CompGen: Generation of Fast Compilers in a Multi-Language VM[*]

Florian Latifi
Johannes Kepler University
Austria
florian.latifi@jku.at

David Leopoldseder
Oracle Labs
Austria
david.leopoldseder@oracle.com

Christian Wimmer
Oracle Labs
USA
christian.wimmer@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Austria
hanspeter.mössenböck@jku.at

Abstract

The first Futamura projection enables compilation and high performance code generation of user programs by partial evaluation of language interpreters. Previous work has shown that it is sufficient to leverage profiling information and use partial evaluation directives in interpreters as hints to drive partial evaluation towards compiled code efficiency. However, this comes with the downside of additional application warm-up time: Partial evaluation of language interpreters has to specialize interpreter code on the fly to the dynamic types used at run time to create efficient target code. As a result, the tie spend on partial evaluation itself is a significant contributor to the overall compile time of a method.

The second Futamura projection solves this problem by self-applying partial evaluation on the partial evaluation algorithm, effectively generating language-specific compilers from interpreters. This typically reduces compilation time compared to the first projection. Previous work employed the second projection to some extent, however to this day, no generic second Futamura projection approach is used in a state-of-the-art language runtime. Ultimately, the problems of code-size explosion for compiler generation and warm-up time increases are unsolved problems subject to research to this day.

To solve the problems of code-size explosion and self-application warm-up this paper proposes *CompGen*, an approach based on code generation of subsets of language interpreters which is loosely based upon the idea of the second Futamura projection. We implemented a prototype of CompGen for *GraalVM* and show that our usage of a novel code-generation algorithm, incorporating interpreter directives allows to generate efficient compilers that emit fast target programs which easily outperform the first Fumatura projection in compilation time. We evaluated our approach with *GraalJS*, an ECMAScript-compliant interpreter, and standard JavaScript benchmarks, showing that our approach achieves $2 - 3X$ speedups of partial evaluation.

CCS Concepts • **Software and its engineering** → **Interpreters**; **Dynamic compilers**; **Translator writing systems and compiler generators**; **Source code generation**.

Keywords partial evaluation, Futamura projection, code generation, interpretation, dynamic compilation

ACM Reference Format:
Florian Latifi, David Leopoldseder, Christian Wimmer, and Hanspeter Mössenböck. 2019. CompGen: Generation of Fast Compilers in a Multi-Language VM. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages.
https://doi.org/10.1145/3359061.3361077

1 Introduction

Many programming language implementations feature a hand written and heavily optimized virtual machine (VM) with an interpreter and one or multiple optimizing just-in-time (JIT) compilers. This violates the principle to not repeat yourself. The Futamura projections [7] generalize the compilation process via partial evaluation (PE) to generate machine from an interpreter, removing the need for language-specific and hand-written compilers. The process of deriving machine code from an interpreter and data is referred to as the *First Futamura Projection (F1)* [8]. With this, a VM can implement a single unified compilation process, and let other language interpreters benefit from optimization and compilation, as well as system components and tooling a VM

[*]This research project is partially funded by Oracle Labs.

SPLASH Companion '19, October 20–25, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6992-3/19/10…$15.00
https://doi.org/10.1145/3359061.3361077

usually provides. GraalVM [21, 24, 27] employs partial evaluation of polyglot AST [29] interpreters implemented using GraalVM's Truffle API to derive high performance machine code [28]. However, experiments with real-world programs in GraalVM have shown that PE in Truffle can make up half of the overall compile time. Truffle PE is compile time intensive because the implementation is guest language agnostic, i.e., the construction of the compiler IR during PE has to function on arbitrary Java AST interpreter code.

To solve this problem we propose a compiler generation approach called *CompGen*. It is loosely based on the idea of self-applying partial evaluation on the partial evaluator, i.e., the second Futamura projection (F2). Our approach generates compilers for individual AST nodes in the Truffle framework. These generated compilers produce the associated compiler IR during PE time specifically for an AST node directly. The output of our compiler is Java code that compiles a particular AST node.

In summary this paper contributes the following:

- A practical compiler generator (called *CompGen*) for *GraalVM* that is loosely based on the idea of F2 [8]. CompGen generates language-specific compilers automatically for a selected subset of a language interpreter.
- A novel code-generation algorithm that considers PE directives of a practical F1 to generate compilers that produce efficient and fast target programs.
- An integration between a practical F1 and CompGen allowing a hybrid PE mode to trade-off between code-size increase and PE time.
- An analysis of compile time, run time and code size, showing that our approach achieves significant speedups in PE, generating compilers that produce target programs with run-time performance similar to a practical F1, competing with highly optimized and hand-tuned language implementations, and does not introduce code-size explosion or additional warm-up in PE.

2 Truffle: Partial Evaluation in a High Performance VM

In this section we present the state-of-the-art F1 application used by the Truffle framework in GraalVM [21, 24, 27], and the major challenges that arise from it that can be optimized using a different PE algorithm.

2.1 Fist Futamura Projection (F1)

The first Futamura projection (F1) proposes compilation of a user program by PE of an interpreter with the assumption that the user program is a constant input to the interpreter. For this, a partial evaluator has to reason about the instructions of the interpreter and decide, with respect to the given user program, if an instruction should be evaluated now, i.e., at compile time time, or if the instruction should be emitted in the target program for evaluation at run time. This is compile-time intensive. In Truffle F1 PE can consume up to 50% of the overall compilation time (partial evaluation time plus time spent in the optimizing compiler to produce optimized machine code). Therefore, PE time is crucial for application warmup and thus is worth optimizing in order to reduce the overall compilation time.

Würthinger et al. [28] showed that a pure generic F1 creates many technical challenges because the PE algorithm has to decide when to stop inlining and evaluating and when runtime-performance can justify larger program size or longer compilation times. For F1 to be practical, especially in the context of dynamic languages, it is sufficient to introduce PE primitives, such as annotations and intrinsics in the interpreter implementation [28]. This allows PE to leverage profiling information gathered during interpretation, and make optimistic assumptions when reasoning about the interpreter instructions, in order to produce efficient target code that speculates on the fast path and deoptimizes [10] when necessary. This comes with the downside of additional warm-up due to profiling, however, greatly increases runtime performance after compilation.

Listing 1 shows a stylized version of Truffle's PE algorithm: It takes an AST node as the starting point. It first parses the node's `exec` method into a compiler IR graph. The `exec` method denotes the logic for the interpretation of this node. Then, `doPE` iterates the IR and performs partial evaluation accordingly. If it sees, e.g., the invocation of another `exec` method, and the receiver represents an AST child node, the receiver is assumed to be constant. This allows parsing the `exec` method of the given child and replacing the invocation, by inlining the child graph into the current graph. Method parsing and dispatching on given IR is what makes partial evaluation slow for large ASTs.

```
IRGraph doPE(ASTNode node) {
  IRGraph graph = parseBytecode(node);
  for (IRInvoke invoke : graph.invokes) {
    if (isExec(invoke.method) && isChild(invoke.receiver)) {
      ASTNode child = (ASTNode) invoke.receiver.asConstant();
      IRGraph childGraph = parseBytecode(child);
      graph.inline(childGraph, invoke);
    }
  }
  return graph;
}
```

Listing 1. Truffle's Partial Evaluation.

