Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance

Matteo Basso
matteo.basso@usi.ch
Università della Svizzera italiana (USI)
Switzerland

Daniele Bonetta
daniele.bonetta@oracle.com
Oracle Labs
Netherlands

Walter Binder
walter.binder@usi.ch
Università della Svizzera italiana (USI)
Switzerland

Abstract

Abstract syntax tree (AST) interpreters allow implementing programming languages in a straight-forward way. However, AST interpreters implemented in object-oriented languages, such as e.g. in Java, often suffer from two serious performance issues. First, these interpreters commonly implement AST nodes by leveraging class inheritance and polymorphism, leading to many polymorphic call sites in the interpreter implementation and hence lowering interpreter performance. Second, widely used implementations of these interpreters throw costly runtime exceptions to model the control flow. Even though Just-in-Time (JIT) compilation mitigates these issues, performance in the first stages of the program execution remains poor.

In this paper, we propose a novel technique to improve both interpreter performance and steady-state performance, lowering also the pressure on the JIT compiler. Our technique automatically generates AST *supernodes* ahead-of-time, i.e., we automatically generate compound AST-node classes that encode the behavior of several other primitive AST nodes before the execution of the application. Our technique extracts common control-flow structures from an arbitrary, given set of ASTs, such as e.g. the functions of popular packages. It is based on matchmaking of AST structures, instantiation of matching supernodes, and replacement of the corresponding AST subtrees with the instantiated supernodes at load-time. We implement our technique in the GraalVM JavaScript engine, showing that our supernodes lead to an average interpreter speedup of 1.24×, an average steady-state speedup of 1.14×, and an average just-in-time compilation speedup of 1.33× on the web-tooling benchmark suite.

CCS Concepts: • **Software and its engineering** → **Interpreters**; **Source code generation**; *Virtual machines*; *Software performance*; Just-in-time compilers.

Keywords: AST, Interpreter, Code Generation, Performance Optimization, JavaScript, GraalVM, Truffle

ACM Reference Format:
Matteo Basso, Daniele Bonetta, and Walter Binder. 2023. Automatically Generated Supernodes for AST Interpreters Improve Virtual-Machine Performance. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '23), October 22–23, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3624007.3624050

1 Introduction

Abstract syntax tree (AST) interpreters allow implementing programming languages in an elegant and straight-forward way. However, AST interpreters frequently suffer from serious performance issues. AST interpreters are often implemented in object-oriented languages, such as e.g. in Java or C++, and exploit features such as class inheritance and polymorphism. Even though these features improve code maintainability, polymorphic call sites in the interpreter implementation lower interpreter performance. Moreover, widely used implementations of AST interpreters rely on costly runtime exceptions to model the control flow of the interpreted language. For instance, exceptions are used to break the execution of a loop iteration or to return from a function.

To mitigate these performance issues, research has mostly focused on the development and improvement of just-in-time (JIT) compilers. The JIT compiler—executed at runtime and usually concurrently with the application—transforms the ASTs into optimized machine code that the system can execute without the need for interpretation. After the JIT compiler has compiled all relevant ASTs, the system can reach a steady state, i.e., a state where the system internals have stabilized and the system executes predominantly JIT-compiled code. JIT compilation aims at high steady-state performance, but does not solve the problem of poor startup performance of AST interpreters.

In this paper, we make a first step towards the investigation of a new technique to improve both interpreter performance and steady-state performance, lowering also the pressure on the JIT compiler (i.e., feeding the JIT compiler with input code that can be more easily optimized). We base our work on the concept of AST supernode [7] (also known as superoperator [16] in the context of bytecode interpreters), i.e., compound AST-node classes that encode the behavior of several other primitive AST nodes. In particular, we aim at answering the following research questions:

RQ1. Can supernodes speed up interpreter performance?
RQ2. Can supernodes help the JIT compiler produce better optimized machine code?
RQ3. Can supernodes reduce the pressure on the JIT compiler?

Contributions. To answer RQ1, RQ2, and RQ3, we propose a new technique to improve virtual-machine performance by exploiting AST supernodes. Our technique leverages automatic ahead-of-time generation of supernodes and runtime installation of matching supernodes, exploring a new point in the design space between ahead-of-time and just-in-time compilation (Section 4). Our technique automatically generates supernodes from a set of ASTs that exercise common control-flow patterns before building the Virtual Machine (VM). At runtime, our technique performs an efficient matchmaking of AST structures, instantiation of matching supernodes, and replacement of the corresponding AST subtrees with the instantiated supernodes. Despite the ahead-of-time generation of supernodes, our technique does not prevent the JIT compiler from exploiting profiling data to fully optimize the emitted machine code [4].

We implement our technique in the GraalVM JavaScript engine (also known as Graal.js [11]) and we evaluate our implementation on the web-tooling benchmark suite [17], showing that supernodes improve both interpreter and steady-state performance. Moreover, we show that supernodes reduce JIT compilation time, at the cost of moderate extra memory, increased VM building time, and some startup costs for loading the supernode classes (Section 5).

We complement the paper by presenting background information (Section 2), a motivating example (Section 3), a discussion of related work and our technique (Sections 6 and 7, respectively), and some concluding remarks (Section 8).

2 Background

In this section we present the required background on AST interpreters (Section 2.1) and Graal.js (Section 2.2).

2.1 AST Interpreters

AST allows representing a program using a simple tree structure where each AST node represents a language operation. For example, AST nodes may represent control-flow constructs, such as `if` and `while`, or primitive operations, such as arithmetic expressions, memory accesses, function calls, etc. We call AST nodes that represent control-flow constructs *control-flow nodes* and AST nodes that represent primitive operations *non-control-flow nodes.*

In an object-oriented implementation, AST nodes are often subclasses of a common abstract class, and ASTs are created exploiting composition—each AST node instance stores references to its children (if any). Each AST node implements the code to perform the behavior of the operation it encodes, returning the produced result. Each AST node is responsible for invoking the execution of its children.

The control-flow of the language to be interpreted is implemented by exploiting the control-flow structures of the language used to implement the interpreter. To implement `break`, `continue`, and `return` statements that span multiple AST nodes, AST interpreters often employ runtime exceptions [9, 27]. An AST node throws a runtime exception of a specific type and an ancestor AST node catches and handles that runtime exception. We note that control-flow nodes are usually found in the lower tree levels of an AST (i.e., closer to the root node), while non-control-flow nodes are usually found in the higher levels (i.e., closer to the leaf nodes).

2.2 GraalVM JavaScript (Graal.js)

We implement our technique in Graal.js [11], an open-source, high-performance implementation of the JavaScript programming language built on top of GraalVM [26]. GraalVM is a managed language runtime system based on the Java Virtual Machine (JVM), capable of executing several different programming languages such as Ruby, R, Python, and JavaScript. GraalVM supports ahead-of-time compilation thanks to native images [24] and yields high performance thanks to the Graal compiler [4].

