Efficient Dynamic Analysis for Node.js

Haiyang Sun
Università della Svizzera italiana (USI)
Switzerland
haiyang.sun@usi.ch

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

Christian Humer
Oracle Labs
Switzerland
christian.humer@oracle.com

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

Abstract

Due to its popularity, there is an urgent need for dynamic program-analysis tools for Node.js, helping developers find bugs, performance bottlenecks, and bad coding practices. Frameworks based on code-level instrumentation enable dynamic analyses close to program semantics and are more flexible than Node.js built-in profiling tools. However, existing code-level instrumentation frameworks for JavaScript suffer from enormous overheads and difficulties in instrumenting the built-in module library of Node.js. In this paper, we introduce a new dynamic analysis framework for JavaScript and Node.js called NodeProf. While offering similar flexibility as code-level instrumentation frameworks, NodeProf significantly improves analysis performance while ensuring comprehensive code coverage. NodeProf supports runtime (de)activation of analyses and incurs zero overhead when no analysis is active. NodeProf is based on dynamic instrumentation of the JavaScript runtime and leverages automatic partial evaluation to generate efficient machine code. In addition, NodeProf makes use of the language interoperability provided by the runtime and thus allows dynamic analyses to be written in Java and JavaScript with compatibility to Jalangi, a state-of-the-art code-level JavaScript instrumentation framework. Our experiments show that the peak performance of running the same dynamic analyses using NodeProf can be up to three orders of magnitude faster than Jalangi.

CCS Concepts • **Software and its engineering** → **Runtime environments**; *Dynamic compilers*;

Keywords JavaScript, dynamic analysis, instrumentation

CC'18, February 24–25, 2018, Vienna, Austria
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179527

ACM Reference Format:
Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.js. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179527

1 Introduction

JavaScript is the most popular programming language on the Web. Thanks to Node.js [25], it has been widely used for server-side applications, too. There are several reasons behind the success of JavaScript and Node.js. Amongst others, Node.js offers a convenient trade-off between developer productivity and application performance, allowing developers to quickly build applications ready for production using thousands of modules in the NPM package repository [19].

One of the main drawbacks of Node.js, however, is the lack of efficient tools to analyze program behavior and performance. Dynamic language runtimes such as JavaScript engines (e.g., V8 [6]) do not support code instrumentation at runtime. Nevertheless, the ability to instrument applications is a key requirement for modern language runtimes, as it enables essential functionalities such as profiling [8], analyses to locate bad coding practice [9], data-race detection [5, 22], memory-utilization profiling [14], etc. The lack of instrumentation capabilities is particularly stringent in Node.js, where applications typically make use of third-party modules imported via NPM. As applications grow in the number of external packages they import, it is crucial to be able to dynamically analyze NPM modules as well, to ensure that the application does not suffer from penalties introduced by third-party code.

The urgent need for dynamic analysis tools in Node.js has motivated several research and industrial projects aimed at providing extensible support for the analysis of Node.js applications. A notable example of such a framework is Samsung's Jalangi [24], a state-of-art dynamic analysis framework for JavaScript based on fine-grained runtime event profiling. Jalangi heavily instruments the application source code and provides developers with several hooks over JavaScript execution events (e.g., property reads or function calls) that can be leveraged to develop complex runtime analyses such as taint tracking [18], non-contiguous object accesses tracing, etc. Despite being popular, Jalangi incurs high performance-overheads: typical dynamic analyses to identify bad coding practices can slow down an application by up to 3 orders of magnitude. All existing frameworks based on Jalangi [3, 4, 24] suffer from such an excessive overhead, which can be a major limitation when running Node.js server-side applications that need high throughput and fast response time. In addition, for analyses requiring full tracing of the application such as data flow analysis, these frameworks fall short in instrumenting the built-in modules library of Node.js. Moreover, dynamic analyses using code-level instrumentation are *irreversible*, meaning that the analyzed application pays the performance overhead for its entire execution life.

In this paper, we introduce a new instrumentation framework for JavaScript and Node.js applications called NodeProf that overcomes the limitations of existing dynamic analysis frameworks like Jalangi. NodeProf relies on efficient runtime instrumentation that leverages VM-internal components such as the JavaScript just-in-time (JIT) compiler to deliver high performance. NodeProf is implemented on top of Graal.js [28], a Java-based Node.js engine included in the GraalVM [10] polyglot language runtime, allowing analyses to be implemented using Java. In addition, NodeProf also provides language interoperability with JavaScript, thus giving NodeProf compatibility with analyses developed using Jalangi. Both for analyses written in Java or in JavaScript, respectively, NodeProf can be up to three orders of magnitude faster than Jalangi.

NodeProf does not rely on code-level instrumentation, and is fully transparent to the instrumented application. This allows analyses to be *hot plugged* at runtime, incurring zero overhead when they are not enabled. Zero-overhead hot plugging is a key feature for monitoring server-side applications, and is particularly convenient for cloud-based Node.js deployments.

This paper makes the following contributions:

1. We introduce NodeProf, a practical solution for the dynamic analysis of Node.js server applications offering significantly better runtime performance than code-level instrumentation tools.
2. NodeProf enables the dynamic instrumentation of the entire JavaScript source code of an application, including the Node.js built-in library and NPM modules. To the best of our knowledge, NodeProf is the only existing dynamic analysis tool for Node.js that is capable of instrumenting all JavaScript code used by an application, as other frameworks cannot instrument dynamically-loaded or built-in modules.
3. Original Jalangi analyses written in JavaScript can be executed on NodeProf without modifications. We also support Java as an alternative language for developing dynamic analyses.
4. NodeProf is hot-pluggable. Instrumentations can be selectively enabled at runtime, and incur no overhead when disabled.

This paper is structured as follows: Section 2 provides background information on Jalangi, and on the technology underlying NodeProf. Section 3 describes the motivation of NodeProf. Section 4 shows the design and overview of NodeProf and Section 5 further introduces the programming models provided. Section 6 explains in detail the algorithm used in NodeProf to support event profiling. Section 7 evaluates the performance of NodeProf. Section 8 discussed related work and Section 9 concludes.

2 Background: Jalangi and GraalVM

In this section, we discuss how dynamic analyses can be developed with Jalangi, a state-of-the-art code-level instrumentation tool and GraalVM which NodeProf is built on.

