Automated Large-Scale Multi-Language Dynamic Program Analysis in the Wild

Alex Villazón
Universidad Privada Boliviana, Bolivia
avillazon@upb.edu

Haiyang Sun
Università della Svizzera italiana, Switzerland
haiyang.sun@usi.ch

Andrea Rosà
Università della Svizzera italiana, Switzerland
andrea.rosa@usi.ch

Eduardo Rosales
Università della Svizzera italiana, Switzerland
rosale@usi.ch

Daniele Bonetta
Oracle Labs, United States
daniele.bonetta@oracle.com

Isabella Defilippis
Universidad Privada Boliviana, Bolivia
isabelladefilippis@upb.edu

Sergio Oporto
Universidad Privada Boliviana, Bolivia
sergiooporto@upb.edu

Walter Binder
Università della Svizzera italiana, Switzerland
walter.binder@usi.ch

Abstract

Today's availability of open-source software is overwhelming, and the number of free, ready-to-use software components in package repositories such as NPM, Maven, or SBT is growing exponentially. In this paper we address two straightforward yet important research questions: would it be possible to develop a tool to automate dynamic program analysis on public open-source software at a large scale? Moreover, and perhaps more importantly, would such a tool be useful? We answer the first question by introducing NAB, a tool to execute large-scale dynamic program analysis of open-source software in the wild. NAB is fully-automatic, language-agnostic, and can scale dynamic program analyses on open-source software up to thousands of projects hosted in code repositories. Using NAB, we analyzed more than 56K Node.js, Java, and Scala projects. Using the data collected by NAB we were able to (1) study the adoption of new language constructs such as JavaScript Promises, (2) collect statistics about bad coding practices in JavaScript, and (3) identify Java and Scala task-parallel workloads suitable for inclusion in a domain-specific benchmark suite. We consider such findings and the collected data an affirmative answer to the second question.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis

Keywords and phrases Dynamic program analysis, code repositories, GitHub, Node.js, Java, Scala, promises, JIT-unfriendly code, task granularity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.20

Category Tool Insights Paper

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 20; pp. 20:1–20:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.11`

Acknowledgements This work has been supported by Oracle (ERO project 1332), Swiss National Science Foundation (scientific exchange project IZSEZ0_177215), Hasler Foundation (project 18012), and by a Bridging Grant with Japan (BG 04-122017).

1 Introduction

Analyzing today's large code repositories[1] has become an important research area for understanding and improving different aspects of modern software systems. Static and dynamic program analyses are complementary approaches to this end. Static program analysis is the art of reasoning about the behavior of computer programs without actually running them, which is useful not only in optimizing compilers for producing efficient code, but also for automatic error detection and other tools that help programmers [45]. Complementary to static analysis is *dynamic program analysis* (DPA), which employs runtime techniques (such as instrumentation and profiling) to explore the runtime behavior of applications under specific workload conditions and input.

In contrast to the large body of work on mining code repositories through static program analysis [39, 52, 37, 53, 38, 7, 11, 51], studies applying DPA to large public code repositories are scarce [35, 42]. Moreover, all such studies are limited at narrow, specific aspects of a particular programming language or framework, and none of them scales to the overwhelming amount of available projects that could potentially be analyzed.

In this paper, we tackle two basic yet important research questions: can we create a tool for automated DPA that can scale to the vast amount of available public open-source projects, and would such a tool be of practical interest?

Given the constant growth of the number of projects in public code repositories and the increasing popularity of different programming languages and runtimes – e.g., JavaScript/Node.js and the many languages targeting the Java Virtual Machine (JVM) – such a tool not only should be scalable, but should also be language-agnostic and resilient to malicious or buggy code. Such aspects already correspond to non-trivial technical challenges. Beyond these technical aspects, developing such a tool would require answering one more fundamental question: *how* should the tool execute code from repositories that are not designed to enable DPA?

Our answer to the question is pragmatic: the tool should *automatically* look for the available executable code in a repository, and *try* to execute anything that could potentially be executed. Such executable code could correspond to existing benchmarks (e.g., workloads defined by the developers via the Java Microbenchmark Harness (JMH) [48]) or software tests (e.g., defined in the default test entry of a Node.js project managed by Node Package Manager (NPM), or based on popular testing frameworks such as JUnit [63]). By replicating the process on the massive size of public repositories, such a tool should be able to identify a very high amount of (automatically) executable code.

With the tool, would running massive DPA on publicly available repositories be of any scientific interest? Our insight is that such a tool can be useful to collect statistics and evidence about code patterns and application characteristics that may benefit language designers, software system designers, and programming-language researchers at large.

[1] In this paper, the term *repository* denotes a code hosting site (such as GitHub, GitLab, or BitBucket), containing multiple *projects* (i.e., open-source code subjected to version control).

One concrete application for such a tool could be the study of the adoption (by a wide open-source community) of new programming-language constructs involving dynamic behavior (i.e., where pure static analysis cannot give any concrete insight). For example, it is currently unclear how the recently-introduced JavaScript Promise API [29] is being used by Node.js developers: while some recent research seems to suggest that developers are frequently mis-using the API [40, 1], its growing adoption by the Node.js built-in modules (e.g., Node.js' file-system module) could result in a more disciplined usage. In this context, a massive analysis of Node.js projects would be of great help to assess the adoption of the API, and to drive its future evolution. Measuring aspects such as the size of a so-called *promise chain* [1] (i.e., the number of promises that are linked together) requires DPA.

A second practical application of a tool for massive DPA is the study of problematic code patterns in dynamic languages. For example, several studies [12, 46, 17] focus on *JIT-unfriendly* code patterns [17], i.e., code that may obstacle dynamic optimizations performed by a Just-In-Time (JIT) compiler. While these studies have shown that such bad code patterns can impair the performance of modern language execution runtimes, none of them has investigated how common problematic coding practices are. Identifying such bad code patterns and assessing their use on the high number of open-source Node.js projects as well as the NPM modules they depend on could be very useful in practice, as it would provide a "bird's-eye view" over the quality of the NPM ecosystem. Similarly to the previous case, DPA is needed to identify such patterns in several runtime-dependent scenarios.

A final useful application for a massive DPA tool is the search for workloads suitable to conduct experimental evaluations. For many domain-specific evaluation needs (e.g., concurrency on the JVM), there is a lack of suitable benchmarks, and creating new benchmark suites requires non-trivial effort for finding proper workload candidates [69]. For example, existing general-purpose benchmark suites including Java and Scala benchmarks (e.g., DaCapo [64] and ScalaBench [56]) have only few task-parallel workloads [5, 58, 61]. Ideally, a fully automated system could discover relevant workloads by massively analyzing the open-source projects in public code repositories. Such a system could find real-world concurrent applications that spawn numerous tasks of diverse granularities, suitable for inclusion in a benchmark suite targeting concurrency on the JVM. Similarly to our previous examples, profiling all parallel tasks spawned by an application and measuring each task's granularity requires DPA.

To support the diversity of DPA scenarios that we have described at the scale of public code repositories, in this paper we present NAB,[2] a novel, distributed, container-based infrastructure for massive DPA on code repositories hosting open-source projects, which may be implemented in different programming languages. NAB resorts to containerization for efficient sandboxing, for the parallelization of DPA execution, and for simplifying the deployment on clusters or in the Cloud. Sandboxing is important to isolate the underlying execution environment and operating system, since NAB executes unverified projects that may contain buggy or even harmful code. Also, parallelizing DPA execution is an important feature for massive analysis, as sequential analysis of massive code repository would take prohibitive time. NAB features both crawler and analyzer components, which are deployed in lightweight containers and can be replicated. They are governed by NAB's coordination component, which ensures scalability and elasticity, facilitating the provisioning of new container images through simple configuration settings. NAB includes a plugin mechanism for the integration of existing DPA tools and the selection of different build systems, testing frameworks, and runtimes for multi-language support.

[2] NAB's recursive name stands for "NAB is an Analysis Box".

Our work makes the following contributions:

- We present NAB, a novel, distributed infrastructure for custom DPA in the wild. To the best of our knowledge, NAB is the first scalable, container-based infrastructure for automated, massive DPA on open-source projects, supporting multiple programming languages.
- We present a novel analysis to collect runtime statistics about the usage of promises in Node.js projects, which focuses on the size of promise chains. The analysis sheds light on the usage of the Promise API in open-source Node.js projects. We present an implementation of the analysis called Deep-Promise, which relies on NodeProf [62], an open-source dynamic instrumentation framework for Node.js based on GraalVM [68].
- We conduct a large-scale study on Node.js projects, searching for JIT-unfriendly code that may impair the effectiveness of optimizations performed by the JavaScript engine's JIT compiler. To this end, we apply JITProf [17], an open-source DPA.
- We perform a large-scale analysis on Java and Scala projects searching for task-parallel workloads suitable for inclusion in a benchmark suite. To this end, we apply tgp [54], an open-source DPA for task-granularity profiling on the JVM.

Our work confirms (1) that NAB can be used for applying DPA massively on public code repositories, and (2) that the large-scale analyses enabled by NAB provide insights that are of practical interest, thus affirmatively answering to our research questions.

This paper is structured as follows. Section 2 describes NAB's architecture and implementation. Section 3 details the experimental setup for our studies. Sections 4, 5, and 6 present the results of our three case studies. Section 7 discusses important aspects such as safety, extensibility, scalability, as well as the limitations of NAB. We discuss related work in Section 8 and conclude in Section 9.

2 NAB

This section presents our tool for massive DPA, NAB. First, we introduce NAB's architecture (Section 2.1); then, we describe how NAB's main components interact (Section 2.2). We continue by detailing the crawling (Section 2.3) and analysis (Section 2.4) process. Finally, we describe NAB's plugin mechanism supporting different DPA tools (Section 2.5), as well as the implementation technologies used to support containerization (Section 2.6).

