A Performant Scheme Interpreter in asm.js

Noah Van Es  Jens Nicolay  Quentin Stievenart  Theo D’Hondt  Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{noahves,jnicolay,qstieven,tjdhondt,cderoove}@vub.ac.be

ABSTRACT
This paper presents the implementation of an efficient in-
terpreter for a Scheme-like language using manually written
asm.js code. The asm.js specification defines an optimiz-
able subset of JavaScript which has already served well as a
compilation target for web applications where performance
is critical. However, its usage as a human-writable language
that can be integrated into existing projects to improve per-
formance has remained largely unexplored. We therefore
apply this strategy to optimize the implementation of an in-
terpreter. We also discuss the feasibility of this approach,
as writing asm.js by hand is generally not its recommended
use-case. We therefore present a macro system to solve the
challenges we encounter. The resulting interpreter is com-
pared to the original C implementation and its compiled
equivalent in asm.js. This way, we evaluate whether manual
integration with asm.js provides the necessary performance
to bring larger applications and runtimes to the web.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—interpreters,
optimization

Keywords
Interpreters, Optimization, JavaScript, asm.js

1. INTRODUCTION
Our study starts with the implementation of an efficient in-
terpreter for a Scheme-like language in JavaScript. Building
an interpreter in JavaScript enables a new language on the
web that inherently becomes available to millions of users,
as nearly each platform today is equipped with a browser
that includes a JavaScript virtual machine. In terms of per-
formance, however, a high-level language such as JavaScript
does not meet the necessary requirements for an efficient
language implementation. We therefore turn to a more op-
timizable, restricted subset of the language, asm.js [6] as
a means to improve the efficiency of performance-critical
JavaScript applications such as an interpreter. By eschew-
ing many of JavaScript’s dynamic features, asm.js promises
to deliver near native performance on the web. It limits the
language to numerical types, top-level functions, and one
large binary heap. With the addition of static typing, an
optimizing JS engine is able to compile and optimize asm.js
code ahead of time. The language remains a strict subset of
JavaScript, so existing engines are automatically backward
compatible with asm.js.

At this time, asm.js is mainly used as a compilation tar-
get. Developers start with an existing C/C++ application
which they can then efficiently port to the web by compiling
it to asm.js using Emscripten [9]. Our approach, however, is
different. We start with an existing JavaScript implementa-
tion and attempt to improve its performance by integrating
asm.js. The idea here is that using asm.js for the core com-
ponents of a JavaScript application improves the overall per-
formance of that application. Such a mix of JavaScript and
asm.js is possible, since the latter can interface with exter-
nal JavaScript code. We therefore apply this strategy to our
interpreter and rewrite its most crucial components (such as
the memory management) into asm.js. As a result, we itera-
tively refactor our application by lowering down its modules
into asm.js one-by-one. This produces a series of successive
implementations, where we expect to see an improvement
in performance for each iteration. We then benchmark each
such milestone to measure the actual performance impact of
this asm.js integration process.

Another point of interest is that we write this asm.js code by
hand. This is unconventional, as asm.js mainly serves as a
compilation target and is therefore not designed to be writ-
ten manually. As a result, we encounter several challenges
in our attempt to do so. For instance, we notice a severe
lack of readability and maintainability in asm.js applica-
tions. These are not really issues for a compiler, but they do
complicate the usage of handwritten asm.js at larger scales.
Furthermore, asm.js can be considered a low-level language,
offering similar functionality as C in a JavaScript syntax.
All data also has to be encoded into numbers and bytes, as
asm.js only supports numerical types. The top-level array
holding these numbers has to be managed manually, since
asm.js does not support any form of garbage collection.

These challenges, however, do not limit the possibilities of
asm.js. In order to deal with the restrictions in readability
and maintainability, we propose a solution using macros. By
using a specialized macro expander, many practical limita-
tions can be hidden into a more convenient syntax. Such
a preprocessor enables writing certain parts of the asm.js
code indirectly as a high-level, domain-specific language, and
therefore defines a more human-writable dialect of the lan-
guage. We illustrate this topic further in Section 3.1.