2.1 Jalangi

Initially developed at UC Berkeley and now maintained by Samsung, Jalangi [24] is based on code-level instrumentation to provide fine-grained execution *hooks* that developers can leverage to intercept runtime events. Examples of such events include object-level access operations (e.g., property reads on a given object), function calls, binary and unary operations, as well as control flow instructions.[1]

By tracking specific runtime events during the execution of a JavaScript application, it is possible to implement program analyses capable of identifying potential performance bottlenecks. A notable class of such analyses is *JIT-compiler profiling* (JITProf [8]) to identify bad coding practices that could lead to *inefficient* code generation by the JavaScript JIT compiler. One example of such bad coding practices is *non-contiguous* access to elements of a JavaScript array: as soon as an array is accessed in a non-contiguous way (e.g., writing to a non-existing index), its in-memory runtime representation could change from a dense, uniform, data structure (e.g., an array of small integers) to a sparse, less-efficient, data structure (e.g., a hash map). Arrays accessed using such non-contiguous patterns often miss the opportunity to be optimized by the JavaScript JIT compiler.

Detecting such inefficient access patterns requires tracking all array writes. The Jalangi code introduced in JITProf [8] for the analysis is depicted in Figure 1. Jalangi analyses consist of a set of event callbacks that map to specific runtime events. The runtime ensures that when such events happens, the corresponding callback will be invoked.

[1]A full list of the events Jalangi supports can be found at https://github.com/Samsung/jalangi2/blob/master/src/js/runtime/analysisCallbackTemplate.js

```
function NonContiguousArray() {
  /* some code omitted here */
  var db = new RuntimeDB(); // a map to store results
  function ifWriteOutsideArrBound(base,offset,iid) {
    if (base.length < offset) {
      //fetch and increment the counter from db for
           the source code region identified by iid
      db.addCountByIndexArr(['JIT-checker',
           'non-cont-array',
           sandbox.getGlobalIID(iid)]);
    } }
  // callback called before a property write.
  this.putFieldPre = function (iid, base, offset,
       val, isComputed, isOpAssign) {
    if (base !== null && base !== undefined) {
      if (Utils.isArr(base) &&
           Utils.isNormalNumber(offset)) {
        ifWriteOutsideArrBound(base, offset, iid);
      } } };
}
```

Figure 1. Non-contiguous array-access analysis in Jalangi.

2.2 Graal.js, Truffle, and GraalVM

As discussed, NodeProf does not rely on code-level instrumentation. Rather, it is based on a tight integration with the language execution runtime that enables dynamic code generation and optimization. Specifically, NodeProf is implemented using the Truffle [29] framework, and runs on top of Graal.js [28], a Node.js-compatible JavaScript execution engine that relies on the Graal dynamic compiler. Truffle, Graal.js, and the Graal compiler are packed in the GraalVM [10], a polyglot language runtime containing a selection of language execution engines implemented using Truffle.[2].

Truffle is a language implementation framework for the development of high-performance language runtime systems. A Truffle-based language runtime is implemented in the form of a self-optimizing Abstract-Syntax-Tree (AST) interpreter [30]: like with typical AST interpreters, each node in the tree corresponds to a specific runtime operation (e.g., reading a property from an object, performing a function call, etc.), which Truffle can optimize by means of partial evaluation [17]. At runtime, each AST node eagerly replaces itself with a specialized version that relies on runtime assumptions, leading to better performance. For example, node rewriting specializes the AST nodes performing a property lookup operation for the actual object types used by the application, and relies on an inline cache [12] optimized for such type. Truffle's self-optimization via node rewriting can result in the elision of unnecessary generality, e.g., boxing and complex dynamic dispatch mechanisms.

Compilation by means of automatic partial evaluation is performed by the Graal dynamic compiler [10]. Graal compiles AST nodes to machine code when the execution profile reaches a certain threshold. In case of speculation failures, Graal performs deoptimization [27], replacing invalidated machine code with less-optimized code.

The NodeProf dynamic analysis framework has been designed to target Truffle-based language runtimes, with an initial focus on Graal.js, a high-performance JavaScript runtime fully compatible with Node.js, and performance in line with those of other JavaScript engines such as Google's V8 [6].

3 Motivation

3.1 Limitations of code-level instrumentation

Due to the lack of VM-level support, code-level instrumentation is one of the few options to analyze JavaScript applications. Jalangi's fine-grained execution events are collected by instrumenting each instruction of a JavaScript application via source-code-level rewriting. This means that every single code location in the original application that corresponds to a profiling event needs to be replaced with function calls to perform the actual profiling.

Code-level instrumentation may introduce enormous overheads. For a typical analysis [8], the slowdown can be up to 3 orders of magnitude. Figure 3 gives some hints as to why the runtime overhead for frameworks like Jalangi can be high: to track each profiling event, a lot of extra code needs to be injected even for a simple function like the one in Figure 2. This voids many optimization opportunities, as all language constructs (e.g., binary comparison operators, property reads, etc.) will be converted to function calls, thus preventing the JIT compiler from applying common optimizations such as canonicalization, inline caching [12], escape analysis, and so on.

Besides the performance penalty, there are several other limitations for code-level instrumentation tools. First, the instrumentation is not transparent to the application. As a result, the instrumented application may yield wrong results when depending on dynamic information such as e.g. the call stack. Secondly, they need to modify the source code files and cannot be applied to the Node.js built-in library. Another limitation is that all modifications to the original sources are *irreversible*: code-level instrumentation is done before execution, and will therefore affect application performance for its entire execution.