2.1 Architecture

At its core, NAB features a *microservice* architecture based on a *master-worker* pattern relying on a *publish-subscribe* communication layer, allowing asynchronous events to be exchanged between its internal components. Figure 1 depicts the overall NAB architecture based on Docker containers [22]. NAB uses existing containerized services (marked in gray) and introduces four new components, three of them running in containers: NAB-Crawler, NAB-Analyzer, and NAB-Master; as well as one external service, NAB-Dashboard.

The NAB-Crawler instances are responsible for mining and crawling code repositories, collecting metadata that allows making a decision on which projects to analyze. The NAB-Analyzer instances are responsible for downloading the code, applying some filtering and eventually running the DPA tool. The results generated by the DPA (such as profiles containing various dynamic metrics) are stored in a NoSQL MongoDB [26] database. NAB provides a *plugin* mechanism to integrate different DPA tools in NAB-Analyzer instances.

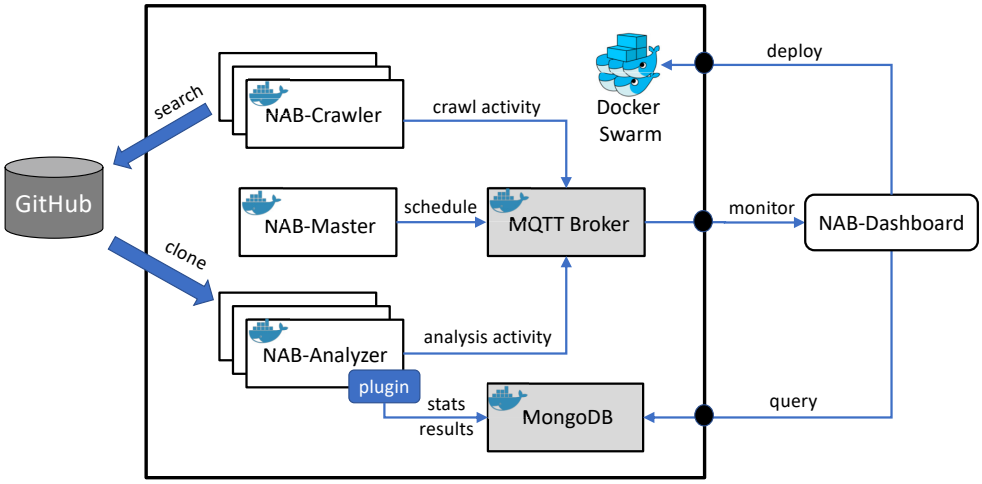

Figure 1 Overview of NAB. The three core NAB containerized services (NAB-Master, NAB-Crawler, and NAB-Analyzer) are shown as white boxes inside the Docker Swarm, whereas existing containerized services are marked in gray. A third-party DPA tool (not shown) is invoked by NAB-Analyzer through a plugin and generates profiles.

NAB-Master orchestrates the distribution of crawling and DPA activities with NAB-Crawler and NAB-Analyzer instances. Finally, NAB-Dashboard is responsible for the deployment of the NAB components through the Docker Swarm [23] orchestration service and monitors the progress of an ongoing DPA.

NAB communication service is based on MQTT (Message Queuing Telemetry Transport), an ISO-standard, lightweight publish-subscribe protocol [14]. The MQTT Broker is the core communication service that receives subscription requests from NAB components to different *topics* and redistributes messages that are published on such topics. This ensures that the communication between NAB components is implemented in a loosely-coupled manner, as they only need to agree on specific topics without knowing each other beforehand.

Some NAB services expose ports (represented as black circles in Figure 1) to allow interaction with them from outside the Docker Swarm. For example, NAB-Dashboard uses the exposed ports to access the MQTT communication service and subscribes to all topics used by the other NAB components. This information allows monitoring the distributed execution of a DPA. Similarly, the NAB-Dashboard can query the MongoDB database to collect execution results of previous runs, e.g., for post-mortem performance analysis, for visualizing the detailed distributed execution, for finding load imbalances, or for debugging purposes.

To improve scalability, NAB can be configured to use several MQTT Brokers through HAProxy [19] (a high-performance TCP load balancer that distributes the exchanged MQTT messages) as well as multiple distributed and replicated MongoDB instances through a *mongos* router [25] coordinating a MongoDB Shard [27] (not shown in Figure 1).

2.2 Interactions between NAB Components

When NAB is started, NAB-Dashboard initializes (through Docker Swarm) all the containerized services shown in Figure 1, passing user-defined specifications about the analysis to be executed to NAB-Master. Such specifications depend on the DPA to use, as well as on the code repository where crawling should be performed. While NAB provides supports for crawling different repositories (e.g., GitHub [24], GitLab [16], Bitbucket [2]), in this paper we focus on GitHub, which is the one providing the most advanced search API.

When performing a DPA on GitHub, the specifications sent to NAB-Master include crawling and analysis settings, the DPA plugin configuration, and a set of user-defined time intervals (which restricts the crawling to specific ranges of dates). The time intervals refer to the dates where the projects had the most recent activity (i.e., commits made to any branch). Since Docker Swarm does not provide a mechanism to enforce a given order for starting containers, we implement a synchronization mechanism such that all NAB components wait for the MQTT Broker container to be ready, since it is the core communication-related component, responsible for receiving and dispatching messages.

NAB-Master handles four lists: `timeIntervals` (storing the time intervals to crawl and initialized according to the specifications set by the user), `projects` (storing the name of the projects to analyze, initially empty), `crawlers` and `analyzers` (storing the IDs of available and ready NAB-Crawlers and NAB-Analyzer instances, respectively, both initially empty). During the initialization, NAB-Master subscribes to three topics: `availableCrawler` (to receive messages from NAB-Crawler instances that are ready), `availableAnalyzer` (same purpose of `availableCrawler`, but reserved for ready NAB-Analyzer instances, and `projectFound` (to receive the name of the projects to analyze from NAB-Crawler instances). Then, NAB-Master simply waits for messages on these topics to arrive. Thanks to the loosely coupled and asynchronous architecture, NAB-Crawler and NAB-Analyzer instances can start at any moment or be spawned on demand.

Figure 2 depicts the interaction between NAB components for crawling and analysis (note that topic subscription is now shown in the figure). Whenever a NAB-Crawler instance starts, it subscribes to the `crawlerAssignment` topic to receive crawling tasks from NAB-Master. To announce that it is ready for crawling, the NAB-Crawler publishes a message to the `availableCrawler` topic together with its ID. NAB-Master stores the received ID in the `crawlers` list. Then, it verifies if there are available time intervals to crawl in the `timeIntervals` list, and, if so, publishes a message to the `crawlerAssignment` topic to start the crawling process, including the ID of the NAB-Crawler that should perform the crawling. Each NAB-Crawler instance receives the message, and verifies if its own ID corresponds to the ID included in the message. In this case, it processes the request and starts crawling.

The NAB-Crawler performs the query on GitHub's search API, and filters out the projects matching the analysis criteria (more details are given in Section 2.3). For each matching project (see the loop frame in Figure 2), a message is published to the `projectFound` topic such that NAB-Master can collect all projects amenable to analysis in the `projects` list. Once the crawling is terminated, a message is sent to the `availableCrawler` topic to signal the availability to crawl projects in a new time interval.

Following a procedure similar to the one employed for NAB-Crawlers, when a NAB-Analyzer instance starts, it subscribes to the `analyzerAssignment` topic and publishes a message to the `availableAnalyzer` topic specifying its ID. Such ID is stored in the `analyzers` list handled by NAB-Master. If there are entries in the `projects` list, NAB-Master publishes a message to the `analyzerAssignment` topic, specifying the ID of the NAB-Analyzer that should execute the DPA and the project to analyze. The corresponding NAB-Analyzer, upon verifying that its ID matches the one contained in the message, starts the DPA activity, by first cloning the project and then running the DPA tool though the selected plugin (more details on the DPA execution are given in Section 2.4). Upon completion, the NAB-Analyzer collects the generated results as well as statistics on the DPA execution (shown as `results` and `stats` in the figure, respectively), and stores them in the MongoDB database. Finally, the NAB-Analyzer discards the temporary data needed for running the

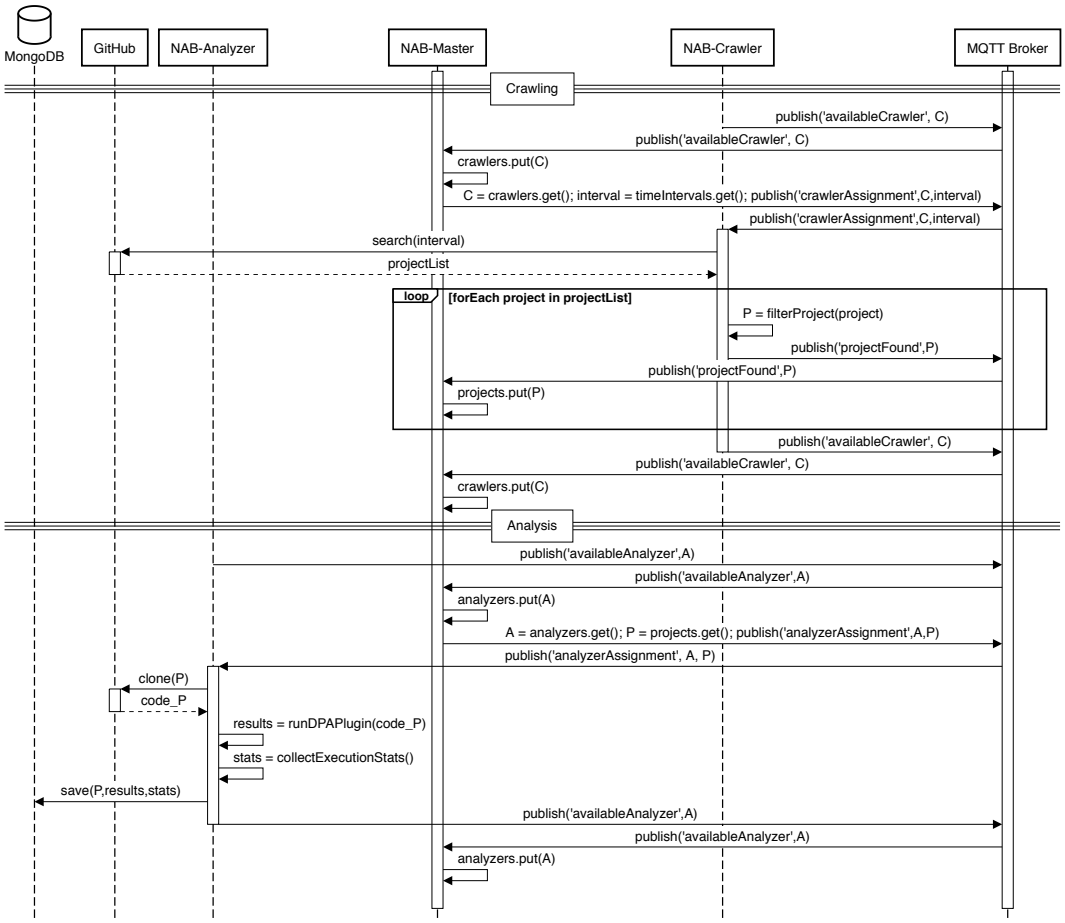

Figure 2 Interactions between NAB's components during crawling and analysis (after component initialization).

