Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR

David Leopoldseder[*] Lukas Stadler[†] Christian Wimmer[†] Hanspeter Mössenböck[*]

[*]Institute for System Software, Johannes Kepler University Linz, Austria [†]Oracle Labs

david.leopoldseder@jku.at {lukas.stadler, christian.wimmer}@oracle.com

hanspeter.moessenboeck@ssw.jku.at

Abstract

We present an approach to cross-compile Java bytecodes to JavaScript, building on existing Java optimizing compiler technology. Static analysis determines which Java classes and methods are reachable. These are then translated to JavaScript using a re-configured Java just-in-time compiler with a new back end that generates JavaScript instead of machine code. Standard compiler optimizations such as method inlining and global value numbering, as well as advanced optimizations such as escape analysis, lead to compact and optimized JavaScript code. Compiler IR is unstructured, so structured control flow needs to be reconstructed before code generation is possible. We present details of our control flow reconstruction algorithm.

Our system is based on Graal, an open-source optimizing compiler for the Java HotSpot VM and other VMs. The modular and VM-independent architecture of Graal allows us to reuse the intermediate representation, the bytecode parser, and the high-level optimizations. Our custom back end first performs control flow reconstruction and then JavaScript code generation. The generated JavaScript undergoes a set of optimizations to increase readability and performance. Static analysis is performed on the Graal intermediate representation as well. Benchmark results for medium-sized Java benchmarks such as SPECjbb2005 run with acceptable performance on the V8 JavaScript VM.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Compiler, Performance, Cross-Compilation, Decompilation

Keywords Java, JavaScript, Graal, ahead-of-time compilation, optimization

1. Introduction

JavaScript has become the standard language available in every web browser. To make existing applications and libraries written in other languages usable in a browser, they need to be translated to JavaScript. This makes JavaScript the assembly language for code generators, despite its shortcomings such as a limited set of types. Languages such as C [50], Java [16], and ActionScript [30] have been translated to JavaScript in many different ways.

We present a novel approach to translate Java bytecodes to JavaScript. Java is a mature language with a large ecosystem for optimized execution. Java virtual machines such as the Java HotSpot VM feature aggressively optimizing just-in-time compilers. Our approach leverages this existing infrastructure: we provide a new back end for an existing compiler that generates JavaScript source code from the compiler's high-level intermediate representation (IR). Since compiler IR usually allows unstructured control flow, we need to reconstruct structured control flow before code generation. We present the details of the algorithm in this paper.

It is infeasible to translate all Java methods of an application and its libraries to JavaScript, since that would include the whole JDK. Therefore, we use static analysis to find out which classes and methods are reachable from the main method of the application, and only translate these reachable parts. This allows us to translate large Java applications such as the SPECjbb2005 benchmark [37]. However, static analysis requires that all method invocations are symbolically analyzable, which precludes the usage of dynamic class loading and reflection.

We use the Graal compiler [32], an aggressively optimizing Java just-in-time compiler written in Java itself. Graal performs all standard compiler optimizations such as method inlining, global value numbering, constant folding, conditional elimination, strength reduction, and partial escape analysis. When used as the optimizing just-in-time compiler

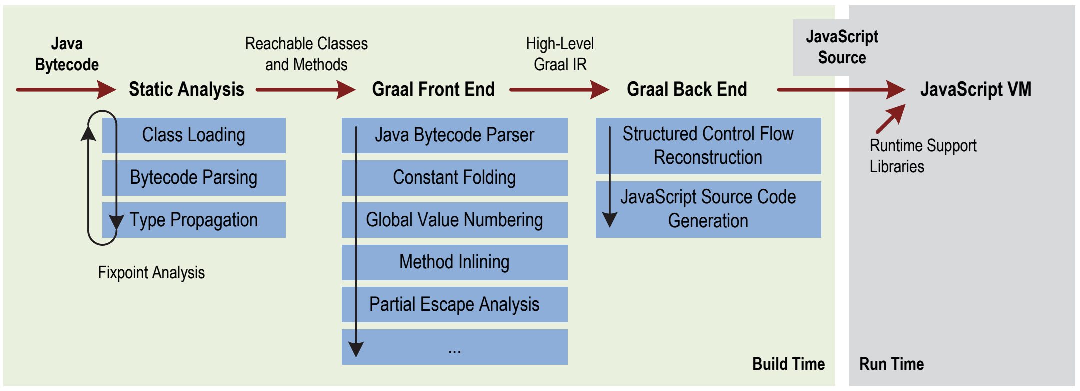


Figure 1. System structure.

of the Java HotSpot VM, the peak performance of the generated code is comparable to that of the Java HotSpot server compiler. Graal is modular and VM independent, which allows us to use it as an ahead-of-time compiler, re-use all high-level optimization on the IR, and only replace the back end. The high-level optimizations improve the quality of the generated JavaScript source code.

The output of our ahead-of-time JavaScript compiler, henceforth called Graal AOT JS, is a single, standalone JavaScript file that contains all reachable methods of all application classes. This code is easy to deploy and can be executed by the JavaScript VM without any additional transformations necessary at run time. The ahead-of-time compiled code is combined with a set of runtime support libraries modeling Java-specific concepts not available in JavaScript. Java objects are mapped to JavaScript objects, i.e., the runtime libraries do not need to include a garbage collector.

We demonstrate the feasibility of the approach with a set of standard Java and JavaScript benchmarks. We compare the generated JavaScript with native Java applications executed on the HotSpot server compiler [33]. While the overall performance heavily varies we can see a trend between $5x$ and $15x$ slowdown of the Graal AOT JS generated code executed on the V8 JavaScript VM to the HotSpot server VM. A large portion of the slowdown is related to additional checks in the generated code which are required by the Java semantics. For smaller benchmarks performance is between $2x$ and $10x$ slower than handwritten JavaScript code. Graal AOT JS can be used to compile arbitrary Java code to JavaScript. The generated code is moderate in size and acceptable in performance. Especially for websites the provided performance is feasible.

In summary, this paper contributes the following:

- We present a novel approach for source code generation from unstructured compiler intermediate representation.
- We describe an implementation of the approach to generate JavaScript source code from Java bytecodes.
- We use static analysis on the input bytecodes in combination with type flow analysis to decrease the resulting code size.
- We present optimizations that improve the readability and performance of the generated source code, leading to a high amount of restructured control flow.
- We present a performance evaluation showing that the approach can handle larger Java applications and produces reasonably fast JavaScript code.

2. System Structure

The presented ahead-of-time (AOT) compiler is based on the Graal OpenJDK project [32] and the underlying Graal virtual machine. Graal is a novel implementation of a Java just-in-time compiler written in Java. It runs on a modification of the HotSpot Java virtual machine. The Graal compiler conventionally serves as a just-in-time compiler, but for the purpose of compiling Java to JavaScript its capabilities have been extended to support AOT compilation.

Graal's compilation process is divided into a front end and a back end. The front end performs platform-independent Java-related optimizations like inlining or partial escape analysis. The back end performs optimizations that are not Java-specific like, e.g., read elimination, prepares the IR for code generation, and performs register allocation. We use the front end of Graal, but replace the default back end with our own implementation.