3.2 Opportunities to use the GraalVM

Node.js applications in Graal.js are executed by means of AST interpretation. During interpretation, the AST of a JavaScript application self-specializes itself by replacing its nodes with

[2]GraalVM currently supports JavaScript, Ruby, R, Python, LLVM, and all languages that compile to Java bytecode such as Java, Scala, and Kotlin

```
function f(a,b,c) {
  return a > b[c];
}
```

Figure 2. A simple JavaScript function

```
J$.iids = {"9":[2,10,2,11], //...Source code mapping
function f(a, b, c) {
    try {
      // Arguments handling
      J$.Fe(57, arguments.callee, this, arguments);
      arguments = J$.N(65, 'arguments', arguments, 4);
      // Variable reads tracking
      a = J$.N(73, 'a', a, 4);
      b = J$.N(81, 'b', b, 4);
      c = J$.N(89, 'c', c, 4);
      // Binary operator tracking
      return J$.X1(49, J$.Rt(41, J$.B(10, '<',
           J$.R(9, 'a', a, 0), J$.G(33, J$.R(17, 'b',
           b, 0), J$.R(25, 'c', c, 0), 4), 0)));
    } catch (J$e) { J$.Ex(121, J$e); }
}
```

Figure 3. Code-level instrumentation applied by Jalangi to the function in Figure 2 to enable runtime event tracking.

more optimized versions; such optimized ASTs are later automatically compiled to machine code by the Graal dynamic compiler. Combining event tracking callbacks of a dynamic analysis directly with AST nodes would allow the optimizations done by Truffle during partial evaluation. For example, the AST node for a JavaScript property lookup operation can be executed together with a NodeProf event callback. In this way, the machine code produced for a NodeProf dynamic analysis will be compiled by the Graal compiler in the same compilation unit of the JavaScript operation. This approach can lead to significantly reduced analysis overhead.

4 NodeProf

In this section, we give an overview of the design of NodeProf. Section 4.1 introduces how NodeProf applies instrumentation at the level of AST nodes to generate various profiling events. Section 4.2 explains how to efficiently pass dynamic data among nodes needed for profiling events.

4.1 Dynamic AST-level Instrumentation

To deal with the limitations of code-level instrumentation as discussed in Section 3.1, NodeProf dynamically instruments the AST of the target application. One notable advantage of AST-level instrumentation is that it preserves the original structure of the application code. The AST intermediate form allows one to map specific source-code regions to nodes

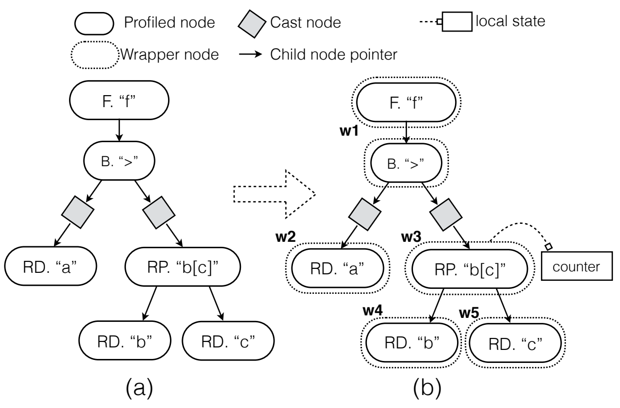


Figure 4. AST instrumentation for the example code (A "local" counter is accessible in the wrapper)

in the tree, ensuring that the AST structure matches that of the applications being executed. This makes dynamic analyses of NodeProf as close to the application semantics as those achieved by code-level instrumentation. Thanks to this key property, it is possible to *inject* executable code into a running AST interpreter by simply replacing some of its nodes with *wrapper* nodes that perform additional operations before or after delegating execution to the original node they are wrapping [23, 26].

An example of AST-node wrapping is depicted in Figure 4(a), where the AST representation of the code in Figure 2 is presented. The sources of the JavaScript function ``f'' have been parsed into an AST, including the function root node (the ``f'' node), a binary node for the ``>'' operator, three local-variable-read nodes, one property-read node, and two helper nodes (represented using diamonds used to cast node execution results into integers for comparison).[3]

AST-node wrapping enables a convenient separation of concerns, as nodes can be wrapped transparently to applications, requiring no changes to the original source code. It also enables efficient execution, as instrumentation nodes can potentially benefit from automatic compilation via partial evaluation in the same way as all other JavaScript AST nodes. Moreover, instrumentation nodes can have *local state* that is relative to a specific source-code location, uniquely mapped by the structure of the AST. The ability to bind state to code locations can be conveniently used by analyses that need to keep track of events happening at a specific location, such as described in the example of Figure 1, where the analysis needs to increment a counter for every non-contiguous array write. With code-level instrumentation frameworks (like Jalangi), such a counter would need to be looked up from a hash map at runtime. With NodeProf, the counter can

[3]The ECMA JavaScript language specification prescribes that each value must be converted to its integer value before executing the ``>'' operator. Therefore, the AST has to convert both values before executing the binary operation.

be conveniently stored in the private state of the AST wrapper node, thus enabling constant-time access of the counter during execution of the instrumented code. This is depicted in Figure 4(b), where `counter` is the local state used by `w3`, i.e., the wrapper node for the property read node.

Another advantage of the AST-level instrumentation of NodeProf is that the instrumentation is done dynamically at runtime and can be applied to any source code. Compared to code-level instrumentation which needs to instrument and replace every source file prior to execution, NodeProf is more convenient because no modification of the original source code is needed , and achieves higher code coverage because also the built-in library can be instrumented.

NodeProf generates various profiling events at different AST nodes called *event nodes*. Every event node needs to get the *dynamic data* necessary for the profiling event. For example, a property-read event needs the name of the property being read and the object instance being accessed. Some dynamic data can be directly retrieved from the event node itself such as the operator for a binary event, or from the runtime stack such as the values of $< this, function, arguments >$ needed for function calls ; we call such directly available data *local values*. Event nodes also need to access other data that is produced by other nodes; we call these values *intermediate values*. The details of the events supported in NodeProf and the dynamic data available for each event node are listed in Table 1. While the retrieval of local values is trivial, handling intermediate values can result in significant overhead as such values may need to be stored in temporary variables. In NodeProf we optimize the access to such values by providing a specialized solution detailed below.

4.2 Intermediate-value Resolution

Since intermediate values are made available only to their direct parent, AST wrapper nodes may not be able to access their values directly, and therefore may need to store those values in a temporary location (with some obvious overhead). In the example in Figure 4, the binary event node for the operator ''>'' can only access the results of its two children, which are internal *cast* nodes used to convert any JavaScript type to an integer value before comparison. Such conversion is required by the JavaScript semantics; an event handler to track the binary operator, however, would need to access the JavaScript values *before* they are converted to numbers.