DPA, initializes a fresh execution environment and announces that is ready for a new DPA activity by publishing a message to the `availableAnalyzer` topic. NAB-Master stops when no further time intervals to crawl and projects to analyze are in the corresponding lists.

2.3 Crawling

Each NAB-Crawler instance receives (from NAB-Master) a set of specifications describing the characteristics of projects to be crawled, including the time intervals to consider during crawling (which can express either the date of project creation or the last update), the programming language(s) or the build system(s) to match, and the maximum number of results per request. This information is used to query the GitHub's search API to collect metadata of the matching projects. Additionally, NAB-Crawler can filter the metadata resulting from the query according to various criteria, such as selecting only projects with a specific entry in the build file (useful for discarding projects that are unrelated to a given DPA), or those with a minimum number of forks, watchers, stars, or contributors. Such criteria are set by the user, and are sent to the NAB-Crawler instance by NAB-Master. Finally, the NAB-Crawler sends the matching projects' names to NAB-Master.

Since the GitHub API has a limit[3] of 1,000 results per search query (independently from the selected time intervals), NAB-Master automatically subdivides the user-defined search intervals such that each NAB-Crawler instance is assigned fewer projects to crawl than the aforementioned limit, thus eventually crawling all the projects in the specified time intervals.

2.4 Executing DPA

The analysis starts with cloning the source code of the project from GitHub. NAB provides support for different build systems, e.g., NPM for Node.js, SBT for Scala, or Maven (MVN) for Java and Scala. NAB-Analyzer spawns a process to run the automatic build system, which typically downloads dependencies and compiles the required code. Finally, the project's testing code is executed, applying a DPA through a plugin (see Section 2.5). Upon DPA completion, the NAB-Analyzer instance stores the analysis results and the execution statistics in the database. By default, the DPA is run on the most recent revision of the default branch (as set up by the manager of the project on GitHub).

Apart from the results of the DPA, NAB collects different statistics on the analysis execution, such as start and end timestamps for every performed activity, and the exit code of the spawned process (indicating success or failure). Such statistics are stored in the database along with the analysis results. NAB-Analyzer sets a timeout (called *analysis timeout*) for the spawned process running the DPA to prevent buggy, malicious, or non-terminating application code (or DPA code) from excessively consuming resources. DPAs exceeding the analysis timeout are forcibly terminated.

NAB also reports projects that fail to build. Build failures are usually caused by developers assuming the presence of pre-installed software or specific settings (e.g., variables or paths) in the environment, which instead are not present. For example, in a Node.js project, some module dependencies may be missing in its configuration. Similar issues can occur in Java projects, where some Maven-managed projects may fail to build due to missing libraries or tools (e.g., parsers or pre-processors).

Since NAB uses a two-level orchestration (i.e., an external orchestration handled by Docker Swarm and an internal one handled by NAB-Master), it can happen that NAB-Analyzer containers are restarted by Docker Swarm without NAB-Master being notified. To handle this case, NAB implements a fault-tolerance mechanism using a timer that is started for each scheduled DPA. If a result arrives before the analysis timeout, the timer is stopped; otherwise, the DPA is re-scheduled for execution for a configurable number of times. Users can configure how NAB should handle multiple results in the case of a restarted DPA (e.g., keep only the first result, keep all results, combine the set of results.)

2.5 NAB Plugins and DPA Tools

To run massive analyses, NAB provides a *plugin* mechanism for different analysis frameworks to run DPA tools.[4] To integrate a third-party DPA tool into NAB, the user needs to provide three shell scripts: (1) to set up the execution environment for the analysis framework, (2) to execute the DPA tool, and (3) to post-process the DPA results. NAB allows one to select a build system and a runtime, and takes care of executing the provided scripts and applying the DPA tools to the projects under analysis. Integrating existing DPA tools into NAB does not require much effort. Typically, creating a new plugin requires about 100 lines of code, divided into the three aforementioned shell scripts. The post-processing script is usually the longest, as it needs to format the DPA results in JSON for storing them in the NAB database.

[3] `https://developer.github.com/v3/search/`

[4] NAB plugins can also directly integrate DPA tools that do not depend on any analysis framework.

Table 1 NAB supported languages, build systems, analysis frameworks, DPA tools, and runtimes. The sign † indicates those used in the following case studies.

Language	Build System	Analysis Framework	DPA Tool	Runtime
JavaScript†	NPM†	NodeProf† [62]	Deep-Promise† JITProf† [17]	GraalVM† [68]
Java†	MVN†	DiSL† [41] AspectJ [34]	tgp† [54] JavaMOP [31]	HotSpot VM† [9] GraalVM
Scala†	SBT† MVN	DiSL†	tgp†	HotSpot VM† GraalVM

For the supported build systems, NAB captures the exact entry point where the runtime is invoked, and plugs in the corresponding DPA tool. For example, to run a DPA tool with a plugin for a weaver or instrumentation framework working at the Java bytecode level (e.g., AspectJ's load-time weaver [34] or the DiSL [41] dynamic instrumentation framework) the user can select the MVN build system and Oracle's HotSpot JVM [9] as runtime; NAB takes care of passing all the required parameters (e.g., instrumentation agents, libraries, or generated harness bytecode) to the JVM, to start the analysis. Existing DPA tools that can run in a containerized environment are suitable for NAB. On the contrary, DPA tools that require access to special OS- or hardware-layer features that are not supported in containerized environments (e.g., hardware performance counters or NUMA-specific CPU/memory policy control) are not candidates for a NAB plugin. Table 1 shows the currently supported programming languages, build systems, analysis frameworks, DPA tools, and runtimes.

Extending NAB to support other languages, build systems and testing frameworks is mostly a straightforward engineering effort. To extend NAB, one needs to modify the NAB-Analyzer component (generating new scripts wrapping the build system to set NAB-specific variables and to invoke the selected plugin) and its building configuration (updating the `Dockerfile`, to install the new build system and to deploy the runtime within the NAB-Analyzer image). All other core NAB components remain unchanged.

2.6 Implementation Technologies for Containerization

NAB-Master, NAB-Crawler, and NAB-Analyzer are implemented in Node.js and are deployed in containers in a Docker Swarm. NAB uses an overlay network inside the Swarm to optimize the communication between the NAB components and to leverage the internal DNS service (to reach NAB services by name, avoiding complex settings). NAB relies on Docker Compose [21], which simplifies the deployment of the whole analysis infrastructure on a cluster or in the Cloud. Since all NAB services are based on Docker container technology, NAB can be easily migrated to other Docker-friendly orchestration engines, such as Kubernetes [4] or Mesos [20], which provide similar functionalities as Docker Swarm. NAB has been tested on Azure Cloud [44] and on a shared cluster with hundreds of nodes.

3 Experimental Setup

Here, we present the experimental setup used for running NAB to analyze massive collections of open-source projects in our case studies. Even though crawling and analysis can be done at the same time in NAB, we separate both processes to first generate a list of projects that is then used for different DPAs.

Table 2 Number of selected GitHub projects. For each programming language (JavaScript/Node.js, Java, Scala), the first row lists the amount of all GitHub projects. Subsequent rows show the number of surviving projects after applying filters (i.e., the predicates in the leftmost column). The number of projects considered in our case studies is shown in the row "≥ 2 Contrib.".

	2013	2014	2015	2016	2017	Total
All Node.js	226,879	380,607	620,211	1,055,799	1,708,261	3,991,757
NPM Build	226,858	332,399	363,181	364,871	363,898	1,651,207
Test Entry	22,956	65,018	100,581	134,917	158,547	482,019
≥ 2 Contrib.	4,311	11,415	20,593	30,889	42,078	**109,286**
All Java	164,208	321,290	636,316	1,001,370	1,477,473	3,600,657
MVN Build	164,146	301,632	362,061	364,861	363,898	1,556,598
Test Entry	17,569	28,411	32,005	32,948	35,773	146,706
≥ 2 Contrib.	2,748	4,361	5,583	6,112	7,114	**25,918**
All Scala	7,875	11,670	18,692	25,501	33,463	97,201
SBT Build	7,853	11,640	18,618	25,402	33,188	96,701
Test Entry	1,802	3,222	5,991	9,217	11,971	32,203
≥ 2 Contrib.	222	404	706	1,021	1,723	**4,076**

3.1 NAB Configuration and Deployment

We run NAB in a shared cluster, composed of 176 nodes (each requiring reservation), where each node is equipped with a 16-core AMD Opteron CPU (2.6 GHz) and 128 GB RAM. We run 1,024 NAB-Analyzer instances using 64 nodes, i.e., 16 NAB-Analyzer instances per node. The nodes are connected to a 10 Gb/s internal network and to a 1 Gb/s external network.