Graal.js is implemented using Truffle [25], a language implementation framework that allows implementing self-optimizing AST interpreters running on GraalVM [27], i.e., implementing AST interpreters using custom APIs that allow the Graal compiler to partially evaluate [5] and efficiently JIT compile ASTs. In particular, the goal of the partial evaluator is to remove the AST-interpretation overhead by following the execution path in the program ASTs, before other JIT-compiler optimizations take place.

We define as *VM startup* the initial VM setup before program interpretation begins, as *warmup* the initial stages of program execution taking place after the VM startup that include program interpretation, partial evaluation, and JIT compilation, and as *steady-state* the stages of program execution where the system has stabilized and all the performance-relevant ASTs have been compiled.

In Graal.js, AST nodes are subclasses of the abstract class `Node` provided by the Truffle API. Each node class implements an `execute` method that defines the behaviour of the node and declares the children the node accepts. To model the control-flow, Graal.js exploits runtime exceptions, as described in the previous subsection.

```
function fibonacci(n) {
  if (n < 2) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Figure 1. JavaScript implementation of function `fibonacci`.

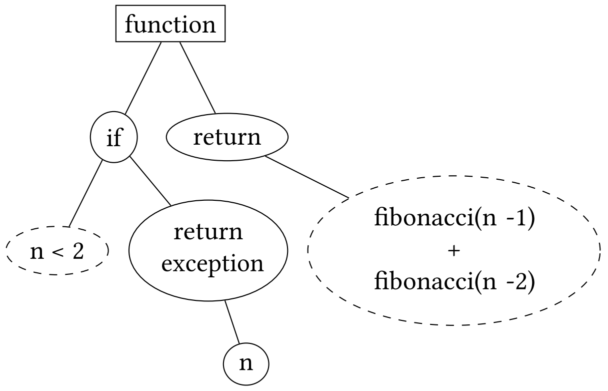

Figure 2. Simplified AST of the JavaScript function `fibonacci` (Figure 1). Collapsed nodes are represented using dashed borders.

3 Motivating Example

AST supernodes are compound AST-node classes that encode the behavior of several other primitive AST nodes. While supernodes can help improving AST interpreter performance, defining what nodes to aggregate in a supernode is non trivial. In this section, we illustrate our technique, showing how an example AST (consisting of primitive nodes only) is transformed into an AST that contains an (automatically generated) supernode.

Figure 1 shows a JavaScript implementation of the function `fibonacci`, i.e., a function that, given a number `n` as parameter, returns the `n`-th element of the fibonacci sequence. A simplified AST corresponding to the `fibonacci` function is shown in Figure 2. For a more compact presentation, we do not show all the nodes of the AST; we collapse some subtrees into single nodes (illustrated with dashed borders). We show simplified implementations of the execute methods of the `function`, `if`, `return`, and `return exception` AST nodes in Figure 3.

In the example, the root `function` node has two children: an `if` node and a `return` node. Unless a control-flow exception is thrown, these children are executed one after the other, as reported in the `execute` method of Figure 3 at lines 2–17. The `if` node first evaluates the condition expression `n < 2` (line 21) and only if the condition evaluates to `true`, the `if` node executes its body (line 23), i.e., the subtree with the `return exception` node as root. The `return exception` node in the body of the `if` statement throws an exception

```
1  // execute method of the `function` node
2  public Object execute() {
3    try {
4      int childrenLength = children.length;
5      if (childrenLength == 0) {
6        return null;
7      }
8
9      for (int i = 0; i < childrenLength - 1; i++) {
10       children[i].execute();
11     }
12
13     return children[childrenLength - 1].execute();
14   } catch (ReturnException re) {
15     return re.getValue();
16   }
17 }
18
19 // execute method of the `if` node
20 public Object execute() {
21   if (condition.executeBoolean()) {
22     if (thenPart != null) {
23       return thenPart.execute();
24     } else {
25       return null;
26     }
27   } else {
28     if (elsePart != null) {
29       return elsePart.execute();
30     } else {
31       return null;
32     }
33   }
34 }
35
36 // execute method of the `return` node
37 public Object execute() {
38   return child.execute();
39 }
40
41 // execute method of the
42 // `return exception` node
43 public Object execute() {
44   throw new ReturnException(child.execute());
45 }
```

Figure 3. Simplified implementation of the `function`, `if`, `return`, and `return exception` nodes of Figure 2.

that encapsulates the value produced by the evaluation of its child node (line 44). This is because this `return exception` node must break the interpretation loop of the `function` node at lines 9–11. The `function` node catches the exception and returns the value the exception encapsulates (lines 14–16). Finally, the `return` node executes its child, returning the produced result (line 38). In this case, since the `return` node is the last child of the `function` node, no exception to model the control flow is required—the `function` node simply returns the result produced by the last child (line 13).

To avoid the use of an exception (that can cause runtime performance degradation), a supernode can be generated that replaces all the control-flow nodes of the AST, as shown in Figure 4. In particular, the control-flow supernode replaces the `function`, `if`, `return`, and `return exception` nodes. For the sake of exemplification, a simplified implementation of the execute method of this control-flow supernode is reported in Figure 5 (the details of the generated code will be shown later in Section 4.2) and consists of a ternary operator

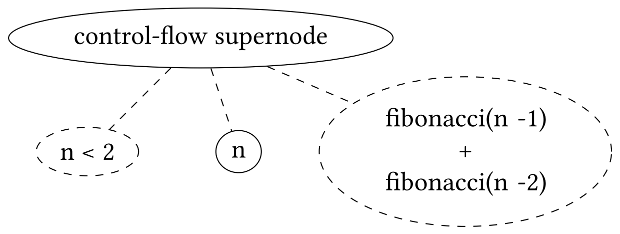

Figure 4. Simplified AST of the JavaScript function `fibonacci` (Figure 1) with a supernode that encodes the control-flow. Collapsed nodes are represented using dashed borders.