2.2 Second Futamura Projection (F2)

The second Futamura projection (F2) proposes a solution to the compile time problem of F1 as it allows to speed up the compilation by generating a language-specific compiler for an interpreter that compiles faster than a generic F1. Originally, this is described as a process of self-application. Instead of partially evaluating an interpreter with respect to a static user program, the partial evaluator is applied to itself, with the assumption that the provided interpreter is static input to the partial evaluator. With this, the overhead of PE in F1, i.e., reasoning about and processing of interpreter instructions when compiling a user program, can in theory be avoided. In this case, the partial evaluator specializes itself for a given interpreter. With this, it can directly emit target programs for given source programs from partially evaluated interpreter instructions. This should yield the target program much faster than a general partial evaluator. However, self-application approaches often do not support all language features, may produce code that only gives modest speedups, or impose other problems which are avoided by a hand-written code generator that creates specialized partial evaluators [2, 17]. For self-application to work in Truffle, partial evaluation itself would need to be re-implemented in form of a dedicated Truffle language with properly incorporated Truffle primitives. However, since these Truffle primitives are currently targeted for actual interpreters (i.e., executing a method) and not abstract interpreters (i.e., converting a method into compiler IR), it is unclear if this self-application can be made practical or even possible in Truffle.

2.3 CompGen: Compiler Generation for High Performance Partial Evaluation in Truffle

In order to speed up PE in Truffle we propose *CompGen*, an approach that uses a hand-written code generator instead of self-application. *CompGen* generates specialized versions of compilers for selected AST interpreter nodes ahead of time, removing the method parsing and instruction dispatching overhead. If partial evaluation is then requested for such an AST node, a partially evaluated IR graph is emitted directly.

We illustrate this for a simple AST interpreter which only adds `long` constants. Listing 2 shows the node classes that implement this logic. `AddNode` has two fields, `left` and `right`. Both are annotated with Truffle's `@Child` primitive, which the `isChild` method in `doPE` checks to enable parsing and inlining of these nodes. The `exec` method calls both children and returns the sum of the two `long` values. `ConstNode` has a `final long` field that is returned by the `exec` method.

```
class AddNode extends ASTNode {
  @Child ASTNode left;
  @Child ASTNode right;
  long exec() { return left.exec() + right.exec(); }
}
class ConstNode extends ASTNode {
  final long value;
  long exec() { return value; }
}
```

Listing 2. AST Interpreter Node Classes.

CompGen generates source code for the specialized versions of `doPE` ahead of time by parsing the `exec` methods of these node classes and traversing the IR graph in reverse post-order. IR nodes describing values already known, e.g., literals, are constant-folded through the graph. IR Nodes describing values annotated with Truffle primitives, e.g., loads of fields annotated with `@Child`, are assumed to be known during partial evaluation. In this case, source code is generated that computes these values without emitting any IR. Nodes describing other values are assumed to be unknown during partial evaluation. In this case, source code is generated that emits IR for the target program to compute the values at run time when the program is executed.

Listing 3 shows the generated partial evaluation methods for the AST node classes, denoted as `doPE_Gen`. The first method dispatches between the other two, based on the given node type. The other methods take concrete `AddNode` and `ConstNode` objects. Since the fields of `AddNode` are annotated with `@Child`, these field loads are assumed to be known during partial evaluation. This allows replacing the original `exec` invocations in the interpreter, by calling `doPE_Gen` for both fields and inlining the resulting graphs, to emit an `IRAdd` node with the return values as inputs. For the `ConstNode`, its value is stored in a `final` field, i.e., the value is also assumed to be known during partial evaluation. This allows emitting an `IRConst` node by reading the `value` field.

To clearly distinguish between nodes of the AST and nodes of the compiler IR, we use the suffix `Node` for everything related to AST nodes and the prefix `IR` for everything related to compiler IR. While in these simple examples it seems as if there is a one-to-one matching between AST nodes and compiler IR nodes, this does not hold for real-world languages like JavaScript where JavaScript addition has semantics far more complicated than just adding two numbers.

```
IRGraph doPE_Gen(ASTNode node) {
  if (node instanceof AddNode) {
    return doPE_Gen((AddNode) node);
  } else if (node instanceof ConstNode) {
    return doPE_Gen((ConstNode) node);
  } else { /* handle other ... */ }
}
IRGraph doPE_Gen(AddNode node) {
  IRGraph leftVal = doPE_Gen(node.left);
  IRGraph rightVal = doPE_Gen(node.right);
  return new IRAdd(leftVal, rightVal);
}
IRGraph doPE_Gen(ConstNode node) {
  return new IRConst(node.value);
}
```

Listing 3. Specialized Partial Evaluation.

Figure 1 shows a comparison of `doPE` and *CompGen* for an example AST which adds two constants. `doPE` parses the `AddNode` and sees both child invocations, so it parses and inlines the IR of both `exec` methods, respectively. Finally, the compiler optimizes the addition in the IR to a constant. In comparison, *CompGen* with its `doPE_Gen` methods directly emits partially evaluated IR, i.e., the addition of both constants, which the compiler then optimizes. Compared to `doPE`, *CompGen* requires two steps less to produce the same IR.

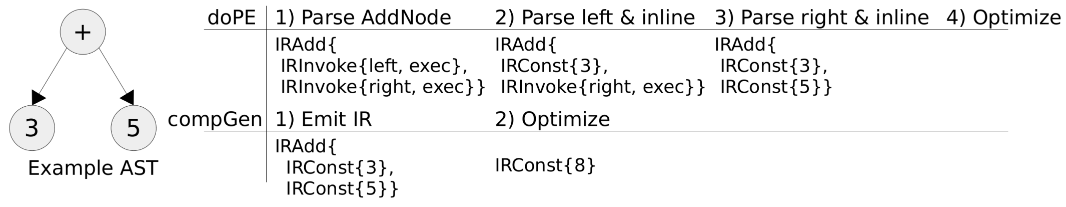


Figure 1. Comparison of doPE and *CompGen*.

3 Approach

In this section we present *CompGen*, an algorithm for fast compiler generation we implemented for the Truffle framework. CompGen (see Figure 2) integrates seamlessly into Truffle's partial evaluation and compilation system. It uses the same AST nodes F1 does, however, for each AST node it creates a specialized compiler plugin that automatically produces the Graal IR of the associated AST node.

In this section we present the problems that arise with compiler generation and how we can optimize them by leveraging profiling information to keep code size increase as low as possible while speeding up the partial evaluation process as much as possible.

Example In order to illustrate the challenges and complexity that arises from a compiler generation we use an example in Listing 4 throughout the rest of this section.

```
class ANode extends Node {
  static final int THRESHOLD = 10;
  @Children Node[] children;

  ANode(Node[] children) { this.children = children; }

  @Override @ExplodeLoop
  int execute(VirtualFrame f) {
    for (int i = 0; i < children.length; i++) {
      int r = children[i].execute(f);
      if (r > THRESHOLD) return r;
    }
    return -1;
  }
}
```

Listing 4. ANode class.

Listing 4 shows a simplified AST node implemented using the Truffle language implementation API. It uses Truffle's core primitives to generate a loop explosion of the compilation final children node array during partial evaluation.