We configure the NAB core services with V8 Node.js 10.9 deployed with Docker-CE 1.18. The NAB containers are built using a Linux Ubuntu 16.04 Docker image. We deploy Eclipse Mosquitto MQTT 1.4 broker and MongoDB 4.0.

For all the analyses performed in the three case studies, the NAB-Analyzer instances are configured with an analysis timeout of one hour.

3.2 Crawling and Project Selection

We crawl 5 years of Node.js,[5] Java, and Scala projects from GitHub (from 2013-01-01 to 2017-12-31). To minimize the number of (typically small) personal projects crawled, we only collect projects with a minimum of 2 contributors, as suggested by [33]. For each crawled project, we check whether an automatic build configuration file is present (i.e., NPM's `package.json` file for Node.js, SBT's `build.sbt` file for Scala, or MVN's `pom.xml` file for Java), such that the project can be automatically built. The analyses presented in this paper are conducted on the unit tests in the projects that can be run by automatic means (e.g., in case of NPM, by executing "`npm test`"). For this reason, we also check whether a test entry exists.

Table 2 shows the total number of crawled and selected projects per year (i.e., projects that existed and whose most recent commit made on any branch occurred that year). For

[5] Node.js projects are identified as JavaScript projects in GitHub, as JavaScript is used as programming language in the project description, but can be recognized thanks to the presence of Node.js-specific configuration files.

each programming language considered, the table reports the number of projects that can be built, that contain a test entry, and those with at least 2 contributors (such filters are applied in cascade). Only projects passing all filters (i.e., 109,286 Node.js projects, 25,918 Java projects, and 4,076 Scala projects, as reported in the last row) are considered in the following case studies.

4 Case Study I: Analyzing the Use of Promises in Node.js

In our first study, we present a novel dynamic analysis (called Deep-Promise) to collect insights on how developers use JavaScript's Promise API, allowing one to understand its actual adoption to handle asynchronous executions. The Promise API is a key novel language feature introduced in the ECMA specification to enable better handling of asynchronous interactions in JavaScript applications. Promises greatly simplify the way asynchronous code can be expressed, by introducing the notion of *promise chain*, i.e., a sequence of asynchronous events with logical dependencies.

The goal of this study is to provide a better understanding of the popularity and usage trend of the API in real-world projects and in the modules they depend on, which can be useful for the Node.js community. To the best of our knowledge, this is the first large-scale study on the use of promises in Node.js projects and the NPM modules they depend on, enabled by NAB.

4.1 Monitoring Promises in JavaScript

The introduction of promises since ECMAScript 6 [29] greatly simplifies the development of asynchronous applications in JavaScript. With promises, asynchronous executions can be implemented elegantly, avoiding the so-called "callback hell" problem, by enabling chaining of (asynchronous) functions [1]. The value that resolves or rejects a promise can be used as the input of the reacting promise(s). The latter promise(s) depend on the previous one, forming a promise chain.

In [40], the authors introduce the notion of *promise graph*, a formal graph-based model for understanding and debugging code developed using the Promise API. A promise graph is composed of several promise chains. Each promise chain is an acyclic data structure showing the dependencies among promises and the values that resolve or reject each promise. The *size* of a promise chain, i.e., the number of promises inside the chain, gives us insights about the use of promise constructs. As promise chains of size one have no subsequent reactions to be executed asynchronously, such chains are not used to handle asynchronous executions. Hence, we denote as *trivial* a promise chain of size one, as opposed to a *non-trivial* promise chain, which size is greater than one.

Building a DPA to accurately capture all promise chains requires an instrumentation framework capable of intercepting every use of the Promise API. Before explaining Deep-Promise, we first introduce the underlying instrumentation framework NodeProf, integrated in NAB through a dedicated analysis plugin.

4.1.1 NodeProf Framework

NodeProf [62] is an open-source[6] dynamic instrumentation framework for Node.js based on GraalVM [68]. NodeProf relies on the dynamic instrumentation of the Abstract Syntax Tree (AST) interpreter of the GraalVM JavaScript engine. In contrast to other dynamic analysis

[6] `https://github.com/Haiyang-Sun/nodeprof.js`

frameworks for Node.js (e.g., Jalangi [57]), NodeProf is compatible with ECMAScript 8, and supports JavaScript constructs such as promises and `async/await`, which are not supported in other frameworks. An additional advantage of NodeProf is that it instruments only loaded code, while other frameworks typically rely on source-code instrumentation, which usually needs to instrument all source code files in advance before execution, even if they are not used. NodeProf's approach can significantly reduce the time needed for the instrumentation, as the NPM modules a project depends on can contain thousands of source code files (while only a small portion of them may be used by the project).

4.1.2 Deep-Promise DPA

Deep-Promise is a DPA implemented on top of NodeProf that constructs the full promise graph of any Node.js application at runtime. Deep-Promise tracks the creation of all promises and the dependent relations between them. There can be three different dependencies between two promises: (1) *fork*, where a promise has one or more reactions registered, e.g., via `Promise.then` or `Promise.catch`; (2) *join*, where multiple promises join into one promise via `Promise.all` or `Promise.race`; (3) *delegate*, where a promise is used as a value to resolve or reject another promise. Additionally, Deep-Promise also tracks usages of `async/await`, which deal with implicit promises.

Deep-Promise directly instruments the built-in promise implementation, capturing the creation of every promise and the dependency between promises. Upon program termination, the promise graph is dumped and stored in the database by the NAB-Analyzer instance executing the analysis. The resulting promise chains are later analyzed with an offline tool to compute statistics.

4.2 Executing Deep-Promise with NAB

For every project to analyze, we run the automatic build system through NPM, by executing "`npm install`", which downloads, compiles, and installs dependencies, including third-party testing frameworks such as *mocha*, *unitest*, or *grunt*. If the installation succeeds, we run the default test program in the project by executing "`npm test`" with Deep-Promise enabled.

From the 109,286 executed Node.js projects (see Table 2 on page 10), NAB reports 23,297 successfully analyzed projects with Deep-Promise. Unsuccessful projects include those with broken tests (e.g., wrong test settings or assuming non-standard pre-installed software), projects with failing tests, and those exceeding the 1-hour analysis timeout of our cluster-based experimental setup.

The total execution time reported by NAB is 7.1 hours. Running the analysis sequentially in a single NAB-Analyzer would take up to about 2.7 months.[7]

4.3 Promise API Adoption for Asynchronous Executions

First, we measure how widely promises are used in project tests by identifying the usage of the Promise API in either application code (i.e., code exercised by tests) or the dependent NPM modules. From the 23,297 successfully analyzed projects, we find that 5,971 projects (i.e., 25.6%) make use of the Promises API. In our analysis, we differentiate trivial promise chains (that are not used to handle asynchronous executions) from non-trivial ones.

[7] This estimation is calculated as the sum of the execution times reported for the analysis of each project by a NAB-Analyzer instance.

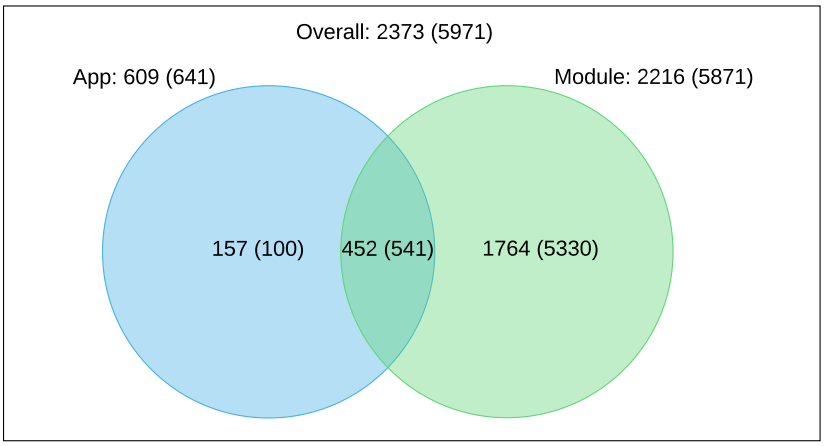

Figure 3 Distribution of projects that use promises only in application code (blue circle), only in modules (green circle), or in both (intersection). The left-side values indicate the number of projects excluding trivial promises, while the right-side values in parentheses include trivial promises.

Figure 3 shows the overall distribution of projects that use promises in the application code (blue circle), in the dependent NPM modules (green circle), or in both (their intersection). The right-side values in parentheses show the total number of projects making use of promises (before excluding trivial promise chains), while the left-side values indicate the number of projects after the exclusion. Overall, 5,971 projects use promises, while only 2,373 projects use non-trivial promise chains. The left-side numbers (after excluding trivial promises) are smaller than the right-side numbers in parentheses, except for the application-only part, because some projects use promises in both application code and in modules, but only trivial promises in modules. After excluding trivial promises, such projects are found using promises only in the application code.

From the figure, we can observe that out of 23,297 projects, only 157 (0.6%) of them use promises only in application code, while 2,216 (9.5%) use promises also indirectly, i.e., in the modules they depend on. Our findings suggest that many projects do not directly depend on promises, but rather rely on the promise support introduced by other modules that they depend on.

Our analysis also reveals that from the 5,971 projects that use the Promise API, a total of 440 different dependent NPM modules use promises, out of which 41 modules create only trivial ones. Figure 4 shows the most frequently used NPM modules that make use of promises on the x-axis (used by at least 100 projects), and the number of projects using such NPM modules on the y-axis. Each bar represents the number of projects and is further divided into two parts according to the maximum size of the promise chain (i.e., showing trivial and non-trivial promise chain usages). Modules *lodash* and *prettier* contain only trivial promise chains, as they use promises only for version detection.[8] The other modules use promises for different purposes. For example, *jest* and *mocha* are test harnesses widely used by NPM modules, which use promises to run tests, while *pify* is a library used to "promisify" a callback by returning a promise-wrapped version of it.