It is therefore crucial to be able to make such values available to the wrapper node *before* they are converted. NodeProf leverages the notion of *frame virtualization* in Truffle [29] to efficiently store and retrieve intermediate values from the descendant node and make them available to the event node. Truffle frame virtualization is an optimization technique enabled by the Graal compiler, and is commonly used in Truffle-based language runtimes to optimize the performance of local variable accesses. With frame virtualization,

Table 1. Profiling events and the dynamic data needed (*) stands for <this, function, arguments>

Profiling Event	Local Values	Intermediate Values
Read (R)	identifier	/
Write (W)	identifier	value
ReadProperty (RP)	/	target, property
WriteProperty (WP)	/	target, property, value
Invoke (I)	(*)	/
New (N)	(*)	/
FunctionRoot (F)	(*)	/
Literal (L)	/	/
Binary (B)	operator	left, right
Conditional (C)	/	condition value

local variables are treated as entries of a map-like data structure. The Graal compiler has special knowledge of this data structure, and performs several optimizations such as inlining and escape analysis over its entries, which very often result in the allocation of such variables to CPU registries. In NodeProf we leverage a similar optimization to store the intermediate values produced by AST nodes into some *virtual slots* of the same data structure and make them available to the proper event nodes. In this way, NodeProf can efficiently resolve intermediate values. This technique operates as follows:

At parsing time: different event nodes in the AST are tagged to be responsible to generate the profiling events listed in the first column of Table 1. In addition, NodeProf allocates one virtual slot for every node providing an intermediate value to an event node. The fewer virtual slots allocated, the more likely they can be stored in CPU registers for fast access. In Section 6, we will present the detailed algorithm used to achieve an optimal slot allocation.

At execution time: the AST nodes execute in a depth-first order from left to right. So the nodes providing the intermediate values will be executed before the event node and store the intermediate values in the allocated virtual slots. Afterwards, these intermediate values can be fetched at the event node from the slots.

5 Programming Model

In this section, we introduce the programming model of NodeProf for writing dynamic analyses. Sections 5.1 and 5.2 present NodeProf's APIs in Java and JavaScript, respectively.

5.1 Profiling API in Java

As introduced, NodeProf analyses can be implemented using Java. The benefit of writing analyses in Java is that we can directly operate on AST nodes, and fully make use of all optimizations enabled by the Truffle framework. Generally speaking, the Java instrumentation API could also support other languages based on GraalVM. However, the focus of this paper is on JavaScript and Node.js.

```
public class NonContiguousArray extends NodeProfAnalysis {
  @Override
  public void setupCallbacks() {
    this.onWriteProperty(new
         AnalysisFactory<WritePropertyNode>() {
      public WritePropertyNode create(EventContext context){
        return new WritePropertyNode(context) {
          @Child IsArrayNode isArrayNode;
          @Child ArrayIndexNode indexNode;
          @Child GetArraySizeNode arraySizeNode;
          final Report report = Report.get(getSourceID());
          public void pre(VirtualFrame frame,
              Object base, Object offset, Object val) {
            if(!isArrayNode.executeIsArray(base))
              return; //continue only when base is an array
            //get the integer value out of offset
            int idx = arrayIndexNode.executeIndex(offset);
            //(e.g., 1, 1.0 and "1" are valid index)
            if (idx >= 0) {
              //read the array size
              int arrSize = arraySizeNode.executeSize(base);
              if (idx > arrSize) {
                //increment the local counter
                report.increCounter();
              } } } }; } }, SourceFilter.appOnly);}
}
```

Figure 5. Using the Java API for the non-contiguous array-access analysis

An example of a Java-based dynamic analysis in NodeProf is depicted in Figure 5, where the non-contiguous-array example of Section 2 is presented. Every dynamic analysis using the Java API is represented as a subtype of class *NodeProfAnalysis* (i.e., *NonContiguousArray*, in this case). Different profiling events are handled with different callback nodes. In this example, one *WritePropertyNode* (WP) will be created for every write property event, and attached to the wrapper of the corresponding property write node in the AST. In order to setup these callbacks, a *NodeProfAnalysis* needs to register the factories to create the callback nodes as shown in method *setupCallbacks*: the callback node for WP events is registered with the method *onWriteProperty*. The callback node will be attached to the wrapper node at instrumentation time, and will be invoked with the correct event-specific runtime arguments. Such arguments are the intermediate values retrieved from the slots computed from the Minimum Indexing Algorithm described in Section 6. A set of utility internal nodes is also provided to enable interaction with JavaScript to fulfill the analysis logic. A few *@Child* nodes are defined in this example: we use *IsArrayNode* to check if a JavaScript object is an array, *ArrayIndexNode* to convert a property value into an array index, and *GetArraySizeNode* to get the size of a JavaScript array. The utility nodes bridge the gap between JavaScript and Java, to ease the development of dynamic analysis using the Java API.

Being able to closely interact with the AST, NodeProf can fully benefit from optimizations like inlining, escape analysis, and polymorphic inline caches. AST-node-local data can be stored in the callback node created for every wrapper. In this example, we keep a *Report* reference in the callback node and benefit from constant-time access as discussed in Section 4.1. Another optimization enabled by the Java API is caching. For example, *isArrayNode* in Figure 5 is used to check if the *base* object is an array. Without caching, we would need to check whether *base* is an array for every property write event. With a cache of the object reference, we first do a cheap check whether the object remains the same, and only do the expensive array check in the case of a cache miss. Such kind of optimizations are extremly useful in loops or other hot code regions where *base* is more likely unchanged for some time. These optimizations are not available in Jalangi.

Another advantage brought by NodeProf compared to code-level instrumentation is the dynamic management of different source-code regions. In the example, reports are created for every source-code region where a non-contiguous array access is found. To this end, an ID needs to be assigned to identify each code region. Jalangi achieves this by hard-coding a map of code locations in each instrumented source-code file. The size of this map can be several times bigger than the original source code itself. In NodeProf, we maintain a source code map in-memory and on demand, i.e., we only keep track of source-code regions which are actually referenced by the dynamic analysis at runtime. This process is encapsulated in method *getSourceID()*, and allows NodeProf to map execution events to source code locations only for locations that are actually executed.