At the end, we take a step back and compare our handwritten implementation to the conventional strategy of compiling an existing C application into asm.js. We also compare the performance of our implementation with that of an equivalent version as well as the native implementation itself. In order to make this comparison, we first add some interpreter optimizations directly into the asm.js code. This also enables us to evaluate the maintainability of macro-enabled asm.js applications. The impact on development effort can then determine whether it is worth to write such asm.js code by hand.

Overall, this paper provides an experience report of our particular usage of asm.js. We make the following contributions:

- An overview of the performance impact that can be achieved by integrating asm.js into existing projects.
- A solution by introducing a macro preprocessor to improve readability, maintainability and performance when writing asm.js code by hand.
- A comparison between two different strategies using either handwritten or compiled asm.js to port runtimes and codebases to JavaScript.
- A handwritten implementation of a garbage-collected Scheme interpreter, written in asm.js to enable good performance on the web.

2. SETTING

We apply the strategy of integrating asm.js to the field of interpreters, where performance is usually a critical requirement. The language that the interpreter executes is Slip\(^1\), a variant of Scheme. An implementation of the language is available in C and is being used in a course on programming language engineering\(^2\). It served as the basis for the design and implementation of our own interpreter, named slipp.js.

The semantics of Slip \cite{slip} closely resembles that of Scheme. Differences are subtle and mostly limited to the usage of certain natives and special forms. Slip intends to go back to the original roots of the language and throws away many of the recent, non-idiomatic additions that are targeted more towards industrial engineering rather than an academic design language. For instance, it considers define to be the most appropriate construct for variable binding, and only provides a single let-form. Slip also enforces left-to-right evaluation of arguments, since not doing so is usually related to an implementation issue rather than a sound design choice.

The first version of the interpreter uses plain JavaScript only. It is ported over from a metacircular implementation of Slip and serves as a useful prototype that can be gradually lowered down to asm.js. Doing so enables the design of an efficient interpreter in a high-level language, without dealing with the complexity of asm.js as yet.

\(^1\)Simple Language Implementation Platform (also an anagram for LISP)
\(^2\)http://soft.vub.ac.be/~tjdhondt/PLE

2.1 Stackless design

The initial design already solves some of the shortcomings of JavaScript and asm.js. For instance, a trampoline and the continuation-passing style (CPS) alleviate the problem of uncontrolled stack growth in JavaScript due to the lack of proper tail-call recursion. The former allows putting all function calls in tail position, while the latter ensures that these tail calls do not grow the stack. This is necessary, since asm.js, as a subset of JavaScript, currently does not offer tail call optimization either.

Formally, we can define a trampoline \cite[p. 158]{lisp} as a function with input set \( S = \{f : \emptyset \rightarrow S\} \cup \{\text{false}\} \), which keeps on calling its argument thunk until it becomes \text{false}. Such a trampoline loop can be implemented in asm.js (or JavaScript) using an iterative construct as illustrated below. The example also shows the usage of bitwise operators and a function table, which are explained later on in Section 3.2.

\begin{verbatim}
function run(instr) {
  instr=instr&0;
  for(;instr;instr=FUNTAB[instr&255](0));
}
\end{verbatim}

Using this iterative loop, each function returns the function table index of the next function to be called.\(^3\) A separate stack is then used to store the continuation frames.

The result is a pure stackless design which does not rely upon the runtime stack of the host language to store the control context. Hence, stack overflows are no longer caused by the JavaScript stack size, and instead depend on the available heap memory. Using a custom stack for the CPS facilitates the implementation of advanced control constructs \cite[Ch. 3]{slip}, such as first-class continuations. It also makes it easier to iterate over the stack for garbage collection.