[8] Such modules execute `Promise.resolve` and check whether the returned object is a promise object or not, to detect whether the current JavaScript version is ECMAScript 6 or higher. Such a promise usage is unrelated to asynchronous executions.

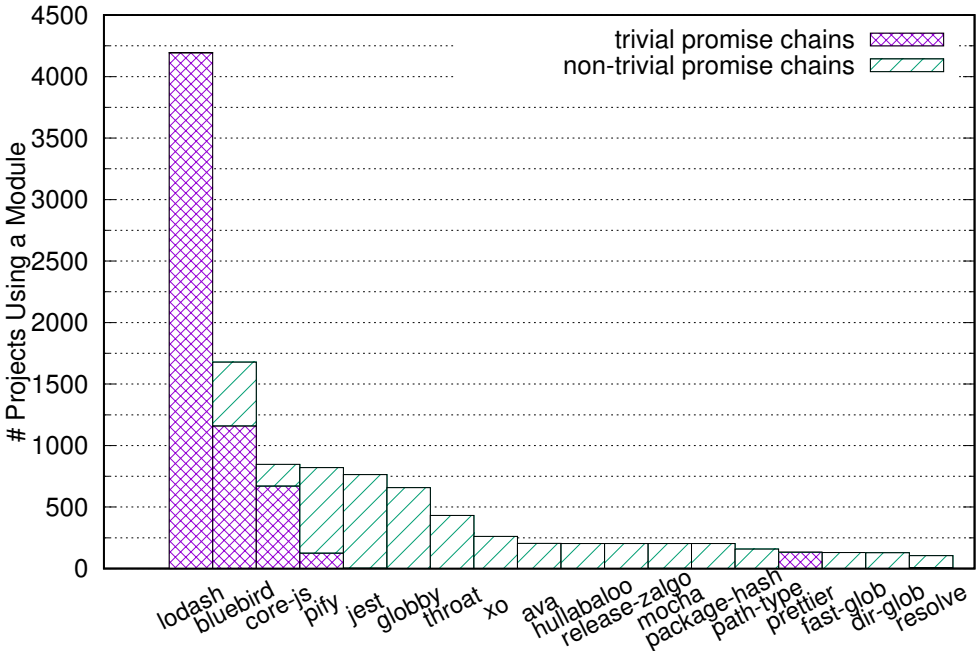

Figure 4 Most frequently used NPM modules that make use of promises, and the number of projects using such modules.

Our results suggest that such modules (i.e., with non-trivial promises) should be preferred over the others (i.e., without promises or with only trivial ones) by researchers as well as language implementers who might want to evaluate and optimize the usage of promises in real-world Node.js applications. Specifically, identifying and optimizing a popular promise usage pattern in one of such modules might have a positive performance impact on several existing applications.

4.4 Frequency of Promise API usage

We further study the frequency of Promise API usages among the 2,373 projects with non-trivial promise chains. The `Promise` constructor and the `Promise.then` method are the most used ones (in 2,287 and 2,363 projects, respectively), as they are the most common way to create and use promises. `Promise.catch` is used less frequently (in 1,732 projects), which reveals that not all programmers add a `catch` statement when programming with promises (which is considered a best practice when using promises in JavaScript [55]). Other APIs, such as `Promise.all` and `Promise.race`, are used even less frequently (in 1,488 and 233 projects, respectively). Finally, only 47 projects use `await` for asynchronous functions, as the `async/await` feature was introduced in ECMAScript 8 [30] (mid 2017).

The size of a promise chain is an important characteristic for understanding how applications use promises. Figure 5 shows the distribution of different maximum promise-chain sizes among the 2,373 projects that use non-trivial promise chains. 50.5% of the projects create only promise chains sized within 10; in 38.3% of the projects the longest promise-chain has a size between 11 and 100; and 11.2% of the projects have at least one promise chain with size greater than 100. We observe the longest promise chain (5,002 promises) in the project *lahmatiy/postcss-csso*.

Our findings suggest that the number of projects with long promise chains is relatively high. As the length of the promise chain is a potential indicator of a long-living application, such projects could be considered as potentially interesting for the development of microbenchmarks stressing the Promise API.

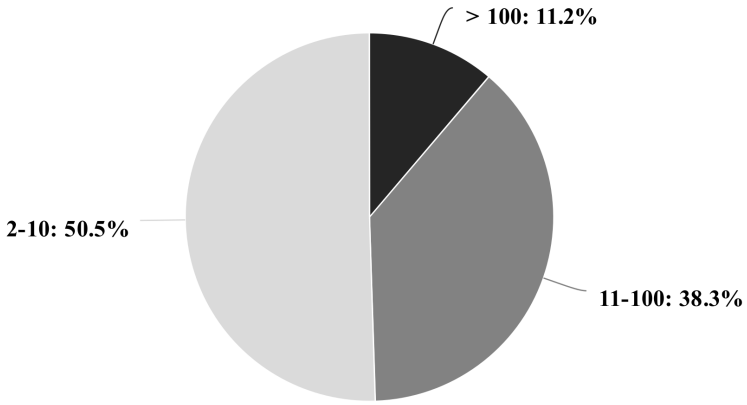

Figure 5 Statistics for the maximum promise-chain size observed in the 2,373 projects that make use of non-trivial promise chains. Percentages indicate projects with maximum chain size between 2 and 10, between 11 and 100, and more than 100.

In conclusion, this study demonstrates how NAB can be used for analyzing the usage of a particular programming-language construct (here, asynchronous executions via promises in Node.js projects) in the wild. The results of our analysis could be useful for Node.js developers to find projects and popular modules that use promises for asynchronous executions. In particular, evaluating and optimizing such modules could be beneficial to several existing applications. Moreover, we found many projects with long promise chains; such projects might be considered as potentially interesting for benchmarking promises on Node.js.

In future work, we plan to re-execute our analysis on new versions of the considered modules, and to track the adoption of the Promise API over time. In this way, the analysis could also indicate which specific parts of the API are gaining more adoption (if any).

5 Case Study II: Finding JIT-unfriendly Code Patterns in Node.js

Our second massive study of public repositories using NAB deals with the quality of the JavaScript code available on the NPM package repository. Specifically, our goal is to execute a comprehensive DPA to identify bad coding practices that are known to affect the performance of Node.js applications. Such coding patterns – also called *JIT-unfriendly* code patterns – may prevent typical JIT compiler optimizations, such as function inlining, on-stack replacement, and polymorphic inline caching.

To this end, we resort to an existing DPA for JavaScript, called JITProf [17], which can identify and collect a variety of JIT-unfriendly code patterns otherwise impossible to identify using static analysis. JITProf is open-source,[9] and relies on the Jalangi [57] instrumentation framework. Since Jalangi does not support the latest ECMAScript standard, we adapt the analysis to run on the NodeProf framework (see Section 4.1.1) for the GraalVM.

This yields the benefits of supporting up-to-date language features (as NodeProf is compatible with ECMAScript 8), and also reduces the time needed for the instrumentation.

JITProf can identify seven different categories of JIT-unfriendly code patterns, namely: `AccessUndefArrayElem`, tracking accesses to undefined array elements; `BinaryOpOnUndef`, to track when `undefined` is used in binary operations; `InconsistentObjectLayout`, to

[9] `https://github.com/Berkeley-Correctness-Group/JITProf`

Table 3 Amount of projects where at least a JIT-unfriendly code pattern in found (out of 26,938 analyzed Node.js projects). Dependent NPM modules are excluded from the analysis.

JIT-unfriendly Pattern	# Projects	%
`AccessUndefArrayElem`	1,253	4.7%
`BinaryOpOnUndef`	757	2.8%
`InconsistentObjectLayout`	9,509	35.3%
`NonContiguousArray`	194	0.7%
`PolymorphicOperation`	3,073	11.4%
`SwitchArrayType`	81	0.3%
`TypedArray`	546	2.0%
At least one	**9,969**	**37.0%**

find object access patterns that lead to inline-cache misses; `NonContiguousArray`, to locate non-contiguous array accesses; `PolymorphicOperation`, to identify polymorphic (including megamorphic) binary and unary operations; `SwitchArrayType`, to find unexpected transitions in arrays internal backing storage (i.e., array strategies [6]); `TypedArray`, to account for unnecessary usages of generic arrays where typed arrays could be used (e.g., to replace contiguous numeric arrays). A detailed description of the analyses required to identify such patterns can be found in [17].

5.1 Executing JITProf with NAB

We run JITProf with NodeProf in two settings, i.e., (1) profiling only application code, and (2) profiling also dependent NPM modules. Out of the 109,286 Node.js projects executed (see Table 2 on page 10), NAB reports 26,938 successfully analyzed projects with application-only profiling, and 3,940 projects when profiling also dependent NPM modules. The analyses in JITProf that identify JIT-unfriendly patterns require heavy instrumentation (as they track object creation, accesses and operations), which may significantly slow down application execution (leading to failures due to the presence of timeouts in the code, or increasing the execution time past the 1-hour analysis timeout) or cause out-of-memory errors, particularly when profiling NPM modules, due to the large amount of code that is instrumented and analyzed in all dependent NPM modules in each project. For this reason, JITProf does not complete on several projects, which are ignored in our analysis. Similarly to the previous use case, we exclude projects with broken or failing tests.

The total execution time reported by NAB is 25.3 hours, whereas a sequential execution in a single NAB-Analyzer would take up to about 8.4 months.

5.2 JIT-unfriendly Patterns in Application Code

We first focus on application code disregarding the dependent NPM modules. The results of the analyses are shown in Table 3. As shown in the second column ("# Projects"), a total of 9,969 projects result in *at least one* JITProf warning, i.e., 37.0% of the successfully analyzed projects suffer from at least one JIT-unfriendly code pattern. The most common pattern found is `InconsistentObjectLayout`, occurring in 9,509 projects. This result implies that these projects perform read or write accesses to objects in a sub-optimal way that may prevent compiler optimizations, and may therefore pay a performance penalty when performing such accesses in frequently executed code.

