JavaScript Performance Tuning as a Crowdsourced Service

Jie Ren, *Member, IEEE*, Ling Gao, and Zheng Wang, *Member, IEEE*

***Abstract*—JavaScript (JS) is one of the most used programming languages for mobile applications. As JS is increasingly used in computation-intensive and latency-sensitive components, JS application performance can significantly impact user experience. While compilers play a crucial role in optimizing JS performance on mobile systems, their optimizations must be simple due to the computation and battery usage limitations of the underlying hardware platforms. We present JSTuner, a machine-learning system to leverage compiler-based autotuning techniques to optimize JS performance by finding a good compiler optimization sequence. JSTuner is designed to reduce the cost of autotuning by using prior knowledge of JS programs collected through a crowdsourcing framework to bootstrap the search process. It allows the user to seamlessly utilize the computation resources of a cloud server to perform the heavy-lifting autotuning process for repeatedly running JS components. This enables aggressive search-based optimizations that are too expensive to run on the user's device. We evaluate JSTuner by applying it to 60 JS benchmarks across three distinct mobile devices and comparing it against four search-based techniques. Experimental results show that JSTuner consistently outperforms prior techniques and improves JS performance by 1.62x on average (up to 3.33x) over the default compiler setting used by the Chrome V8 JS engine.**

***Index Terms*—JavaScript optimization, mobile computing, machine learning, program autotuning, crowdsourcing.**

I. Introduction

JAVASCRIPT (JS) is one of the most popular programming languages [1]. It is widely used to develop mobile and web-based applications running on diverse computing hardware and operating system environments. Optimizing JS performance is essential for ensuring fast response time and a good user experience, which becomes increasingly important as JS is often used for computation-intensive and latency-sensitive tasks like animation, interactive maps and video playing.

A JS program is dynamically interpreted or compiled by a JS engine to run in the underlying computing environment. While performance optimization for JS code is crucial, the capability of today's JS engines is restricted by the computation resources of a mobile phone or embedded device. Due to the limited CPU power and memory capacity, they cannot apply aggressive analysis and code optimization strategies [2], which, while expensive, can significantly improve the performance of the code. As a result, many JS are poorly optimized, leading to poor user experience and energy inefficiency of computing systems.

Autotuning is a viable means of assisting compilers with code optimization [3], [4], [5], [6]. Instead of relying on hand-craft decision models to determine the best optimization options, autotuning techniques automatically search and model a typically large compiler optimization space to find a set of parameter settings to improve the performance of the compiled code. This technique can outperform hand-crafted compiler optimization strategies on a wide range of applications and tasks [6], [7], [8]. A key advantage of auto-tuning is that it can adapt to a wide range of computing environments as the technique makes no prior assumptions about the program and the underlying hardware.

This work exploits autotuning techniques to optimize JS programs for mobile and embedded devices. Auto-tuning permits us to target a wide range of JS engines and execution environments with little change to the JS compiler implementation. One of the key hurdles in applying autotuning to optimize JS code on resource-contradicted mobile systems is its expensive resource requirements and search time. A typical autotuning algorithm like generic search [9] and Bayesian optimization [10] require thousands or even hundreds of thousands of search iterations to find a good optimization setting [11], [12]. During this search process, the search algorithm needs to evaluate a (partially) transformed program by running the program on the target hardware to gather feedback (e.g., program execution time) to guide the search direction. Because running programs takes time, this search process can take hours or weeks on a single desktop machine, which is prohibitively expensive for optimizing JS code on a mobile device.

We present JSTuner, a machine-learning guided JS autotuning framework to speed up JS autotuning by reusing knowledge from prior program tuning and cloud computing. Our key insight is to map JS programs onto a carefully designed program feature space, where programs that are similar in the feature space can benefit from similar compiler optimization settings. This allows us to leverage the known good optimization configurations (e.g., the combination of `JS` compiler flags) of a previously seen JS program to speed up the search process of a *new*, *unseen* program.

Manuscript received 3 February 2023; revised 5 September 2023; accepted 13 September 2023. Date of publication 18 September 2023; date of current version 4 April 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 62372281, in part by Key Research and Development Program in Shaanxi Province under Grants 2023-YBGY-265 and 2023-YBGY-241 and in part by the Shaanxi Key Science and Technology Innovation Team Project under Grant 2022TD-26. Recommended for acceptance by X. Liu. *(Corresponding author: Jie Ren.)*

Jie Ren is with the School of Computer Science, Shaanxi Normal University, Xi'An, Shaanxi 710062, China (e-mail: renjie@snnu.edu.cn).

Ling Gao is with the School of Information Science and Technology, Northwest University, Xi'An, Shaanxi 710062, China (e-mail: gl@nwu.edu.cn).

Zheng Wang is with the School of Computing, University of Leeds, LS2 9JT Leeds, U.K. (e-mail: z.wang5@leeds.ac.uk).

Digital Object Identifier 10.1109/TMC.2023.3316167

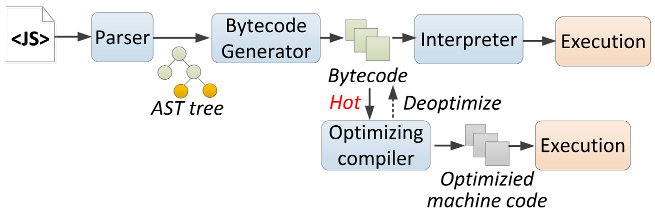


Fig. 1. Workflow of the Chrome V8 JS engine.

To mitigate the search overhead of autotuning, JSTUNER utilizes the high-performance computing power of cloud computing infrastructures. It supports the deployment of a search engine on a remote server to perform autotuning remotely to return the optimal JS compiler options to the end-user device to perform the optimization locally. Offloading the search computation to the cloud server speeds up the search process and reduces the battery power consumption on mobile devices for running the iterative autotuning process. On the remote server, JSTUNER utilizes the optimal setting found on similar JS benchmarks as the initial search point and skips poor configurations to accelerate the search process. Since JSTUNER does not alter the JS compiler implementation nor the user program, it can work with any JS engine and does not introduce new security vulnerabilities.

JSTUNER further incorporates a crowdsourcing module to collect data in real-time from various platforms, testing new compiler optimizations of target JS programs and learning how to optimize the unseen JS programs for new hardware platforms. This, in turn, allows us to use collective knowledge to improve our predictive model and the auto-tuning process. This permits us to periodically update the machine-learning-based decision model using JS program workloads collected through crowdsourcing. This ability for continuous learning improves the generalization ability of JSTUNER and enables it to adapt to the change in application workloads. The more JSTUNER, the better it knows what works.

We have implemented a working prototype of JSTUNER and integrated it with the Chrome V8 JS engine. We evaluate JSTUNER on 60 JS benchmarks using three mobile devices and one laptop. Experimental results show that JSTUNER achieves an average of 1.27x speedup by reusing the previously searched configuration. With the help of cloud computing, our approach performs a fine-tuning search and achieves a 1.62x speed up over the Chrome V8 default configuration used to compile and run the JS. It improves a state-of-the-art compiler tuning engine [13] by an average of 1.12x. In addition, we consider the cases for porting an existing model to different programs and environments. We show that JSTUNER provides portable performance but incurs significantly less training overhead over prior strategies.

Contributions: This paper is the first to:

- introduce a JS autotuning framework incorporating deep learning and crowdsourcing into the existing autotuner to provide a general, high quality and efficient JS optimization service specifically designed for mobile and embedded devices (Section III);
- show that the new coming, unseen JS can get benefit from prior JS tuning knowledge, which can help to skip the poor configuration to accelerate the search process (Sections IV and V);
- apply the crowdsourcing to gather JS real-time data, and speed up learning to address the predictive model portability issue across programs and devices (Section VI).

II. Background and Motivation

A. JS Engine

JS execution dominates the computation delays of mobile web browsers and mobile web-based applications [14]. JS is widely used to develop cross-platform mobile apps that seamlessly work across various devices and operating systems, and it is a popular choice for mobile app development, including games [15], augmented reality [16] and machine learning based tasks [17], such as object detection [18] and speech recognition [19].

Fig. 1 illustrates the working process of Chrome V8, a powerful and high-performance open-source JS engine developed by Google for use in their Chrome web browser. V8 is also popular for building web applications and running JS on the server side (using tools like Node.js [20]). To execute JS, V8 follows a number of steps: parsing, generating bytecodes, interpreting/optimizing compilation and execution. First, the input JS source code is parsed to construct an Abstract Syntax Tree (AST), where each node of the tree represents a token by breaking down the JS code into smaller units, such as variable type, value and kind. Next, the bytecode generator takes in the AST to produce the bytecode, which is then interpreted and executed by the interpreter on the target machine. At the same time, the interpreter collects profiling data. Frequently used bytecodes are sent to the optimizing compiler, which translates the bytecode into highly optimized machine code using techniques like hidden class optimization, inline caching, and hidden class transition elimination. In cases where optimization fails, the compiler de-optimizes codes and lets the interpreter executes the original bytecodes. Once the JS code has completed execution, the V8 engine frees up any memory that was used during the process, ensuring efficient memory usage.

B. Program Autotuning

Autotuning is a powerful technique that automatically fine-tunes program or system parameters to optimize performance. It is widely used in high-performance computing to improve speed and efficiency by adapting the program to the underlying hardware. However, autotuning can be resource-intensive, time-consuming, and demands substantial effort for implementation and maintenance. Typically, it leverages search techniques to explore the space of possible optimizations and identify the best-performing configuration for the target program on a specific platform [6], [13], [21]. Due to the large search space size, recent program autotuning integrates machine learning into the search technique [22], which proves effective for navigating a sizeable discrete space, outperforming traditional search techniques on a range of optimization tasks. Until now, several successful program autotuning frameworks have been developed for program optimization, such as OpenTuner [13], CompilerGym [21] and SuperSonic [6].