2.2 Optimized memory model

Instead of relying on the underlying memory management of JavaScript, the interpreter allocates its own memory chunks. It is accompanied by an iterative mark-and-sweep garbage collector. The memory model takes over many optimizations from the original C implementation, such as the usage of tagged pointers. This allows us to inline simple values and avoids indirections to unnecessary memory chunks.

Using a custom chunked memory model is a necessary provision, since asm.js does not provide any support for objects or garbage collection. Moreover, Slip keeps all its runtime entities in a single large address space. This makes it easier to map this heap onto the single memory array that is used to store values in asm.js.

2.3 Register-machine architecture

The interpreter also avoids the usage of local variables and arguments and opts for a register-machine architecture instead. This is possible because the evaluation process is transformed into CPS and therefore only uses tail calls. Such an iterative process is known to require only a fixed amount of iteration variables. In summary, around twenty dedicated registers are available for this purpose and shared

\(^3\)due to the lack of first-class functions in asm.js
throughout the entire interpreter infrastructure. Each one of them serves a specific purpose. For instance, the KON-register stores the current continuation, whereas the FRM and ENV hold the current frame and environment.

With only tail calls and no local variables or arguments, the host stack remains untouched. This not only facilitates the implementation of garbage collection, but also provides significant performance improvements. Furthermore, asm.js is able to store the contents of these registers very efficiently by mapping them to raw 32-bit words.

2.4 Imperative style

Finally, due to the transformation to CPS and the usage of registers, the interpreter follows a very low-level, imperative style. In fact, the evaluator shows a lot of similarity with the explicit-control evaluator from the SICP handbook [1, pp. 547–566]. Having such code in the initial prototype makes the transition to the low-level constructs of asm.js easier later on.

3. ASM.JS INTEGRATION

3.1 Integration process

The integration process lowers down as many of the modules in the interpreter to asm.js, starting with the most critical ones. Each iteration is expected to improve performance of the previous one by refactoring another component. Completing this process then results in an interpreter that is almost entirely written in asm.js.

System decomposition. Figure 1 shows the interpreter pipeline. The program input is first preprocessed by two analyzers, a parser and a compiler. The former constructs a basic abstract syntax tree (AST), while the latter performs some optimizations at compile-time. The compiler employs a rich abstract grammar that is able to provide more static information to the interpreter than the original AST. This takes away some of the processing work for the evaluator and thus improves run-time performance. A pool is used to store all the symbols for enabling efficient comparisons using pointer equivalence. The resulting AST is then interpreted by the evaluator, which forms the core of the interpreter. Finally, a printer presents resulting values appropriately. Two other important modules are the abstract grammar and the memory management. As indicated by the heavy line in Figure 1, these critical modules form the foundation for the entire interpreter infrastructure, since all modules rely on the memory management for allocation and garbage collection. Likewise, slip.js uses the unified abstract grammar extensively for both values as well as expressions.

Milestones While lowering down the modules into asm.js, four different milestones were identified. asm0 refers to the original prototype version in plain JavaScript. It makes no use of asm.js. asm1 is the first step in the integration process. It lowers down the memory management into asm.js, as it is one of the most critical components in the interpreter. asm2 also refactors the abstract grammar and merges it with the memory management into a single asm.js module. asm3 is the biggest leap in the refactoring process. Due to limitations in working with multiple asm.js and non-asm.js modules (cf. Section 5) most of the other components are lowered down into a single asm.js module at once. Figure 2 shows how much of the total code in each version is written in asm.js. These ratios are not representative of the actual contribution of asm.js, as some components are more important than others. Instead, they merely visualize how the asm.js and JavaScript mix evolved over time.

