

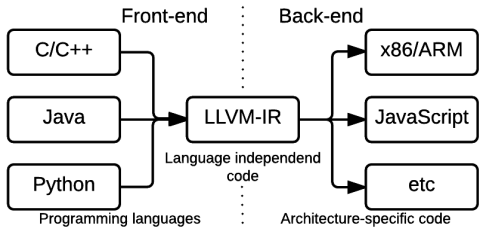

Figure 1 – LLVM architecture

7. PERFORMANCE

A computation of the Fibonacci sequence was chosen as a simple test for initial performance comparison. You can see the source codes below. The experiment was made with plain JavaScript (executed in NodeJS), for C++ (compiled by clang) and with asm.js (also executed in NodeJS).

This microbenchmark is indeed very simple; however, even this simple code can demonstrate that the performance asm.js is quite higher than hand-written JavaScript.

```
#include <iostream>

long fib(int x) {
    if (x < 2) {
        return 1;
    } else {
        return fib(x - 1) + fib(x - 2);
    }
}

int main() {
    long result = fib(50);
    std::cout << result << std::endl;
    return 0;
}
```

Source code 1 – Computation of 50th number of Fibonacci sequence in C++

Code written in JavaScript is not very different. Besides the data types they are substantially identical.

```
function fib(x) {
    if (x < 2) {
        return 1;
    } else {
        return fib(x - 1) + fib(x - 2);
    }
}

var result = fib(50);
console.log(result);
```

Source code 2 – Computation of 50th number of Fibonacci sequence in plain JavaScript

In the following example you can see the code generated by Emscripten. It is a JavaScript code compiled from C++ code (Source code 1). You can see, for example, the type annotations mentioned in chapter 4 or that the AOT optimization removed one of the recursive calls.

```
function __Z3fibl($x) {
    $x = $x|0;
    var $0 = 0, $1 = 0, $2 = 0, $3 = 0, $4 = 0,
$5 = 0, $accumulator$tr$lcssa = 0,
$accumulator$tr1 = 0, $x$tr2 = 0, label = 0, sp
= 0;
    sp = STACKTOP;
    $0 = ($x|0)<(2);
    if ($0) {
        $accumulator$tr$lcssa = 1;
        return ($accumulator$tr$lcssa|0);
    } else {
        $accumulator$tr1 = 1;$x$tr2 = $x;
    }
    while(1) {
        $1 = (($x$tr2) + -1)|0;
        $2 = (__Z3fibl($1)|0);
        $3 = (($x$tr2) + -2)|0;
        $4 = (($2) + ($accumulator$tr1))|0;
        $5 = ($3|0)<(2);
        if ($5) {
            $accumulator$tr$lcssa = $4;
            break;
        } else {
            $accumulator$tr1 = $4;$x$tr2 = $3;
        }
    }
    return ($accumulator$tr$lcssa|0);
}
```

Source code 3 – Computation of 50th number of Fibonacci sequence – source code generated by Emscripten

On the following graph you can see the comparison of execution times. You may notice that the code compiled from C++ to JavaScript is quite faster than hand-written JavaScript.

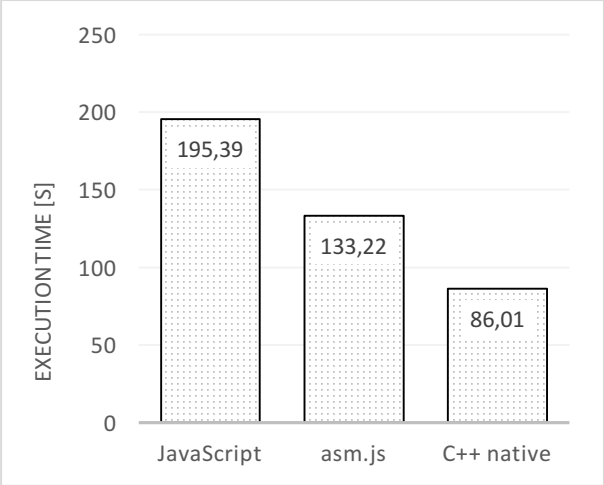

Figure 2 – Comparison of execution time of calculation of 50th number of Fibonacci sequence

As you can see on Figure 3, also other benchmarks confirm that JavaScript code compiled from C++ is almost always faster than hand-written JavaScript.