Figure 1 shows an overview of our system. The figure is divided into the two major parts of our system. The left hand side shows the ahead-of-time compilation of Java to JavaScript and the generation of JavaScript code. The first part happens at build time. The right hand side denotes the final standalone JavaScript application for deployment. At run time, the generated code and support libraries modeling the runtime system are executed by a JavaScript VM.

Static Analysis Compiling Java bytecodes ahead-of-time has one major drawback: code size. Java programs heavily use elaborate class libraries of the JDK. Thus, Java applications have numerous dependencies into the JDK, which produce a large call tree even for simple applications such as a trivial `HelloWorld` program. AOT compilation of a Java program with all its dependencies is therefore not feasible.

As Java applications only use portions of the imported classes, we need to remove unused methods and types. Our approach uses static analysis offered by Graal to reduce the size of the generated JavaScript code by removing unused elements. The analysis is based on a *closed world assumption*. For a given *entry point method* our analysis iteratively processes all transitively reachable types, methods and fields that are necessary to execute the code of the entry point method. In this process all required types and their fields and methods are identified. A complete call tree as well as a list of used methods and types is built, which allows us, e.g., to treat classes without subclasses as final.

Compilation Pipeline The following list describes the steps of the compilation pipeline on a finer granularity:

1. Class loading: Classes are loaded, meta-information is collected and all static initializers are executed. Several optimizations require access to constants at compile-time, therefore the associated classes need to be loaded for compilation. Accessing constants at compile time requires code generation to keep identity semantics for all constants. Graal AOT JS writes constant as well as static data created by static initializer methods to an initial JavaScript heap. Alternatively static initializers could also be compiled to JavaScript, but this would prohibit many optimizations.
2. Static analysis: The static analysis phase uses Graal to build the IR for the methods encountered during analysis. Every transitively reachable method is parsed and the IR is built. Those parts of the class path that were not discovered to be reachable can be excluded from compilation. Discovering classes leads to class loading, i.e., the first two steps are executed until a fixpoint is reached and no more new classes are discovered.
3. Graal optimizations: Standard compiler optimizations of Graal are applied to the IR, e.g., global value numbering [7], constant folding, strength reduction [3], conditional elimination [38], method inlining, and partial escape analysis [39]. We can leverage these sophisticated compiler optimizations without any additional implementation effort. The performance impact of some of these optimizations is discussed in Section 5.
4. Graph canonicalization to structured control flow: A set of control flow transformations rewrites certain structures in the IR to produce structured control flow.
5. Control flow reconstruction optimization: The graph is analyzed for structured control flow, additional structural rewritings on the IR are performed and analysis information is saved for code generation.
6. Code generation: Each type and method is processed and JavaScript code is emitted using the information of the previous step for generation of JavaScript control flow statements.

Graal IR The intermediate representation of Graal [10, 11] is structured as a directed graph in static single assignment (SSA) form [8]. Each IR node produces at most one value. To represent data flow, a node has *input* edges pointing to the nodes that produce its operands. To represent control flow, a node has *successor* edges pointing to its successors. In summary, the IR graph is a superposition of two directed graphs: the data-flow graph and the control-flow graph. Note that the two kinds of edges point in opposite directions.

In the front end, Graal IR mirrors the Java bytecodes parsed during graph building. We use this high-level IR for Graal AOT JS because it is still platform independent, i.e., it does not introduce concepts that have no high-level representation in JavaScript. The disadvantage of using a higher-level IR is that certain optimizations are not possible yet, e.g., array bounds-check elimination as presented in [46]. In the high-level IR, an array load is represented as a single operation, not yet modeling the array bounds check.

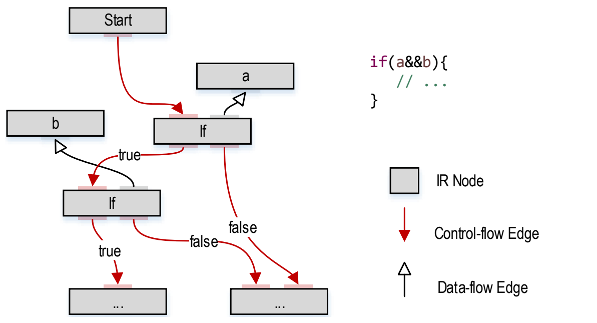
3. Structured Control Flow Reconstruction

Generating structured JavaScript source code from possibly unstructured Java bytecodes is challenging. A structured program disallows jumps to arbitrary locations in the program and consists of (or can be mapped to) a sequence of structured high-level statements such as if, switch, and while statements. Since Java bytecodes could have been written by hand instead of being generated by a Java compiler such a mapping is not always trivial to find. Furthermore, there are several patterns in Java that introduce unstructured control flow. We list two common ones of them below, but further details can be found in [27]. Since Graal IR must be able to cope with unstructured bytecodes it has to be unstructured, too.

We use the standard terminology of a *control flow graph* (CFG) and *basic block*. A control flow graph is a directed graph modeling the control flow of a method. The nodes of a CFG are basic blocks. A basic block denotes a list of instructions with just one entry and one exit point, so no jumps transfer control out of the basic block except for the last instruction of the block. For the IR we introduce the terms *split* and *merge* for special nodes. Split nodes are nodes with more than one successor, whereas merge nodes are nodes with more than one predecessor.

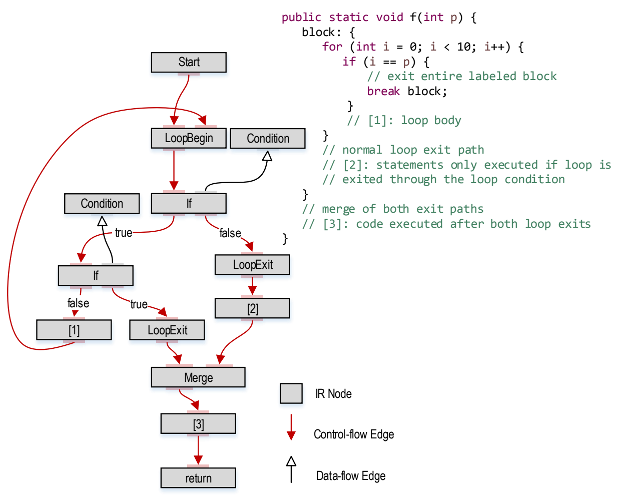
Compound Conditions Java's short circuit evaluation of compound conditions uses a common merge block. In Figure 2, for example, the false successor block for both evaluations of `a && b` is the same.

Multi-Exit Loops Loops with multiple exits leading to different successor nodes or loops with multiple backward



```
if(a&&b){
   // ...
}
```

Figure 2. Unstructured control flow: compound condition.