Finally, different analyses may need to instrument different scopes of source code. NodeProf allows for flexibly configuring which source files will be instrumented. Developers can choose to instrument any of the application code, the built-in library, or selected NPM modules. In the example, the option *appOnly* is used to limit the instrumentation scope to the application code.

5.2 Compatibility with Jalangi

As discussed, NodeProf is compatible with most Jalangi features. In other words, many existing analyses written in Jalangi can be executed without any modifications. To achieve this, NodeProf implements a layer on top of its Java API that can *expose* instrumentation events and callbacks to Jalangi's JavaScript-based hooks.

The compatibility is obtained by inlining a function call node (i.e., a JavaScript call node) into the AST instrumentation wrapper as illustrated in Figure 6. We slightly modify Jalangi's library, such that NodeProf is aware of all Jalangi analyses that are registered. When an event node is executed, its wrapper node performs a direct call to the corresponding callback defined in the Jalangi analysis. A special Truffle API called *DirectCallNode* is used to ensure that the call is always

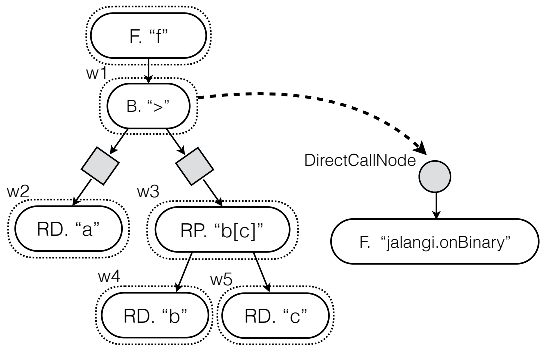


Figure 6. Call from wrapper to Jalangi via DirectCallNode

inlined. This API can pass values from the Java space to the JavaScript environment, transparently, without introducing boxing or conversion overheads. We currently support the major features of Jalangi including the callbacks listed in Table 1, and fully support the analyses we use for the evaluation. In contrast to Jalangi, we do not support changing the return value of the instrumented code, because we focus only on dynamic analyses that do not alter the semantics of the observed program.

6 Minimum Indexing Algorithm

In this section, we explain the details of a Minimum Indexing Algorithm (MIA) used in NodeProf to allocate slots for intermediate values. Section 6.1 defines the algorithm and Section 6.2 gives a proof that MIA computes a correct and optimal solution.

6.1 Definition

Every event node must get the correct intermediate values. This poses two fundamental requirements for the correctness of a slot allocation algorithm:

1. The nodes producing intermediate values for an event node should be known and unchanged.
2. The intermediate values stored in the slots should never be overwritten until they are retrieved by the event node needing them.

We distinguish the nodes in two categories:

1. **External nodes** that have a source-code mapping. Not all external nodes are event nodes, e.g., a block representing a sequence of statements is external but does not produce any profiling event. However, all event nodes and nodes providing intermediate values must be external.
2. **Internal nodes** which are not bound to any source code. For example, the *cast* nodes of a binary node in the previous example. Internal nodes are usually helper nodes in the language implementation.

The first requirement can be satisfied for the fact that: for every event node N in Graal.js, the nodes providing the intermediate values for N are all the external nodes whose first external ancestor node is N. We define this fixed set of nodes for N as the intermediate-value-node set: *ivnSet(N)*.

Assuming there are an infinite number of available slots, allocating a unique slot for every intermediate value would be a trivial solution to the second requirement. However, to benefit from storing slots in CPU registers, NodeProf aims at minimizing the number of used slots.

As a result, a minimum indexing approach is described in the procedure *AllocateSlots* in Algorithm 1. Assume every slot is identified by an integer index starting from 0. *AllocateSlots* takes two input parameter: $SubTree(N)$ a subtree of an AST whose root node is N, and $startIndex$ which is the minimum ID free to allocate. *AllocateSlots* allocates an incremental unique ID, starting from $startIndex$, for each node C_i from $ivnSet(N)$ and recursively allocates slots for subtrees whose root is C_i (i is the iteration number, the nodes are traversed in the AST execution order which is post-order from left to right).

Algorithm 1 Algorithm to allocate intermediate-value slots

```
1:  procedure AllocateSlots
2:  input:
3:      Subtree N, an AST subtree whose root is node N
4:      Integer startIndex, the minimum available index
5:  output:
6:      Integer result, the number of intermediate values found for N
7:  body:
8:    $ivnSet(N) \leftarrow$ external nodes whose first external ancestor is N
9:    foreach $C_i \in ivnSet(N)$ (traversal in AST execution order) do
10:       if N expects intermediate values then
11:           $AllocateSlots(C_i, startIndex + i)$
12:           $C_i.setSlotIndex(startIndex + i)$
13:       else
14:           $AllocateSlots(C_i, startIndex)$
15:   end foreach
16:   return i
```

Our Minimum Indexing Algorithm is applied to all root nodes representing JavaScript functions and the slot allocation for the previous example is shown in Figure 7. In the figure, we can see that three slots are needed to store intermediate values. In this case, three slots are the best solution because two slots have to be reserved at the time when `"b[c]"` is executed, while slot 0 is still occupied by the unfinished execution of the binary node `">"`.

6.2 Proof

We define the external depth e-depth of $SubTree(N)$ as the most number of external nodes included between N and any of its leaf nodes. It is clear that if the e-depth of $SubTree(N)$ is $k + 1$, all the subtrees whose root is in $ivnSet(N)$ are of e-depth k. Then we can prove by induction that our algorithm provides a correct slot allocation and the number of allocated slots is minimal [2]:

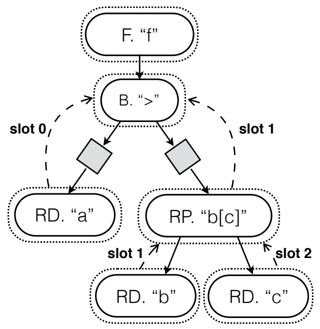


Figure 7. Slot assignment for *a=b[c]* using MIA

1. When applying *AllocateSlots* to any subtree whose *e*-depth is one, meaning only node N is external, there will be no intermediate values needed and our algorithm gives the optimal result.
2. If *AllocateSlots* gives the correct solution for all subtrees whose *e*-depth is k, it also holds for any subtree whose *e*-depth is $k + 1$: if the root does not need any intermediate value, it is obvious that AssignSlots return the correct result 0; Otherwise, each intermediate value node from $ivnSet(N)$ will get an incremental ID in the for-each loop. Since the traversal is in AST execution order, the slots assigned for C_i in the i-th iteration are free to be used in the later iteration. So there will be no conflicts for the slots allocated for N. So the algorithm is correct.
3. Since the slot allocation for every C_i is already optimal, our algorithm will always give the optimal result as below:

$$Optimal(N) = \min_{\forall C_i \in ivnSet(N)} \{i + Optimal(C_i)\} \qquad (1)$$

By induction, for any subtree of the input AST in any *e*-depth (including the input AST itself), our algorithm gives the optimal solution.