F1 In order to generate efficient machine code for this AST node Truffle's F1-based partial evaluation algorithm has to parse the associated bytecode of an AST node. If during parsing an invocation is encountered partial evaluation forces the invoke to be inlined if possible. For this F1 has to decide whether the invoke can be inlined (i.e., check if the receiver is constant so it can be devirtualize the target method and continue PE in the new method. In the Truffe framework it is ensured node fields are compilation final (effectively final during compilation) so the compiler can devirtualize the receiver of a virtual dispatch on the AST level[1]. Listing 4 shows compilation final AST node children as they are annotated with *@Children*. During partial evaluation of *execute* in Listing 4 the F1 algorithm has to inline every execute method of the children[2]. For each *children*[*i*] receiver that is constant the F1 PE algorithm has to

- parse the execute method of the ANode class into a new graph
- replace the *this* parameter in this graph with a ConstantNode that represents the receiver object
- replace the normal path of the Invoke with the starting node of the new graph
- iterate the IR nodes and try to further simplify the graph

Given that a generic F1 algorithm never knows, which methods it encounters during PE, the simplification algorithm is a generic loop that dispatches between simplification logic for different kinds of IR nodes. Listing 5 shows a simplified, generic version of the F1 algorithm in Truffle. Given the number of different node classes in the Graal IR and the number of local transformations, F1 becomes a very compile time task spending a tramendeous amount of time allocating simplifiable nodes, doing typechecks and allocating temporal memory.

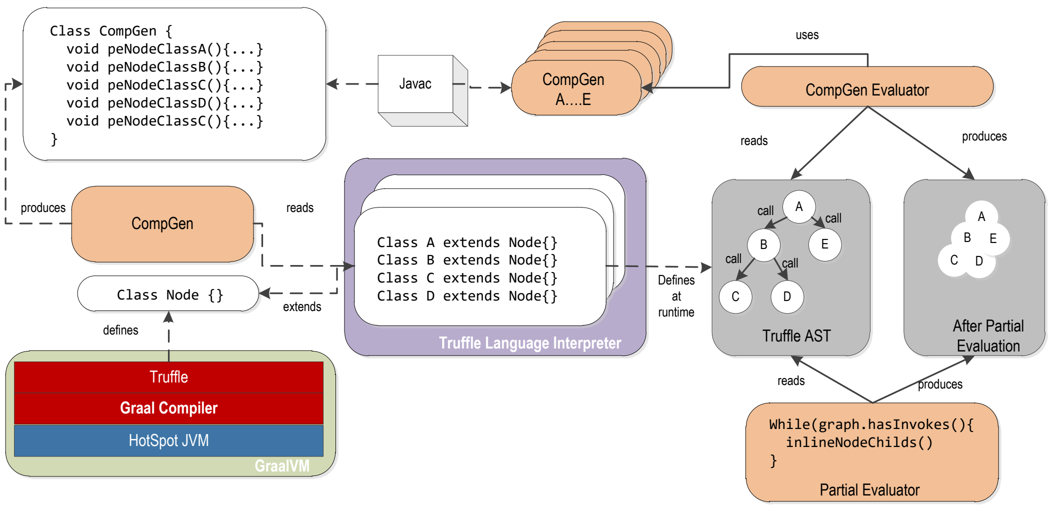
```
graph = parse(method);
worklist = graph.start;

do {
  node = worklist.pop();

  if (node instanceof IfNode) {
    if (node.condition instanceof ConstantNode) {
      graph.remove(node.condition.value ?
             node.falseSuccessor : node.trueSuccessor);
      worklist.push(node.condition.value ?
             node.trueSuccessor, node.falseSuccessor);
    } else {
      worklist.push(node.trueSuccessor);
      worklist.push(node.falseSuccessor);
    }
  } else if (node instanceof LoadFieldNode) {
    if (node.object instanceof ConstantNode &&
           node.field.allowsFolding()) {
      graph.replace(node, readConstantField(node.object,
             node.field));
    }
    worklist.push(node.successor);
```

[1]The truffle framework allows to use receivers from annotated field loads that can be constant folded, since we assume the AST is fixed; if they come from other sources and can't be constant folded, the invoke stays in the graph, which is later reported as "language performance bug").

[2]Note that in Listing 4 also the array length is final given that the array contents is considered final.



```
Class CompGen {
  void peNodeClassA(){...}
  void peNodeClassB(){...}
  void peNodeClassC(){...}
  void peNodeClassD(){...}
  void peNodeClassC(){...}
}
```

```
Class Node {}
```

```
Class A extends Node{}
Class B extends Node{}
Class C extends Node{}
Class D extends Node{}
```

```
While(graph.hasInvokes(){
  inlineNodeChilds()
}
```

Figure 2. CompGen System Structure.

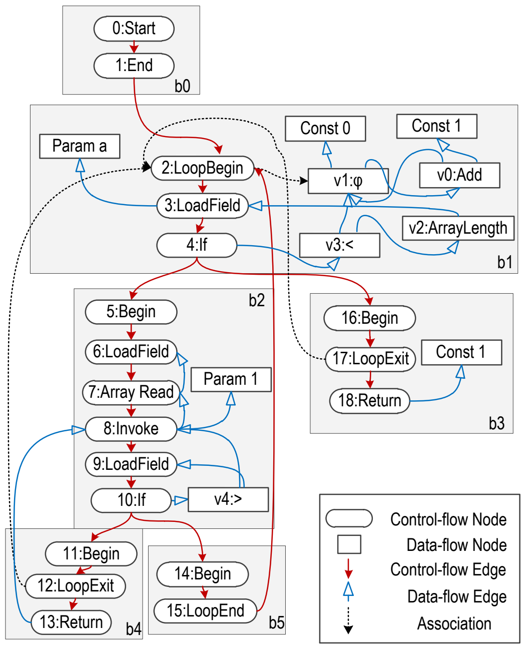


Figure 3. GraalIR for ANode.execute.

```
  } else if (...) {
      ...
  }
} while (worklist not empty);
```

Listing 5. F1 algorithm.

CompGen CompGen generates PE code for specific execute methods that are parsed and created ahead-of-time. We can leverage as much static information as possible to generate efficient PE code that produces as efficient code as F1 at run time. In order to generate specific compilers for Truffle IR nodes we have to take the original Graal IR of the execute method of a node. Figure 3 shows the Graal IR of Listing 4.

Note that the IR for the execute method after parsing and the IR generated during partial evaluation for such a method are conceptually different. In that sense, F1 can be seen as an "inlining-during-parsing" algorithm that eagerly explores and inlines the complete search space. To generate a compiler for Listing 4 that produces the same Graal IR as F1 after PE we have to identify, when values represented by IR nodes are available and what static information about these nodes can we leverage for generating efficient PE code. The generated compiler code can leverage constant information during compiler generation already. While a regular F1 algorithm runs compilation task at run time concurrently to the user program executing, we can use constant information inside final fields of AST nodes already during compiler generation and shift the optimization time from run time to ahead of time. Conceptually, there are 3 different compilation times in a compiler generator setting

- Compiler Generation Time (GenTime): The compiler produces compilers for user AST nodes.
- Partial Evaluation Time (PETime): The compiler produces partially evaluated code for user ASTs.
- Run Time: The generated code executes.