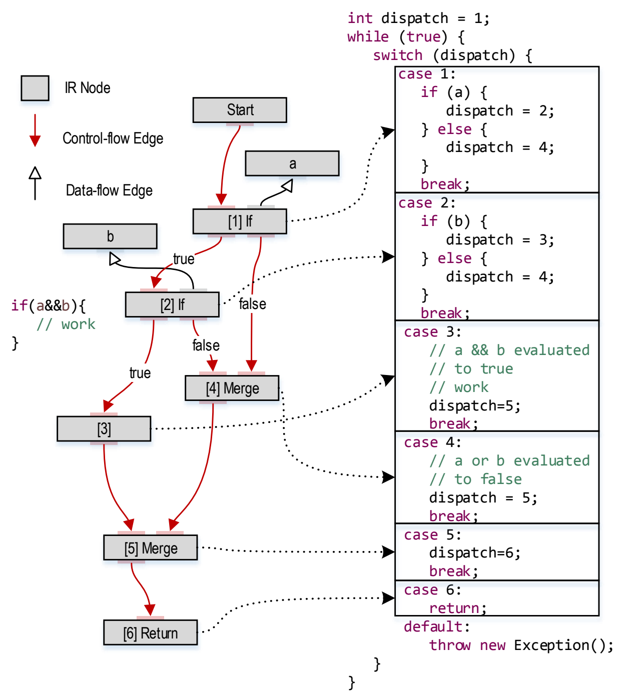
```
public static void f(int p) {
   block: {
      for (int i = 0; i < 10; i++) {
         if (i == p) {
            // exit entire labeled block
            break block;
         }
         // [1]: loop body
      }
      // normal loop exit path
      // [2]: statements only executed if loop is
      // exited through the loop condition
   }
   // merge of both exit paths
   // [3]: code executed after both loop exits
}
```

Figure 3. Unstructured control flow: multi-exit loop.

edges are unstructured [5]. Figure 3 shows an example of a multi-exit loop in Java. After the compilation to bytecodes there is no corresponding high-level representation for the given code snippet, except with the usage of labeled blocks. Code generation using labeled blocks requires one labeled block per loop exit. Graal AOT JS uses a different approach for the reconstruction of multi-exit loops. Section 3.2 illustrates how we handle the reconstruction of multi-exit loops in our compiler.

3.1 Control Flow Graph Block Interpreter

The presented examples of unstructured control flow motivate the need for a generic solution for modeling arbitrary control flow and for transforming it to structured control flow in a high-level language like JavaScript. We use a generic solution for modeling and transforming arbitrary control flow during code generation. It is based on an approach for the removal of `goto` statements presented by Erosa and Hendren [12]. In the domain of control flow obfuscation the approach is known as *control flow flatting* [20, 44]. Control flow is expressed by an endless loop dispatching CFG successors with a `switch` statement. Figure 4 illustrates the code resulting from a compound condition of the form `a && b`. Every case in the `switch` statement is a basic block of the CFG.



```
if(a&&b){
   // work
}
```

```
int dispatch = 1;
while (true) {
   switch (dispatch) {
   case 1:
      if (a) {
         dispatch = 2;
      } else {
         dispatch = 4;
      }
      break;
   case 2:
      if (b) {
         dispatch = 3;
      } else {
         dispatch = 4;
      }
      break;
   case 3:
      // a && b evaluated
      // to true
      // work
      dispatch=5;
      break;
   case 4:
      // a or b evaluated
      // to false
      dispatch = 5;
      break;
   case 5:
      dispatch=6;
      break;
   case 6:
      return;
   default:
      throw new Exception();
   }
}
```

Figure 4. Control flow graph block interpreter.

This generic pattern, although elegant and easy to adapt, has major disadvantages. The performance is between $2x$ and $> 10x$ slower than a structured representation of control flow. In order to improve the performance and the readability of the code it needs to be restructured.

3.2 Structured Control Flow Reconstruction Algorithm

To produce efficient JavaScript code we have to model as much control flow as possible with high-level control flow structures. We present a novel approach for reconstructing structured code from possibly unstructured control flow. Below we refer to the graph tagging algorithm in particular, as it marks the final structured control flow detection step. We refer to [5] for the theory behind structured control flow and its reconstruction.

Merging of Loop Exits To overcome the problem of having different successors for multiple loop exits we use a mechanism to have a structured set of loop exits. We merge all loop exits on a common node. We introduce a switch that dispatches the successors of the exits depending on the exit that was taken out of the loop. Figure 5 shows the example from Figure 3 after the merging of loop exits. Merging of loop exits removes potentially unstructured control flow introduced by a combination of multiple loop exits from the loop body. This enables the structuring of the loop body. Although control flow after the merge of all loop exits can be unstructured, the loop can be compiled to a structured piece of JavaScript code. Mapping the control flow of loops to structured JavaScript statements is crucial for performance since loops are the most frequently executed paths in a program. The structural rewritings in the IR during the merging of loops exits are intrinsic to the control flow reconstruction algorithm, thus necessary for the structuring of loop exits.

Graph Tagging For the reconstruction of structured control flow we use a heuristic that is based on general control flow concepts and is not limited to our Graal IR. We use the terms *link* and *walk*. A walk is an ongoing or currently stalled bottom-up traversal of the CFG to find structured sub-graphs. A link is a sub-graph that has already been identified to be structured. During the walks previously established links are skipped as they already contain structured control flow. *Final paths* are those walks that start at an instruction, which has no successor in the CFG (except one reached via a back edge), e.g., `break`, `continue`, `throw` and `return`.

The control flow tagging algorithm is applicable to deduce structured control flow, thus to ease the process of analysis, Graal AOT JS applies a set of structural transformation prior to control flow analysis. Cases where a callee unwinds with an exception and the caller unwinds to its caller again are removed completely, as JavaScript supports native exception handling. Exception handlers that cannot be removed are rewritten to invocations followed by if-statements that dispatch to the exception handler, which allows us to analyze exception edges with the tagging algorithm. For unwind and return nodes we expand them across the IR to produce as many final paths as possible. For those graphs containing unstructured control flow, or on which the tagging algorithm bails out, code generation uses the *block interpreter* as illustrated in Figure 4.

Our heuristic is based on the definition of structured control flow presented in [5]:

- All upward paths from a merge node m lead to the same split node s. Similarly, all non-final downward paths from a split node s lead to the same merge node m. In both cases, there must not be other merge or split nodes between s and m except in links that are already known to be structured.
- Every final path can always be mapped to JavaScript code. A loop can always be exited early or continued and a return can always be emitted in JavaScript.
- Additionally to the two definitions it is required that the number of predecessors of merge node m is less than or equal to the number of successors of the split node s. There might be fewer predecessors of m than successors of s if top-down paths starting at s are final.

In our algorithm (see Listing 1) we perform the following steps:

1. From every merge node and from every final node traverse all incoming edges upwards until a split or another merge is encountered. Such a traversal is called a walk (In Listing 1 this operation is denoted by the method `walkBack(curr,prev,start)`).
2. If a merge is encountered during a walk do the following:
   - If the merge is already associated with a split (`isMergeOfLink(merge)`), and is thus marked as being structured, skip the entire link and continue the walk at the split's predecessor.
   - If the merge is not associated with a split save the walk in a map with the merge as the key, thus stall the entire walk for possible later continuation (`save(key,Walk(curr,prev,start))`).
3. If a split is encountered during a walk do the following:
   - If the split is already part of a link, i.e., if it has already been encountered during a different walk, this can only happen with unstructured control flow, thus abort the walk.
   - If the split is not yet part of a link, store this walk at the split node. Once the number of stored walks for a split node equals the number of successors for this split node, check if all these walks started at the same merge node. If so, consider the sub-graph between the split and the merge as a link and consider it as structured. As the sub-graph between the merge and the split (and further potential links) is tagged, all walks that were stalled at the particular merge are restarted.
4. For all other nodes that are encountered continue the walk at the predecessor.