Since autotuning is traditionally performed offline, the search overhead is less of a concern than our scenarios of online autotuning on resource-limited mobile systems. JSTUNER is designed to make online autotuning practical on mobile systems through low-cost predictive modeling and leveraging the computation capabilities of cloud servers.

C. Problem Scope

While many of our optimization techniques can be ported to other computing systems used by PC and high-performance servers to speed up the autotuning process, this work specifically focuses on autotuning JS performance on mobile systems - an area that has received relatively little attention despite its significance.

The distinct nature of mobile systems presents two unique challenges that differentiate them from high-performance and desktop computing environments. First, mobile devices are characterized by their compact form factor and thermal limitations, which constrain the available computational resources and computing capabilities. These restrictions require optimization techniques that are mindful of resource consumption, ensuring that the performance improvements achieved are feasible within the mobile device's limitations. Second, unlike PCs and servers that benefit from a continuous power supply, mobile devices rely on battery power in most cases. Consequently, energy consumption becomes the first class constraint, and any optimization approach must prioritize lightweight methodologies with minimal computational overhead on the end-user device.

JSTUNER is designed to work with the aforementioned mobile device characteristics. By leveraging the computation power of a remote server through a lightweight predictive model running on the user device, JSTUNER ensures that the autotuning process remains efficient, imposing minimal impact on the end-user device while still delivering substantial performance enhancements.

D. Basic Ideas

Using autotuning techniques in JS can be an effective way to improve the performance of a program, especially in cases where the JS is running in environments with limited resources, such as smartphones and embedded systems. However, it is important to carefully consider the tradeoffs involved, as autotuning can be time-consuming. For example, Chrome V8 has over 250 options, and search spaces can be intractably large, up to 10^{158} possible configurations. It is a big challenge to find the right configuration for performance optimization in such a large search space. Additionally, popular websites and web-based applications frequently update the JS source code frequently to enhance user experience, further increasing the overhead of searching for the right configuration.

Our idea is to leverage prior JS tuning knowledge to guide autotuning. If the new incoming JS has a similar workload to a previously optimized program, we can use this knowledge to guide the search for the best configurations for the new program, thus greatly accelerating the JS autotuning process. Furthermore, to adapt to changing programs and runtime environments, we also develop a crowdsourcing system to collect profiling data to make our predictive model more robust. Our approach can complement existing autotuning techniques, like OpenTuner, and help JS developers to optimize JS performance on the target platform.

E. Motivation Examples

Setup: As a motivating example, consider running two typical JS programs (*pdfjs* and *box2d*) on three mobile devices (from high-end to low-end, Table IV lists details). *Pdfjs* is a JS-based PDF renderer supported by Mozilla labs [23]. It has been adopted as the default PDF viewer by many browsers, such as Firefox and Opera. *box2d* is a JS port of the Box2D Physics Engine [24], a well-tested library for developing popular games like Angry Birds [15]. The JS engine used in this work is the Chrome V8. We leverage OpenTuner (running on the cloud server) to search for the configuration that can provide the best performance for each JS program on the respective devices. The networking condition is WiFi 6 with an average upload speed of 1232 Mbps and download speed of 1543 Mbps.

Results: Fig. 2 presents the performance improvement achieved by the configuration found by OpenTuner against the V8 default setting. To find the best-performing configuration, we use OpenTuner to automatically search all the flags and parameters supported by V8. As we can see from this figure, the configurations found by OpenTuner give a noticeable performance improvement on all three platforms, which can reduce an average of 38.1% (71 ms), 25.7% (58.5 ms) and 36.5% (127 ms) latency on Pixel 6, Xiaomi 9 and Huawei P9, respectively. However, we apply the best-performing configuration of *pdfjs* to run *box2d* on the same device, leading to an average of 3.7% slowdown. Additionally, we test the compatibility of applying the same JS (JS) configuration across different devices. For example, using the optimal configuration of *box2d* on the Pixel 6 to run the same program on the Huawei P9 takes 313 milliseconds, which is less effective compared to using the best-performing configuration searched on the Huawei P9 (257 milliseconds). Specifically, we list part of the selected options of OpenTuner configuration for *box2d* and *pdfjs* across all platforms in Table I. We can see that the OpenTuner configuration of *pdfjs* prefers to disable the flag of *turbo-inlining* and *use-osr* on all three platforms. On the contrary, the *box2d* configuration remains consistent with the default settings. For garbage collection, both *box2d* and *pdfjs* enable the *parallel-marking* flag, which can reduce the garbage collection time. It is worth noting that the best-performing value of *min-semi-space-size*, *max-optimized-bytecode-size* and *max-inlined-bytecode-size* have great diversity. Therefore, simply using one optimal configuration discovered for one JS program on a specific device is likely to miss out on potential optimization opportunities.

F. Insights

The two examples show the enormous potential of performance optimization for JS engine on the mobile device. However, finding the right optimization configuration can be quite

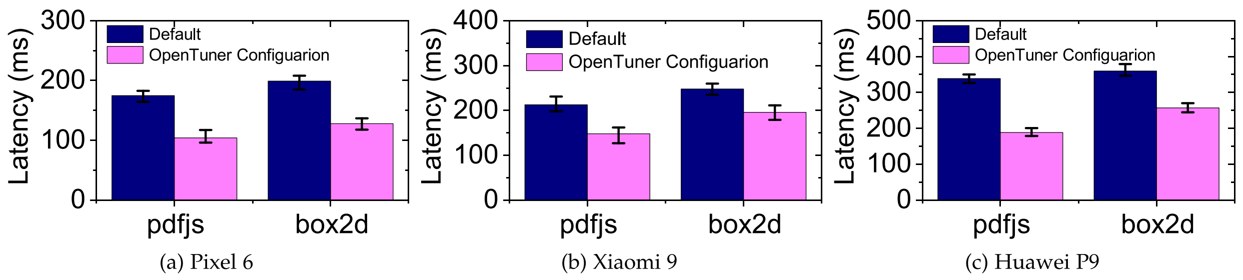


Fig. 2. Performance of two typical JS programs run on the Pixel 6 (a) Xiaomi 9 (b) and Huawei P9 (c) with the V8 default setting and the best-performing configuration found by the OpenTuner. We set the search time to 500 seconds. We can see that the OpenTuner configuration outperforms the V8 default setting.

TABLE I
PART OF THE SELECTED OPTIONS OF THE BEST PERFORMING CONFIGURATIONS FOUND BY THE OPENTUNER

option	Description	default	Pixel 6		Xiaomi 9		Huawei P9	
			box2d	pdfjs	box2d	pdfjs	box2d	pdfjs
turbo-inlining	enable inlining in TurboFan	on	on	off	on	off	on	off
use-osr	use on-stack replacement	on	on	off	on	off	on	off
minor-mc-sweeping	perform sweeping in young generation mark compact GCs	off	off	on	off	on	off	on
parallel-marking	use parallel marking in atomic pause	on	on	on	on	on	on	on
min-semi-space-size	min size of a semi-space (in MBytes)	0	36	68	41	62	28	43
max-optimized-bytecode-size	maximum bytecode size to be considered for optimization	61440	61094	122154	31679	113397	43763	60911
max-inlined-bytecode-size	maximum size of bytecode for a single inlining	460	852	306	847	585	103	72

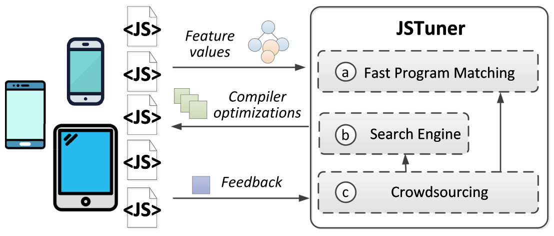


Fig. 3. Overview of JSTUNER.

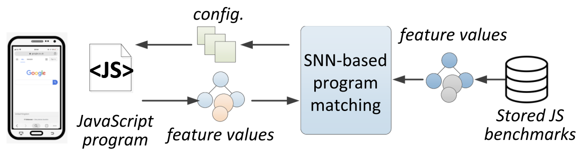


Fig. 4. Workflow of FPM.

tricky as it depends on the characteristics of the JS program and the performance of the hardware. Besides, the time required for searching the right configuration in a large search space is unacceptable for mobile JS. The following section will describe how our approach tackles these challenges by analyzing the program complexity and modeling the impact of the compile configuration on JS performance through predictive modeling.

III. OVERVIEW OF JSTUNER

Fig. 3 presents the high-level architecture of JSTUNER, a machine learning-guided JS autotuner. The core idea is to leverage the collective knowledge to deliver fast optimization service for the target JS. To achieve this, we first design a Fast Program Matching (termed as FPM) module to provide a fast and effective optimization for the new coming JS. Based on the idea that similar programs can benefit from the same configuration, the FPM calculates the similarity between the target JS and the stored benchmarks, and then compile and execute the target JS by using the previously found best-performing configuration of the matched JS benchmark. Moreover, to further improve the JS runtime performance, we employ a deep neural network-based search engine to perform a fine-grained optimization, and this process will find the right configuration within a large search space in a short time. Finally, we design a crowdsourcing module to collect the JS features on various types of devices with different compiler configurations and corresponding performance, which are used to update the deep neural network used in FPM and the search engine. As the number of JS codes and hardware platforms continues to grow, JSTUNER can test new compiler optimizations for their applications and learn how to optimize each JS for different mobile devices, resulting in faster JS execution.