```
1  public Object execute() {
2    return condition.executeBoolean()
3                ? thenPart.execute()
4                : fallThrough.execute();
5  }
```

Figure 5. Simplified execute method of the control-flow supernode of Figure 4.

where each expression calls the execute method of a child node (lines 2–4). Child nodes are stored in the instance fields `condition` (line 2), `thenPart` (line 3), and `fallThrough` (line 4). To reduce the number of polymorphic call sites, the declared type of these fields is not `Node`, i.e., the common ancestor class of all the AST nodes. Instead, the declared types of the fields `condition`, `thenPart`, and `fallThrough` are the leaf classes `LessThanNode`, `ParameterNode`, and `AddNode`, respectively. In the figure, dashed edges represent monomorphic calls that can generally be better optimized, hence improving performance.

4 Supernode Generation and Installation

In this section, we first give an overview of our technique to automatically generate and install AST supernodes (Section 4.1). Then, we detail the three steps of our technique, namely *Supernode Generation* (Section 4.2), *Lookup-Tree Generation* (Section 4.3), and *Supernode Installation* (Section 4.4).

4.1 Overview

Figure 6 shows how our technique is integrated into the VM building process and workload execution. The three steps introduced by our technique are represented using gray nodes with dashed borders.

The first step of our technique is *Supernode Generation*. This step takes place at build time, before the building of the production VM, and aims at generating a supernode for the lower tree levels of each AST in a collection of functions, i.e., the part of the AST that contains the control flow of the function the AST encodes.[1] We call this collection of functions *generation set*, which will consist of the functions of popular packages. Supernode generation is eager—it creates the maximal supernode that encodes the whole control-flow of the provided AST. We do not create supernodes that encapsulate only part of an AST's control-flow nodes. Moreover, we do not create supernodes for non-control-flow AST nodes (such as arithmetic operations, function calls, or memory accesses). As detailed in Section 3, to avoid polymorphic call sites within supernodes, we generate supernodes that store their children in instance fields whose declared types are the dynamic types of the children nodes. We assume that the interpreter is implemented in a statically typed language, i.e., Java in the case of Graal.js.

After the supernodes have been generated, the supernodes need to be organized for efficient matchmaking. This is done in the second step of our technique, *Lookup-Tree Generation*. In our technique, each supernode is associated with a sequence of unique IDs (henceforth called *supernode structure* or simply *structure*) that encodes the structure of the supernode; the structure contains the types IDs of the AST control-flow nodes that the supernode encapsulates (in a fixed traversal order) and the types IDs of the children nodes that the supernode accepts, e.g., the type IDs of the corresponding non-control-flow nodes. At execution time, before parsing the source code, we load the supernode classes and use the supernode structures to generate a lookup-tree for the subsequent supernode installation step.

Finally, after supernodes have been generated and organized for efficient matchmaking, suitable supernodes are installed at runtime. The parser needs to be aware of their existence and should instantiate them when appropriate. This is achieved in the third step of our technique, *Supernode Installation*. After the creation of each AST, we match AST structures, instantiate matching supernodes, and replace the corresponding AST subtrees with the instantiated supernodes. To do so, we traverse the lookup-tree generated in the second step of our technique together with each parsed AST, collecting children nodes that will be used to instantiate a supernode, if a matching one is found.

After the supernode installation step, the VM executes the user code by interpreting the optimized ASTs that contain our supernodes (eventually JIT compiling the optimized ASTs to machine code).

4.2 Supernode Generation

Before the VM building phase, we generate the supernodes that will be included in a production build of the VM, e.g., by using the ASTs of the functions of popular packages.

Algorithm 1 depicts the pseudocode for generating supernodes. As input, the algorithm takes a set of ASTs for which supernodes shall be created and produces the supernodes

[1]For simplicity, in this paper, we refer to the ASTs of functions written in the interpreted language, whereas our approach is applied to the root of any generated AST (i.e., for functions, procedures, methods, constructors, etc.).

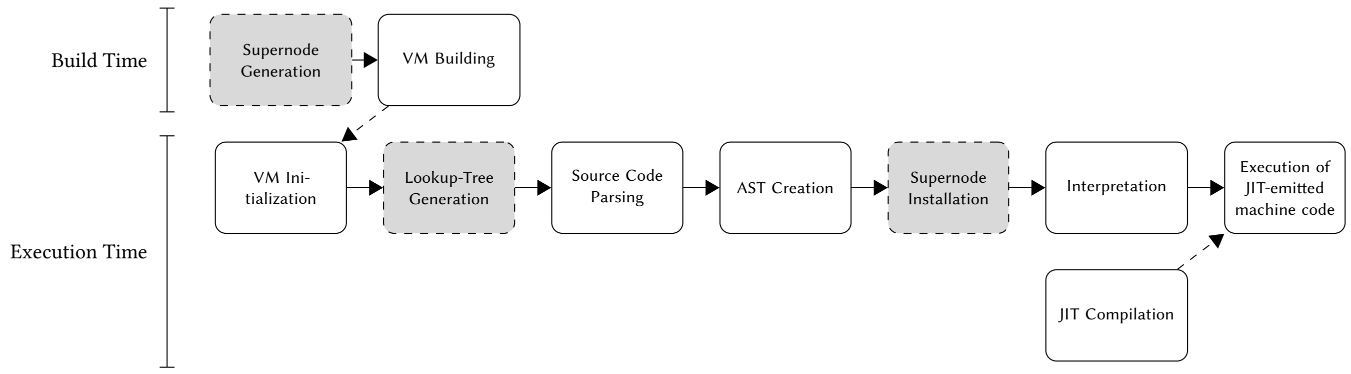

Figure 6. Integration of the proposed technique into the VM building process and workload execution. White nodes with solid borders represent baseline steps. Gray nodes with dashed borders represent the steps introduced by our technique.

Algorithm 1: Supernodes generation

```
generateSupernodes(A): generate supernodes for the provided ASTs
Input: A, the set of ASTs to be used to generate supernodes
Output: Supernode classes dumped as Java files
1  $S \leftarrow$ new $Set()$
2  foreach $a \in A$ do
3      $root \leftarrow a.root()$
4      $structure, code \leftarrow generateSupernode(root)$
5      if $exceedsThreshold(structure)$ and $!S.contains(structure)$ then
6          $code.addStructureField(structure)$
7          $code.dump()$
8          $S.add(structure)$
```

as output in the form of Java files. In particular, the algorithm iterates over the provided ASTs (line 2), extracts the root of each AST (line 3), and calls a subroutine that recursively generates the supernode, returning the supernode structure and the corresponding code. Then, our algorithm checks whether the generated supernode should be saved or discarded (line 5). We save supernodes whose structure encodes a minimum number of control-flow nodes (in our experiments, we set this threshold to 3) and whose structure has not yet been encountered before, i.e., we save supernodes that are sufficiently complex and we avoid duplicates. If a supernode is saved, we store its structure in the supernode class in a `static final` field (line 6), we dump the supernode class as a Java file (line 7), and we update the set S (declared at line 8) that contains the already saved supernodes (line 1).

Algorithm 2 shows the recursive subroutine to generate supernodes. As input, the algorithm takes an AST node for which a supernode is to be created and returns a pair as output. The first element of the pair is the supernode structure

Algorithm 2: Supernode generation