Table 4 Top 3 dependent NPM modules suffering from JIT-unfriendly code. Column "# Modules" indicates the number of unique modules where a given pattern occurs. For each module, the values in parentheses indicate the number of projects using that module.

JIT-unfriendly Pattern	# Modules	Top 3 NPM Modules		
`AccessUndefArrayElem`	252	*commander* (637)	*glob* (569)	*abbrev* (178)
`BinaryOpOnUndef`	83	*strip-json-comments* (110)	*jsbn* (110)	*sinon* (83)
`InconsistentObjectLayout`	523	*commander* (687)	*chai* (369)	*tape* (337)
`NonContiguousArray`	49	*semver* (167)	*jsbn* (110)	*eslint* (51)
`PolymorphicOperation`	453	*lodash* (311)	*glob* (178)	*mime-types* (174)
`SwitchArrayType`	16	*babylon* (4)	*lodash* (3)	*eslint* (3)
`TypedArray`	144	*lodash* (51)	*jshint* (48)	*regenerate* (38)
At least one	**900**	***commander*** **(963)**	***glob*** **(569)**	***lodash*** **(432)**

5.3 JIT-unfriendly Patterns in NPM Modules (top 3)

We also collect statistics on JIT-unfriendly code in the NPM modules used by the exercised tests, which is reported in Table 4. From the set of 3,940 Node.js projects successfully analyzed, 900 dependent NPM modules execute at least one JIT-unfriendly code pattern, as shown in the second column ("# Modules"). The most common JIT-unfriendly code pattern is `InconsistentObjectLayout` (in 523 modules), while only 16 modules show occurrences of `SwitchArrayType`. For each code pattern, we also identify the top 3 modules (i.e., the 3 NPM modules used most frequently among projects where we found the pattern). The values in parentheses below a module name in Table 4 show the number of projects using that module. For example, as shown in the first row of the table, there are 637 projects using the NPM module *commander*, 569 projects using *glob*, and 178 projects using *abbrev*.

Several modules suffer from more than one JIT-unfriendly code pattern. For example, the popular NPM module *lodash* frequently executes 3 kinds of JIT-unfriendly code patterns (i.e., `PolymorphicOperation`, `SwitchArrayType` and `TypedArray`). The 3 top modules with at least one JIT-unfriendly code pattern are *commander*, *glob* and *lodash* (as reported in the last row of the table). These modules are very popular, being imported by almost 130K other distinct NPM modules overall.[10]

In summary, our study reveals that Node.js developers frequently use code patterns that could prevent or jeopardize dynamic optimizations and have a potential negative impact on applications performance. Such patterns occur both in application code and in dependent NPM modules used by a project.

[10] The estimation of the number of dependent modules is taken from the global NPM registry (`https://www.npmjs.com/`) and dated November 2018.

6 Case Study III: Discovering Task-parallel Workloads

For many specific evaluation needs, there is a lack of suitable domain-specific benchmarks, forcing researchers to resort to less appropriate general-purpose benchmarks. In our third case study, we focus on discovering workloads that can fulfill the needs of a domain-specific evaluation. We consider a researcher requiring task-parallel applications on the JVM exhibiting diverse task granularities, to analyze concurrency-related aspects. Currently, only few task-parallel workloads can be found in well-known benchmark suites [5, 58, 61, 60] targeting the JVM. Moreover, except for DaCapo,[11] such suites were last updated several years ago, thus they may be not representative of state-of-the-art applications using task parallelism.

6.1 Executing tgp with NAB

`tgp` [54] is an open-source[12] DPA for detecting the granularity of tasks spawned by multi-threaded, task-parallel applications running on the JVM. `tgp` profiles all tasks spawned by an application (defined as subtypes of the Java interfaces/classes `java.lang.Runnable`, `java.util.concurrent.Callable`, and `java.util.concurrent.ForkJoinTask`), collecting their *granularity*, i.e., the amount of work carried out by each parallel task, in terms of the number of executed bytecode instructions. The DPA runs on top of DiSL [41], a dynamic analysis framework for the JVM that ensures complete instrumentation coverage (i.e., it can instrument every method with a bytecode representation), thus enabling the detection of tasks used inside the Java class library.

For this case study, we run `tgp` on top of DiSL (which can be attached to NAB via a dedicated plugin) to measure the granularity of all tasks spawned during the execution of testing code in Java and Scala projects from GitHub. Our goal is to discover projects with a high diversity of task granularities, which could be good workload candidates for benchmarking task parallelism on the JVM.

The total execution times reported by NAB for this DPA are 9.5 hours (Java projects) and 1.8 hours (Scala projects), whereas a sequential execution in a single NAB-Analyzer instance would take up to about 1.9 months (Java) and 12.5 days (Scala).

6.2 Results for Java Projects

Out of the 25,918 Java projects analyzed (see Table 2 on page 10), 1,769 projects make use tasks and successfully complete all tests within the 1-hour analysis timeout, thus they are considered for the following analysis.

The total number of tasks spawned by the analyzed projects is 1,406,802. The minimum granularity found is 1 and the maximum granularity is 187,673,879,636. Table 5 (Java section) shows the distribution of tasks wrt. their granularities. The first column shows all ranges of task granularities found. The second column reports the number of tasks with granularity in the corresponding range. The third column indicates the number of projects having at least one task with a granularity in the considered range.

The analysis results obtained with NAB reveal that test methods in project `https://github.com/rolfl/MicroBench` (a Java harness for building and running microbenchmarks written in Java 8) spawn a total of 55 tasks with granularities spanning all ranges

[11] Dacapo 9.12-MR1-bach was released on January 2018. However, the workloads were not significantly modified since the previous release (dated 2009) and no new workload has been added.
[12] `https://github.com/Fithos/tgp`

Table 5 Distribution of all tasks spawned in the considered Java and Scala projects wrt. their granularities.

Granularity Range	Java		Scala	
	Tasks	Projects	Tasks	Projects
$[10^0$ - $10^1)$	137,468	686	301,066	771
$[10^1$ - $10^2)$	278,765	466	280,244	710
$[10^2$ - $10^3)$	215,211	673	2,795,702	860
$[10^3$ - $10^4)$	285,196	1,092	1,278,974	769
$[10^4$ - $10^5)$	247,284	1,367	124,473	771
$[10^5$ - $10^6)$	128,992	1,492	74,989	769
$[10^6$ - $10^7)$	89,710	1,327	13,002	806
$[10^7$ - $10^8)$	17,178	1,046	4,555	677
$[10^8$ - $10^9)$	5,696	581	1,789	619
$[10^9$ - $10^{10})$	1,164	177	430	276
$[10^{10}$ - $10^{11})$	120	53	22	20
$[10^{11}$ - $10^{12})$	18	8	1	1

except $[10^4$ - $10^5)$. In addition, the project `https://github.com/47Billion/netty-http` (a library to develop HTTP services with Netty [65]) makes use of a total of 123 tasks in its tests, with granularities in all ranges except $[10^{11}$ - $10^{12})$. Both Java projects can be good candidate workloads for task benchmarking, as they exhibit a high diversity of task granularities.

6.3 Results for Scala Projects

Out of the 4,076 Scala projects analyzed (see Table 2 on page 10), 860 projects contain task-parallel workloads and successfully complete all tests within the analysis timeout. Such projects spawn a total of 4,875,247 tasks. The minimum granularity found is 2 while the maximum is 204,418,653,894. Table 5 (Scala section) shows the distribution of tasks wrt. their granularities.

The analysis results pinpoint three candidate workloads spawning tasks with granularities spanning all ranges except $[10^{11}$ - $10^{12})$. First, the project `https://github.com/iheartradio/asobu` (a library for building distributed REST APIs for microservices based on Akka cluster [36]) spawns a total of 19,880 tasks in its tests. Second, project `https://github.com/TiarkRompf/virtualization-lms-core` (a library for building high performance code generators and embedded compilers in Scala) executes a total of 5,759 tasks. Finally, tests inside project `https://github.com/ryanlsg/gbf-raidfinder` (a library for tracking gaming-related tweets from Twitter) run a total of 20,934 tasks. This set of projects provides candidate Scala workloads for benchmarking task execution on the JVM, due to their high diversity of task granularities.

Overall, our analysis results show that NAB can help discover good candidate workloads satisfying domain-specific benchmarking needs. Moreover, this study demonstrates NAB's support for multiple programming languages.

7 Discussion

In this section, we discuss the strengths and limitations of our approach, focusing on different aspects of massive DPA, including safety, extensibility, scalability, and code evolution.

7.1 Safety

As NAB is executing unknown projects, sandboxing is crucial to protect the execution platform from malicious or erroneous code. NAB relies on Docker containers to isolate the execution of DPAs from the host environment. Thus, a project may only crash a NAB-Analyzer instance running in a Docker container, without harming the underlying platform. Such crashes are handled by NAB's fault-tolerance mechanism. A project that repeatedly crashes or takes excessive time to execute will be excluded from any further DPA. The fault-tolerance mechanism also mitigates problems caused by unstable third-party DPA tools or by experimental runtimes.

7.2 Extensibility

NAB has been designed for extensibility, as code repositories have different search APIs, and projects may use different version-control systems, programming languages, build systems, and testing frameworks. Moreover, third-party DPA tools may require specific execution environments, such as a modified JVM. NAB uses a plugin mechanism to handle the large variety of systems it needs to interact with. The currently supported analysis settings are listed in Table 1 on page 9. It is straightforward to implement additional plugins; each existing NAB plugin has only about 100 lines of code.