Special attention lies on the reconstruction of loops. Final paths are always structured, therefore, if the tagging algorithm is able to deduce that every control flow element in the body of a loop is structured, the entire loop can be compiled to a structured `while-true` loop. Final paths guarantee that loop ends and exits are mapped correctly to JavaScript.

There are different prominent approaches dealing with control flow reconstruction. The decompiler for the C language presented by Cifuentes in [5] is based on the construction of *derived sequences*, which basically iteratively collapses regions in the CFG until it is trivial, or can no longer be simplified. The approach from [5] differs from the presented graph tagging algorithm as graph tagging inplace restructures the control flow graph, without the need for collapsing of sub-graphs as already established links are skipped during walks.

A prominent decompiler for Java building on the Soot [41] bytecode optimizing framework is *Dava* [26] which decompiles Java bytecode based on the theory of *staged encapsulation (SET)*. The theory of staged encapsulation builds a tree based on the CFG. The SET tree contains nodes representing Java high level language constructs. Edges requiring an unstructured control flow transfer are resolved with labeled blocks. SET based decompilation requires several phases

```
// list of all nodes
List nodes = ir.allNodes();
void tagGraph(){
  foreach(Node n:nodes){
    if(n isa Merge){
      foreach(Node e:n.predecessors){
        walkBack(e, n, n);
      }
    }else if(isFinalNode(n)){
      walkBack(n.predecessor, null, null);
    }
  }
}
void walkBack(Node curr, Node prev, Node start){
  if(curr isa Merge){
    if(isMergeOfLink(curr)) {
      Node split = splitOfLinkAtMerge(curr);
      // skip link - continue walk after the link
      walkBack(split.predecessor, split, start);
    }else{
      // stall the walk
      save(curr /*key*/, new Walk(curr,prev,start));
    }
  }else if(curr isa Split){
    if(splitMustBeTagged(split)){
      tagWalk(curr/*split*/,prev,start/*merge*/);
    }
    if(allTagged(curr)){
      // create new link
      createLink(curr);
      // respawn walks stalled at the new link's merge
      respawnSavedWalk(mergeOfLinkAtSplit(curr));
    }
  }else{
    // arbitrary CFG node
    walkBack(curr.predecessor,curr,start);
  }
}
```

Listing 1. Control flow tagging algorithm.

whereas graph tagging only needs one for control flow reconstruction. Cases of unstructured control flow are handled differently by Dava. Dava tries to generate labeled blocks whereas Graal AOT JS requires phases prior to control flow reconstruction to rewrite unstructured control flow to structured one directly in the IR. Graal AOT JS then generates code containing unstructured control flow using the *block interpreter* approach.

4. Graal IR to JavaScript Code Generation

This Section describes the runtime system used for the execution of JavaScript code emulating Java semantics. It presents the relevant concepts of Java and their representation in the generated JavaScript code. In the first part we list all major features of Java and their realization in JavaScript. In the second part we present certain optimizations applied to the generated code that increase its performance.

4.1 Java to JavaScript

To overcome limitations in the language semantics of JavaScript an elaborated runtime system is necessary. The generation of a type system in JavaScript reflecting Java concepts requires semantically equivalent constructs for typing, dynamic binding and exception handling. In the following we present the relevant concepts of Java and their semantically equivalent constructs in JavaScript.

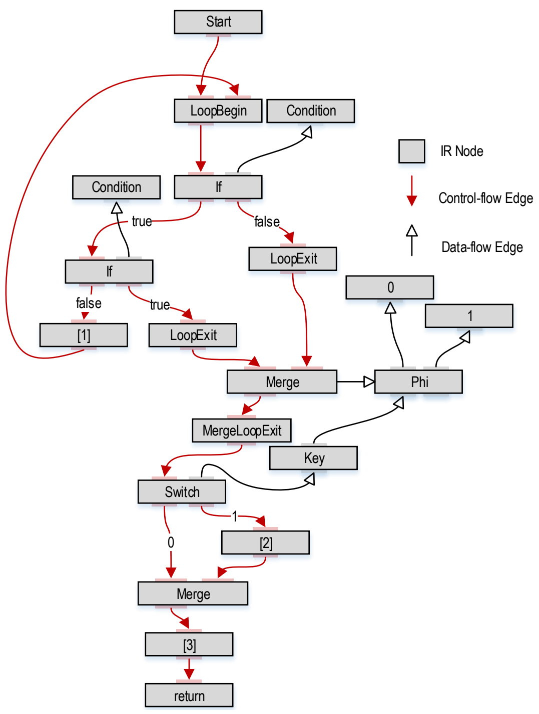


Figure 5. Structured control flow: multiple loop exits are merged.

Primitive Types

boolean, char, short, int Primitives are modeled with the support of ASM.js [28] as native JavaScript integers. JavaScript integers are 32 bit signed integers in two's complement representation. Runtime checks are necessary to ensure the valid value ranges of Java types smaller than `int`, e.g., for `short`. Operations on short might lead to values outside the 16 bit range and must therefore be checked for overflow and adjusted. Techniques to do so are presented in [45] and implemented in our back end.

long Longs needs to be emulated since the standard number type in JavaScript is the IEEE 754 double precision floating point type which has a maximum integral part in the range of $[-2^{53} : 2^{53}]$. We emulate longs with a JavaScript object that has two properties representing the high and low word of the long value, each of which is a signed 32 bit two's complement number.

float Java expressions of type float never exceed single precision. However, JavaScript only provides double precision numbers. To ensure an expression of type float never exceeds its precision every expression of type float must be checked. The ECMAScript 6 (Harmony) standard defines a function `Math.fround` [29] that rounds a double precision floating point number to nearest single precision number. To enable competitive performance AOT JS features a mode that omits checks for single precision floating point ranges and therefore might lead to floats that exceed single precision.

double Doubles are mapped to JavaScript primitive numbers.

Arrays JavaScript offers an intrinsic `Array` object that is capable of representing a Java array. Graal AOT JS features an optional compilation mode using `TypedArrays` [29] for primitive arrays.

Inheritance The Java inheritance model is directly mapped to JavaScript's prototype-based inheritance model. We build a JavaScript prototype chain reflecting the Java type tree with a `java.lang.Object` prototype as the root prototype.

Interfaces A Java variable can have an interface as its static type. JavaScript is an untyped language and does not support the concept of static types. To support type checks against interfaces we compile additional type bits to compiled `java.lang.Class` objects enabling those checks. Additionally, default methods introduced in Java 8 [21] require interface types to be compiled to JavaScript.