A. Fast Program Matching

Fig. 4 illustrates the working process of Fast Program Matching (termed as FPM), designed to promptly deliver a coarse-grained optimization configuration. To this end, we characterize a set of selected benchmarks on mobile devices and document the best-performing configuration and its associated feature values from benchmark and hardware. The FPM leverages the Siamese Neural Network (SNN) [25] to predict the similarity between the incoming JS programs on specific hardware and the previously stored feature values. We assume that similar features of programs and hardware can benefit from the same compilation configuration. The FPM outputs a configuration from a stored set of previously characterized benchmarks with

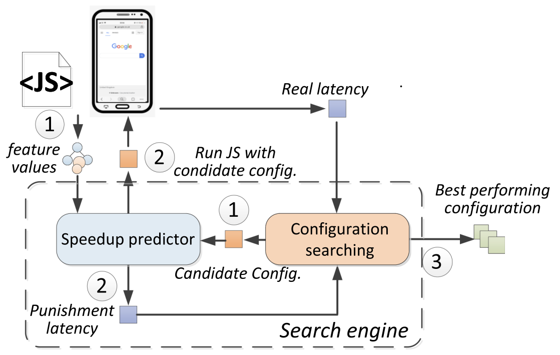


Fig. 5. Workflow of search engine.

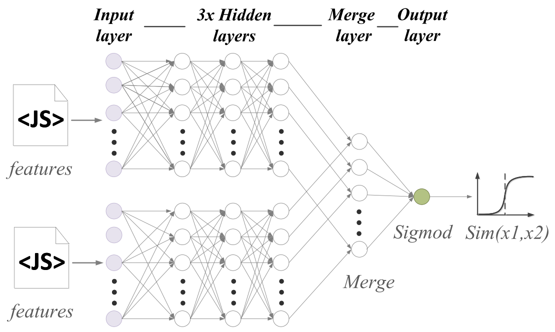


Fig. 6. Structure of our siamese neural network.

the highest similarity score and uses this configuration to run the new coming JS program. The program is executed using default settings when the similarity score falls below a specific threshold. Furthermore, we employ the autotuning technique to optimize the JS runtime performance for conducting a detailed search (refer to Section V).

B. Search Engine

Fig. 5 illustrates the working process of the search engine, which aims to find the best-performing configuration by fine-tuning the V8 support parameters and flags for the coming JS within a limited time. While there is a critical challenge related to this goal. How to speed up the fine-tuning process within a large search space. We integrate a speedup predictor into the OpenTuner, a generic tuning framework to address this issue. The predictor is a classification model that classifies the coming JS with candidate configurations into four speedup levels. For the configuration that may degrade the target JS runtime performance, we report back a high cost to the autotuner, thus reducing the device measurement cost. By giving up testing the configuration candidate that may slow down the performance, our search engine can significantly accelerate the tuning process and reduce the energy cost on the target platform needed for executing the tuning tests.

C. Crowdsourcing

To improve the generalizability of JSTUNER, we develop a crowdsourcing module that gathers JS runtime data from a variety of devices. This module evaluates the performance of different compilation configurations, updates the models, and provides users with customized compiler optimizations. For instance, when a new, unseen device requests the JS optimization service, the crowdsourcing module will run the eight representative benchmarks while the device is charging and report the dynamic and static characteristics of the benchmarks and the runtime performance. The profiling data is utilized to recalibrate the speedup predictor and SNN model. Additionally, we apply the conformal prediction technique to monitor the SNN performance and use the automated runtime measurements on incorrectly predicted inputs to improve the speedup model over time. In cases where the configuration provided by SNN degrades the performance, we will leverage OpenTuner to find the best-performing configuration at first, update the benchmark dataset and add a new class at the last layer of the SNN model. With the support of the crowdsourcing module, the predictive models used in JSTUNER are continually updated, ensuring optimization for newly encountered programs.

IV. Fast Program Matching

The FPM aims to deliver a quick optimization for any JS. By calculating the similarity between incoming JS features with the hardware characteristics and the pre-stored feature values, we use the previously best-performing configuration of the stored features with the highest similarity score to compile and execute the incoming JS. To do so, we employ the SNN to rank the similarity between the input features and the stored candidates and then output the configuration corresponding to the maximum similarity. We have evaluated several alternative modeling techniques, including KNN, SVM, Random Forest, and ANN. We chose the SNN because it extracts semantic similarity between the projected representation of the two input features, gives the best performance in our small dataset, and has a moderate training overhead (see Section VIII-F2). Our prototype is implemented using the Pytorch and scikit-learn machine learning package [26].

A. Problem Modeling and Training Data Generation

1) Model Structure: Fig. 6 depicts our SNN, including two parallel, fully connected, feed-forward ANN with 3 hidden layers and 42 nodes per hidden layer. The number of nodes in the input layer is determined by the dimensionality of the input features. This structure is automatically determined by applying the AutoML [27] tool on the training dataset. The two ANN subnetworks have the same architecture, parameters, and weights and any updates to the parameters are mirrored across both subnetworks. During training, the two identical feedforward neural networks read the two input features (Table II lists part of the essential features) and process their values through three hidden layers, and then feed into the merge layer to calculate the difference between the outputs of the two subnetworks by applying the torch.nn.abs() function. The result of the merge layer is then passed through a sigmoid activation, mapping the result onto the interval [0, 1]. Finally, we use the Binary

TABLE II
THE SELECTED FEATURES FOR SIAMESE NEURAL NETWORK

Feature	Description	Feature	Description
Cyclomatic complexity	Metric to measure the program complexity	TotalParseSize	Amount of parsed source code
Physical LOC	Physical lines of code	Cyclomatic complexity density	The avg. cyclomatic complexity for the function
Branches	# of branches	JS Size	Code size (B)
Functions	# of functions	AST depth	The depth of Abstract Syntax Tree (AST)
NumberOfSymbols	# of all symbols in a JS program	AST Nodes	# of AST nodes
CPU frequency	The maximum CPU frequency (GHz)	AllComparisons	# of comparision operators
RAM size	The size of RAM (GB)	Parameters	# of parameters
Cache size	The size of cache memory in SoC (MB)	Total Operators	# of operators
RAM frequency	RAM frequency (GHz)	Loops	# of loops
RAM channels	# of RAM channels in SoC.	TotalPreparseSkipped	Amount of code skipped over using preparsing
Halstead Volume	The product of program length and logarithm of vocabulary size		

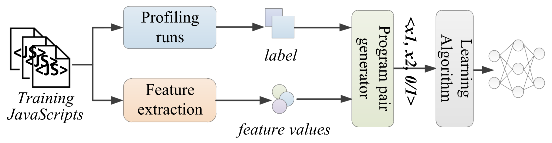


Fig. 7. Training process of SNN.

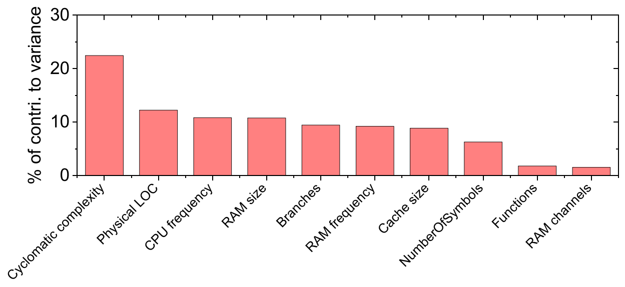


Fig. 8. Importance of top ten important features for siamese neural network.

CrossEntropy Loss as the loss function and update the weights through error back-propagation.

2) Training Data Generation: We apply cross-validation to train and test our models (see Section VI-C). Training data are generated by profiling 60 JS benchmarks on three mobile devices, where the benchmarks come from JetStream 2 (see Section VII-B for details). Fig. 7 depicts the process for learning the baseline model on our training data.

Profiling: First, we leverage OpenTuner [13] to search for the best-performing configuration for each JS benchmark on every mobile device. Since we have 60 JS benchmarks and three mobile devices, we obtained a total of 180 samples and 180 optimal configurations. Subsequently, we perform cross-testing using the best-performing configuration for each JS benchmark. Specifically, we exhaustively executed the 180 samples using the previously discovered 180 optimization configurations. Samples are classified into the same group when performance improvements are observed using the same configuration. Ultimately, we identify 8 configurations that cover all 180 samples, allowing us to categorize all benchmarks into 8 classes.

Feature: The SNN model is based exclusively on JS code, JS dynamic features, and hardware features. Code features are extracted from the JS source code, and dynamic features are collected using hardware performance counters during the initial profiling run of the target JS. We extract V8 dynamic features at runtime and collect code features by using complexity-report [28] and IstanbulJS [29]. The dynamic features, such as the number of Symbols, TotalParseSize, and TotalPreparseSkipped can be extracted during the JS parsing stage. The hardware features, such as CPU frequency, RAM size, and cache size, significantly impact JS runtime performance. Finally, we considered 35 raw features in this work. To learn effectively over a small training dataset, we apply the correlation coefficient (remove values above 0.75) and principal component analysis [30] to reduce the dimensionality of raw features from 35 to 21. Table II lists the features used for training. We also scale each of the extracted feature values to a common range (between 0 and 1) to prevent the range of any single feature from being a factor in its importance. We record the minimum and maximum values of each feature in the training dataset to scale the feature values of an unseen JS. We also truncate a feature value to make sure it is within the expected range during deployment.