Overview The integration process starts with the most performance-critical components of our application, in this case the memory management and abstract grammar. Afterwards, other core modules such as the evaluator and natives are lowered down into asm.js as well. The colors in Figure 1 indicate the final result after all the refactoring up to asm3. Modules colored black are completely written in asm.js, whereas the grey boxes indicate the presence of regular JavaScript. The only component that does not use any asm.js is the printer, hence the white box in the diagram. The memory management and the abstract grammar are the first components lowered down into asm.js. This choice is motivated by the fact that they are frequently used throughout the interpreter, and therefore have a considerable impact on performance. This transition is also quite gentle, since the memory model was already designed at the bit-level in the prototype. Similar performance considerations also hold for the evaluator and the natives that make up the core of the interpreter and should therefore be optimized as much as possible. Having the compiler and parser partially lowered down into asm.js has more to do with convenience to facilitate interaction with other modules, rather than true performance concerns. They are only invoked once before execution and are not considered a bottleneck in the interpreter. For this reason, the parser is not entirely written in asm.js. String manipulation is also much easier in traditional JavaScript, so the parser relies on a separate external module to iterate over the input program string. The same argument also holds for the pool. Designing an efficient map between Slip symbols (i.e. strings) and their respective index in the pool is not trivial in asm.js. This is much easier in JavaScript, since we can simply use an object for this map. As a consequence, the pool module still communicates with
expressions.

if lowing high-level description to define AST-nodes for the memory manager. For instance, we can use the fol-

tion of an AST node, the macro expander generates the flexibility and avoid duplication. Given a high-level descrip-

tion of the interpreter. The macro transforms the descri-

tion name automatically gets replaced with its index into the function table. Padding involves adding extra

nop functions at the end of the table. Table 1 demonstrates the usage of this macro and compares it with the expanded source code. Using symbolic names to refer to functions clearly improves both readability as well as maintainability. Moreover, the trampoline that is generated by the macro ensures that no jumps between labels grow the stack. The example shows how if-expressions are evaluated in the evaluator. For this purpose, it uses the accessors that were generated by the struct-macro. We omit certain parts of the original source code here (indicated by ‘...’) for the sake of brevity.

4. OPTIMIZATION

The previous sections discussed how the interpreter was first designed in a high-level language (JavaScript), and then systematically translated into a low-level subset (asm.js). In or-
der to evaluate the maintainability of handwritten, macro-enabled asm.js applications, however, it is also interesting to add new functionality directly into that asm.js code. We
elaborating this strategy using a Huffman encoding of tags such as a small integer, instead of an actual pointer. Further, if the LSB is set, the other 31 bits can hold any immediate value, making the usage of these bits more efficient. This enables more values to be inlined, which reduces memory access even further, while still maintaining a reasonable value range for each immediate type. For instance, small integers only use two bits for their tag, leaving the remaining 30 bits free to represent a numerical value. Local variables on the other hand require five bits to recognize their tag. This still gives them a substantial range of $2^{27}$ values to indicate the offset of the variable in the frame.

### 4.1 Interpreter optimizations

Most of the optimizations included in asm4 are traditional interpreter optimizations [8, Ch. 6]. We highlight some of them below.

**Lexical addressing** The first major improvement is the implementation of lexical addressing. Slip, as a dialect of Scheme, employs static binding, where free variables are looked up in lexical environments. The exact frame and offset where a variable can be found therefore can be determined without executing the program. Lexical addressing builds up a static environment at compile-time, and replaces each variable occurrence with the index of the frame in the environment and the variable’s offset into that frame (also known as lexical addresses or De Bruijn indices [2]). The evaluator can then use these indices to access a variable in constant time instead of looking up the variable at run-time.

**Rich abstract grammar** Other large performance improvements are achieved by further diversifying the abstract grammar and detecting more static features at compile-time. Doing so provides more information to the evaluator and further improves run-time performance of the interpreter. For instance, any proper Slip implementation should support tail call optimization. Whether a function call is in tail-position is a static feature, and therefore it makes sense to detect such tail calls at compile-time. Another major heuristic observation is that most function applications use a simple expression (such as a variable) in operator position. Detecting and marking such applications optimizes their execution at run-time by avoiding unnecessary stack operations.