```
generateSupernode(n): recursively generate a supernode for the lower levels of the provided AST that contain the control-flow nodes
Input: n, root of an AST subtree for which a supernode may be created
Output: A pair that consists of:
(1) The supernode structure as a list
(2) The code of a supernode class encoding the lower tree levels of the input AST
1  $code \leftarrow$ new $ClassBuilder()$
2  $structure \leftarrow$ new $List()$
3  $id \leftarrow nodeTypeId(n)$
4  $structure.append(id)$
5  if $isControlFlowNode(n)$ then
6      $C \leftarrow children(n)$
7      $code$ += $emitSourceCodeBeforeChildren(n, C)$
8      for $k \leftarrow 1$ to $length(C)$ do
9          $c \leftarrow C_k$
10         $code$ += $emitSourceCodeBeforeChild(n, c)$
11         $cs, ccode \leftarrow generateSupernode(c)$
12         $structure.appendAll(cs)$
13         $code$ += $ccode$
14         $code$ += $emitSourceCodeAfterChild(n, c)$
15     $code$ += $emitSourceCodeAfterChildren(n, C)$
16 else
17     $code$ += $emitChildInvocation(n)$
18 return $structure, code$
```

represented as a list, while the second element is the code of a supernode that corresponds to the returned structure. First, the algorithm creates a new code builder (line 1), a new

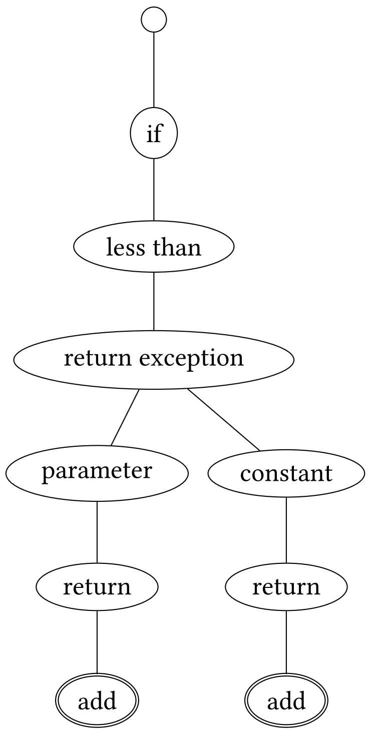

Figure 7. Lookup-tree encoding two supernodes. Nodes containing a reference to a supernode class are represented using double borders. For the sake of exemplification, each illustrated node contains the AST node type name instead of the unique ID of the corresponding node type.

structure (line 2), and appends the id of the input node to the newly created structure (lines 3–4). Then, the algorithm checks whether the input node is a control-flow node (line 5).

If the input node is not a control-flow node (lines 16–17), the algorithm generates a field in the supernode class with the same declared type as the dynamic type of the child node. The dynamic node type is uniquely identified by *id* and the field will be used to store the child node. The algorithm then generates code that invokes the execute method of the child node (line 17).

If the input node is a control-flow node, the algorithm emits Java source code that corresponds to the primitive operations encoded by the input node (lines 7, 10, 14, and 15). We provide more detail on the generated Java source code in the next paragraph. The algorithm recursively traverses the child nodes of this input node (line 11), generating the corresponding structures and code. The algorithm merges the generated structure and code into the current structure and code (lines 12–13), allowing a recursive supernode generation. Finally, we return the structure and code as a pair (line 18).

On the one hand, extracting supernode structures from the generation set, creating the lookup-tree of supernodes, and installing supernodes at runtime are interpreter-independent algorithms and will work for any Truffle AST interpreter out-of-the-box. On the other hand, the generation of the Java source code of supernodes requires knowledge of the interpreter implementation and currently is a manual effort by the developers of an AST interpreter. In our implementation, we have dedicated source-code generation functions for 45 different control-flow node types (Graal.js contains more than 300 AST node types). Figure 8 shows the code emitted by our generation functions that corresponds to the control-flow supernode of Figure 4, without the simplifications reported in Figure 5. Consider the `if` AST node encapsulated by the supernode (Figure 2), the `if` generation function emits a Java `if` statement (line 3) that accepts (as its condition) the result produced by the execution of the condition child node (line 2) and (as its body) the code produced by the traversal of the if-body subtree (lines 4 and 5). In supernodes, AST `return` nodes that throw exceptions to model the control flow are simply replaced by Java `return` statements (line 5). Even though not present in this example, when generating Java source code, we generate also Java labels that can be later referenced by `break` and `continue` statements. In this way, we can avoid the usage of exceptions to model the control flow. We are investigating techniques to automate code generation and further reduce the burden on the interpreter developer.

```
1 public Object execute() {
2   boolean tmp1 = child1.executeBoolean();
3   if (tmp1) {
4     Object tmp2 = child2.execute();
5     return tmp2;
6   }
7   Object tmp3 = child3.execute();
8   return tmp3;
9 }
```

Figure 8. Execute method of the control-flow supernode of Figure 4 without simplifications.

4.3 Lookup-Tree Generation

After the VM Initialization, we iterate over the structures of all the generated supernodes and build a lookup-tree that efficiently maps AST structures to supernodes. The lookup-tree is a trie (i.e., a prefix tree), for which the alphabet is the set of node IDs that occur in the supernodes' structures. Apart from the root, each node in the lookup-tree stores the type ID of an AST node.

Algorithm 3 reports our pseudocode for generating the lookup-tree. As input, the algorithm takes the supernodes generated by the *Supernode Generation* step and returns the root of the generated lookup-tree as output. In particular, the lookup-tree has an empty root, where later the matchmaking of AST structures will start (line 1). We iterate over the input supernodes (line 2) and we extract the structure of each supernode (line 4). Then, we iterate over the ids that compose the structure of each supernode (line 5) and

Algorithm 3: Lookup-tree generation using supernode structures.