We plan to add support for other version-control systems such as Mercurial [43], and for other programming languages and their ecosystems. Moreover, supporting the Java Microbenchmark Harness (JMH) [48] is straightforward, allowing leveraging existing benchmarks in open-source projects that use MVN. Furthermore, we are extending NodeProf to support additional dynamic languages offered by GraalVM, such as Python, Ruby, and R. These extensions will enable studies on an even larger code base, targeting many popular programming languages.

Finally, in addition to the direct interaction with the standard GitHub search API, NAB can be extended to interact with offline mirrors and metadata archive dataset of GitHub, such as GHTorrent [18] and GHArchive [15], which may improve the crawling time. However, the suitability of using such offline services strongly depends on the type of DPA and required metadata. For example, for the case studies presented in this paper, we found that GHTorrent and GHArchve lack part of the required metadata (e.g., information about build system and the number of contributors).

7.3 Scalability

NAB's analysis infrastructure has been designed with scalability as a driving principle, leveraging a container-based microservice architecture. We executed NAB in clusters varying the number of nodes (16, 32, 48, and 64) with a constant number of containers per node (16), observing close to linear scalability when testing the core analysis infrastructure, i.e., running a test project with fixed execution time and without any analysis plugin, thus confirming the low overhead of NAB's distributed infrastructure.

Although NAB features load-balancing mechanisms for its publish-subscribe communication infrastructure and for database accesses, during massive DPA (as in our case studies) we started observing some performance degradation with more than 1K NAB-Analyzer containers, when the number of messages exchanged for analysis coordination, result notification and result storing increases significantly. This is due to the limits reached by Docker Swarm's internal overlay network, which will be improved as Docker evolves. This issue can be mitigated by running several NAB deployments (i.e., running in separate Docker Swarms) that coordinate themselves using external MQTT brokers.

The optimal number of NAB-Analyzer instances to run on each host depends on the resource demands of the DPA tool to be executed. Even though Docker Swarm's scheduler tries to distribute the load fairly, it cannot distinguish the containers' roles and may stop and restart them without making any difference. While the restart of NAB services is handled by the fault-tolerance mechanism, it is important to avoid deploying critical NAB services (notably NAB-Master and the MQTT Brokers) on the same host where high-demanding NAB-Analyzer instances are deployed, so as to avoid performance penalties in running the analysis. NAB provides configurable deployment settings to properly place the core services.

7.4 DPA Reproducibility and Code Evolution

NAB can differentiate between different versions of a DPA (e.g., when an existing DPA is updated, it results in a new version). Moreover, NAB can apply an arbitrary DPA version to an arbitrary project revision. When a DPA completes, NAB stores provenance metadata, which identifies the DPA version and project revision. This metadata makes DPAs reproducible, as NAB allows one to re-execute a specific DPA version on a specific project revision even if they have been updated.

For version dependency management supporting wildcards, as in the case of Node.js, since many versions may match a given wildcard for a single dependency, it would be extremely costly (and thus impractical) to test each valid version for every dependency. NAB follows the default behavior of "`npm install`", which installs only the latest version of a dependent module matching the wildcard. Thus, provenance metadata includes the exact dependency version installed.

NAB enables the analysis of multiple revisions of the same project, which helps gain insight into the changing runtime behavior during project evolution. NAB supports this through incremental analysis: the user may request the analysis of only those project revisions that have not been analyzed yet, avoiding to re-analyze projects that have not changed since the last run of the same DPA. While such studies would significantly increase the number of analyses to run, NAB enables such computationally expensive analyses thanks to its distributed and scalable architecture, allowing the deployment of an analysis on a large cluster or in the Cloud.

Preserving provenance metadata is also useful to understand the evolution of a DPA. The user may apply the new version of a DPA to exactly the same project revisions analyzed previously (with an older version of the DPA). Exploring the differences between the analysis results can help identify bugs in the DPA implementation.

7.5 Limitations

In the following text, we outline the main limitations of our approach.

7.5.1 Low-level Metrics

One inherent limitation when running in a virtualized environment (which container technology is based on) is that low-level performance counters such as hardware performance counters [28] are restricted or not accessible. Thus, low-level dynamic metrics related to the CPU or memory subsystem cannot be collected when running the NAB components in Docker containers. Moreover, if multiple NAB components are deployed on the same machine, performance interference will prevent the collection of accurate time-based metrics.

This limitation is due to the trade-off between the safety offered by Docker containers and the flexibility of collecting arbitrary low-level metrics. While NAB also supports a deployment

setting without containerization, such a setting would sacrifice the safety properties needed when automatically executing the code of unknown (and hence untrusted) projects. Thus, for the measurements presented in this paper, we always use NAB with containerization.

7.5.2 Security Vulnerabilities and OS/Kernel Dependencies

NAB relies on Docker's default sandboxing (i.e., it does not use any privileged setting). However, DPA tools that require host's kernel system calls (e.g., the linux "`perf`" tool) will need to run NAB with privileged access (thus, potentially insecure). Furthermore, since the real host kernel is used by such tools, this setting will not work in virtualized environments. This is a well-known limitation of hardware-specific and OS-dependent profilers running on Docker.[13]

7.5.3 Representativeness of Testing Code

A general limitation of the presented use cases is that DPAs are applied only to the existing testing code of open-source projects. Such workloads may not be representative for a real usage scenario of an application in production. However, since testing code in projects also exercises library code extensively (as is the case of Node.js projects executing dependent NPM modules), significant information on the dynamic behavior of library code can be collected and analyzed. Thus, our approach ensures that the analyzed code stems from real applications and libraries, and the analyses presented in this paper yield relevant results. Moreover, in [69] the authors point out that many projects have relatively long-running testing code, different from simple and short unit tests.

As mentioned before, we will provide a plugin for JMH to analyze existing benchmarks in open-source projects, in addition to testing code. Furthermore, we plan to apply techniques for automated test-case generation [50, 59] to yield executable (and hence dynamically analyzable) code that maximizes various coverage metrics [42].

7.5.4 Analyzed Codebase and Analysis Timeout

The cluster used for obtaining our evaluation results requires a reservation and is heavily booked. For this reason, the projects considered in our use cases cover only the period 2013–2017. An extension of the use cases to cover also year 2018 is already scheduled. The choice of 1 hour as analysis timeout for DPAs also stems from the need of limiting the computational effort of the DPAs, to complete them within the limited timeframe of the cluster reservation.

8 Related Work

In this section, we provide an overview of the most significant related work. First, we review massive analysis of code repositories. Next, we discuss DPAs targeting JavaScript. Finally, we focus on previous approaches for generating benchmarks.

[13] `https://docs.docker.com/engine/security/seccomp/`

8.1 Massive Analysis of Code Repositories

Analyzing publicly available code repositories has become an important research area for understanding and improving different characteristics of software. Most related work relies on static analysis, notably for evaluating code quality [39, 52], predicting program properties [53], detecting code duplication [37], checking contracts in Java [11], identifying effects of software scale [38], automatically documenting code modifications [7], and summarizing bug reports [51]. Although static analyses can shed light on several aspects of software, there is a large body of properties that can be observed only when applications are executed [8], as DPA exposes the system's actual behavior. Our work facilitates the application of third-party DPAs to the projects in large code repositories.

Studies massively applying DPA are scarce. Legunsen et al. [35] use the JavaMOP [31] runtime-verification tool to check the correct usage of Java APIs. Even though the authors target 200 projects, their evaluation does not rely on any automated system enabling massive DPA. Marinescu et al. [42] present a framework to analyze how open-source projects evolve in terms of code, tests, and coverage by collecting both static and dynamic metrics. Although the authors apply DPA using containers to easily deploy several versions of the studied applications, their evaluation is limited to only 6 open-source projects. Overall, the aforementioned work dynamically analyzes a small and fixed set of projects, lacking an automatic and scalable system supporting massive custom DPA in the wild, as offered by NAB.

8.2 DPA for JavaScript

Dynamic analysis of JavaScript and Node.js applications is an active area of research. In [40] the authors introduce the notion of *promise graph*, a graph-based model to reason about the usage of JavaScript Promise objects through graphical visualization. Promise graphs are built using DPA, and can be used to identify bugs and API misuses in Node.js. A follow-up paper [1] from the same authors expands the work on promise graphs to perform automatic bug detection on real-world Node.js applications. The main focus of promise graphs is bug detection, while our DPA Deep-Promise focuses on the characterization of the Promise API usage and on asynchronous application behaviors (non-trivial promise chains) in the wild. To the best of our knowledge, no other large-scale study on the usage of the Promise API on Node.js projects and the NPM modules they depend on has been conducted.

Beyond JavaScript promises, DPA has been applied to JavaScript and Node.js in a variety of forms. As an example, JITProf [17] is a DPA tool that can identify JIT-unfriendly code patterns in JavaScript programs. JITProf lacks the ability to perform analyses on large code bases, and the JITProf paper evaluates only 50 client-side JavaScript applications. With NAB, we are able to scale analyses similar to those of JITProf up to a significantly higher number of JavaScript applications, enabling more representative results.

8.3 Benchmark Generation

Several studies focus on the creation of hand-coded synthetic benchmarks [10, 67], synthetic workload traces [49, 47, 13], and automatically synthesized benchmarks [3, 66, 32]. Overall, these techniques *generate* short workloads exhibiting a set of desired behaviors (e.g., intensive use of CPU, memory, I/O) to enable estimating and comparing the performance of hardware and applications. In contrast, our approach massively applies DPAs to *existing* testing code at the scale of public code repositories, as a technique for automatically discovering potential workload candidates satisfying domain-specific benchmarking needs.

To the best of our knowledge, NAB is the first system that can automatically run third-party DPAs in the wild. Similar in spirit, the AutoBench [69] toolchain can be used to look for potential benchmarks in Java workloads. In comparison to NAB, AutoBench lacks scalability, multi-language support, failure handling, sandboxing, and parallel code-repository crawling and analysis on clusters or in the Cloud. Moreover, AutoBench only supports MVN projects, relies exclusively on JUnit, and lacks any plugin mechanism for integrating third-party DPA tools that are fundamental for conducting analyses in the wild.