GenTime Values which are already available when we generate compiler code and which we can safely be re-used in generated compiler code, e.g., primitive and string literals. Here, we have to be careful, because we can't re-use object constants or static-final fields, because of multiple reasons: 1) user-level code would reference objects from the wrong object space 2) initializing classes ahead-of-time could change program semantics[3]. When we see Truffle annotations (ExplodeLoop, CompilationFinal, Child/Children, TruffleBoundary), we can generate more optimized PE code

[3]For example class constructors with calls to `static final DateTime time = DateTime.currentTime()`.

which skips meta-information fetching (e.g. field descriptors) and annotation checks that are performed by generic F1 PE. Example: for *@CompilationFinal* annotated fields we can generate optimized PE code which accesses necessary field descriptors from specific global constants and constant folds the field accesses by directly reading the specific fields. In contrast, generic F1 during PE would first fetch necessary field descriptors and check if the field is annotated with *@CompilationFinal*, before it tries to constant fold the field access. An example for a constant that can already be optimized during GenTime in Listing 4 is the field *THRESHOLD*. It is a constant primitive literal that can already be constant folded during compiler generation, i.e., we can replace the loadfield and constant pool lookup with a constant integer node 10.

PETime Values which are constant during PE time. In principle, everything could eventually become constant during PE. For example, PE of a child's execute method could give a constant node. An if-condition depending on this value could eventually be constant-folded. This is why we have to generate PE code that, before creating a node, checks, if the required inputs are constant, so we can create a more simplified version of the node (which is where, e.g., our node factories kick in). Examples for PETime constants are all the Truffle API core primitive usages in Listing 4: *@Children*, i.e., at PE time the length of the children array is constant as well as teh content of each array cell.

Run Time Values which are not available during PE can only become available at run time, i.e., we must emit the instructions in the target code. When a constant-check yields false, or we depend on a run-time parameter, we have to emit the instruction in the target program, so the value is computed by execution at run time. A standard example would be an if-condition that did not constant fold, which means the compiler has to emit the if-statement in the target code so the condition is evaluated at run time.

3.1 Compiler Generation

In order to generate a specific compiler for a single AST node we have to:

1. Parse the target execute method of the AST node (e.g. Listing 4 the *execute* method).
2. Propagate GenTime constants into the IR and simplify the graph based on these constants.
3. Schedule the IR so all nodes have a mapping to corresponding basic blocks.
4. Iterate the basic blocks in reverse post order.
5. Output PE code that carries out simplification logic specific to each visited IR node.
6. Use glue code to properly connect nodes and basic blocks simplified during PE.

Source code VS machine code generation Generating compilers AOT for AST nodes gives substantial compile time benefits over PE at run time. However, traditional compiler generated machine code that is loaded and executed at run time . In order for a better development experience and debuggability we propose to generate Java source code instead of machine code. This source code uses the Graal compiler IR's directives to create the associated AST node's execute method programatically once being executed. The source code itself can be compiled with javac to bytecode which can be compiled to native executables/shared libraries with GraalVMs native image if needed.

Code Generation Algorithm We present a pseudo-code version of our proposed compiler generation algorithm in Listing 6. The algorithm first parses the IR graph for the given method, populates GenTime constants and optimizes the graph accordingly. Then, it schedules the IR, i.e., each IR node is mapped to a basic block in the control flow. Finally, the PE code is generated by visiting the basic blocks in reverse post order and emitting PE code based on the encountered IR nodes. The reverse post-order iteration ensures that generated PE code does not reference values or IR nodes that have not been instantiated yet. The `handleBeginNode` and `handleEndNode` methods emit the PE glue code which properly connects the basic blocks and sets up the continuation points where nodes are appended to the control flow of the IR graph. For example, when a `MergeNode` is encountered, PE code is emitted which potentially merges predecessor paths if they have been visited before. When an `IfNode` is encountered, PE code is emitted which checks if the condition was constant-folded or not, so only one successor or both successors plus the `IfNode` itself must be prepared/instantiated. Both methods also emit the glue code that is necessary for Truffle's loop explosion mechanism. The `handleIntermediateNodes` method emits a snippet for each remaining IR node of a basic block, which tries to create a simplified version of the node during PE, e.g., through constant folding. When, e.g., an `AddNode` that takes two inputs is encountered, a snippet is emitted that tries to directly produce a `ConstantNode` of the sum during PE if both inputs are constant, and instantiates an `AddNode` if not. A more complex example would be an `InvokeNode`. Here, a snippet is emitted that checks if the receiver is constant and whether the invoke should be inlined. If both is true, the snippet dispatches to generated PE code that is capable of handling the encountered target method. If not, the invoke is simply instantiated in the IR graph.

```
PECode compGen(Method m) {
  // create and optimize original IR
  Graph g = parse(m).optimize().schedule();
  PECode c = new PECode();
  for (Block b : g.blocksInRPO) {
    handleBeginNode(g, c, b);
    handleIntermediateNodes(g, c, b);
    handleEndEnd(g, c, b);
  }
  return c;
}
void handleBeginNode(Graph g, PECode c, Block b) {
  IRNode n = b.beginNode;
  if (n instanceof MergeNode) {
    // potentially create BB merge
    c.maybeMergePredecessors(n);
    // potentially create phi nodes
    n.phis.forEach(p -> c.maybePhiValues(p));
  } else if (n instanceof LoopBeginNode) {
    // handle Truffle loop explosion
    if (g.hasLoopExplosion()) {
      c.enterNextLoopExplosionIteration(n);
      c.maybeMergePredecessors(n);
      n.phis.forEach(p -> c.maybePhiValues(p));
    } else {
      c.instantiate(n);
    }
  } else if (n instanceof LoopExitNode) {
    // handle Truffle loop explosion
    if (g.hasLoopExplosion()) {
      c.exitLoopExplosionIteration(n);
      c.maybeMergePredecessors(n);
      n.proxies.forEach(p -> c.maybePhiValues(p));
    } else {
      c.instantiate(n);
    }
  }
  // new nodes will be appended to this node
  c.continueWith(n);
}
void handleIntermediateNodes(Graph g, PECode c, Block b) {
  b.intermediateNodes.forEach(n -> c.trySimplify(n));
}
void handleEndEnd(Graph g, PECode c, Block b) {
  IRNode n = b.endNode;
  if (n instanceof IfNode) {
    c.simplifyIfConstant(n);
    c.instantiateElse(n);
  } else if (n instanceof LoopEndNode) {
    if (g.hasLoopExplosion()) {
      c.prepareNextLoopExplosionIteration(n.loopBegin);
    } else {
      c.instantiate(n);
    }
  } else if (n instanceof EndNode) {
    if (n.merge instanceof LoopBeginNode) {
      if (g.hasLoopExplosion()) {
        c.prepareInitialLoopExplosionIteration(n.merge);
      } else {
        c.instantiate(n);
      }
    }
  } else if (n instanceof ReturnNode) {
    c.propagateReturnValue(n);
  } else if (n instanceof UnwindNode) {
    c.propagateExceptionValue(n);
  }
}
```

Listing 6. F2 Code Generation Algorithm.