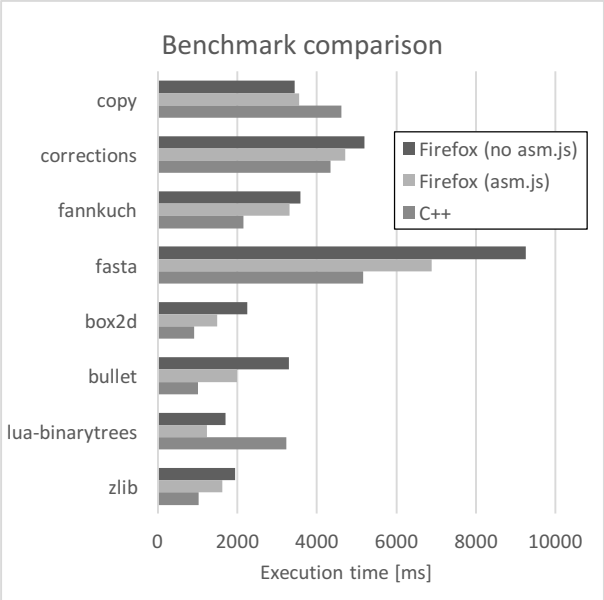

Figure 3 – Comparison of benchmark results [Data source: arewefastyet.com on March 24, 2016]

8. CASE STUDY

In 2013 Daniel Schroeder from Department of Physics at Weber State University, Utah has implemented a simple fluid dynamics simulator in JavaScript based on Lattice-Boltzman methods. You can see this simulator on Figure 4.

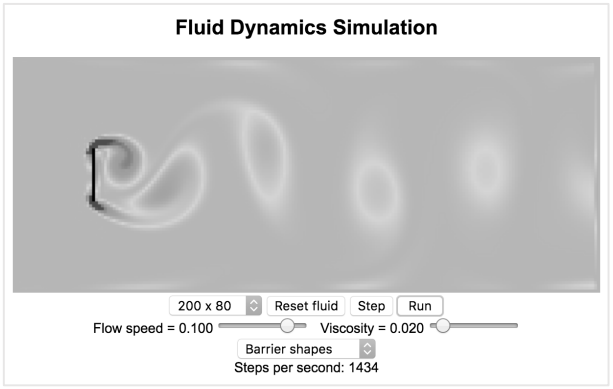

Figure 4 – Fluid dynamics simulator

This simulator was chosen as a reference for the comparison. It was reimplemented in C++ and transpiled back to JavaScript using Emscripten. It was chosen because fluid dynamics is not entirely trivial in terms of mathematical calculations and it is simple to animate.

As you can see on Figure 5, there is a difference in how many simulation steps per second are executed. The original JavaScript implementation performs 1050 steps per second while the C++ implementation compiled to JavaScript performs 1434 steps (average number of steps after 5 minutes of simulation). That is 36.57% higher performance.

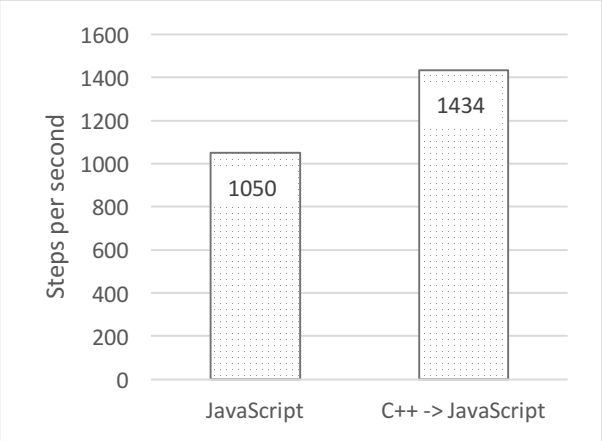

Figure 5 – Performance test of fluid dynamics simulators

9. CONCLUSION

Performance optimization is an important and frequently discussed topic, especially in the context of web-based simulation. Results presented in this paper indicate that implementing simulators in languages with strong type system like C++ and their

compiling to JavaScript is worth considering. JavaScript optimized by Emscripten gave often better performance than hand-written JavaScript.

There is also a lot of mathematical and physical libraries for C++ and this approach allows you to use them easily in your web-based simulators.

REFERENCES

Hanenberg S., 2010. An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications, pp. 22–35, New York, NY, USA.

Herman D., Wagner L., Zakai A., 2014. Specification of asm.js. Available from: http://asmjs.org/spec/latest/ [November 2015].

Karták Š., 2015. Web simulation as a platform for training software application. In Proceedings of the 27th European Modeling and Simulation Symposium (EMSS 2015), pp. 70-78, Bergeggi, Italia.

Lattner C., Adve V., 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '04), pp. 75-, Washington, DC, USA.

Sanghoon J., Jaeyoung C., 2012. Reuse of JIT compiled code in JavaScript engine. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12), pp 1840-1842, New York, NY, USA.

Voráček J., 2015. Web Simulation with Support of Mobile Agents. In Proceedings of the 27th European Modeling and Simulation Symposium (EMSS 2015), pp. 32-35, Bergeggi, Italia.

Zakai A., 2011. Emscripten: an LLVM-to-JavaScript compiler. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (OOPSLA '11), pp. 301-312, New York, NY, USA.