**Tagged pointers** In order to avoid unnecessary chunks and indirections, the initial prototype already employs tagged 32-bit pointers to inline small values. More precisely, if the LSB is set, the other 31 bits can hold any immediate value, such as a small integer, instead of an actual pointer. Further elaborating this strategy using a Huffman encoding of tags makes the usage of these bits more efficient. This enables more values to be inlined, which reduces memory access even further, while still maintaining a reasonable value range for each immediate type. For instance, small integers only use two bits for their tag, leaving the remaining 30 bits free to represent a numerical value. Local variables on the other hand require five bits to recognize their tag. This still gives them a substantial range of $2^{27}$ values to indicate the offset of the variable in the frame.

### 4.2 asm.js optimizations

Another improvement in performance involved optimizing the code we write and generate in the underlying language, in this case asm.js. One weak point in writing asm.js by hand is that it is designed as a compilation target. Some JavaScript engines therefore assume that common optimizations, such as the inlining of procedures, are already performed while generating the asm.js code in the first compilation step. This enables faster AOT-compilation of asm.js later on. To compensate for this, our handwritten application requires some profiling to manually identify and optimize certain bottlenecks in performance.

We therefore inline the most common functions in our application by replacing them with macros. Doing so avoids the overhead of function calls by replacing the call with the
functions body at compile-time. A macro expander enables us to factor out these function bodies into isolated macros, thereby maintaining the benefits of procedural abstraction. The multi-step expansion of macros in sweet.js also makes it possible to define macros that generate other macros. For instance, the struct-macro generates macros for the accessors and mutators of the AST nodes. Such a technique achieves significant performance improvements with a relatively small amount of effort.

5. EVALUATION
In order to evaluate performance, the runtimes of a fixed benchmark suite is measured for different versions of the interpreter. These include the four milestones discussed in Section 3.1 (asm0, asm1, asm2, asm3), as well as a final version asm4 that implements the additional interpreter optimizations described in Section 4. This final version can also be compared with the original Slip implementation and the asm.js output that the Emscripten compiler [9] generates from this C implementation.

A description of the benchmarks is given below. Most of them originate from the Gabriel and Larceny R7RS benchmark suites.

- **tower-fib** A metacircular interpreter is executed on top of another metacircular interpreter. In this environment, a slow recursive fibonacci is called with input 16. This benchmark also serves as a useful test case, since it provides almost full coverage of the interpreter.

- **qsort** Uses the quicksort algorithm to sort 500000 numbers.

- **hanoi** The classical hanoi puzzle with problem size 25.

- **tak** Calculates takSU (up to 5) using a recursive definition.

- **cpstak** Calculates the same function as tak, this time using a continuation-passing style. A good test of tail call optimization and working with closures.

- **ctak** This version also calculates the Takeuchi function, but uses a continuation-capturing style. It therefore mainly tests the efficiency of call-with-current-continuation.

- **destruct** Test of destructive list operations.

- **array1** A Kernighan and Van Wyk benchmark that involves a lot of allocation/initialization and copying of large one-dimensional arrays.

- **mbrot** Generates a Mandelbrot set. Mainly a test of floating-point arithmetic.

- **primes** Computes all primes smaller than 50000 using a list-based sieve of Eratosthenes.

Each version of the interpreter runs on top of the three major JavaScript VMs found in today’s browsers: SpiderMonkey, V8, and JavaScriptCore. SpiderMonkey deserves particular attention here, as this is the only VM implementing AOT-compilation for asm.js and should thus benefit from asm.js code. The other engines optimize asm.js to a certain extent, and should also benefit from other generic optimizations that are applicable to asm.js code [6]. The native version is compiled using Apple’s version (6.1) of the LLVM compiler. The test machine is a Macbook Pro (Mid 2012), 2.6GHz Intel Quad-Core i7, 16GB 1600Mhz DDR3 RAM. The VMs were allocated with a 1GB heap to run the benchmarks.