Exception Handling Java's exception class hierarchy is vital for the distinction between logical errors, runtime exceptions or even control flow dispatches. JavaScript makes a less effective distinction. In JavaScript every object can be thrown. The intrinsic `Error` object denotes runtime errors but exception rules are less strict, e.g., division by zero does not produce a runtime exception. Graal AOT JavaScript is able to use the native JavaScript exception handling mechanism. Our IR contains special edges for exception handlers [11]. Code for exception handlers is generated with the `try-catch` statement. Code generation for the code in the IR's exception edge emits an if statement after the `try-catch` block that is entered if the preceding statement produced an exception.

Unchecked Array Access and Unchecked Receivers The Java virtual machine specification [21] requires array accesses to be checked at runtime. For performance reasons we omit code generation of these kinds of checks. JavaScript does not produce an error if an array index is out of bounds, because such an array access is just treated as a new property access at the given index. If the checks are desired Graal AOT JS features a *safe array* mode emitting code for safe array access. Additionally Graal AOT JS features a mode omitting null checks on receivers, which removes null checks on every callsite.

Limitations In general, there are four concepts of Java that cannot be handled with AOT compilation to JavaScript.

- **Dynamic Class Loading:** Graal AOT JavaScript cannot support dynamic class loading as this would require runtime compilation which would require the compilation of Graal itself to JavaScript. Currently runtime compilation is not supported but this might change in the future in order to support the self-optimizing AST interpreter framework *Truffle* [47], on-top of Graal.
- **Reflection:** Java has an elaborate reflection mechanism enabling programmers to examine or modify the behavior of applications at runtime. AOT support for the Java reflection API is limited for the same reasons that dynamic class loading is not supported.
- **Multithreading:** Multithreaded applications introduce two problems with the presented approach. Our current static analysis is not capable of analyzing multithreaded applications appropriately. Another problem is JavaScript's inherent lack of a real concurrency model. JavaScript has `WebWorkers` [29] which offer the capability to execute code in a different thread. However this model does not feature shared memory. Data is exchanged via message passing and callbacks. Java and JavaScript do not share a common semantic notion of concurrency, thus this feature of Java is not supported by Graal AOT JS.
- **Synchronous APIs:** JavaScript does not support synchronous I/O whereas Java does. It is not possible to map synchronous Java APIs to JavaScript with AOT compilation. Such APIs like, e.g., the Java Socket API, must be re-written to work asynchronously. Currently, this is not supported by Graal AOT JS.

4.2 Optimizations

In this Section we present some major optimizations on the generated JavaScript code for improving performance. Some of the optimizations are particularly designed for the V8 virtual machine, but have no performance penalty on different JavaScript engines. We evaluated the optimizations on the benchmarks presented in Section 5.

Local Variable Inlining As presented in Section 2 Graal's IR is in static-single-assignment form [10]. Every node producing a value represents a new local variable in the underlying program. Performance and code size of the generated code would be compromised if every node producing a value were associated with a new local variable in JavaScript. Thus we use an optimization for inlining static-single-assignment nodes at their usages. Our static-single-assignment node inlining heuristic is based on the subsequent list of assumptions

- Nodes with less than 2 usages can be inlined.
- Null can always be inlined
- Constants of primitive types can always be inlined
- Nodes with side-effects are never inlined
- Nodes whose usage contains a conditional expression of the form a ? b : c are not inlined.

- Array accesses are not inlined when safe array mode is enabled as this requires additional bounds checks.

Using our static-single-assignment inlining heuristic between 20% and 30% of all IR data nodes can be inlined at their usage(s).

Exception Wrapping This optimization mainly pays off in JavaScript VMs that do not optimize exception handlers. The optimizing compiler of Google's V8, for example, bails out on methods containing the `try-catch` keywords.

Graal AOT JS is aware of all points in the code where Java exceptions might be thrown which enables us to model exception handling completely without using "native" JavaScript exception handlers. Whenever an implicit exception would be thrown such as de-referencing null or when violating an array bound we do not throw an exception with the JavaScript keyword "throw" but rather return a custom exception and exit the function on the return path. Listing 2 illustrates the approach. Function `caller1` calls function `f1` in plain exception mode and `caller2` calls `f2` in wrapped exception mode. It is crucial to point out that with exception wrapping *every* callsite must allocate the call's return value to a local variable. The return value must be checked for a pending exception even if the called function's return type is void. The advantage of this optimization is that (although the return type is not fixed and may be megamorphic) the function is still compiled with V8's optimizing compiler, which makes a big difference in performance as presented in Section 5.

```
// native JS Exceptions
function f1(){
  if(a){
    throw new java_lang_NullPointerException();
  }
}
function caller1(){
  try{
    f1();
  }catch(e){
    if(e instanceof java_lang_NullPointerExcpetion){
      ...
    }
    // else unwind
  }
}
// wrapped exception handlers
function f2(){
  if(a){
    return new ExceptionWrapperType(new
     java_lang_NullPointerException());
  }
}
function caller2(){
  var ret=f1();
  if(isException(ret)){
    var exc=ret.exception;
    if(exc instanceof java_lang_NullPointerException)
     {
      ...
    }
    // else unwind
  }
}
```

Listing 2. Exception wrapping optimization example (JavaScript).

5. Evaluation

We evaluated our AOT JS compiler, which has been implemented on-top of the Graal just-in-time compiler, by running and analyzing a set of benchmarks of different sizes. All benchmarks were executed on a desktop-class Intel i5 processor (2010) with 2 cores, 4 virtual threads featuring 8GB of RAM and a core speed of 2.4 GHz running Fedora 21(64 bit).

We ran each of the benchmarks presented below with different optimization configurations. As we wanted to compare the results against the Java HotSpot VM and against hand-written JavaScript versions of (some) of the benchmarks, we not only measure the peek performance but also the start-up performance. The HotSpot server compiler has a slow start-up performance whereas Google's JavaScript engine V8, which compiles every method upon its first execution with a simple base-line compiler has a fast start-up performance. On the other hand, the server compiler reaches a better peak performance than V8. Each test run consisted of three different configurations of the benchmark and ten iterations for each configuration. We took the arithmetic mean of all configurations and put them in direct comparison to the HotSpot server compiler. For the benchmarks binarytrees, deltablue, nbody, fasta and SPECjbb2005 we measured their execution time in milliseconds whereas for Linpack, SciMark2a and JBox2D we measured their score result.

We now give a short description of the used benchmarks. Their performance is shown in Figure 6.

binarytrees is part of the Computer-Language-Benchmark-Game (CLBG); it is a small benchmark performing many binary tree allocations and recursive calls up to a certain depth [6].

deltablue is a prominent one-way constraint solver, originally developed for the *Smalltalk* language by Maloney and Wolczko. [9].

nbody is a simple numeric benchmark performing an nbody simulation [6].

fasta (V#4) is also part of the CLBG generating and writing random DNA sequences.

linpack is a benchmark solving a set of linear equations, heavily relying on floating-point performance [22].

SciMark2a is a benchmark performing scientific computations including a fast Fourier transformation, a Jacobi successive over-relaxation, a Monte Carlo simulation, a sparse matrix multiplication and an LU matrix factorization [36].

JBox2D is a Java port of the prominent physics engine Box2d[4]. The Java port is released with its own benchmark called "Piston" [19] which sets up and performs a physical simulation over several iterations in various configurations.