Listing 7 presents the generated F2 Java compiler code for ANode.execute without optimizations[4][5]. Here, the generated PE code simply replicates without any modification the IR graph that has been parsed at GenTime. The decode method prepares the starting anchor point and then calls all the gen methods in reverse post order to emit the basic blocks. A LoopScope class is generated which stores nodes and values created during a single loop iteration for a loop that is unrolled during PE - a LoopScope instance is passed to all gen methods so they can reference nodes if necessary. Even though we don't explode/unroll any loop in this example, one single LoopScope object is created. This is because we treat the body of the execute method as a giant loop with a single iteration, so we don't have to treat code without loops in a special way.

```
class LoopScope {
  boolean genB1 ... genB5;
  IRNode n0 ... n18;
  IRNode v0 ... v4;
}
void decode(GraphBuilder b, MethodScope m) {
  LoopScope l = new LoopScope();
  l.n0 = b.current();

  genB0(b, m, l);
  if (l.genB1) genB1(b, m, l);
  if (l.genB2) genB2(b, m, l);
  if (l.genB3) genB3(b, m, l);
  if (l.genB4) genB4(b, m, l);
  if (l.genB5) genB5(b, m, l);
}
void genB0(GraphBuilder b, MethodScope m, LoopScope l) {
  b.continueWith(l.n0);
  l.n1 = b.append(new EndNode());
  l.genB1 = true;
}
void genB1(GraphBuilder b, MethodScope m, LoopScope l) {
  l.n2 = b.add(new LoopBeginNode());
  l.n2.addForwardEnd(l.n1);
  l.v1 = b.add(new PhiNode(l.n2));
  l.v1.addInput(ConstantNode.of(0));
  b.continueWith(l.n2);
  l.v0 = AddNode.tryFold(b, l.v1, ConstantNode.of(1));
  l.n3 = LoadFieldNode.tryFold(b, "ANode#children",
           m.args[0]);
  l.v2 = ArrayLengthNode.tryFold(b, l.n3);
  l.v3 = LessThanNode.tryFold(b, l.v1, l.v2);

  if (l.v3.isTautology()) {
    l.n5 = b.current();
    l.genB2 = true;
  } else if (l.v3.isContradiction()) {
    l.n16 = b.current();
    l.genB3 = true;
  } else {
    l.n5 = b.add(new BeginNode());
    l.genB2 = true;
    l.n16 = b.add(new BeginNode());
    l.genB3 = true;
    l.n4 = b.append(new IfNode(l.v3, l.n5, l.n16));
  }
}
void genB2(GraphBuilder b, MethodScope m, LoopScope l) {
  b.continueWith(l.n5);
  l.n6 = LoadFieldNode.tryFold(b, "ANode#children",
           m.args[0]);
  l.n7 = ArrayReadNode.tryFold(b, l.n6, l.v1);
  l.n8 = trySimplifyInvoke(b, "execute", l.n7, m.args[1]);

  if (l.n8.isConstant()) {
    if (l.n8.value() > 10) {
      l.n11 = b.current();
      l.genB4 = true;
    } else {
      l.n14 = b.current();
      l.genB5 = true;
    }
  } else {
    l.v4 = b.add(new GreaterThanNode(l.n8,
             ConstantNode.of(10)));
    l.n11 = b.add(new BeginNode());
    l.genB4 = true;
    l.n14 = b.add(new BeginNode());
    l.genB5 = true;
```

[4]For simplicity we assume all values will only be available at run time, so no loop explosion, no constant checks, no primitive literals.

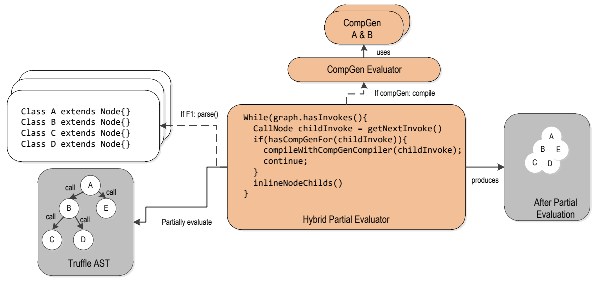
[5]We only show the code for the loop body blocks, without exception handling.

```
      l.n10 = b.append(new IfNode(l.v4, l.n11, l.n14));
    }
}
void genB3(GraphBuilder b, MethodScope m, LoopScope l) {
  b.continueWith(l.n16);
  l.n17 = b.append(new LoopExit(l.n2));
  m.returnPaths.add(l.n17);
  m.returnValues.add(ConstantNode.of(1));
}
void genB4(GraphBuilder b, MethodScope m, LoopScope l) {
  b.continueWith(l.n11);
  l.n12 = b.append(new LoopExit(l.n2));
  m.returnPaths.add(l.n12);
  m.returnValues.add(l.n8);
}
void genB5(GraphBuilder b, MethodScope m, LoopScope l) {
  b.continueWith(l.n14);
  l.n15 = b.append(new LoopEnd(l.n2));
}
```

Listing 7. Generated F2 Code - without optimizations.

3.2 Hybrid Compiler Generation

CompGen is capable of generating compilers for arbitrary Java source code. However, to improve partial evaluation time of Truffle interpreters only those AST node methods reachable during interpretation and thus subject to partial evaluation account for PE time later during compilation. Language implementations based on the GraalVM not only consist of AST interpreter method but also runtime library support, memory subsystem, parser etc. Less frequent language features make up a significantly large portion code. However, the number of methods subject to partial evaluation at run time is commonly less than the entire language support. Plainly using CompGen to generate compilers for all methods ever subject to partial evaluation comes at a high code size cost. GraalJS [23] for example roughly has 10 000 run-time compilable AST methods, i.e., the partial evaluator may only see those during any execution of a JavaScript program. Generating source code, which ultimately is compiled to Java bytecode, for 10k methods still incurs a significant code size overhead. For GraalJS, we measure about 40MB of additional bytecodes in the VM if we generate compilers for every method ever subject to PE. Thus, we propose a so called *hybrid* execution mode for the existing F1-based PE algorithm in Graal that only creates CompGen compilers for important AST node functions. A hybrid F1 PE algorithm proceeds as follows: if a compiler was generated by CompGen for a particular method, then the algorithm can use a generated compiler for this particular node, resulting in significant PE time reductions. If, however, no compiler was generated, the algorithm falls back to the F1-based PE algorithm, performing a regular "node interpretation" loop for the AST method. On the other side, every AST root node starts PE in the F1 algorithm, if a AST method is reached for which a compiler was generated the F1 algorithm calles into the CompGen generated code resulting in less compilation time. Figure 4 outlines the major execution flow of the hybrid approach: initially an AST's root node's partial evaluation starts in the regular (hybrid) F1-based PE algorithm. PE happens iteratively until an AST node is reached (A & B in this example) for which CompGen generated compilers have been registered in a map data structure in the PE algorithm. Execution will then resume in the generated compiler for A & B nodes respectively, after which the compiled compiler calls again back into the hybrid F1 PE to evaluate nodes C,D and E. We experimented with various different policies for a proper selection of suitable CompGen methods, we present the best performing ones in Section 4. To the best of our knowledge, combining F1 PE with pre-compiled compilers is a novel contribution in the domain of PE and shows that compiler generation can be used to effectively improve partial evaluation time a production system.



```
Class A extends Node{}
Class B extends Node{}
Class C extends Node{}
Class D extends Node{}
```

```
While(graph.hasInvokes(){
  CallNode childInvoke = getNextInvoke()
  if(hasCompGenFor(childInvoke)){
    compileWithCompGenCompiler(childInvoke);
    continue;
  }
  inlineNodeChilds()
}
```

Figure 4. Hybrid Partial Evaluation Approach.