7 Evaluation

NodeProf has been designed for low overhead. In this section, we first compare NodeProf with Jalangi, a state-of-the-art code-level instrumentation framework. Then, we evaluate the performance of NodeProf's hot-plugging feature on a real-world Node.js application.

7.1 Experimental Setup

All measurements were performed on an Intel(R) Core(TM)2 Quad CPU (Q9650) with 2 physical cores (4 virtual cores) running at 3.0 GHz, 4 GByte main memory, running Ubuntu Server release 16.04 (kernel version 4.4.0-59-generic). NodeProf runs on Graal.js v0.18 OTN release [10] and compares against Jalangi [13] (last commit on April 5, 2017).

We evaluated six existing analyses originally developed for Jalangi, namely:

Table 2. Analyses and events involved in the evaluation.

Analysis	Profiling events involved
Typed arrays	RP, WP, L, I, N
Non-contiguous arrays	WP
Objects per allocation site	L, I, N
Branch coverage	C
Undefined offsets	RP, WP
Concat Undefined to String	B

1. Non-contiguous arrays (as described in Section 2): reports non-contiguous writes to arrays that may cause expensive deoptimizations.
2. Typed arrays: tracks non-numeric stores into numeric arrays .
3. Objects per allocation site: reports the number of object allocations per allocation site.
4. Branch coverage: profiles branch execution coverage.
5. Undefined offsets: finds any array read/write access to an "undefined" offset.
6. Concat *Undefined* to String: finds cases where one operand of "+" is undefined and the result is a string.

The analyses exercise different event hooks in NodeProf, as summarized in Table 2. The original implementation and description for each analysis can be found in Jalangi-related publications [8, 9], and on GitHub [13, 15].

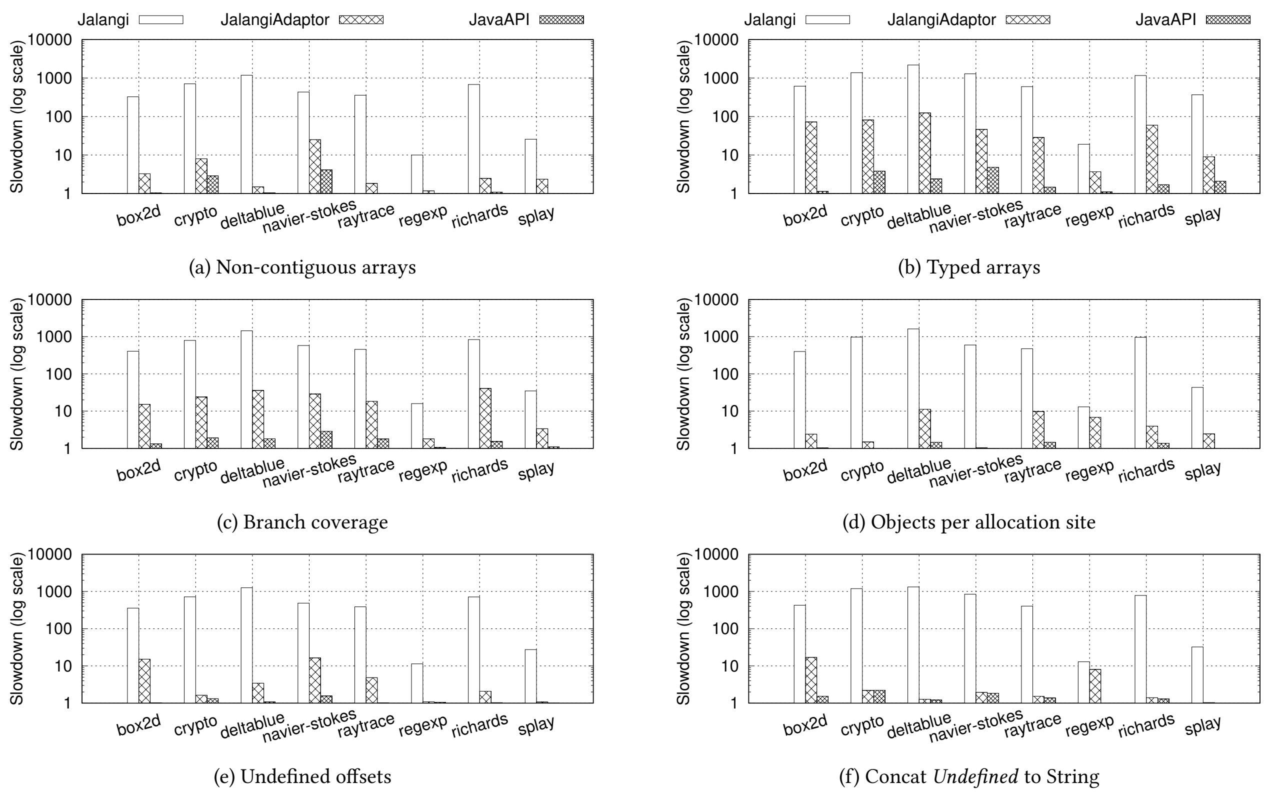
7.2 Octane Benchmarks

We first evaluate the cost of running a Jalangi analysis on the popular JavaScript Octane [20] benchmark suite. The goal of this evaluation is to measure the performance improvement of NodeProf wrt. Jalangi. We compute the slowdown of the peak performance as collected by the (unmodified) Graal.js Octane benchmarking harness (included in the GraalVM distribution [10]). The harness reports the average over 100 benchmark runs, collected after 100 warm-up iterations. The warm-up phase is used to let the engine reach a steady state, and is not counted for the final grade.