5.1 Integrating asm.js
We first evaluate the performance impact of integrating asm.js into an existing JavaScript application. We compare the benchmark results of asm0, asm1, asm2 and asm3, representing different milestones in the asm.js integration process (cf. Section 3.1). We slightly modify asm0 to use the underlying JavaScript memory management for allocation and garbage collection, instead of the memory model described in Section 2.2. This enables a more representative comparison between JavaScript and asm.js, as JavaScript already provides built-in memory management. Figure 3 shows a relative performance ratio for each version in SpiderMonkey, which optimizes the execution of asm.js using AOT-compilation. We obtain these ratios by normalizing all measurements with respect to those of asm0 and summarize them with their geometric means [4]. These results show that integrating asm.js into the original JavaScript prototype (asm0) does not yield the expected performance improvement. In fact, lowering down the memory management, a crucial component in the interpreter, slows down the entire system by a factor greater than 5. On the other hand, the final version with all modules refactored into asm.js does significantly perform better than the original version. In this case, we are seeing a performance improvement of around 80%.

In order to explain these results, we profile each version to determine what causes the initial slowdown. As it turns out, a lot of overhead in SpiderMonkey is caused by calling in and out of asm.js code. This is a known issue with asm.js code that is compiled ahead of time: external JavaScript interfaces asm.js modules through a set of exported functions and vice versa. Passing arguments to those functions requires JavaScript values to be converted and (un)boxed, even between two asm.js modules. Moreover, the function call itself causes trampoline entries and exits in asm.js that build up a severe performance penalty as well. For this reason, it is recommended to contain most of the computation inside a single asm.js module.

For our application, asm1 and asm2 have a lot of calls in and out of asm.js modules, as they refactor the memory model (asm1) and abstract grammar (asm2). Other components
rely heavily on these modules, as previously discussed in Section 3.1. In this case, the overhead caused by frequently calling into these asm.js modules is apparently larger than the performance benefits we achieve, hence the slowdown. On the other hand, asm3 uses only a single asm.js module for all the components in the interpreter. Moreover, all major computation resides inside this module. It only calls to external JavaScript for special services (such as I/O) and infrequent tasks (such as parsing and printing). This explains why it does not suffer from the aforementioned overhead and thus greatly benefits from the integration with asm.js.

It is also interesting to examine the performance impact of asm.js on other, non-optimizing engines. After all, we expect asm.js to provide us with general performance improvements, as it claims to be an optimizable subset of JavaScript. Figure 4 shows how JavaScriptCore, an engine that does not perform AOT-compilation for asm.js code, handles the different iterative versions. The results shown are geometric means of normalized runtimes. In general, we can conclude that the integration of asm.js is beneficial for the other engines in terms of performance. These engines do not compile asm.js ahead-of-time, and therefore do not benefit as much from its presence compared to SpiderMonkey. However, even typical JIT execution of asm.js is able to achieve a significant performance increase over the original version here up to 70%. Moreover, the engine does not suffer from the performance overhead of asm1 and asm2, unlike SpiderMonkey.

5.2 Comparison

We now look at the final, optimized version of slip.js, which we refer to as asm4. Table 2 shows how this version performs in today’s most common JavaScript engines. These results clearly demonstrate that the AOT-compilation of asm.js (in SpiderMonkey) is able to provide a significant performance improvement over traditional JIT execution (in JavaScriptCore, V8). To put these numbers in a better perspective, we can compare the results from SpiderMonkey with the runtimes of an equivalent native C implementation of Slip. Additionally, we compile this native version to asm.js using Emscripten. We refer to these versions as native and compiled, respectively. Figures 5 and 6 illustrate how these versions compare in terms of performance.