3.3 PE Optimized CompGen Compilers

Listing 8 shows the generated F2 code for the hybrid PE model of ANode.execute with optimizations[6]. We generate the `LoopScope` class for handling the single loop iteration of the execute method as before, but this time, we explode the loop in the execute method. So we additionally store fields in `LoopScope` for handling the start and end of the loop explosion, like the initial `LoopScope2` object, lists of exit paths that must be merged and which are populated when a single loop explosion iteration hits an exit, and corresponding values for ProxyNodes that must be phied. A second `LoopScope2` class handles values produced within a single loop explosion iteration.

In the decode method, we start with a `LoopScope` object and call the gen method of the first basic block B0. Because the next block starts with a LoopBegin node, the gen method of B0 creates the first instance of `LoopScope2`, sets its anchorB1 to the current anchorB0, and initializes the IR field for variable i with ConstantNode(0). Then, a while loop processes a worklist of `LoopScope2` instances, and calls the gen methods of the basic blocks scheduled inside the loop in reverse post order. These gen methods take both `LoopScope` and `LoopScope2` instances, so the gen methods can reference both values scheduled inside and outside of the loop. The most important fact in this version is that we don't use

[6]We assume values may eventually constant fold during PE time, with loop explosion. Additionally, we only show the code for the interesting basic blocks, without exception handling

constructors to replicate the IR nodes as parsed ahead of time, but we use static factory methods that try to constant fold based on the given inputs. This can be seen in the genB2 method, where we try to constant fold the field and array access of the AST node's children array, so we can simplify the invoke of the children's execute methods. Another important usage of constant-folding is in genB5, where the next loop explosion iteration is prepared by creating the next LoopScope2 object. Here, the usage of AddNode.tryFold ensures that i+1 is constant folded and that the next value for i is propagated to the next loop scope. Note that in general Truffle loop explosion is not guaranteed to terminate, API mis-use can cause non-constant PE primitives. To circumvent this, we check before the next loop explosion iteration if the current iteration count has reached a defined maximum, and bailout from PE if necessary.

```
class LoopScope {
  /* generate flags, node & anchor fields */
  ..

  /* initial L2 scope */
  LoopScope2 initialL2;

  /* exit paths */
  List<Node> exitPathsN12;
  List<Node> exitPathsN17;

  /* proxy values */
  List<Node> var5Values;
}
class LoopScope2 {
  /* generate flags, node & anchor fields */
  ..

  /* iteration count */
  int iteration;

  /* next loop iteration scope */
  LoopScope2 nextIteration;
}
void decode(GraphBuilder b, MethodScope m) {
  LoopScope1 l = new LoopScope();
  l.anchorB0 = b.current();
  genB0(b, m, l);
  LoopScope2 l2 = l.initialL2;
  while (l2 != null) {
    if (l2.iteration > MAX_ITERATION) bailout();
    if (l2.genB1) genB1(b, m, l, l2);
    if (l2.genB2) genB2(b, m, l, l2);
    if (l2.genB5) genB5(b, m, l, l2);
    l2 = l2.nextIteration;
  }
  if (l.genB3) genB3(b, m, l);
  if (l.genB4) genB4(b, m, l);
}
void genB0(GraphBuilder b, MethodScope m, LoopScope l) {
  b.continueWith(l.n0);
  l.initialL2 = new LoopScope2();
  l.initialL2.n2 = b.current();
  l.initialL2.v1 = ConstantNode.of(0);
  l.initialL2.v0 = AddNode.tryFold(b, ConstantNode.of(1),
        l2.v1);
  l.initialL2.genB1 = true;
}
void genB1(GraphBuilder b, MethodScope m, LoopScope l,
      LoopScope2 l2) {
  // see unoptimized listing
}
Node hybridPE(String name, ValueNode[] args){
  if(hasCompGenCompiler(name,args[0]){
    return compileCompGen(name,args);
  }
  return F1PE.partialEval(name,args);
}
void genB2(GraphBuilder b, MethodScope m, LoopScope l,
      LoopScope2 l2) {
  b.continueWith(l2.anchorB2);
  l2.n6 =
        ConstantFoldUtil.readConstantField("ANode#children",
        m.args[0]);
  if (l2.n6 == null) l2.n6 = b.append(new
        LoadFieldNode("ANode#children", m.args[0]));
  l2.n7 = ArrayAccessNode.tryFold(b, l2.n6, l2.v1 /* value
        for variable i */);
  l2.n8 = hybridPE("execute", new ValueNode[] { l2.n7,
        m.args[1] /* a ValueNode for the VirtualFrame object
        */ });
  l2.v4 = GreaterThanNode.tryFold(b, l2.n8, l.c10 /* const
        int node */);
  l2.anchorB2 = b.current();

  if (l2.v4.isTautology()) {
    l.exitPathsN12.add(l2.anchorB2);
    l.v5Values.add(l2.n8);
    l.generateB4 = true;
  } else if (l2.v4.isContradiction()) {
    l2.anchorB5 = l2.anchorB2;
    l2.generateB5 = true;
  } else {
    l2.n10 = b.append(new IfNode(l2.v4, new BeginNode(), new
          BeginNode()));
    l.exitPathsN12.add(l2.n10.trueSuccessor);
    l.v5Values.add(l2.n8);
    l2.anchorB5 = l2.n10.falseSuccessor;
    l.generateB4 = true;
    l2.generateB5 = true;
  }
}
void genB5(GraphBuilder b, MethodScope m, LoopScope l,
      LoopScope2 l2) {
  b.continueWith(l2.anchorB5);
  l2.nextIteration = new LoopScope2();
  l2.nextIteration.iteration = l2.iteration + 1;
  l2.nextIteration.genB1 = true;
  l2.nextIteration.anchorB1 = l2.anchorB5; /* propagate
        anchor */
  l2.nextIteration.v1 = l2.v0; /* propagate i */
}
```

Listing 8. Generated CompGen Hybrid Compiler - with PE optimizations.

4 Evaluation

We evaluated our CompGen approach on top of the GraalVM by running and analyzing a industry-standard benchmark suite by using GraalJS, a high-performance JavaScript implementation based on Truffle.

4.1 Hypothesis

We want to show that the CompGen approach allows a runtime system performing partial evaluation to significantly reduce compilation time at a medium static code size increase. We tested this hypothesis by running a set of industry standard benchmarks in a JavaScript runtime and replaced thed partial evaluation algorithm with parts of our generated compilers.