```
generateLookupTree(supernodes): generates a lookup-tree for the provided supernodes
Input: supernodes to register in the lookup-tree
Output: A lookup-tree that contains the provided supernodes
1  root ← new Node()
2  foreach m in supernodes do
3      c ← root
4      ids ← structure(m)
5      for k ← 1 to length(ids) do
6          id ← ids_k
7          t ← c.get(id)
8          if t == null then
9              t ← new Node()
10             c.addChild(t)
11         c ← t
12     c.setSupernode(m)
13 return root
```

for each supernode we create a path in the tree. This path starts from the root (line 3) and has an edge for each id in the supernode structure (lines 6–7). When generating the lookup-tree, the algorithm traverses the edges that already exist and creates new edges only if one cannot be found (lines 8–10). After the creation of the path for each supernode, the algorithm associates the supernode class to the last node of the path. This supernode class represents the supernode whose structure is equal to the path that led to that node. We note that each node, including non-leaf nodes, potentially may contain a reference to a supernode class, because a supernode may have a structure that is a prefix of another supernode's structure. Finally, the algorithm returns the (empty) root (line 13).

Figure 7 shows an example lookup-tree that stores two supernodes. The path that starts from the root and ends with the `add` node on the left side of the Figure encodes the example supernode reported in Figure 4. The path that starts from the root and ends with the `add` node on the right side of the Figure encodes a supernode that returns a constant (instead of a function parameter) in the body of the if-statement. In practice, the only difference between the two supernodes is the declared type of the field storing a reference to a child node.

4.4 Supernode Installation

After the source code parsing and the creation of each AST, we match AST structures, we instantiate supernodes, and we replace the matched subtrees with the instantiated supernodes. To do so, we traverse each AST and the lookup-tree at the same time, accumulating children nodes and potentially finding a supernode that replaces part of the AST.

Algorithm 4 reports the pseudocode for replacing AST subtrees with supernodes. The algorithm takes (as input) the root node of an AST subtree that may be modified with supernodes and the root node of the lookup-tree that contains the supernode classes. The algorithm returns (as output) the root node of an AST subtree that contains the installed supernodes. If no supernode is installed, the algorithm returns the root of the unmodified AST subtree provided as input.

The algorithm starts by searching for the matching AST structure, using the recursive subroutine `lookupTreeSearch` (line 1). This subroutine returns a pair that contains (1) a node of the lookup-tree potentially containing the supernode class matching the provided AST or an empty result if the lookup-tree cannot be traversed due to missing edges, and, (2) a list that contains the roots of the subtrees of the input AST needed for supernode instantiation. We will describe this subroutine in the next paragraph. After performing the search, the algorithm checks whether the subroutine found a lookup-tree node and whether this node is associated with a supernode class (line 2). If not, the algorithm returns the root of the unmodified AST subtree (line 6). Otherwise, the algorithm extracts the supernode class associated with the lookup-tree node (line 3) and returns a modified AST, i.e., an instance of the matched supernode class that takes as children nodes the nodes contained in the list returned by the recursive subroutine (line 4).

Algorithm 5 reports the pseudocode of the recursive lookup-tree search subroutine. The algorithm takes (as input) an AST node and a lookup-tree node to start the search, and returns (as output) the aforementioned pair. The algorithm first initializes a list that will contain the childen nodes to be used for (potential) supernode instantiation (line 1). Then, the algorithm extracts the unique id of the AST node provided as a parameter (line 2) and traverses the lookup-tree using this id, updating the current lookup node (line 3). If an edge for this id cannot be found in the lookup-tree (line 4), the algorithm returns a pair of `null` values (line 5). Otherwise, the algorithm checks whether the input node is a control-flow node (line 6). If the input node is not a control-flow node, the algorithm appends the input node to the children list (line 15).

If the input node is a control-flow node, the algorithm iterates over the children of the input node (line 8). In particular, the algorithm performs a recursive search for each child, providing the current child and the current lookup node as parameters (line 9). If the lookup node returned by the recursive call is `null` (line 11), meaning that the lookup-tree cannot be traversed, the algorithm terminates early by returning a pair of `null` values (line 12). Otherwise, the algorithm updates the current lookup node, assigning the lookup

Algorithm 4: Supernode installation

```
installSupernode(n, l): tries to install a supernode
                        in the provided AST
Input:
n, root of an AST subtree for which a lookup-tree
node may be found
l, root node of a lookup-tree
Output: Root node of an AST subtree containing
        installed supernodes. Otherwise, the input
        root.
1  $c, vs \leftarrow lookupTreeSearch(n, l)$
2  if $c$ != $null$ and $c.hasSupernode()$ then
3      $m \leftarrow c.getSupernode()$
4      return $m.instantiate(vs)$
5  else
6      return $n$
```

Algorithm 5: Lookup-tree search