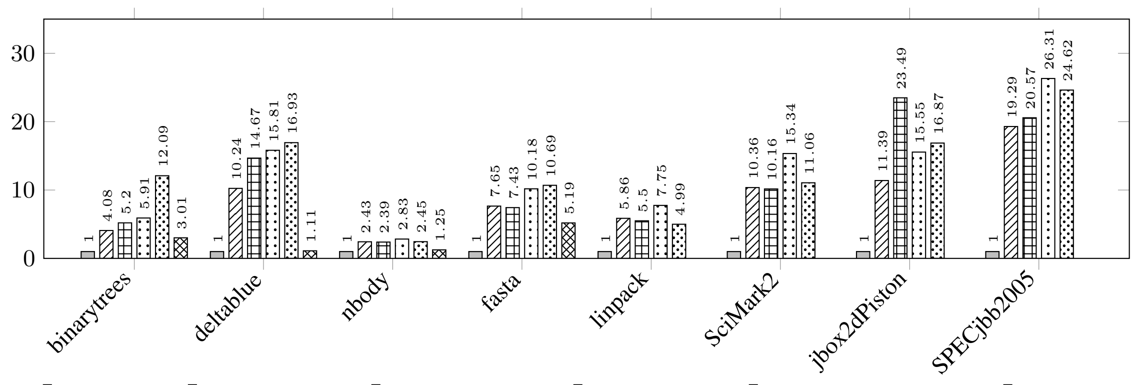


Figure 6. Relative performance of Graal AOT JS in different configurations compared to the Java HotSpot Server Compiler and native JavaScript(lower is better, missing number indicates the absence of a native benchmark for JS).

SPECjbb2005 A single-threaded sequential version of the original *SPECjbb2005* Java server benchmark [37] emulating a three tier client-server application. The modifications include a removal of the file logging and execution of all threads on the main thread. For AOT JS usage warehouses are loaded sequentially and a warehouse thread runs on the main thread. Iterations are limited to 5000 per warehouse. The execution was measured on four to eight warehouses with an increment of one.

5.1 Control Flow Analysis Optimization

Our control flow tagging optimization presented in Section 3.2 works on arbitrary Java bytecodes and yields a high amount of restructured control flow. The evaluation of the algorithm on the presented benchmarks can be seen in Table 1.

Table 1. Percentage of reconstructed control flow relative to the number of splits in an entire benchmark compilation.

Benchmark	Nr. of Splits	Nr. of Restructured Splits	Nr. of Methods with Unstructured Splits
binarytrees	131	120 (92%)	3 (8%)
delta blue	1177	918 (78%)	54 (20%)
nbody	8229	5910 (72%)	425 (28%)
fasta	1415	1008 (71%)	74 (24%)
linpack	976	714 (73%)	56 (25%)
Sci Mark 2a	1010	750 (74%)	61 (26%)
JBox2D	9576	6810 (71%)	499 (28%)
specjbb2005	12341	8871 (72%)	618 (28%)
Arithmetic Mean		75%	23%

More than 75% of the control flow splits could be restructured leaving less than 25% of the methods with unstructured control flow. The numbers leave space for further optimizations. Especially nested loops with shared exception handlers may currently lead to improperly restructured control flow.

The performance impact of our control flow reconstruction optimization can be seen Figure 6 and ranges from $2x$ to $5x$ on average.

5.2 Code Size

Table 2. Total code sizes of the benchmarks.

Java Benchmark	Java LOC (approx.)	JS Code Size
binarytrees	58	305 KB
delta blue	607	1.6 MB
nbody	130	10.6 MB
fasta	125	1.7 MB
linpack	284	1.3 MB
Sci Mark 2a	1890	4.3 MB
JBox2D	13 478	12.9 MB
SPECjbb2005	12 800	14.2 MB

Table 2 shows the sizes of the source code and the target code of the benchmarks when compiled with the Graal AOT JS compiler. The sizes range from a few lines (CLBG) to several thousand lines of code. Even small programs may result in an enormous amount of compiled code due to heavy usage of the JDK. For example, the nbody benchmark requires big parts of the JDK to be compiled, which results in a code size comparable to the one of SPECjbb2005. The general overhead in code size is high. Reasons for that are the required features that are compiled this includes each reachable type, a virtual method table for each type, a field offset table for unsafe memory access mapping to JavaScript and unstructured control flow. 20% to 50% of the code size can be attributed to the initial heap initialization. If more compact code is desired tools like [15] or [23] can be used, which achieve high compression rates and would decrease the code size by $30 - 60\%$. For the presentation of benchmarks we decided to evaluate the performance with unmodified code produced by AOT JS.

The benchmarks were compiled with the optimizations discussed in Sections 2, 3 and 4.

5.3 Runtime Performance

The performance measurements relative to the HotSpot server compiler[1] are presented in Figure 6. The server compiler offers the highest peak performance of all Java virtual machines currently on the market. For the execution of the JavaScript benchmarks that were generated from Java as well as for the original JavaScript benchmarks we used Google's Chrome Browser[2] as the target platform. Internally Chrome uses Google's V8 [14] engine for executing JavaScript. For details about V8 we refer to [14] and for Crankshaft, V8's optimizing compiler to [13] .

There are no JavaScript implementations of linpack, SciMark2a, JBox2D and SPECjbb2005, because these are Java benchmarks and would have been too tedious to rewrite them in JavaScript by hand.

As we can see in Figure 6, JavaScript generated by AOT JS is slower than Java on the server compiler and slower than JavaScript on V8. The benchmarks binarytrees, nbody and fasta are $1.3x$ to $1.4x$ slower under AOT JS than on V8 which is reasonable if one considers the overhead introduced by the Java semantics. Also linpack performs quite well, although a native JavaScript baseline is missing.

Figure 6 also presents different configurations for optimizations such as inlining, control flow analysis and exception wrapping. The optimizations have different impacts on the benchmarks. Small benchmarks such as binarytrees profit from the inlining of constructors which enables escape analysis and increases the performance. Nbody and linpack are examples for benchmarks that do not rely on elaborate Java features. Applying certain optimizations on them, besides control flow analysis, does not result in significant speedups. Nbody has basically zero call- and type-overhead. Linpack, even without exception wrapping, which does not compile methods with exception handlers with V8's optimizing compiler, as they do not optimize exception handlers, is simple enough for V8's baseline compiler to be optimized properly. Binarytrees has a high call overhead with call stack depths of 15 and more. Compiling it with the optimizing compiler makes a huge difference compared to the full compiler. The "number crunching" benchmarks generally profit less from the optimizations as they rely less on an elaborate type system and exception handling.

The interesting benchmarks of Figure 6 are deltablue, JBox2D, SciMark2a and SPECjbb2005. The high slowdown of deltablue is mainly due to the control flow analysis. Deltablue has a set of hot loops with control flow that cannot be fully reduced by the control flow reconstruction optimization. Slowdown due to unstructured control flow ranges from $1.36x$ to $1.54x$ to the AOT JS baseline which has all optimizations enabled.