![Figure 4: normalized runtimes in JavaScriptCore](lower is better)

| Table 2: runtimes of asm4 (in milliseconds; lower is better) |
|---------------------------------|-----------------|-----------------|
|                               | SpiderMonkey   | JavaScriptCore  | V8       |
| tower-fib                     | 3518           | 6865            | 12142    |
| nqueens                       | 1296           | 2433            | 4677     |
| qsort                         | 3948           | 6934            | 14219    |
| hanoi                         | 4046           | 8899            | 18711    |
| tak                           | 878            | 1629            | 3374     |
| cpstak                        | 985            | 2110            | 4412     |
| ctak                          | 5222           | 7112            | 20380    |
| destruct                      | 4350           | 10029           | 19643    |
| array1                        | 3518           | 7724            | 16161    |
| mbrot                         | 12838          | 23648           | 49252    |

![Figure 5: comparison of runtimes using SpiderMonkey](milliseconds; lower is better)

Overall, we see that the performance of asm4 is comparable to that of the native version. We only experienced a slowdown factor of 1.19 in our benchmarks. The traditional strategy, however, uses asm.js as a compilation target. Typically, asm.js code that is compiled from C/C++ with Emscripten is only twice as slow as the native version [9]. In our experiments, the slowdown factor for the compiled version in SpiderMonkey was 1.74. This makes it around 46% slower than our handwritten implementation.

One case were both native and compiled significantly outperform asm4 is the ctak benchmark. Due to a small difference in the design of the original implementation and asm4, the latter requires that the construction of the current continuation into a first-class value performs a shallow copy of
the stack, while the original implementation only needs to copy the pointer to the current stack. This explains why call-with-current-continuation, the main point of interest in the ctak benchmark, is implemented more efficiently in native and compiled.

Compilation to asm.js is not always straightforward. Emscripten requires that the original source code is “portable”. For instance, the C implementation of Slip is not completely portable due to unaligned memory reads and writes. This resulted in some problems when working with floats in SpiderMonkey, which is why we have omitted the mbrot benchmark here. Similarly, JavaScriptCore was unable to run the compiled version unless we lowered the LLVM optimization level (which in turn worsened performance). Emscripten specifies the behavior of unportable C code as undefined.

6. RELATED WORK

Emscripten [9] provides a compiler back-end to generate asm.js from the LLVM intermediate representation format [7]. Any language that compiles into the LLVM IR can therefore be compiled into asm.js. We used this toolchain to compile the original C implementation of Slip to asm.js. Existing language implementations, such as LuaVM and CPython, have also been ported to the web with Emscripten.

PyPy.js [8], which implements Python in a restricted subset of Python which compiles to C and subsequently to asm.js, uses a different strategy and features a customized JIT back-end which emits and executes asm.js code at runtime.

LLJS [7] is a low-level dialect of JavaScript that was initially designed to experiment with low-level features and static typing in JavaScript, but it never reached a mature and stable release. While our approach uses domain-specific macros that translate directly into asm.js, LLJS defines a more general-purpose, low-level language that can also translate into asm.js.

7. CONCLUSIONS

Overall, our experiments allow us to evaluate the impact of integrating asm.js and writing asm.js applications by hand in general. In terms of performance, our strategy yields considerable improvements, as we achieve near-native performance on the web. The conventional toolchain of compiling an existing C application to asm.js using Emscripten in fact performed almost 50% slower than our handwritten implementation. Additionally, we can make the following conclusions on asm.js:

- Frequently calling in and out of asm.js modules compiled ahead-of-time causes a major overhead in terms of performance. Integrating asm.js into an existing JavaScript application is therefore only beneficial if all computation can reside in a single asm.js module.
- A macro preprocessor is necessary to alleviate the challenges in readability, maintainability and performance when writing asm.js by hand.
- Using asm.js to improve the efficiency of web applications comes down to a tradeoff between development effort and performance.
- At nearly native performance, writing asm.js by hand appears to be the best solution to get the most out of an application that can run on a billion-user VM.

8. REFERENCES


---

8http://www.py py.js.org
7http://www.lljs.org