```
lookupTreeSearch(n, l): tries to find a node in the
                        lookup-tree for the
                        provided AST
Input:
n, root of an AST subtree for which a lookup-tree
node may be found
l, current lookup node in the lookup-tree
Output: A pair that consists of:
(1) A node of the lookup-tree potentially containing
the supernode class matching the provided AST or
null if the lookup-tree cannot be traversed further
due to missing edges
(2) A list that contains the roots of the subtrees of the
input AST needed for supernode instantiation
1  $vs \leftarrow$ new $List()$
2  $i \leftarrow nodeTypeId(n)$
3  $l \leftarrow l.get(i)$
4  if $l == null$ then
5      return $null, null$
6  if $isControlFlowNode(n)$ then
7      $C \leftarrow children(n)$
8      for $k \leftarrow 1$ to $length(C)$ do
9          $lc, vz \leftarrow lookupTreeSearch(C_k, l)$
10         $l \leftarrow lc$
11         if $l == null$ then
12             return $null, null$
13         $vs.appendAll(vz)$
14 else
15     $vs.append(n)$
16 return $l, vs$
```

node returned by the recursive call (line 10), and appends the children list returned by the recursive call to the local children list (line 13). After processing the input node and potentially all its children nodes, excluding the case of early termination, the algorithm returns the latest lookup-tree node and the updated children list (line 16).

We note that we stop traversing the AST and we do not install any supernode as soon as matching fails. This is because our supernode-generation algorithm is eager and each supernode encodes a specific structure and accepts a specific number of children nodes of exact types—a single failure indicates a mismatch in the AST structure. Since the installation step is deterministic, there is only one match for one supernode, the same AST cannot match two different supernodes. Moreover, if the algorithm does not stop because of missing edges, the algorithm may return a non-leaf node of the lookup-tree. This node may contain a supernode class. We do not need to traverse the lookup-tree up to the leaves. The complexity of the supernode installation algorithm is $O(n)$ where n is the number of nodes of the input AST.

5 Evaluation

In this section, we first present our experimental setup (Section 5.1). Then, we present the interpreter speedups (Section 5.2), steady-state speedups (Section 5.3), and compilation-time speedups (Section 5.4) achieved by our technique. Finally, we discuss the memory overhead (Section 5.5) and the VM build-time overhead (Section 5.6) introduced by our supernodes, as well as the overhead of lookup-tree generation (Section 5.7).

5.1 Evaluation Settings

We run our experiments on a machine equipped with an 18-core Intel i9-10980XE (3.00 GHz) and 256 GB of RAM running Linux Ubuntu (kernel v. 5.4.0-58-generic). Frequency scaling, turbo boost, and hyper-threading are disabled, CPU governor is set to "performance". We conduct our experiments on Graal.js 22.3.0 community edition, based on OpenJDK 11 that uses the Graal compiler. In particular, we modify both Graal.js and Graal to implement our technique. We perform our experiments on the web-tooling benchmark suite [17], consisting of 18 benchmarks.

To generate our supernodes, as generation set, we use a collection of popular JavaScript packages for web development. We set the supernode threshold to 3, i.e., we create supernodes that encode at least 3 control-flow nodes. In this setting, our supernode generation creates ~6500 supernodes.

5.2 Interpreter Speedup

In this section, we answer RQ1 by evaluating the impact of our technique on interpreter performance. To do so, we force interpretation by disabling JIT compilation, we run 10

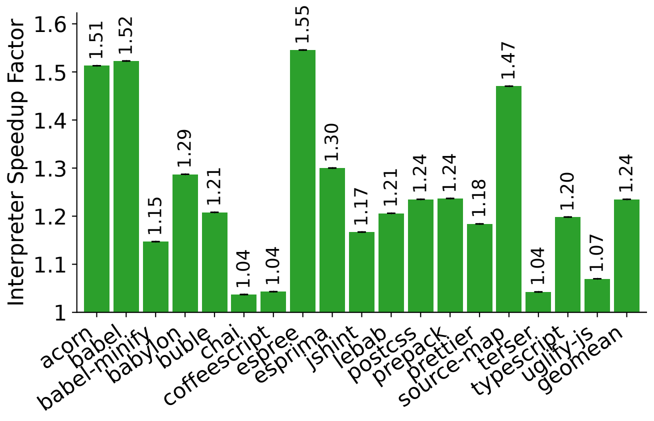

Figure 9. Interpreter speedup factors achieved by the proposed technique.

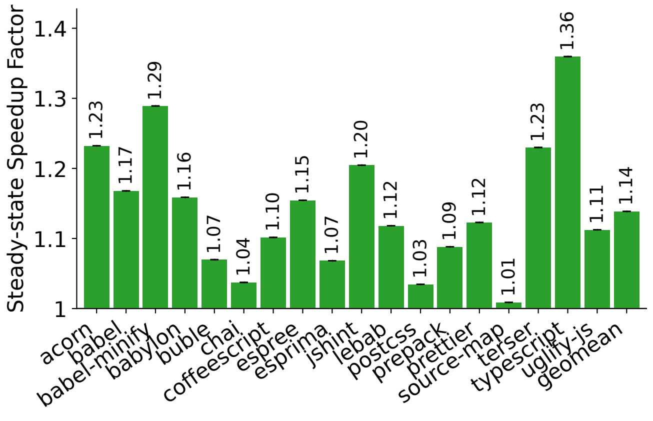

Figure 10. Steady-state speedup achieved by the proposed technique.

iterations of each benchmark, and we take the average of the last five time measurements. In this way, we let cache behaviors stabilize and we avoid the potential measurement perturbations of the first iterations. To mitigate measurement noise, we repeat this process five times.

Figure 9 reports the interpreter speedup as $T_{baseline}/T_{supernodes}$, where $T_{baseline}$ refers to the average execution time obtained without using our technique and $T_{supernodes}$ refers to the average execution time obtained using our technique. The benchmarks are reported on the x-axis of the plot, while the speedup factor is reported on the y-axis. Above each bar, we report the exact speedup factor. The black error bars represent the 95% confidence intervals (CI) of the measurements. We note that the error bars are narrow for most of the experiments thanks to the stable measurements obtained, both with and without supernodes. The last bar on the right represents the geometric mean of all the other speedup factors.

We notice that our technique does not introduce any slowdown for any benchmark. Interpreter speedups range from 1.04× (*chai*, *coffeescript*, and *terser*) to 1.55× (*espree*), 1.24× on average.[2] This is because the VM needs to traverse less AST nodes upon interpretation—our supernodes reduce the size of the ASTs with possible cache improvements. Moreover, our supernodes help reduce the overhead of using expensive exceptions to model the control flow as well as the number of polymorphic call sites. Hence, we positively answer RQ1.

5.3 Steady-state Speedup

Here, we answer RQ2 by evaluating the impact of our technique on steady-state performance, i.e., we investigate the impact of supernodes on JIT compilation, to understand whether supernodes lead to better optimized JIT-compiled code and consequently to speedups in steady state. We run 1 000 iterations of each benchmark (without disabling JIT compilation) and take the average of the last 10 iteration time measurements. In this way, we let JIT compilation stabilize and we take our measurements only after all the relevant ASTs have been compiled. We repeat this process five times.

Similarly to Figure 9, Figure 10 reports the steady-state speedup as $T_{baseline}/T_{supernodes}$, where $T_{baseline}$ refers to the average execution time obtained without using our technique and $T_{supernodes}$ refers to the average execution time with our technique.

Steady-state speedups range from 1.01× (*source-map*) to 1.36× (*typescript*), 1.14× on average. Our experimental results positively answer RQ2, confirming that supernodes improve JIT compilation. As part of our future work, we plan to conduct an in-depth study on the effect of supernodes on JIT-compiler budget-driven optimization heuristics, which may lead to better optimized JIT-emitted code.

5.4 Compilation-time Speedup

We evaluate now the impact of our technique on compilation time, answering RQ3. We run 1 000 iterations of each benchmark (without disabling JIT compilation) and take the overall compilation time.

Figure 11 reports the compilation-time speedup as $T_{baseline}/T_{supernodes}$, where $T_{baseline}$ refers to the compilation time obtained without using our technique and $T_{supernodes}$ refers to the compilation time obtained using our technique.

Compilation-time speedups range from 0.62× (*babylon*) to 2.86× (*lebab*), 1.33× on average. The benchmarks *lebab* (2.86×) and *typescript* (2.76×) benefit the most from our supernodes, because these benchmarks contain functions with complex control flow that is captured by supernodes. Our technique yields a compilation-time slowdown on the

[2]Average speedup factors across multiple benchmarks are computed using the geometric mean.

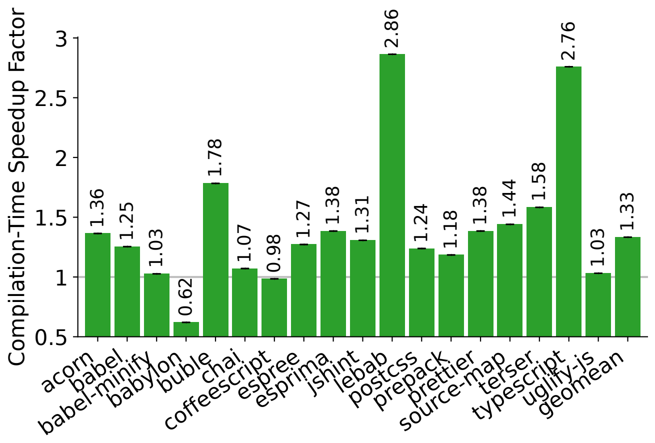

Figure 11. Compilation-time speedup achieved by the proposed technique.

benchmarks *babylon* (0.62×) and *coffeescript* (0.98×). By investigating the root cause of the compilation-time slowdowns, we find that in *babylon* supernodes lead to a significant increase of compilations w.r.t. the baseline. When using supernodes, the JIT compiler compiles the same functions multiple times with different specializations, increasing the number of compilations. We plan to conduct a thorough investigation of this phenomenon as part of our future work.

Overall, we can positively answer RQ3, stating that supernodes reduce the pressure on the partial evaluator and the JIT compiler. Since the ASTs contain fewer nodes, the partial evaluator needs to traverse and partially evaluate fewer nodes. Moreover, the JIT compiler does not need to optimize runtime exceptions and polymorphic call sites in the supernodes, saving compilation time.

5.5 Memory Overhead

We discuss now the memory overhead introduced by our supernodes. For all our experiments, we install supernodes generated by using a collection of the most commonly used JavaScript packages. For this reason, we do not report the overheads for each benchmark. Instead, we report a single memory overhead factor computed as $M_{supernodes}/M_{baseline}$, where $M_{supernodes}$ is the size of a VM build that contains our supernodes, and $M_{baseline}$ is the size of a VM build without supernodes, respectively.