The SPECjbb2005 benchmark has several performance problems. The main reasons for its slow performance are its large number of memory allocations and the emulation of `long` arithmetic. About 15% of the benchmark's time is spent in long arithmetic, mostly in addition, multiplication and division. For every computation using `long` values, immutable long objects are created. About $7 - 10\%$ of the benchmarks's time is spent in the GC. Considering the allocations of `long` objects, it seems that V8's escape analysis is not able to remove all of them, although inlining of the arithmetic functions should never allocate more than one object, namely the result of the computation. In addition to that, SPECjbb2005 also suffers from performance problems introduced by unstructured control flow.

The SciMark2a and JBox2D benchmarks illustrate further examples of performance penalties for unstructured control flow. The benchmarks spend a lot of time in hot loops which were not fully analyzed by the control flow tagging optimization. A large set of those loops is not identified as structured control flow and thus compiled with the generic block interpreter. This certainly leaves space for optimizations.

JBox2D is the benchmark that profits most from partial escape analysis. Inlining of small functions and constructors removes many object allocations which results in a performance increase of $12x$.

Performance slowdown to code compiled with the server compiler ranges from $2.44x$ to $20x$, which is a significant slowdown. However, for many cases this speed is certainly enough, especially for code that runs in websites. The slowdown compared to handwritten JavaScript ranges from $1.3x$ to $9x$, but for small benchmarks is around $2 - 3x$. This illustrates the usability of our approach. $2x$ is still a significant slowdown but for most client applications this speed is sufficient.

6. Related Work

Cross-Compilation to JavaScript is a well-known target in the compiler research community. For a list of languages that have cross compilers for JavaScript see [18]. Nearly every prominent language has a representative in there.

Below we give short overview over the most interesting approaches in relation to Graal AOT JS.

Emscripten [50] is Mozilla's back end for the Low-Level-Virtual-Machine (LLVM) [25] that generates JavaScript code from LLVM assembly. The uniqueness of this approach is the usage of ASM.js [28]. ASM.js is a low-level subset of JavaScript that can be easily optimized as it allows the executing JavaScript VM to specialize on 32 bit integers for expressions. Emscripten is mainly targeting the LLVM C/C++ front end *Clang* [24]. Their approach tries to produce fast and small JavaScript code. For the compilation of

[1] JDK1.8.0_11 (64 bit)

[2] Chrome 42.0.2311.135 (64-bit)

Table 3. Compile to JS Comparison: *N/A* indicates that a certain approach is not design to be evaluated to a given property, or literature does not provide detailed information.

	JDK Support	Threading	Reflection	GC	Control Flow Reconstruction	AOT Optimization	Compilation Source	Dependency Analysis
Graal AOT JS	Compilation	No	limited	JS VM	Yes	Yes	Java Bytecode	Yes
Emscripten XMLVM	Compilation	No	Full	manually	Yes	Yes	Java Bytecode	No
GWT	Custom	No	limited	JS VM	N/A	N/A	Java SourceCode	N/A
teavm	Custom	Yes	limited	JS VM	Yes	N/A	Java Bytecode	Yes
Bck2Brwsr	Custom	No	Full	JS VM	No	No	Java Bytecode	N/A
Doppio JVM	Interpretation	Yes	Full	JS VM	N/A	No	Java Bytecode	No
Shumway	N/A	N/A	N/A	JS VM	Yes	Yes	ActionScript	N/A
WhaleSong	N/A	Yes	N/A	JS VM	Limited	Limited	Racket Source	N/A
JS_of_ocaml	N/A	No	N/A	JS VM	Yes	Yes	OCaml ByteCode	N/A

Java code to JavaScript, emscripten offers a tool-chain described in [34]. The approach pipelines a Java to C and a C to JavaScript cross-compiler. The Java to C transformation uses the XMLVM [34] and the C to JavaScript transformation emscripten.

Google Web Toolkit (GWT) is Google's client-sided JavaScript web development toolkit, which includes a Java to JavaScript compiler. The main focus of GWT lies on performance and readability of the generated JavaScript code. GWT features its own implementations of `java.util.*` classes, so its JDK usage is limited. The *Closure compiler* [15] that can be optionally used features aggressive optimizations.

Bck2Brwsr [17] is a Java VM that compiles bytecodes to JavaScript. The approach is based on parsing Java classes, extracting meta information and generating JavaScript code. Bck2Brwsr supports just-in-time compilation in the browser via loading required classes from a web server, which enables support of the Java reflection API.

teavm is a Java bytecodes to JavaScript compiler [40]. The compilation pipeline includes bytecode parsing, IR generation in static-single-assignment form, followed by a dependency analysis over the imports to reduce the code size. It applies AOT optimizations, control flow reconstruction and a control flow optimization step. The approach's *dependency checker* is used to analyze imports of Java classes and to remove unused imports. Teavm's documentation lacks precise information about AOT optimizations. It just says that *"all major optimizations should be applied"*. Teavm does not offer native support for the JDK, but features a custom compatibility API for it.

Doppio [42] is a JavaScript-based runtime system for general purpose language support. The system features a large set of low-level API and runtime functionality emulations. The original paper presents a case study of a Java bytecode interpreter running on-top of Doppio. However the approach aims for general-purpose language support rather than generation of efficient JavaScript code.

Mozilla Shumway [30] is Mozilla's web-native implementation of the small web format (SWF) [1]. Shumway uses HTML5 and JavaScript for interpretation of SWF applications. It features an ActionScript [2] interpreter as well as a just-in-time compiler generating JavaScript code.

Whalesong [49] is a *Racket* [35] to JavaScript compiler. The approach features different compilation strategies, the fastest being a Racket to bytecode to JavaScript compiler. Slowdown to native Racket ranges from $25x$ to more than $100x$. Problems arise from the mapping of Racket's advanced language constructs to JavaScript like, e.g., preemption and advanced control. The paper presents limited control flow reconstruction optimizations during compilation but it lacks detailed description.

JS_of_ocaml [43] is an *OCaml* [31] bytecode to JavaScript compiler. The presented compiler builds a SSA based IR from OCaml bytecode, applies several optimizations and control flow reconstruction. Peak performance of the generated JavaScript code is very high. Compared to native JavaScript the compiled OCaml code is very fast and nearly as good as handwritten JavaScript.

Comparison to Graal AOT JS In Table 3 the presented approaches are evaluated according to their capabilities and features for transformations to JavaScript.

Emscripten is targeting a different source language than Graal AOT JS. It uses the Clang front end for LLVM and thus compiles unmanaged C & C++ code, which is not comparable to a managed language like Java. Their XMLVM extension fully supports the JDK and also features a way of reducing the code size via so-called *red lists*. However, this approach still suffers from larger code sizes as it lacks a static analysis. The approach does not use native JavaScript objects and compiles its own GC. Execution of allocation-intensive Java benchmarks such as SPECjbb2005 may lead to performance issues if the garbage collection is performed in JavaScript rather than, by a highly tuned JavaScript VM.

Google's web toolkit operates on the Java source-code level and features a compiler-intrinsic compatibility set of the JDK, which limits the usage prospects of the system. The approach results in better performance because it does not need control flow reconstruction on the source-code level. Future work on Graal AOT JS will tackle this issue and improve the control flow reconstruction optimization.