B. Features Used in Fast Program Matching

To understand the usefulness of each feature, we apply a factor analysis technique called Varimax rotation [31] to the feature space transformed by the principal component analysis (PCA). This technique quantifies the contribution of each feature to the overall variance in each of the PCA dimensions. Intuitively, the more variances a feature brings to the space, the more useful information the feature carries. Fig. 8 shows the top 10 features chosen for the SNN model. These include static code features, such as Cyclomatic complexity and Physical LOC, as well as hardware features cache size, RAM size, RAM frequency, and RAM channels. Cyclomatic complexity indicates the complexity of a JS code, and Physical LOC is the lines of code. The hardware feature cache size influences the number of cache hits during compilation, the RAM size affects the available memory size for the heap memory and stack memory, and the RAM frequency relates directly to the rate of reading/writing from/in the RAM memory. Additionally, the number of RAM channels integrated into the SoC directly impacts data transfer rates.

C. Building the Model

We first combine the collected code feature, dynamic feature and hardware feature values with a label for each training sample, where each label represents a class that the same configuration can optimize. The training process requires building input pairs for the same and different class samples. Then we traverse the training dataset, combine two samples arbitrarily, check their labels, and set the new label as 1 if they belong to the same class and 0 otherwise, termed as $< x1, x2, 0/1 >$. This results in a new training set of over thirty thousand paired samples. Finally, we feed the training samples into the supervised learning algorithm, and the output of our learning algorithm is an SNN model, where the weights of the model are determined from the training data. Model parameter tuning is performed on the training dataset for each targeting hardware architecture using cross-validation (see also Section VII). Since training is performed only once "at the factory", this is a one-off cost.

D. Runtime Deployment

Once we have built and trained our SNN as previously described, we can use it to find the best matching stored feature and then output the corresponding optimization configuration for the new program. Specifically, we first extract static code features, hardware characteristics, and dynamic features as soon as the JS code is downloaded onto the user's device. Next, we upload the collected features to the server and combine them with the pre-stored features of each class (8 classes in this work) to form a paired sample and feed them into the SNN to determine similarity. The SNN predicts whether input features are similar or dissimilar and output the class label with the highest similarity score. Finally, the selected configurations are sent to the mobile device to compile and execute the JS on the specific platforms (Section VIII-F2 shows the cost details). Considering the potential variations in networking conditions and inference costs, the FPM will first use the default configuration to execute a newly arrived JS program so that there is no delay in executing the code. However, for subsequent executions of the same JS program, the FPM utilizes SNN prediction to optimize the program. Since a JS program (or function) is invoked many times by an application or webpage, our techniques can be advantageous for repeatedly executed programs. Repeated execution is a common pattern for JS programs, examples of such patterns include JS scripts periodically checking for live updates, like social media feeds or chat messages, using polling mechanisms or WebSockets for real-time communication like video streams, computing gaming logic, or rendering graphics. Moreover, web applications use event-driven programming to respond to user interactions and system events. Events like clicks, form submissions, keyboard input, and timers trigger code execution in response. Each time these events occur, the associated code is executed, leading to repeated execution. For JS program that only gets executed once on a webpage loading, our approach can store the tuning results in a local cache to be used the next time the same page is loaded.

E. Continuous Adaptation

The FPM aims to promptly provide configurations for optimizing JS performance. This is achieved through the utilization of a lightweight SNN model that is trained to effectively extract key features from incoming JS code and categorize them into 8 distinct classes. Each class corresponds to a specific compiler configuration that enhances performance. As the JS compiler implementation evolves or new types of JS workloads emerge, we may need to expand the number of compiler option classes. JSTuner is designed to support the continuous update of the decision model to allow FPM adapt to changes in the application landscape. To facilitate this adaptability, JSTuner incorporates a crowdsourcing framework that allows the underlying prediction model to evolve over time. For instance, the decision model can be periodically retrained using new application workloads gathered from diverse users, where our methodology for learning program features and model training can remain unchanged.

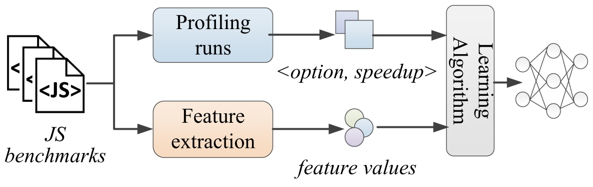


Fig. 9. Training process of speedup predictor.

V. Search Engine

Our search engine aims to find the best-performing configuration within a large search space. To this end, we build a speedup predictor to classify the coming JS with the candidate configurations into four levels of speedup, ranging from 0 (delivers over 1.2x speedup) to 3 (gives over 1.2x slowdown). We then integrate the predictor with existing autotuning techniques like OpenTuner, genetic algorithm, greedy algorithm, etc. By utilizing the predictor outputs, the autotuner can significantly speed up the search process (see Section VIII-D). In this section, we describe how to employ machine learning to train the speedup predictor to guide the tuning process.

A. Speedup Modeling

The central component of our search engine is the speedup predictor, built upon a Multi-layer Perceptron (MLP) Artificial Neural Network (ANN). We selected ANNs as they give superior and more consistent performance compared to alternative methods (as discussed in Section VIII-F3). Additionally, using ANNs allows us to employ transfer learning, which reduces the training overhead in the deployment environment.

1) Training the Speedup Predictor: Fig. 9 depicts the process of building the speedup predictor. To learn a classification model, we begin by profiling the runtime performance of JS using a large number of candidate configurations. We then calculate the speedup factor over the default configuration (V8 default setting) for each configuration and classify the speedup values into four levels. Finally, we use the configurations, JS feature values and speedup levels to train the classification model.

Generating Training Data: We apply cross-validation to train and test our models (see Section VII). The training data is generated by profiling JS using various configurations. In detail, we use OpenTuner to search for the optimal performance configuration for compiling and executing a particular JS on the

TABLE III
TOP 15 IMPORTANT COMPILER OPTIONS FOR SPEEDUP PREDICTOR

Feature	Description
max-optimized-bytecode-size	maximum bytecode size to be considered for optimization
turbo-inlining	enable inlining in TurboFan
inline-new	use fast inline allocation
stack-size	default size of stack region v8 is allowed to use
min-semi-space-size	min size of a semi-space (in MBytes), the new space consists of two semi-spaces
scavenge-task-trigger	Scavenge task trigger in percent of the current heap limit
compact-on-every-full-gc	perform compaction on every full GC
aseline-batch-compilation-threshold	the estimated instruction size of a batch to trigger compilation
always-sparkplug	directly tier up to Sparkplug code
minor-mc-sweeping	perform sweeping in young generation mark compact GCs
bytecode-size-allowance-per-tick	increases the number of ticks required for optimization by bytecode.length/X
flush-baseline-code	flush of baseline code when it has not been executed recently
max-inlined-bytecode-size	maximum size of bytecode for a single inlining
page-promotion-threshold	min percentage of live bytes on a page to enable fast evacuation
lazy-new-space-shrinking	Enables the lazy new space shrinking strategy

target platform, while simultaneously recording the JS performance with the corresponding tuning configuration generated by OpenTuner during the search process. However, performing a full, exhaustive search of such a large space (up to 10^{158} possible configurations) is not feasible due to the excessive overhead it would incur. Therefore, we train eight speedup predictors for eight FPM clusters, each of which is trained on over 240,000 (3 devices $\times$ $\sim$8 benchmarks $\times$ 10 thousand samples) automatically generated training samples.

Building the Model: Each evaluated configuration is appended to the JS and hardware feature values to form input for the model. The model inputs and the corresponding speedup levels for all JS training benchmarks used for training are passed to the learning algorithm. The algorithm finds a correlation between the input vector and the desired prediction. The output of our learning algorithm is an MLP model, whose weights are determined based on the training data. Using cross-validation, we fine-tune the model parameters on the training dataset for each target hardware architecture. In our case, the overall training process for all training benchmarks takes two weeks on a single machine (dominated by training data generation). Since training is performed only once "at the factory", this is a one-off cost.

2) Features: Our speedup models rely on code, dynamic features of the target programs and candidate configurations. Code features are extracted from the JS source code, and dynamic features are obtained during the initial profiling run of the target application (see also Section IV-A2), configurations are built upon the V8-supported flags and parameters for ARM.

Feature selection from compiler options: To build an accurate model through supervised learning, the training sample size typically needs at least one order of magnitude greater than the number of features. In this work, we start from 241 raw features, including 21 FPM features (see Section IV-A2) and 220 compiler options. Our process for feature selection is fully automatic, described as follows. First, we calculate the impact of the V8-supported options one by one by using the equation below:

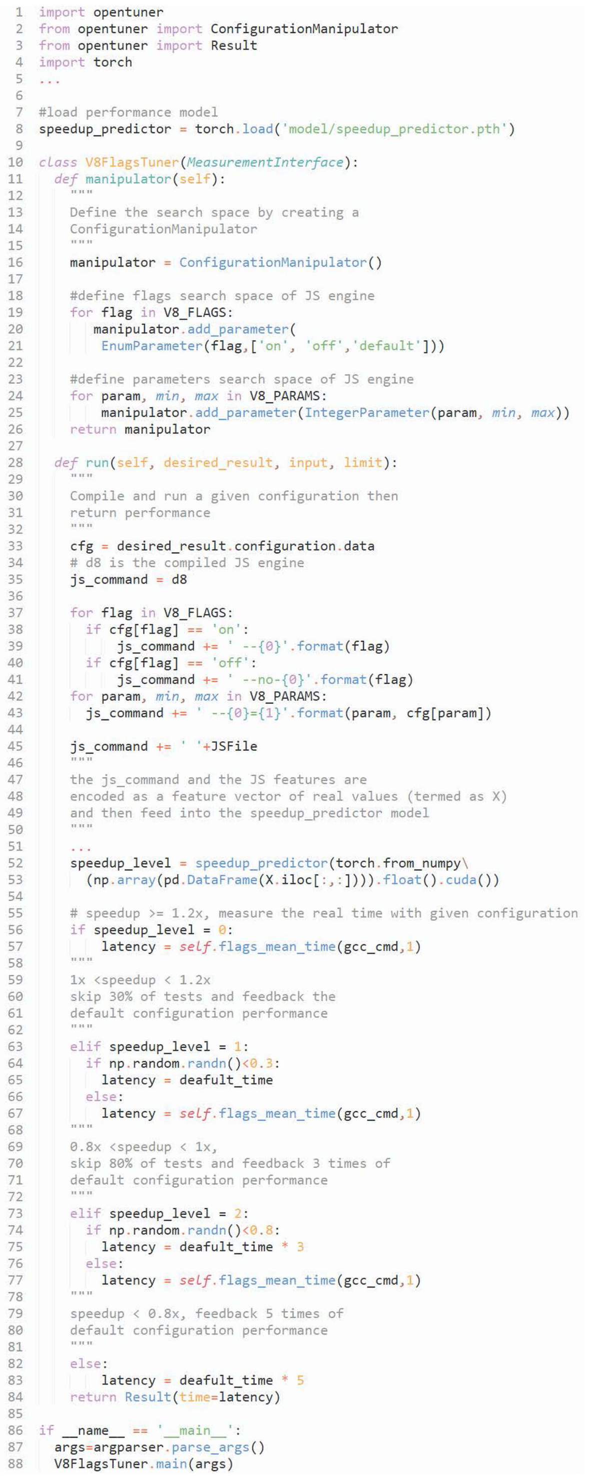
$$Impact = \frac{|runtime - runtime_{baseline}|}{runtime_{baseline}}$$

where $runtime_{baseline}$ is the JS performance under the V8 default setting, and $runtime$ is a measurement of the JS performance when removing one flag. We then calculate the impact variance of each option for all programs in the training set. Next, we eliminate the flags or parameters that consistently improve performance for the whole JS benchmarks on all three platforms, such as *–use-ic*, *–inline-new*, *–sparkplug*, *–no-jitless*. This reduces the number of V8-supported options to 181, with 21 FPM features (as discussed in Section IV-A2). We use 202 features to predict the speedup levels of the incoming program with different configurations. Table III lists the top 15 most important V8-supported options for the speedup predictor. Since our approach for feature selection is automated, it can be applied to other sets of candidate features. It is important to note that feature selection is also performed using cross-validation.

Feature normalization: In the final step, we scale each of the extracted feature values to a common range (between 0 and 1) to prevent the range of any single feature being a factor in its importance. We record the minimum and maximum values of each feature in the training dataset, in order to scale the feature values of an unseen JS and configuration. We also clip a feature value to make sure it is within the expected range during deployment.

B. Runtime Deployment

Once we have built and trained our speedup predictor described above, we can use it as a cost function to search for the best-performing configuration for any new JS. Fig. 10 shows the simplified code example of using our search engine to search over the space of the V8-supported options on ARM. First, we load the speedup predictor built upon the PyTorch framework (line 8). Next, the selected configuration of the autotuner in each tuning iteration and the FPM features are encoded as a feature vector of real values (from line 46 to line 51). Then, feeding the feature vector into the speedup predictor to estimate the speedup level (line 52). When the speedup_level is zero ($speedup \geq 1.2x$), which represents the given configuration may significantly improve the performance over the default setting, the search engine will measure the latency in the target device(line 56 to line 57). When the speedup predictor outputs 1 ($speedup \geq 1x$ and $speedup < 1.2x$), the search engine has a 30% chance of feeding back the default latency directly (line 63 to line 67). Furthermore, when the speedup_level is 2 ($speedup \geq 0.8x$ and $speedup < 1x$), the search engine has



```
 1 import opentuner
 2 from opentuner import ConfigurationManipulator
 3 from opentuner import Result
 4 import torch
 5 ...
 6
 7 #load performance model
 8 speedup_predictor = torch.load('model/speedup_predictor.pth')
 9
10 class V8FlagsTuner(MeasurementInterface):
11   def manipulator(self):
12     """
13     Define the search space by creating a
14     ConfigurationManipulator
15     """
16     manipulator = ConfigurationManipulator()
17
18     #define flags search space of JS engine
19     for flag in V8_FLAGS:
20        manipulator.add_parameter(
21         EnumParameter(flag,['on', 'off','default']))
22
23     #define parameters search space of JS engine
24     for param, min, max in V8_PARAMS:
25         manipulator.add_parameter(IntegerParameter(param, min, max))
26     return manipulator
27
28   def run(self, desired_result, input, limit):
29     """
30     Compile and run a given configuration then
31     return performance
32     """
33     cfg = desired_result.configuration.data
34     # d8 is the compiled JS engine
35     js_command = d8
36
37     for flag in V8_FLAGS:
38       if cfg[flag] == 'on':
39           js_command += ' --{0}'.format(flag)
40       if cfg[flag] == 'off':
41           js_command += ' --no-{0}'.format(flag)
42     for param, min, max in V8_PARAMS:
43       js_command += ' --{0}={1}'.format(param, cfg[param])
44
45     js_command += ' '+JSFile
46     """
47     the js_command and the JS features are
48     encoded as a feature vector of real values (termed as X)
49     and then feed into the speedup_predictor model
50     """
51     ...
52     speedup_level = speedup_predictor(torch.from_numpy\
53       (np.array(pd.DataFrame(X.iloc[:,:]))).float().cuda())
54
55     # speedup >= 1.2x, measure the real time with given configuration
56     if speedup_level = 0:
57         latency = self.flags_mean_time(gcc_cmd,1)
58     """
59     1x <speedup < 1.2x
60     skip 30% of tests and feedback the
61     default configuration performance
62     """
63     elif speedup_level = 1:
64       if np.random.randn()<0.3:
65         latency = deafult_time
66       else:
67         latency = self.flags_mean_time(gcc_cmd,1)
68     """
69     0.8x <speedup < 1x,
70     skip 80% of tests and feedback 3 times of
71     default configuration performance
72     """
73     elif speedup_level = 2:
74       if np.random.randn()<0.8:
75         latency = deafult_time * 3
76       else:
77         latency = self.flags_mean_time(gcc_cmd,1)
78     """
79     speedup < 0.8x, feedback 5 times of
80     default configuration performance
81     """
82     else:
83         latency = deafult_time * 5
84     return Result(time=latency)
85
86 if __name__ == '__main__':
87   args=argparser.parse_args()
88   V8FlagsTuner.main(args)
```

Fig. 10. Simplified code for our search engine.

an 80% of feeding back three times of default latency (line 73 to line 77). When the speedup_level is 3 ($speedup < 0.8x$), it returns five times of default latency directly (line 82 to line 83). With the help of the speedup predictor, the search engine can perform more tests and reach convergence more quickly (see Section VIII-D). We use the search engine to perform aggressive, fine-grained searches to find a good compiler configuration for the target JS programs on specific hardware architectures offline. We store the best-performing configuration for future reference or use it to extend the speedup classes for FPM. Since the search engine is not designed to provide real-time optimization and the search is performed offline, this overhead is excluded from the runtime evaluation of JSTUNER.

We also attempt to construct a regression model for the speedup factor, where the predictor enables the autotuner to receive a specific speedup value for a given configuration, eliminating the need for running tests on real devices. However, it is important to note that the input features from the configuration may contain conflicting compiler options. For instance, the heap management-related options like –max-heap-size, –max-old-space-size, and –max-semi-space-size cannot be specified simultaneously, as doing so would lead to a memory trap. Pruning these options may be a solution to avoid conflicts, but it may overlook non-intuitive, yet effective configurations. Despite this, the regression model may deliver a wrong prediction, such as giving a speedup value above 1x, which can lead the autotuner to explore an invalid search space. Our search engine is designed to leverage the classification model for predicting the speedup level. If the predicted speedup result is above 1x, the given configuration is also evaluated on the real device. Skipping such configurations may likely result in a slowdown of JS performance. Our experiment results show that the search engine outperforms the other SOTA methods for the averaged performance across test benchmarks.

VI. CROWDSOURCING

To maintain the effectiveness of JSTUNER with the growing number of JS and the increasing diversity of the available hardware devices, we design the crowdsourcing module to continuously collect the incoming data, and retrain our predictive model if we find that the data distribution has deviated significantly from the original training data distribution. However, there are two critical challenges we need to consider when we design the crowdsourcing module. The first challenge is how to efficiently incorporate new data into the FPM service scope, as it is important to keep our predictive model up-to-date and aligned with the latest JS and hardware devices. The second challenge is how to minimize the training overhead across different JS and platforms, as retraining the predictive model on new data can be computationally expensive and time-consuming.

A. *Adaptive Learning for FPM*

There are two scenarios in which the input features may not match all classes of the SNN, resulting in a similarity score below 0.5. First, it is possible that the SNN has not learned the specific data distribution, although the previously searched configuration can optimize the upcoming JS. In this situation, we can address the issue by combining the new data with existing data from the same class and retraining the SNN. This process ensures that the new data is placed close to the existing data from its class, improving the model's performance.

Second, there may be cases where the new data does not benefit from any existing configurations. In such scenarios, we

TABLE IV
EVALUATION PLATFORMS

Device	CPU	GPU	RAM (GB)	OS
Pixel 6	Google Tensor @ 2.8 GHz	Mali-G78 MP20	8	Android 13
XiaoMi 9	Snapdragon 855 @ 2.84 GHz	Adreno 640	8	Android 9
Huawei P9	Kirin 955 @ 2.5 GHz	Mali T880	3	Android 8

can extend the baseline model by adding a new class and apply transfer learning to update the SNN model. Transfer learning involves utilizing the newly observed programs to update the hidden layers of an already trained SNN. This approach allows the model to generate embeddings for the new class that are distinct from the existing classes, as discussed in Section VIII-E1.