9 Conclusions

Motivated by the vast amount of today's public open-source code and available ready-to-use software components, this paper tackles two important research questions: whether it would be possible to develop a tool to automate large-scale DPA on public open-source software at a large scale, and whether such a tool would be useful for the community.

To positively answer the first question, we develop NAB, a novel, distributed infrastructure for executing massive custom DPA on open-source code repositories. NAB resorts to containerization for efficient DPA parallelization (fundamental to obtain analysis results in reasonable timeframes), sandboxing (to isolate buggy or malicious code) and for simplifying the deployment on clusters or in the Cloud. NAB features both crawler and analyzer components, which are deployed in lightweight containers that can be efficiently replicated. Moreover, NAB supports different build systems, testing frameworks, runtimes for multi-language support, and can easily integrate existing DPA tools. To the best of our knowledge, NAB is the first scalable, container-based infrastructure for automated, massive DPA on open-source projects, supporting multiple programming languages.

To positively answer the second question, we present three case studies where NAB enables massive DPA on more than 56K open-source projects hosted on GitHub, leveraging unit tests that can be automatically executed and analyzed. We present a novel analysis that sheds light on the usage of the Promise API in open-source Node.js projects. We find many projects with long promise chains, which can potentially be considered for benchmarking promises on Node.js. Moreover, the results of our analysis could be useful for Node.js developers to find projects and popular modules that use promises for asynchronous executions, which optimization could be beneficial to several existing applications. We conduct a large-scale study on the presence of JIT-unfriendly code on Node.js projects. Our study reveals that Node.js developers frequently use code patterns that could prevent or jeopardize dynamic optimizations and have a potential negative impact on applications performance. Finally, we perform a large-scale analysis on Java and Scala projects, searching for task-parallel workloads suitable for inclusion in a benchmark suite. We identify five candidate workloads (two in Java and three in Scala) that may be used for benchmarking task parallelism on the JVM.

Regarding ongoing research, we are exploring to which extent the testing code executed by NAB is representative for real-world usage scenarios of applications. We are applying automated test-case-generation techniques to increase the amount of dynamically analyzable code. Finally, we are also extending NAB to different repositories (including offline mirrors and datasets) and programming languages.

References

1 S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding Broken Promises in Asynchronous JavaScript Programs. *OOPSLA*, pages 162:1–162:26, 2018.
2 Atlassian. Bitbucket API. `https://developer.atlassian.com/bitbucket/api/2/reference/`, 2018.
3 R. Bell Jr and L. K. John. The Case for Automatic Synthesis of Miniature Benchmarks. In *MoBS*, pages 4–8, 2005.
4 D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
5 S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.
6 C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA*, pages 167–182, 2013.
7 R. P.L. Buse and W. R. Weimer. Automatically Documenting Program Changes. In *ASE*, pages 33–42, 2010.
8 B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.
9 Oracle Corporation. Java SE HotSpot at a Glance. `https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html`, 2018.
10 H. J. Curnow and Brian A. Wichmann. A Synthetic Benchmark. *Comput. J.*, 19:43–49, 1976.
11 J. Dietrich, D.J. Pearce, K. Jezek, and P. Brada. Contracts in the Wild: A Study of Java Programs. In *ECOOP*, pages 9:1–9:29, 2017.
12 Y. Ding, M. Zhou, Z. Zhao, S. Eisenstat, and X. Shen. Finding the Limit: Examining the Potential and Complexity of Compilation Scheduling for JIT-based Runtime Systems. In *ASPLOS*, pages 607–622, 2014.
13 L. Eeckhout, K. de Bosschere, and H. Neefs. Performance Analysis Through Synthetic Trace Generation. In *ISPASS*, pages 1–6, 2000.
14 International Organization for Standardization. ISO/IEC 20922:2016. `https://www.iso.org/standard/69466.html`, 2018.
15 GHArchive. Public GitHub Timeline and Archive. `https://www.gharchive.org/`, 2018.
16 GitLab. GitLab API. `https://docs.gitlab.com/ee/api/`, 2018.
17 L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *ESEC/FSE*, pages 357–368, 2015.
18 G. Gousios. The GHTorrent Dataset and Tool Suite. In *MSR*, pages 233–236, 2013.
19 HAProxy. HAProxy Community Edition. `http://www.haproxy.org/`, 2018.
20 B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI*, pages 295–308, 2011.
21 Docker Inc. Docker Compose. `https://docs.docker.com/compose/`, 2018.
22 Docker Inc. Docker Technology. `https://docs.docker.com/`, 2018.
23 Docker Inc. Swarm Mode Overview. `https://docs.docker.com/engine/swarm/`, 2018.
24 GitHub Inc. REST API v3. `https://developer.github.com/v3/search/`, 2018.
25 MongoDB Inc. mongos. `https://docs.mongodb.com/manual/reference/program/mongos/`, 2018.
26 MongoDB Inc. Scalable and Flexible document database. `https://www.mongodb.com/`, 2018.
27 MongoDB Inc. Sharding. `https://docs.mongodb.com/manual/sharding/`, 2018.
28 Innovative Computing Laboratory (ICL) - University of Tennessee. PAPI. `http://icl.utk.edu/papi/`, 2017.
29 ECMA International. ECMAScript 2015 Language Specification (ECMA-262 6th Edition). `https://www.ecma-international.org/ecma-262/6.0/`, 2015.

30 ECMA International. ECMAScript 2017 Language Specification (ECMA-262 8th Edition). https://www.ecma-international.org/ecma-262/8.0/, 2017.

31 D. Jin, P. O. N. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE*, pages 1427–1430, 2012.

32 A. Joshi, L. Eeckhout, and L. K. John. The Return of Synthetic Benchmarks. In *SPEC Benchmark Workshop*, pages 1–11, 2008.

33 E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The Promises and Perils of Mining GitHub. In *MSR*, pages 92–101, 2014.

34 G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

35 O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov. How Good Are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *ASE*, pages 602–613, 2016.

36 Lightbend, Inc. Cluster Specification. https://doc.akka.io/docs/akka/2.5/common/cluster.html, 2018.

37 C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. DéjàVu: A Map of Code Duplicates on GitHub. *OOPSLA*, pages 84:1–84:28, 2017.

38 C. V. Lopes and J. Ossher. How Scale Affects Structure in Java Programs. In *OOPSLA*, pages 675–694, 2015.

39 Y. Lu, X. Mao, Z. Li, Y. Zhang, T. Wang, and G. Yin. Does the Role Matter? An Investigation of the Code Quality of Casual Contributors in GitHub. In *APSEC*, pages 49–56, 2016.

40 M. Madsen, O. Lhoták, and F. Tip. A Model for Reasoning About JavaScript Promises. *OOPSLA*, pages 86:1–86:24, 2017.

41 L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *AOSD*, pages 239–250, 2012.

42 P. D. Marinescu, P. Hosek, and C. Cadar. COVRIG: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. In *ISSTA*, pages 93–104, 2014.

43 Mercurial. Mercurial Source Control Management. https://www.mercurial-scm.org/, 2018.

44 Microsoft. Microsoft Azure. https://azure.microsoft.com/en-us/, 2018.

45 A. Møller and M. I. Schwartzbach. Static Program Analysis, October 2018. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

46 R. Mudduluru and M. K. Ramanathan. Efficient Flow Profiling for Detecting Performance Bugs. In *ISSTA*, pages 413–424, 2016.

47 S. Nussbaum and J. E. Smith. Modeling Superscalar Processors via Statistical Simulation. In *PACT*, pages 15–24, 2001.

48 OpenJDK. JMH - Java Microbenchmark Harness. http://openjdk.java.net/projects/code-tools/jmh/, 2018.

49 M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Designs. In *ISCA*, pages 71–82, 2000.

50 C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, May 2007.

51 S. Rastkar, G. C. Murphy, and G. Murray. Summarizing Software Artifacts: A Case Study of Bug Reports. In *ICSE*, pages 505–514, 2010.

52 B. Ray, D. Posnett, P. Devanbu, and V. Filkov. A Large-scale Study of Programming Languages and Code Quality in GitHub. *CACM*, pages 91–100, 2017.

53 Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting Program Properties from "Big Code". In *POPL*, pages 111–124, 2015.

54 A. Rosà, E. Rosales, and W. Binder. Analyzing and Optimizing Task Granularity on the JVM. In *CGO*, pages 27–37, 2018.

55 O. Rudenko. Best Practices for Using Promises in JS. https://60devs.com/best-practices-for-using-promises-in-js.html, 2015.

56 Scala Benchmarking Project. ScalaBench. http://www.scalabench.org/, 2018.

57 K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE*, pages 488–498, 2013.

58 A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. DaCapo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *OOPSLA*, pages 657–676, 2011.

59 S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *ASE*, 2015, pages 201–211, 2015.

60 K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 Performance Characterization. In *SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, 2009.

61 L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *SC*, pages 6–16, 2001.

62 H. Sun, D. Bonetta, C. Humer, and W. Binder. Efficient Dynamic Analysis for Node.Js. In *CC*, pages 196–206, 2018.

63 The JUnit Team. JUnit. `https://junit.org`, 2018.

64 The DaCapo Benchmark Suite. DaCapo. `http://http://www.dacapobench.org/`, 2018.

65 The Netty project. Netty project. `https://netty.io/`, 2018.

66 L. Van Ertvelde and L. Eeckhout. Benchmark Synthesis for Architecture and Compiler Exploration. In *IISWC*, pages 1–11, 2010.

67 R. P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Commun. ACM*, 27(10):1013–1030, 1984.

68 T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *PLDI*, pages 662–676, 2017.

69 Y. Zheng, A. Rosà, L. Salucci, Y. Li, H. Sun, O. Javed, L. Bulej, L. Y. Chen, Z. Qi, and W. Binder. AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses. In *SANER*, pages 639–643, 2016.