Since Jalangi cannot instrument the Node.js built-in library, for fairness, we configure NodeProf to instrument the same parts of the code as Jalangi.

The performance of Jalangi, NodeProf's JalangiAdaptor, and its JavaAPI are shown in Figure 8. We report the slowdown relative to the same baseline which is running the benchmarks on Graal.js without any instrumentation enabled. Once its language runtime reaches a steady state, the performance of Graal.js [28] is on par with other leading JavaScript engines such as V8.

From the result, we can see that Jalangi always suffers from excessive overhead. E.g., applying *Typed Arrays* for the *deltablue* benchmark results in a slowdown of 3000x. Conversely, the performance overhead is greatly reduced with NodeProf: in most cases, NodeProf's JalangiAdaptor is one or two orders of magnitude faster than Jalangi, and



(a) Non-contiguous arrays

(b) Typed arrays

(c) Branch coverage

(d) Objects per allocation site

(e) Undefined offsets

(f) Concat *Undefined* to String

Figure 8. Slowdown factors (wrt. the uninstrumented Graal.js) for Jalangi, JalangiAdaptor, and JavaAPI running the same analyses for Octane.

the Java API is two to three orders of magnitude faster than Jalangi. The advantage can be very significant in some cases where the Java API achieves almost no overhead.

This result is expected, as analyses written in JavaScript (i.e., Jalangi) still pay the price of the dynamic nature of the JavaScript semantics, whereas analyses expressed in the Java API can be further optimized by the Graal compiler. Moreover, as discussed in Section 5, the Java API of NodeProf enables optimized access to AST-local data structures (e.g., node-local counters) as well as caching of runtime references. The effect of such optimizations can be seen e.g., in Figure 8 (a), where the Java API has almost zero overhead for *raytrace*. In this case, the benchmark uses only a very limited number of array objects. Therefore, the number of events detected by the analysis (i.e., non-contiguous array accesses) is minimal: while Jalangi still instruments the entire source code –even when events are never detected–, NodeProf’s Java API allows one to efficiently check for object types, leading to almost zero overhead when very few objects match the type considered relevant for the analysis (i.e., array, in this case).

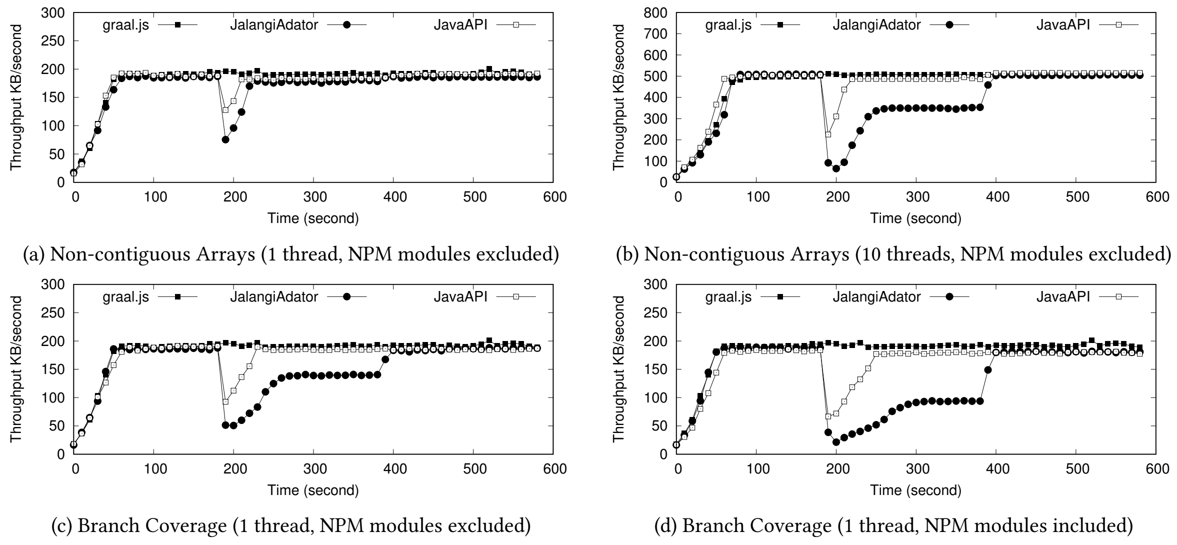
7.3 Hot-plugging for Acme Air

NodeProf makes profiling practical for long-running server-side Node.js applications thanks to its hot-plugging feature. In this section, we illustrate the performance impact of this feature on a popular server-side Node.js benchmark called Acme Air [1]. Acme Air is a complex benchmark that simulates a flight booking system whose server-side backend is implemented using Node.js. Performance is measured using a JMeter [16] testing suite simulating realistic workloads, collecting throughput and latency for the web server.

We deployed Acme Air using different machines for the workload generator and the web server, to reduce measurement perturbations. The two server machines are connected with a network bandwidth of 100 Mb/s.

We report the performance impact of enabling (and disabling) two dynamic analyses (Non-contiguous Arrays and Branch Coverage) on a Graal.js web server running Acme Air. The benchmark was executed changing the number of workload threads (1 and 10) as well as the number of Node.js built-in library modules to instrument. The results are depicted in Figure 9. We omit the performance numbers for Jalangi on this benchmark, as the Jalangi framework failed to instrument all of the 1833 JavaScript files used by the NPM modules in the benchmark. On the contrary, NodeProf is able to instrument the whole application: application code (App), the NPM modules (NPM) it depends on, and the Node.js built-in library (Built-in).