Since the SNN is hosted on a crowdsourced server, the service provider has the capability to retrain the model from scratch using data collected on new JS programs with the new compiler optimization sequence. This ensures that our system, JSTUNER, becomes increasingly effective and adaptable to changes in application workloads over time.

B. Transfer Learning of Search Engine

When we need to fine-tune the coming JS, we apply the DNN-based search engine to search for the best-performing configuration. During the search process, we measure the runtime performance on the device, calculate the speedup factor, and then compare it with the predicted speedup levels. If the variance is greater than 10%, we perform transfer learning on the current speedup predictor. This is done by automatically running the coming JS with 40 configurations generated by OpenTuner on the target device. The JS features, hardware features and configuration with the corresponding speedup levels are then combined as the transfer learning dataset. When the device is charging, JSTUNER starts collecting profiling data and runs the learning algorithm to update the predictors. which freezes the first two layers (Section VIII-E2 gives the experimental details).

VII. EVALUATION SETUP

A. Evaluation Platforms

We use Google V8 (ver. 10.2.0) as our JS engine and compile it for ARM to enable evaluation on mobile devices. Table IV gives details of our hardware platforms. For training, we use a cloud server with a 10-core 3.7GHz Intel i9-10900X CPU and two NVIDIA RTX 3090 GPUs. Our mobile test board and the server communicate through WiFi-6 using Huawei AX6 remoter, but we use Netem [32] to control the network delay and server bandwidth (with an upload bandwidth of 1232 Mbps and download bandwidth of 1543 Mbps). Considering the rapid development of network technology, we only add 5% of variances (which follow a normal distribution) to the bandwidths and delay to simulate a dynamic network environment. Our predictive models build on PyTorch v1.2.

B. JS Workloads

We use the JetStream 2 [33] benchmark suite, which includes a range of JS benchmarks such as SunSpider, Octane 2, ARES-6, and Web Tooling programs. These benchmarks are often used as a standardized measure for evaluating the performance of JavaScript applications. We exclude the JavaScript-based machine learning benchmarks and Web Assembly benchmarks in this work.

Fig. 11 shows the cumulative distribution function (CDF) of their JS size, cyclomatic complexity, and runtime performance for our selected JS benchmarks. As we can see from this figure, the JS size and cyclomatic complexity range from small (204 B and 1) to large (over 11 MB and 3972), indicating that our test data cover a diverse set of JS programs. Moreover, Fig. 11(c) presents the JS runtime performance on three representative mobile platforms, the latency range from 0.03 ms to 2159 ms, representing a great diversity of runtime performance of our benchmarks on different platforms.

C. Competitive Approache

We compare our FPM and search engine against the following widely used autotuning techniques:

OpenTuner [13] is a framework for program autotuning, which is the process of automatically finding the optimal configuration of a program for a specific platform and input. It is designed to efficiently explore the large and often high-dimensional search space of possible program configurations. This is achieved through the use of intelligent search algorithms, such as Bayesian optimization and genetic algorithms, that can guide the exploration towards the most promising regions of the search space.

GGA [34] is a modified genetic algorithm for solving constrained optimization problems. It combines a genetic algorithm for global search with a gradient-based local search for fine-tuning the solutions found. This combination allows GGA to effectively explore the search space while ensuring that the solutions found are close to the global optimum.

Greedy [35] is a type of optimization algorithm that makes choice at each step based on a specific criterion, aiming to select the best immediate option. Its objective is to find a globally optimal solution by iteratively making locally optimal choices.

Random [36] search algorithm utilizes randomness or probability in its definition to achieve optimal solutions. This algorithm demonstrates the capability to efficiently solve large-scale problems and has proven useful for a wide range of ill-structured global optimization problems involving continuous and/or discrete variables.

D. Model Evaluation

Like [37], we use five-fold cross-validation to train all machine learning models. Specifically, we randomly partition our 60 JS benchmarks into 5 sets, each containing 12 programs. We use one set as the validation data for testing our model and the remaining 4 sets as training data to learn a model. In each iteration, we exclude one set for predictions from the training

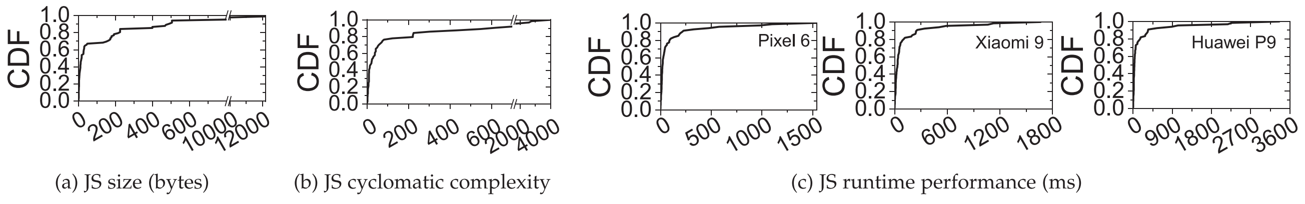


Fig. 11. CDF of JS program size (a) cyclomatic complexity (b) and runtime performance (c).

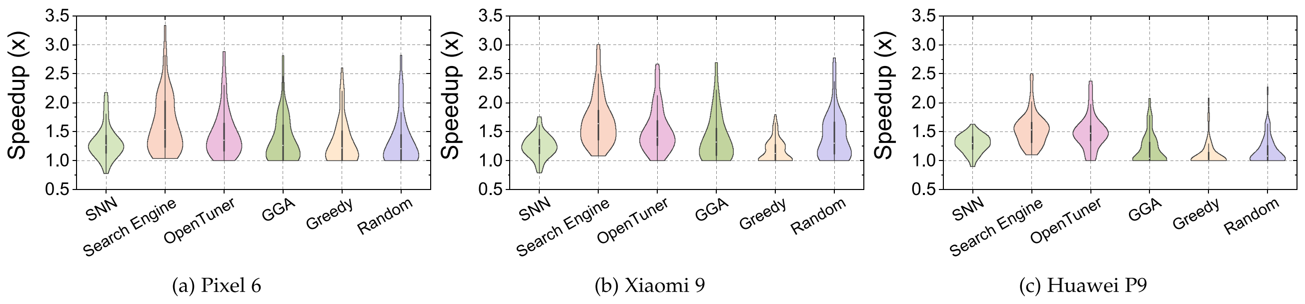


Fig. 12. JS speedup over the V8 default compilation configuration on three representative mobile devices. Search engine outperforms alternative methods for the averaged performance across test benchmarks.

program set, and learn a model using the remaining 4 sets. This process is repeated five times (folds), ensuring that each of the 5 sets is used exactly once as the validation data.

E. Reporting

We run each program under a configuration multiple times and report the geometric mean of the runtime. The geometric mean is widely seen as a more reliable performance metric over the arithmetic mean [38]. Moreover, configurations are automatically generated by OpenTuner. To have statistically sound data, we run each approach on a test case repeatedly until the confidence-bound under a 95% confidence interval is smaller than 5%. In our experiments, we set three levels for the search time limit, 50 seconds, 100 seconds and 200 seconds, as our experimental results show that our approach can get good enough speedup for all the benchmarks within 200 seconds (see Section VIII-D), and the benchmark can choose different search time limit depending on the runtime performance.

VIII. Experimental Results

A. Overall Performance

We compare our approach against OpenTuner, GGA, Greedy and Random search techniques for performance improvement. We set the search time to 200 seconds for all search algorithms. Fig. 12 compares the speedup performance of all strategies. In this experiment, we exclude the cost feature extraction, data transfer and model inference from measuring the speedup of SNN. This evaluation allows us to determine the maximum achievable performance of FPM when the JS needs to be executed multiple times. Later in Section VIII-F2, we show the end-to-end performance of FPM by including all the overhead associated with feature extraction, data transfer and model inference.

The diagram shows that SNN improves runtime performance compared to the default setting across test benchmarks, with average improvements of 1.31x, 1.24x, and 1.28x on Pixel 6, Xiaomi 9, and Huawei P9, respectively. Although there are ∼8% programs (4-6 of 60) that perform worse than the default settings because the SNN gives the wrong matching configuration, the SNN still delivers an average of 89% performance of default settings for these cases. As we expected, because of the high performance of our predictive model, the search engine improves OpenTuner by an average of 1.12x (up to 2.51x), improving the default configuration by an average of 1.63x, 1.69x, and 1.54x on Pixel 6, Xiaomi 9 and Huawei P9, respectively. The other three search techniques: GGA, Greedy and Random give an average speedup of 1.31x, 1.20x, and 1.30x over the default setting.

Specifically, Fig. 13 presents the speedup details for all 60 JS benchmarks on the Pixel 6. We observe that the SNN makes the wrong prediction on zlib, Air, sjlc, and bitops-bits-in-byte. The search engine improves the OpenTuner on 91.6% of benchmarks (55 of 60) and achieves 96.2% performance of OpenTuner on the rest of the benchmarks. On the other hand, the search engine improves the GGA, Greedy and Random techniques with an average of 1.29x, 1.32x and 1.35x speedup. In detail, the GGA, Greedy and Random deliver the optimization configuration on 75.0%, 73.3% and 76.3% of benchmarks, and we apply default configuration for the rest of the benchmarks.