Benchmarks We used the JavaScript Octane [6]. JavaScript Octane is a widely used JavaScript benchmark suite containing workloads ranging from 500 LOC to 300 000 LOC. Which is a good proxy for non-trivial JavaScript applications. The

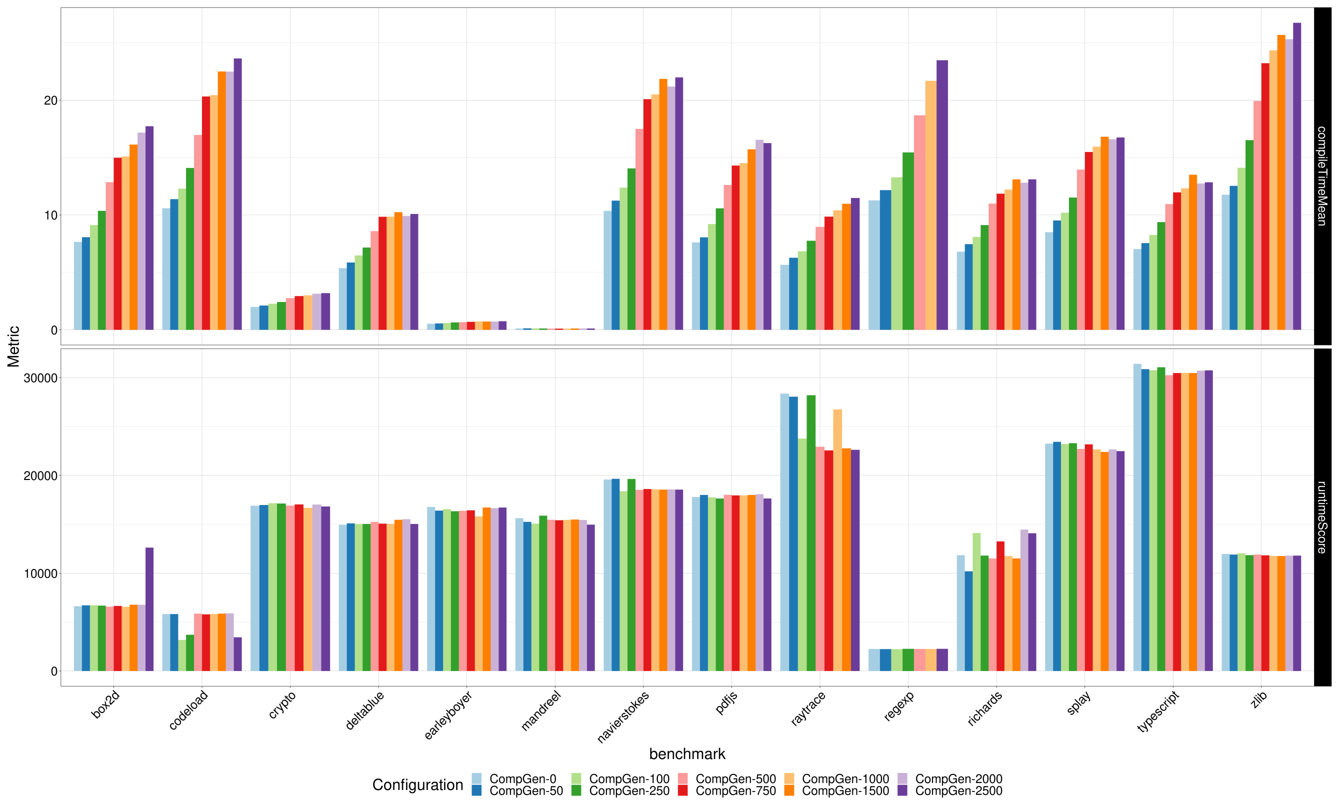


Figure 5. Octane Benchmarks Performance.

suite addresses JIT compilation, garbage collection, code loading and parsing of the executing JS VM. We measured Octane performance using Graal JS [28], the JavaScript implementation on top of Truffle, which is competitive in performance to Google's V8 [9]. Graal JS is on average 17% slower compared to the V8 JavaScript VM [34].

Environment All benchmarks were executed on a mobile dell workstation, equipped with an Intel I7-6820HQ cpu running at a clock speed of 2.7GHZ. The cpu features 4 hardware cores, hyperthreading was enabled and CPU throttling was disabled. The entire machine is equipped with 32 GB DDR-4 memory (running at 2133 MHz). GraalJS on GraalVM CE performs compilation in background threads concurrently to the interpreter executing the program.

Metrics For each benchmark we measured three distinct metrics:

- **Partial Evaluation Time**: We measured partial evaluation time of Graal when compiling the individual JS functions. PE time in Graal is the time needed to start partial evaluation on the function AST root node until the set of compilation final methods are inlined an simplified. To ensure stability of these benchmarks we ported them into a jmh [22] harness and extracted partial evaluation operations per second.
- **Runtime Performance**: In order to ensure that our generated compilers create the same IR as the regular (F1 based) partial evaluator, we measured runtime performance of regular benchmark executions with the different F2 configurations.
- **Bytecode Size Increase**: Generating compiler code increases the static bytecode footprint of the entire application. We measured bytecode size after source code generation and compilation with javac.

(Hybrid) CompGen Configurations For evaluating the performance of different *hybrid* (see Section 3) configurations we took the number of maximum runtime compiled methods from GraalJS [7] and derived buckets of the hottest n AST interpreter methods during partial evaluation when executing the Octane benchmark suite. For this we once ran the entire Octane suite, for each partial evaluation counted which AST node methods have been partially evaluated and sorted them by highest (hottest) to lowest. Then we took the n first methods of the resulting profiles resulting in several buckets where $CompGen - 0$ is pure F1 PE.

[7] https://github.com/oracle/graaljs/blob/master/graal-js/mx.graal-js/native-image.properties

4.2 Synthesis

The results of our experiments can be seen in Figure 5 and Table 1. We report static bytecode size of the CompGen generated compilers as well as partial evaluation time for each hybrid configuration and the runtime performance of the generated machine code. The octane benchmarks report a throughput value upon execution end. Generally we are interested in a hybrid configuration that improves PE performance significantly without sacrificing too much code size. Generating CompGen compilers for all 1000 js runtime compilable methods results in more than 40MB of bytecode size inrease. However, the "profiled" hot hybid configurations we tested with bucket sizes of 50-2500 already outperform the baseline F1 based PE significantly.

Not that the size ranges from 0 to $12MB$ while we have not yet optimized code size of the generated code, bytecode minification however may be still able to significantly reduce this.

The most notable hybrid configuration appears to be the $CompGen - 750$ which generates compilers for the hottest 750 GraalJS AST interpreter methods (out of 10 000). It outperforms F1 PE at every single benchmark in compilation time while never creating a worse peak performance result than CompGen-0 (F1). This configuration comes at a moderate bytecode size of 3.1 MB which is acceptable for up to 2x of average PE time improvements.

Note that in the peak performance numbers in Figure 5 different CompGen configurations sometimes cause peak improvements or regressions, this is due to the fact that compilation heuristics in the Graal compiler, which is run after PE, can make different decisions based on the time of compilation, i.e., earlier or later compilation can impact performance. While those are interesting findings, we consider them out of scope of our contribution of improving PE time.

Configuration	Bytecode Size (B)
CompGen-0	0
CompGen-50	109937
CompGen-100	282670
CompGen-250	802705
CompGen-500	1884316
CompGen-750	3116428
CompGen-1000	4143801
CompGen-1500	6601132
CompGen-2000	9196842
CompGen-2500	12061368

Table 1. CompGen Configuration Bytecode Size

5 Related Work

In this section we compare our *CompGen* compiler generation system to several other approaches in the domain of PE [8] and code generation.