Teavm and Bck2Brwsr feature custom solutions for the JDK and do not natively compile it to JavaScript. While teavm's approach also compiles an application to one piece of executable JavaScript code, Bck2Brwsr supports just-in-time compilation in the browser. However the loading of classes and runtime-compilation require additional server requests.

The Doppio JVM covers the execution of the entire JDK. It supports threading, the entire file system API and full reflection. However, its Java bytecode interpreter suffers from bad performance. Their presented benchmarks are compared to the HotSpot interpreter, which is significantly slower than the server compiler.

7. Conclusions

We presented a novel approach for AOT compilation of Java to JavaScript. The approach is able to produce structured JavaScript code from possibly unstructured Java bytecode. We presented an algorithm for structured control flow reconstruction yielding a reasonable amount of restructured control flow. We also presented a mapping of all Java concepts to corresponding JavaScript concepts. The evaluation of our approach on a set of benchmarks shows that AOT compilation from a just-in-time compiler's IR to JavaScript is feasible, but there is some slowdown to hand-written code depending on the benchmark. Benchmarks which rely more on numeric computations than on a complex class hierarchy show that the slowdown to native JavaScript mainly consists of the additional operations carried out in Java such as type checks, numeric range checks and the emulation of long values.

Future work will include improvements of the control flow restructuring optimization especially for shared exception handlers and compound conditions. In addition to improving the code generator, the support of the Truffle framework [47] running on-top of Graal is a desired research goal. First steps in this direction have already been made. Feasibility prototypes of Graal's Simple Language, which is a simple untyped language for illustration of Truffle language development, Truffle Interpreter and the Truffle JavaScript interpreter are running on Graal AOT JS, but without runtime compilation. Beside the general support for Truffle interpreters, Graal AOT JS's capabilities for the support of runtime compilation for Truffle as presented in [48] will be explored. Such a system will require Graal itself to be compiled to JavaScript which is a challenge in terms of code size rather than in terms of performance.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute for System Software at the Johannes Kepler University Linz for their support and contributions. We especially thank Thomas Würthinger, Andreas Wöss, Matthias Grimmer, Roland Schatz, and Christian Wirth for their constant feedback. The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

References

[1] Adobe. Small Web Format, 2015. URL http://www.adobe.com/devnet/swf.html.

[2] Adobe. ActionScript, 2015. URL http://www.adobe.com/devnet/actionscript/documentation.html.

[3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.

[4] Box2d. Box2D Physics Engine, 2015. URL http://box2d.org/.

[5] C. Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, 1994.

[6] CLBG. Computer Language Benchmark Game, 2015. URL http://benchmarksgame.alioth.debian.org/.

[7] C. Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257. ACM Press, 1995. DOI: 10.1145/207110.207154.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. DOI: 10.1145/115372.115320.

[9] DB. DeltaBlue Benchmark, 2015. URL https://github.com/xxgreg/deltablue.

[10] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. ACM Press, 2013.

[11] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, pages 1–10. ACM Press, 2013. DOI: 10.1145/2542142.2542143.

[12] A. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *In Proceedings of the International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994. DOI: 10.1109/ICCL.1994.288377.

[13] Google. Crankshaft: V8's optimizing compiler, 2012. URL http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html.

[14] Google. V8 JavaScript Engine, 2012. URL http://code.google.com/p/v8/.

[15] Google. Closure Compiler, 2015. URL https://developers.google.com/closure/compiler/.

[16] Google. Web Toolkit [GWT], 2015. URL http://www.gwtproject.org/.

[17] Jaroslav Tulach. DukeScript: Bck2Brwsr VM, 2015. URL http://wiki.apidesign.org/wiki/Bck2Brwsr.

[18] jashkenas. List of Languages that compile to JavaScript, 2015. URL https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS.

[19] JBox2D. Piston Bechnmark, 2015. URL http://www.jbox2d.org/.

[20] T. László and Á. Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[21] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*, 2015. URL https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[22] Linpack. Linpack Benchmark, 2015. URL http://www.netlib.org/benchmark/linpackjava/.

[23] Lisperator. UglifyJs2, 2015. URL http://lisperator.net/uglifyjs/.

[24] LLVM. Clang, 2015. URL http://clang.llvm.org/.

[25] LLVM. Low-Level Virtual Machine, 2015. URL http://llvm.org/.

[26] J. Miecznikowski and L. Hendren. Decompiling java using staged encapsulation. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 368–374. IEEE, 2001. DOI: 10.1109/WCRE.2001.957845.

[27] J. Miecznikowski and L. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *Compiler Construction*, volume 2304, pages 111–127. Springer Berlin Heidelberg, 2002. DOI: 10.1007/3-540-45937-5˙10.

[28] Mozilla. ASM.js, 2015. URL http://asmjs.org/.

[29] Mozilla. Developer Network (MDN): JavaScript, 2015. URL https://developer.mozilla.org/de/docs/Web/JavaScript.

[30] Mozilla. Shumway, 2015. URL http://mozilla.github.io/shumway/.

[31] OCaml. The OCaml Language, 2015. URL https://ocaml.org/.

[32] OpenJDK. Graal, 2015. URL http://openjdk.java.net/projects/graal/.

[33] M. Paleczny, C. Vick, and C. Click. The Java HotSpot^{TM} Server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*, pages 1–12, 2001.

[34] A. Puder, V. Woeltjen, and A. Zakai. Cross-compiling Java to JavaScript via tool-chaining. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 25–34. ACM Press, 2013. DOI: 10.1145/2500828.2500831.

[35] Racket. The Racket Language, 2015. URL http://racket-lang.org/.

[36] SCI2. SciMark 2 Benchmark, 2015. URL http://math.nist.gov/scimark2/.

[37] Spec. SPECjbb2005 Java Server Benchmark, 2015. URL https://www.spec.org/jbb2005/.

[38] L. Stadler, G. Duboscq, H. Mössenböck, T. Würthinger, and D. Simon. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings of the 4th Workshop on Scala*, page 9. ACM, 2013. DOI: 10.1145/2489837.2489846.

[39] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*, page 165. ACM, 2014. DOI: 10.1145/2544137.2544157.

[40] TEA. TEA VM, 2015. URL http://teavm.org/.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[42] J. Vilk and E. D. Berger. Doppio: breaking the browser language barrier. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 508–518. ACM, 2014. DOI: 10.1145/2594291.2594293.

[43] J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44 (8):951–972, 2014.

[44] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, 2000.

[45] H. S. Warren. *Hacker's delight*. Addison-Wesley, Upper Saddle River, NJ, 2nd ed. edition, 2013. ISBN 0321842685.

[46] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5-6), 2009. DOI: 10.1016/j.scico.2009.01.002.

[47] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the Dynamic Languages Symposium*, page 73. ACM Press, 2012. DOI: 10.1145/2384577.2384587.

[48] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 187–204, 2013. DOI: 10.1145/2509578.2509581.

[49] D. Yoo and S. Krishnamurthi. Whalesong: Running racket in the browser. In *Proceedings of the Dynamic Languages Symposium*, pages 97–108. ACM, 2013.

[50] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 301–312. ACM Press, 2011. DOI: 10.1145/2048147.2048224.