B. Compare to the Extended Search Techniques

We extend the GGA, Greedy and Random by leveraging our speedup predictor, termed as GGA-Speedup, Greedy-Speedup and Random-Speedup. In detail, we integrate the speedup

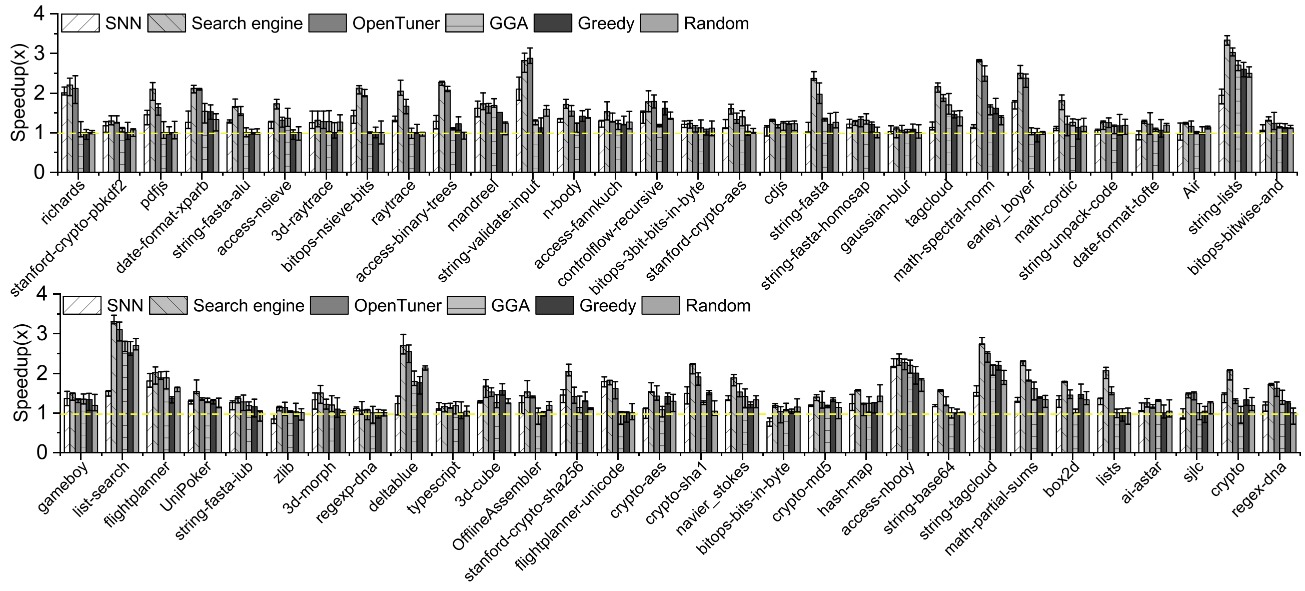


Fig. 13. Performance speedup on Pixel 6.

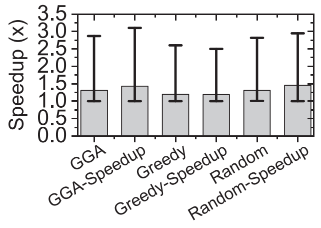


Fig. 14. Performance of extended search techniques on Pixel 6.

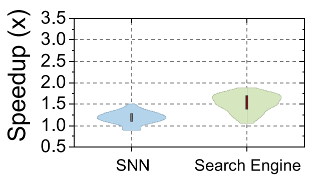


Fig. 15. JSTuner on laptop.

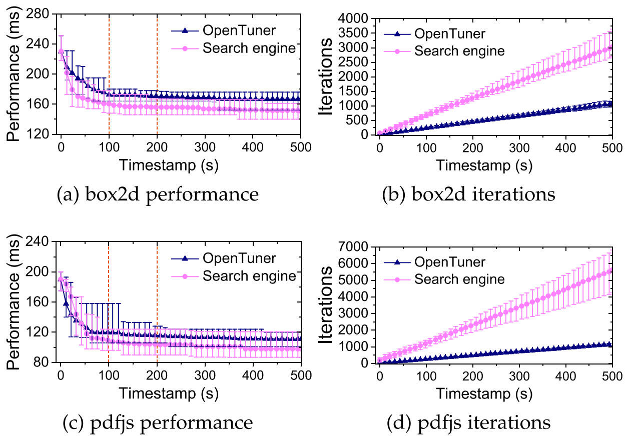


Fig. 16. Aggregated performance of 10 runs of OpenTuner and our search engine for box2d (a) and pdfjs(c) on Pixel 6 within 500, error bars indicate the maximum and minimum value. Our approach speeds up the convergence and improves the performance achieved by the OpenTuner.

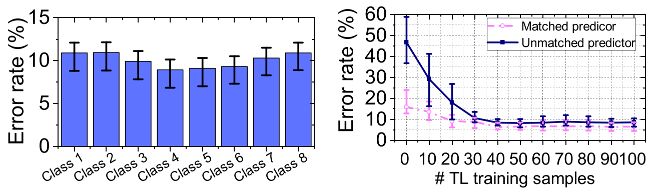
predictor into the three autotuning techniques. The GGA-Speedup, Greedy-Speedup and Random-Speedup take the input features (JS and hardware) with the candidate configuration and then feed them into the speedup predictor to estimate the speedup level, and then report back to the search technique to perform the next search (see Section V-B). As we can see from Fig. 14, the GGA-Speedup and Random-Speedup benefit from the speedup predictor, with 1.09x and 1.11x speedup over the original GGA and Random, respectively. The Greedy-Speedup seems can not benefit from the speedup predictor on the performance improvement, as the core of the Greedy algorithm focuses on the local optimal. However, the speedup predictor can improve the speed of convergence for all the search techniques.

C. Applying JSTUNER on Laptop

We apply JSTUNER to the high-performance laptop Alienware m15 r4, which is equipped with an 8-core 10th Generation Intel Core i7-10870H CPU, an NVIDIA GeForce RTX 3070 GPU, and 16 GB RAM memory. Fig. 15 presents the improvement of our approach on 60 JS benchmarks over the default compilation configuration. We observe an average of 1.18x and 1.51x improvement for SNN and search engine, respectively. These results demonstrate that our approach can effectively enhance the JS performance on high-performance devices. As knowledge of devices and JS grows, the system will make those devices faster and more energy efficient.

D. Searching Engine Performance

Fig. 16 reports the performance and the number of tuning iterations when applying OpenTuner and our search engine to search the best-performing configuration for *box2d* and *pdfjs* on



(a) Add a new class to the SNN (b) Perf. w.r.t. #training samples

Fig. 17. Applying crowdsourcing to collect data for JSTuner continuous learning.

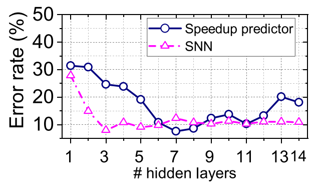


Fig. 18. Impact of the number of hidden neural layers for two models.

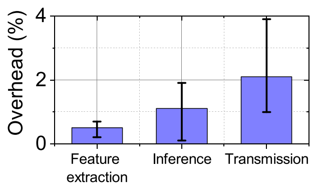


Fig. 19. Breakdown of runtime overhead.

Pixel 6. We can see that our search engine can speed up the convergence by 1.53x over OpenTuner. As the search engine leverages the speedup predictor to skip the configuration that slows down the performance and feedback a high penalty cost to the OpenTuner, thus can save time to run the configuration on the real environment. Fig. 16(b) and (d) present the number of tuning iterations with time. We can see that the search engine conducts up to 1068 and 2803 tests for *box2d* and *pdfjs* in 200 seconds, which is 2.84x and 5.28x over OpenTuner. Over the large tuning tests, the configuration performance of our search engine improves OpenTuner by an average of 1.13x on both JS programs.

E. Evaluation of CrowdSourcing

We employ crowdsourcing to collect data for retuning two predictive models used in JSTuner, when the models are failures on the new coming data. In detail, we evaluate the SNN model performance when adding a new class and test the speedup model for applying transfer learning (TL) to tune baseline predictors for the new coming JS.

1) Adding a New Class to SNN: Our baseline SNN model includes 8 classes. To test the performance when adding a new class, we use one out of all 8 classes as the testing class, while the rest samples from the other 7 classes are used for training a baseline model. Fig. 17(a) presents the SNN performance when adding a new class, the average error rate achieves 8.9%, which is comparable to the training on the whole dataset. Adding a new class in SNN is to learn the similarity function and make the new class far away from the existing classes, which do not need to retrain the SNN from scratch.

2) Tuning Speedup Predictors for New Coming Feaures: The speedup predictor performs Transfer Learning (TL) when the bias of the estimated speedup level and the measured speedup is above 10%. In such case, we consider the predictor fails for the new coming input features. For TL, we need to determine whether the input vector belongs to one of the eight classes of FPM. As we train 8 baseline speedup predictors for the 8 FPM classes, the input features belonging to the same class use one speedup predictor. Fig. 17(a) plots the results for using TL to port the baseline speedup predictor for the new input features in two situations. When the input features match one of eight FPM classes, we can leverage the corresponding speedup predictor to perform TL directly. As expected, TL performs better when the input vector belongs to one of our existing classes, which gives the lowest error rate of between 4.4% and 7.5%. For unmatched cases, we see a slight increase in the error rate when applying TL with the unmatched speedup predictor but it still gives an acceptable error rate, which is 8.43%. The experiment results show that the error rates reach a plateau when the training samples reach 40. With the FPM outputs, the new input features are able to select a suitable speedup predictor to accelerate the training process. We also train a new speedup predictor for when the FPM adds a new class.