Using a supernode threshold of 3 (i.e., the minimum number of control-flow nodes subsumed by a supernode, as described in Section 4.2), our technique yields a memory overhead of 1.09× (the sizes are ~671MB and ~615MB for the VM build that contains our supernodes and the VM build without supernodes, respectively).

On modern machines where memory consumption is often not a major issue, we consider such a memory overhead acceptable. Nonetheless, we note that memory consumption may be reduced by performing a study to identify and keep only the most commonly used supernodes (of the ~6500 automatically generated supernodes from our generation set).

5.6 VM Build-time Overhead

Here, we discuss now the VM build-time overheads introduced by our supernodes. Similarly to Section 5.5, we do not report the overheads for each benchmark but a single VM build-time overhead factor computed as $T_{supernodes}/T_{baseline}$, where $T_{supernodes}$ is the time required to build a VM that contains our supernodes, and $T_{baseline}$ is the time required to build a VM that does not contain our supernodes, respectively.

Using ~6500 automatically generated supernodes, our technique yields a VM build-time overhead of 1.73× (121 seconds with supernodes, versus 70 seconds without supernodes). We note that a high VM build-time overhead factor is a minor drawback of our technique, considering the achieved runtime speedups. Indeed, once supernodes have been generated, the VM needs to be built only once before the production release, without any impact on the final user.

5.7 Lookup-Tree Generation Overhead

In this section, we evaluate the overhead of generating the lookup-tree, i.e., the overhead that our technique introduces upon VM startup. In our experiment, we measure the time required to build the lookup-tree and to load and link the Java class files of the supernodes in a fully sequential setting without any optimization—we parallelize neither lookup-tree generation nor supernode class loading, and we do not run the lookup-tree generation concurrently with other VM initialization steps.

Our implementation requires ~3 seconds to generate the lookup-tree and to load the code of ~6500 supernodes. We note that the startup is rather expensive in Graal.js and the lookup-tree generation is not critical for performance. For this reason, we have not optimized this phase yet. To lower this overhead in the future, we note that the lookup-tree can be built asynchronously and in a parallelized manner (including the loading and linking of the supernode classes), or serialized at VM building time and then deserialized upon VM startup. Moreover, the supernode classes may also be lazily loaded upon the first match in the process of supernode installation. With such optimizations, one can hide (part of) the costs of lookup-tree creation and supernode class loading.

6 Related Work

Different techniques try to improve warmup performance of managed language runtime systems by optimizing and implementing efficient interpreters.

The concept of supernode was initially proposed in the context of bytecode interpreters. In particular, Proebsting

[16] reduce the cost of instruction dispatching in bytecode interpreters by creating superoperators, i.e, compound operations composed of many smaller primitive operations that avoid costly per-operation overheads. Larose et al. [7] apply a similar technique to AST interpreters, manually generating 20 supernodes. Differently from their method, we automatically generate and install supernodes, reducing the burden on interpreter developers. Our technique requires only the implementation of the interpreter-specific generation functions. Moreover, we implement our technique in a more complex Truffle implementation used in industry (Graal.js), in contrast to the toy language TruffleSOM targeted in [7].

Sun's JVM implementation [22] dynamically replaces occurrences of certain bytecode instructions—after their first execution—with more efficient *_quick* pseudo-instructions. The pseudo-instructions speed up the execution by taking advantage of the work done the first time the associated normal instructions are executed. In contrast to this method, we perform supernode instantiation right after AST creation, before an AST is executed for the first time.

Static compilation, also known as ahead-of-time (AOT) compilation, improves warmup performance by compiling applications before execution and hence removing interpretation costs. This feature is available for widely used languages, such as Java [6, 14] and JavaScript [19, 20]. The main issue of AOT compilation is the degradation of steady-state performance—specifically in the case of dynamic languages—since aggressive, speculative optimizations may not be performed due to the lack of profiling data. While our supernodes are generated ahead-of-time, our technique does not compile the source code of the application ahead-of-time. For this reason, in contrast to static compilation, our technique allows for steady-state performance improvements.

Other techniques try to improve the performance of either bytecode or AST interpreters. Threaded code [2] solves the branch prediction problem in bytecode interpreters. Savrun-Yeniçeri et al. [18] speed up hosted interpreters on the JVM by providing annotations that enable the generation of efficient threaded code and avoid the insertion of unnecessary runtime checks produced by the JIT compiler. Brunthaler [3] illustrate inline-caching optimizations, a technique to unfold code, a new reduced instruction format, a technique to eliminate reference counting operations in interpreters, and a technique to cache local variables of the host language in the stack frame of the executing language. Sullivan et al. [21] partially evaluate sequences of native instructions with respect to the in-memory representation of the program being interpreted by using instrumentation and a dynamic optimizer. Truffle [25] applies AST specialization [23, 27] during interpretation, enabling partial evaluation [5] and hence the execution of highly optimized code.

Finally, related work proposes strategies to reduce the runtime overhead of JIT compilation and hence improve warmup performance. `ShareJIT` [28] is a technique to cache and share JIT-compiled code across processes. Even though this technique improves warmup performance, differently from our technique, it leads to steady-state performance degradation since the compiler cannot emit shared JIT-compiled code that uses absolute addresses. To overcome this limitation, instead of sharing JIT-compiled code, other techniques [1, 8, 15] share profiling data that is used to JIT compile the application either before or during the execution of the application itself. The main limitation of these approaches is that the code for which no profiling data is available is still interpreted.

7 Discussion

In this section, we first discuss use cases of our technique. In particular, we detail how long-running programs can benefit from our technique and how our technique can be employed to speed up specific workloads. Then, we detail ongoing work on native images and how our technique can be used to analyze JIT compilers. Finally, we discuss the generation set used in our experiments and the portability of our technique.

Long-running Programs. For long-running programs where steady-state performance is more relevant than interpreter performance, our technique can be slightly modified to dynamically generate supernodes at runtime. In particular, we compile the generated Java source code and we link the corresponding bytecode at runtime. In this way, we do not separate the supernode-generation, lookup-tree-generation, and supernode-installation steps as in Figure 6, but we perform them altogether after AST creation.

Workload-specific Supernodes. On server machines that frequently execute the same workload, our technique can be employed to generate workload-specific supernodes and so create a dedicated and optimized VM. For instance, we can create workload-specific supernodes for user-provided cloud lambda functions that typically have a short lifetime, which impairs the ability of the system to collect profiling data and JIT-compile the lambda function.

Moreover, we can create workload-specific supernodes for embedded systems with limited hardware resources and power supply. These embedded systems usually cannot JIT compile code due to significant runtime compilation overhead. In both cases, our supernodes may help improve interpreter performance.