To demonstrate NodeProf’s hot-plugging feature, we measure the impact of an analysis on the throughput of the



(a) Non-contiguous Arrays (1 thread, NPM modules excluded)
(b) Non-contiguous Arrays (10 threads, NPM modules excluded)
(c) Branch Coverage (1 thread, NPM modules excluded)
(d) Branch Coverage (1 thread, NPM modules included)

Figure 9. Hot plugging feature with the Acme Air benchmark and different levels of instrumentation for two dynamic analyses
.

running server. To this end, we first execute the Graal.js server without instrumentation. Then, after 200 seconds, we enable a NodeProf analysis, and we let it collect profiling data for 200 additional seconds. Finally, after 400 seconds, we disable the analysis. The throughputs along with time are shown in Figure 9. The experiment reveals the following:

1. When the profiler is enabled, a performance drop is detected (at around 200 s). This is because the newly enabled analysis code has not been optimized by the Graal.js JIT compiler. Still, the server is running and accepts incoming requests. After a while, the performance stablizes. The throughput of the web server is lower than before, as now the analysis is enabled.
2. After disabling the analysis, the throughput of the system recovers as before, showing that NodeProf can be hot-plugged with no overhead when disabled.

8 Related Work

Prevailing approaches for program analysis written in dynamic languages such as JavaScript can be divided into two main categories: frameworks based on code-level instrumentation, and frameworks relying on VM-level support.

For what concerns VM-level support, common JavaScript engines (e.g., V8 [6]) provide a basic built-in profiling module that can collect minimal profiling data such as the number of method executions [21]. Such built-in solutions offer low overhead, but are not extensible, and enable only very limited analyses. Typically, developers can only diagnose problems using performance logs, and have no way to customize analyses. As a consequence, many analyses—including those described in this paper—cannot be expressed using built-in profilers. Moreover, built-in profilers for Node.js cannot be dynamically enabled/disabled. In contrast, NodeProf offers the flexibility to define a wide variety of analyses for Node.js, with the support for hot-plugging.

Jalangi [24] has already been discussed in this paper, and is the most popular code-level instrumentation framework for JavaScript. Several other examples using code-level instrumentation exist [3, 4], and many analyses [8, 9, 14] have been built on top of them. As we have shown, the overhead of code-level instrumentation can be excessive. Sampling can be used [24] to mitigate this problem, but sampling-based approaches sacrifice the accuracy of the analysis, and cannot be used when total accuracy is critical (e.g., for taint analysis [18]). Finally, code-level instrumentation solutions have limitations with respect to code coverage: in the case of Node.js, none of them can instrument the built-in library, while special launchers or tools are required to instrument NPM modules.

Other, less popular approaches exist. Goldshtein et al. [7] gave a tutorial of several ways to use low-level system events (e.g., SystemTap, perf, DTrace [11]) for Node.js profiling. Such approaches have low overhead, but cannot easily map low-level performance events to high-level JavaScript constructs. In comparison, NodeProf instruments the AST, which is still a high-level representation of the application, and can write analyses as flexible as code-level instrumentation frameworks.

9 Conclusion

In this paper, we introduced NodeProf, a new dynamic analysis framework for Node.js. Powered by the GraalVM polyglot language runtime, NodeProf leverages automatic partial evaluation to generate efficient machine code. NodeProf relies on AST-level instrumentation with efficient profiling event generation using frame virtualization. The instrumentation is done at runtime and enables comprehensive coverage of the whole Node.js application, including the Node.js built-in library and all NPM modules.

NodeProf provides two programming models. First, it can run Jalangi analyses in JavaScript out of the box. Second, dynamic analyses can be developed using a Java API, which can further benefit from the AST-level optimizations performed by the Graal compiler. With both APIs, NodeProf analyses run up to three orders of magnitude faster than Jalangi.

In contrast to other approaches, NodeProf is hot-pluggable such that dynamic analyses developed using NodeProf can be enabled or disabled at runtime, and incur zero overhead when disabled.

The focus of this paper was JavaScript and Node.js. Our results show the great potential of writing dynamic analyses with awareness of the language runtime. In the future, we plan to extend NodeProf to support other GraalVM-based languages such as R, Ruby, or Python.

Acknowledgments

The work presented in this paper was supported by Oracle (ERO project 1332). We thank the VM Research Group at Oracle 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] Acme Air. Last visited: October 2017. https://github.com/acmeair/acmeair-nodejs.
[2] John A Bather. 1994. Mathematical induction. (1994).
[3] L. Christophe, E. G. Boix, W. D. Meuter, and C. D. Roover. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 1. 260–270.
[4] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. 2015. Dynamic Analysis Using JavaScript Proxies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. 813–814.
[5] James Davis, Arun Thekumparampil, and Dongyoon Lee. 2017. Node.Fz: Fuzzing the Server-Side Event-Driven Architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*. 145–160.
[6] Google V8 JavaScript Engine. Last visited: October 2017. https://developers.google.com/v8/.
[7] Sasha Goldshtein. 2017. Profiling Node Applications.
[8] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 357–368.
[9] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 94–105.
[10] Oracle GraalVM. Last visited: October 2017. http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html.
[11] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.
[12] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*. 21–38.
[13] Samsung Jalangi2. Last visited: October 2017. https://github.com/Samsung/jalangi2.
[14] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. 2015. MemInsight: Platform-independent Memory Debugging for JavaScript. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 345–356.
[15] JITProf. Last visited: October 2017. https://github.com/Berkeley-Correctness-Group/JITProf.
[16] Apache JMeter. Last visited: October 2017. http://jmeter.apache.org/.
[17] Neil D. Jones and Arne J. Glenstrup. 2002. Program Generation, Termination, and Binding-time Analysis. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. 283–283.
[18] James Newsome, Dawn Song, James Newsome, and Dawn Song. 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the 12th Network and Distributed Systems Security Symposium*.
[19] Node Package Manager (NPM). Last visited: October 2017. https://github.com/npm/npm.
[20] Google Octane. Last visited: October 2017. https://developers.google.com/octane/.
[21] Google V8 Profiler. Last visited: October 2017. https://nodejs.org/en/docs/guides/simple-profiling/.
[22] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages*. 151–166.
[23] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. 2014. Debugging at Full Speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*. Article 2, 13 pages.
[24] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.
[25] S. Tilkov and S. Vinoski. 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (Nov 2010), 80–83.
[26] Michael L. Van De Vanter. 2015. Building Debuggers and Other Tools: We Can "Have It All". In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. Article 2, 3 pages.
[27] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. 2017. One Compiler: Deoptimization to Optimized Code. In *Proceedings of the 26th International Conference on Compiler Construction*. 55–64.
[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 *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 662–676.
[29] 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 *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 187–204.
[30] 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*. 73–82.