Partial Evaluation In 1952 Kleene [14] proved via the *smn*-theorem the mathematical foundation and feasibility of partial evaluation. In 1964 the term "partial evaluation" was probably first used in the context of incomplete Lisp data structures [18]. Futamura [7] was 1971 one of the first to consider self-applying partial evaluation to generate compilers. Beckman et al. [1] proposed 1975 a partial evaluator for a subset of Lisp, which probably was the first description of a using partial evaluation on a partial evaluator itself (Second futamura projection) to perform compiler generation. Work by Jones et al. [13] followed that proposed the first self-applicable partial evaluator for a subset of Lisp. This approach was the first to generate small compilers and compiler generators. In the next two decades many partial evaluators have been developed for variaous languages like C, Lisp, Scheme, Prolog etc. [12]. However, all of the above approaches including a full self application according to Fumatura (F2) suffer from the same short comings:

- Self Application: If the approaches support F1 they typically lack support for F2
- Polyglot PE: They do not support multiple programming languages.
- Compilation Time: Compilation time is exhaustive.
- Code Size: The generated code for a self applicable approach explodes as inlining boundaries are missing

Partial Evaluation in Truffle The foundation for our work was done by Würthinger et al. [29] why they first proposed the *Truffle* framework. Later they presented, after many years of polyglot F1 based language implementation an approach for practical F1 Würthinger et al. [28].

Today our approach can be most notably compared to other virtual machines that employ any form of automated code generation most and for all our main competitor is GraalVMs default partial evaluator Würthinger et al. [28] that is based on the first futamura projection. In the other sections of the paper we outlined how our CompGen approach can, for the cost of AOT cmopilation time and code size, outperform regular partial evaluation by up to $3x$. Given that there is no F2 based or automated compiler generation approach for polyglot VMs we cannot compare our compiler generation approach directly to other approaches. However, we can compare to other approaches that generate compiled machine code from interpeters like: meta-tracing employed by the PyPy Bolz et al. [3, 4], Bolz and Rigo [5, 5]. Meta-tracing Marr and Ducasse [19] functions at runtime and observes interpreter execution traces to derived compiled code. Tracing compilation without runtime feedback is impossible. Our ComGen approach can create compilers for arbitrary Java code (Graal IR graphs) and thus does not need any dynamic data to speed up the compilation process. Tracing compilation would need to create traces during execution as templates for AOT compiler generation. At runtime a particular trace can be matched against a trace and if they are the same the compiler could be applied.

Source Code Generation Finally, we want to relate to the domain of source code generation. CompGen generates compiler source code for language AST nodes so there is a large domain of related work in the domain of code generation. Especially parser generators Johnson and Sethi [11] are most notably here: a prominent parser generator is ANTLR Parr and Fisher [25], Parr and Quong [26] that also generates Java source code. CocoR Mössenböck et al. [20] is a recursive descendant LL(1) parser generator for Java.

Transpilers and Source-to-Source Compilers Transpilation, i.e., the process of transforming one source language to another is also challenged with the task of source code generation. The most used transpiler for C++ to Javascript emscripten Zakai [30] also faces similar problems as CompGen: irreducible control flow as well as unstructured ones pose challenges on the code generation algorithm. Emscripten solves this by the relooper algorithm and recently added support for the stackifier. However, for CompGen the challenge for unstructured Graal IR is less critical given that we can create one method per basic block and thus can have early returns/exits.

Graal aotjs [15, 16] is a Java bytecode to JavaScript transpiler that uses a different control flow reconstruction algorithm than emscripten. Conceptually CompGen could re-use AOTJS source code generation logic but create AST nodes instead, however aotjs was optimized for efficient source code and thus contains hard-coded code generation patterns we could not re-use.

6 Conclusion

In this paper we presented *CompGen*, a practical second Futamura projection based compiler generator for fast partial evaluation of guest language interpreters. We proposed a novel algorithm to generate compilers for guest language ASTs allowing the usage of F1 compiler intrinsics to generate fast automatic compilers for guest languages. CompGen is a versatile approach allowing debuggability of the generated code as well as an integration with existing PE based compilation systems.

We implemented our CompGen approach in the GraalVM and showed that the approach can increase partial evaluation time by up to 3x at moderate static bytecode size increases of up to 3MB.

References

[1] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. 1976. A partial evaluator, and its use as a programming tool. *Artificial Intelligence* 7, 4 (1976), 319–357.

[2] Lars Birkedal and Morten Welinder. 1994. Hand-writing program generator generators. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 198–214.

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. 18–25.

[4] C. F. Bolz, M. Leuschel, and A. Rigo. 2010. Towards Just-In-Time Partial Evaluation of Prolog. In *LOPSTR*.

[5] Carl Friedrich Bolz and Armin Rigo. 2007. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*.

[6] Stefano Cazzulani. 2012. Octane: The JavaScript benchmark suite for the modern web. *Retrieved December* 21 (2012), 2015. https://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html

[7] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, computers, controls* 2, 5 (1971), 45–50.

[8] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

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

[10] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *PLDI'92*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[11] Stephen C Johnson and Ravi Sethi. 1990. Yacc: a parser generator. *UNIX Vol. II: research system* (1990), 347–374.

[12] Neil D Jones. 1996. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)* 28, 3 (1996), 480–503.

[13] Neil D Jones, Peter Sestoft, and Harald Søndergaard. 1985. An experiment in partial evaluation: the generation of a compiler generator. In *International Conference on Rewriting Techniques and Applications*. Springer, 124–140.

[14] S. C. Kleene. 1952. Introduction to Metamathematics.

[15] David Leopoldseder. 2015. Master Thesis: Graal AOT JS: A Java to JavaScript Compiler. http://epub.jku.at/obvulihs/content/titleinfo/912629.

[16] David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. 2015. Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR. In *DLS'15*. ACM, New York, NY, USA, 91–103. https://doi.org/10.1145/2816707.2816715

[17] Michael Leuschel, Jesper Jørgensen, Wim Vanhoof, and Maurice Bruynooghe. 2004. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming* 4, 1-2 (2004), 139–191.

[18] Lionello A Lombardi. 1964. Lisp as the language for an incremental computer. (1964).

[19] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 821–839.

[20] Hanspeter Mössenböck, Albrecht Wöss, and Markus Löberbauer. 2003. *Der Compilergenerator Coco/R*. Citeseer.

[21] OpenJDK 2017. GraalVM -New JIT Compiler and Polyglot Runtime for the JVM;. http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html

[22] OpenJDK. 2021. Java Microbenchmark Harness (JMH). https://github.com/openjdk/jmh

[23] Oracle. 2018. GraalJS Repository. https://github.com/graalvm/graaljs

[24] Oracle. 2018. GraalVM. https://www.graalvm.org/

[25] Terence Parr and Kathleen Fisher. 2011. LL (*) the foundation of the ANTLR parser generator. *ACM Sigplan Notices* 46, 6 (2011), 425–436.

[26] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[27] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making Collection Operations Optimal with Aggressive JIT Compilation. In *SCALA 2017*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/3136000.3136002

[28] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *PLDI 2017*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[29] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages (DLS ’12)*. ACM Press, 73–82.

[30] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 301–312.