Native Images. We conducted our experiments on a Graal.js VM based on OpenJDK, as discussed in Section 5.1. In addition, we are currently investigating the use of our technique for native images [24], i.e., our technique can be employed also when compiling Graal.js to a standalone executable. A Graal.js native image contains the VM internals compiled to machine code ahead-of-time, including the interpreter, the partial evaluator, and the JIT compiler (implemented in Java). At runtime, the partial evaluator and the

JIT compiler compile the input program to machine code, but not the VM internals.

In this setting, our supernodes can still help in reducing the number of polymorphic call sites and thrown control-flow exceptions. When using native images, the lookup-tree generation can take place at build time, the lookup-tree may be serialized and stored in the executable, reducing startup time. The supernode installation process remains unaltered when using native images, i.e., the native image parses the input program, creates the ASTs, and installs the supernodes. Supernodes can be generated before the native image building from the same generation set we used in our experiments.

We plan to conduct an in-depth evaluation of our implementation on native images as part of our future work. Preliminary results show performance improvements similar to those reported in Section 5. The main advantage of native image w.r.t. our technique is that the extra startup overhead associated with supernodes, i.e., class loading, linking, and possibly JIT compilation of supernode classes, becomes a cost of native-image building, but is avoided when the native image is executed. The only remaining sources of overhead are the loading of the lookup-tree (negligible) and supernode installation (which already is a very efficient $O(n)$ algorithm, where n is the number of nodes in an AST).

Analyzing Compiler Optimizations. As shown in Section 5.3, our technique yields speedups also in steady-state performance. One could employ our technique to compare the JIT-emitted machine code executed in the steady-state between the original VM and a VM that uses our supernodes. The JIT-emitted machine code can reveal inefficient code patterns executed in the original VM, and hence shortcomings in the JIT-compiler optimizations or in the heuristics the JIT compiler employs.

Generation Set. In our experiments, we considered a single generation set consisting of the functions of popular web-development packages to generate our supernodes. We note that our supernodes could also be generated using popular Node.js packages or characteristic workloads.

Portability. Our technique could be easily implemented to speed up other Truffle languages such as Ruby, Python, and R. Moreover, the algorithms depicted in Section 4 are interpreter-independent and do not leverage any internal AST-interpreter implementation details, increasing the portability of our technique.

As mentioned in Section 4.2, the Java code generation of the supernodes is the only interpreter-specific part of our technique. Interpreter developers implementing our technique may need to manually write and maintain generation functions for the most common control-flow nodes. We consider the implementation of the code-generation functions an acceptable effort considering the performance gains thanks to supernodes.

8 Concluding Remarks

To conclude, we summarize our contributions, discuss the limitations of our technique, and outline our plans for future research.

Contributions. In this paper we propose a novel technique to generate AST supernodes. Our technique automatically improves the performance of AST interpreters, as it helps reducing typical interpretation overheads related to polymorphic call sites and control-flow-related exception handling. Our technique employs ahead-of-time code generation to automatically create executable supernodes, and runtime installation of matching supernodes.

We implement our technique in the GraalVM JavaScript language runtime (also known as Graal.js), and evaluate our implementation using the well-known web-tooling benchmark suite. Our evaluation shows that supernodes help reducing compilation-time and improve both interpreter and steady-state performance up to a factor of 1.33×, 1.24×, and 1.14×, respectively. Hence, the answers to our research questions RQ1, RQ2, and RQ3 are affirmative.

Our technique is specific to AST interpreters, and is implemented targeting the Truffle language implementation framework of GraalVM. Our technique could be easily ported to other existing Truffle AST interpreters such as, e.g., TruffleRuby [13], FastR [10], or GraalPy [12].

Limitations. The main limitation of our technique is that runtime lookup-tree generation and supernode installation increase VM startup time. We are investigating techniques to serialize the lookup-tree and further reduce the overhead of our technique during VM startup.

Even though our technique reduces the size of the code emitted by the JIT compiler by removing exceptions that model the control flow and several checks in polymorphic call sites, our technique increases the size of the interpreter source code and the VM build time (as shown in Section 5.5 and Section 5.6). For these reasons, when using our technique to build a production VM, it is necessary to find a proper trade-off between code size and performance improvement.

Finally, to generate effective supernodes, it is crucial to select a generation set that contains functions that exercise common control-flow patterns.

Future Work. As part of our future work, in addition to providing an in-depth explanation of the sources of steady-state speedups, we plan to expand our technique to create supernodes that encode non-control-flow nodes as well as supernodes that encode both control-flow and non-control-flow nodes. Moreover, we plan to conduct a large-scale analysis to identify the most frequently used supernodes that may be included in a production build of Graal.js. Finally, we plan to conduct an in-depth evaluation of our implementation on native images.

Acknowledgments

This work has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200020_188688). We thank the VM Research Group at Oracle Labs for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving Virtual Machine Performance Using a Cross-Run Profile Repository. In *OOPSLA*. 297–311.

[2] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (Jun 1973), 370–372.

[3] Stefan Brunthaler. 2010. Efficient Interpretation Using Quickening. In *DLS*. 1–14.

[4] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*. 1–10.

[5] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (Sep 1996), 480–503.

[6] Vladimir Kozlov. 2018. JEP 295: Ahead-of-Time Compilation. https://openjdk.java.net/jeps/295

[7] Octave Larose, Sophie Kaleba, and Stefan Marr. 2022. Less Is More: Merging AST Nodes To Optimize Interpreters. (February 2022). https://kar.kent.ac.uk/93936/

[8] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *ManLang*. 105–118.

[9] Robert Nystrom. 2023. Crafting Interpreters. https://craftinginterpreters.com/

[10] Oracle. 2023. fastr. https://github.com/oracle/fastr

[11] Oracle. 2023. graaljs. https://github.com/oracle/graaljs

[12] Oracle. 2023. graalpython. https://github.com/oracle/graalpython

[13] Oracle. 2023. truffleruby. https://github.com/oracle/truffleruby

[14] Oracle and/or its affiliates. 2021. GraalVM: Native Image. https://www.graalvm.org/22.0/reference-manual/native-image/

[15] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *CGO*. 340–350.

[16] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *POPL*. 322–332.

[17] V8 project authors. 2023. Web Tooling Benchmark. https://github.com/v8/web-tooling-benchmark

[18] Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Eric Seckler, Chen Li, Stefan Brunthaler, Per Larsen, and Michael Franz. 2014. Efficient Hosted Interpreters on the JVM. *ACM Trans. Archit. Code Optim.* 11, 1, Article 9 (Feb 2014), 24 pages.

[19] Manuel Serrano. 2018. JavaScript AOT Compilation. In *DLS*. 50–63.

[20] Manuel Serrano. 2021. Of JavaScript AOT Compilation Performance. *Proc. ACM Program. Lang.* 5, ICFP, Article 70 (aug 2021), 30 pages.

[21] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. 2003. Dynamic Native Optimization of Interpreters. In *IVME*. 50–57.

[22] Sun Microsystems, Inc. 1996. The Java Virtual Machine Specification. https://book.huihoo.com/the-java-virtual-machine-specification/first-edition/Quick.doc.html

[23] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. 1997. Declarative Specialization of Object-Oriented Programs. In *OOPSLA*. 286–300.

[24] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter Bernard Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29.

[25] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *SPLASH*. 13–14.

[26] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Onward!* 187–204.

[27] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. *SIGPLAN Not.* 48, 2 (Oct 2012), 73—82.

[28] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 124 (Oct 2018), 23 pages.

Received 2023-07-14; accepted 2023-09-03