F. Model Analysis

1) Impact of Neural Layers: Fig. 18 presents the error rate when the SNN model and speedup model with different number of hidden layers. We train each model with the same training set and record the error rate. We can see that the SNN model and speedup model give the best performance when the model is constructed with 3 and 7 hidden layers, respectively.

2) Overhead Breakdown of FPM: Fig. 19 presents a quantitative analysis of the overhead associated with each processing stage of FPM when searching for a coarse-grained optimization configuration for a newly arrived JS program. Once the JS code is downloaded onto the user device, we then start extracting JS features, which are then sent to the server for feature matching. It is important to note that during the initial loading of the JS, we will use the default configuration for execution (refer to Section IV-D for further details) so that the user does not experience an additional delay in our processing pipeline. Subsequently, the FPM carries out inferences based on the incoming features and applies the feedback configuration to compile and execute the JS program. Considering the volatile nature of networking conditions, the feature transfer stage can incur the highest cost in terms of processing overhead. However, due to the parallel execution of the FPM with JS parsing, the average overhead introduced by the FPM only contributes to an end-to-end turnaround time of less than 5%. Fig. 20 presents the end-to-end latency of FPM, where FPM delivers an average improvement of 1.19x over the default compiler configuration used by Chrome V8. We can observe that in a few test cases (8% of the test benchmarks), our predictive model yields sub-optimal

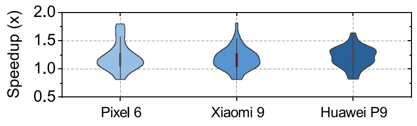
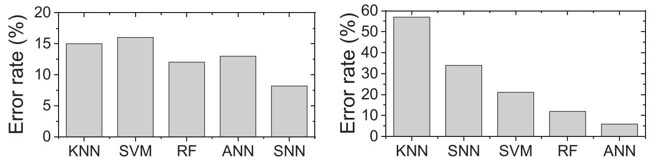


Fig. 20. End-to-end performance of the FPM.



(a) Different modeling techniques performance for fast program matching.

(b) Different modeling techniques performance for speeedup predictor.

Fig. 21. Comparing our SNN-based FPM and ANN-based speedup predictor with other modeling techniques.

predictions, while still achieving an average performance of 86% compared to the default settings, and the remaining cases outperform the average performance of the default configuration. To mitigate the slowdown caused by incorrect predictions, we can enhance model prediction accuracy through increased training samples. Our framework offers this continuous learning capability by employing an offline search engine to identify optimal configurations, which can then be utilized to enhance the predictive model through retraining.

At the same time, our compiler strategy may change across JS programs of the same web page. The overhead introduced by our scheme for this multi-program scenario would solely be the prediction overhead-less than 1 ms. This is because the default JS engine will also compile each JS program individually. Our configuration prediction can also be performed for all JS programs at once after they are loaded, where the model inference overhead can be hidden while executing the first JS program. Therefore, the overhead of using program-specific compiler configurations is negligible.

3) Alternative Modeling Techniques: Fig. 21(a) compare our SNN-based fast program matching module against four alternative classification methods: K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Random Forest (RF) and ANN. On the other hand, we also try to build our speedup predictor by using KNN, SVM, RF, SNN and ANN, and Fig. 21(b) presents the performance for all modeling techniques. All the alternative techniques were trained and evaluated by using the same method and training data as our models. Our approach achieves the lowest error rate and enables us to employ transfer learning.

G. Discussions and Future Work

Application scenarios: JSTUNER can be useful in multiple practical scenarios. One such scenario is the application of JSTUNER within an app store environment, where it can be harnessed to provide tailored JS optimization services, thereby enhancing the user experience for individual mobile apps and improving the competitiveness of the mobile app. This optimization service can be made available to application developers as a paid API, with the cost based on the number of API requests made per month. Consequently, the service provider can leverage the aggregated information from the crowd to continuously enhance the quality of the optimization services offered. Furthermore, JSTUNER empowers application developers by providing crowdsourcing capabilities that facilitate evaluating JS performance across diverse mobile hardware platforms during the development phase. This enables developers to conduct comprehensive performance testing and deliver the JS code with the appropriate configuration directly to the intended mobile devices, ensuring optimized performance on each platform.

Portability: JSTUNER can be easily ported into other computing platforms, including desktop and high-performance computing systems. Doing so would require collecting training data on the target platform and JS engine. However, our approach to training and using the machine learning model can remain largely unchanged. JSTUNER can also work with off-the-shelf web browsers, including Chrome, as we do NOT change the internal implementation of the JS engine. Instead, we use the compiler options exposed by the JS engine to optimize the program and treat the JS engine as a black box. Consequently, our techniques are compatible with any web browser implementation.

Dynamic offloading: Our evaluation was conducted within a high-quality network environment, primarily focusing on computationally intensive benchmarks. However, we also realize that real network conditions often exhibit instability. Consequently, the detrimental effects of a poor network status may outweigh the performance advantages gained through FPM, particularly for non-computation-intensive JS programs. A hybrid scheme can be employed to tackle this challenge, wherein the prediction task is executed locally on the mobile device or offloaded to the cloud. The decision to offload depends on network status, available hardware resources, and the complexity of the upcoming JS task. The objective is to minimize the cost associated with FPM on the mobile device. To reduce the FPM cost on the mobile device, we are exploring the design of a lightweight SNN model on mobile the highly efficient and lightweight deep learning framework [39] for future research.

We also consider how to combine offline auto-tuning and online compiler option prediction to offer a high-quality real-time optimization service. Specifically, we can deploy a lightweight predictive model on the mobile device to predict a configuration for the coming JS to provide real-time coarse-grained optimization. Additionally, we can utilize the search engine to find the optimal configuration when the device is idle. This optimal setting is applied in subsequent JS runs and updates the on-device predictive model.

Privacy and security: Our techniques do not alter the JS code or the internal implementation of a JS engine. Therefore, it should not introduce new security issues. Furthermore, since the JS code is downloaded from the internet and we only offload the code rather than user data, JSTUNER should not lead to the leakage of private data. Furthermore, JSTUNER can leverage other privacy-preserving frameworks like gg [40] to enhance privacy and data security. For the predictive model, JSTUNER can seamlessly integrate with the Federated Learning framework [41] to train and deploy our predictive model directly on the local device. This ensures that the JS code remains on the device, enhancing privacy and reducing the need for data transmission to external servers.

IX. Related Work

Our work is broadly related to the literature in three areas:

JavaScript optimization: There is considerable work on JavaScript optimization. One common approach is to use code optimization techniques, such as minification, tree shaking, and dead code elimination, to reduce the size of the code and reduce the number of operations that need to be performed at runtime [42], [43]. While some work focused on improving the efficiency of JS runtime engines, such as by implementing Just-In-Time (JIT) compilation, optimizing garbage collection algorithms [44], and leveraging hardware features such as SIMD instructions to accelerate certain operations [45]. Furthermore, there has also been research on optimizing the use of specific JS features, such as the use of functional programming constructs, the use of asynchronous programming patterns, and the optimization of object-oriented design patterns. Recently, there has been research on using machine learning techniques to optimize the performance of Javascript applications, such as by using neural networks to predict the performance of different code paths [46] or using natural language information to predict JavaScript function types [47], [48]. However, these works give limited performance optimization and can not always deliver optimization across different platforms and usage scenarios. Prior works also try to offload the intensive-compute JavaScript task to cloud [49], [50] or convert JavaScript to the executable binary file [51] on the server, while offloading may pose numerous security problems and convert to the binary file can not make up the compilation cost, as the user may only run the program once in most cases. JSTuner is designed to address these limits, focuses on JavaScript engine optimization, and can adapt to different programs, platforms and JavaScript engines.

Program autotuning: Researchers have developed a wide range of algorithms and search strategies for program autotuning, including genetic algorithms, Bayesian optimization [10], and reinforcement learning [6]. These approaches aim to efficiently explore the search space of possible configurations and select the best ones. Besides, researchers have also explored the use of machine learning techniques [52], [53] to learn performance models of programs, which can be used by autotuners to guide the search for the best configurations. This can reduce the number of program runs required to find the optimal configuration. While they mainly focus on tuning the hyperparameters and layers of the neural network. Furthermore, many works [6], [13], [21] integrate reinforcement learning with other search techniques, such as evolutionary algorithms [54], Bayesian optimization and particle swarm optimization [55], which can further improve the performance of programs and make it easier to optimize them for specific hardware platforms. However, to find a good solution, the autotuning techniques need to perform fine-grained searches with a long search time which is not acceptable for JS on the mobile side. JSTuner leverages the similarities between programs, and delivers a portable configuration to optimize the target program, then use a performance predictor to accelerate the configuration search process.

Predictive modeling: Machine learning has been used to model power consumption [56], task scheduling [57] of mobile systems and program tuning in general [6]. Our work is the first to deploy machine learning for tuning mobile JavaScript. To build a consistently valid model, JSTuner employs crowdsourcing to gather a large amount of data from a diverse group of mobile devices and JS, which can lead to more accurate and robust models. Furthermore, Our work tackles an outstanding problem of porting a model to a new computing environment or JS. Transfer learning was recently used for wireless sensing [58] through randomly chosen samples. JSTuner leverages transfer learning to avoid training from scratch, as it allows the JSTuner to make use of the knowledge and experience gained from other JS or platforms to inform the search.

X. Conclusion

This paper has presented JSTuner, a simple and effective machine-learning guided autotuning system to optimize JS performance on mobile devices by delivering a good compiler optimization sequence. JSTuner provides a significant improvement over the other autotuning algorithms in terms of run time performance. Central to JSTuner is two predictive models, one is the siamese neural network-supported program matching model for the configuration selection based on the similarity between the input features and the pre-stored features; the annual neural network-based speedup predictor skips the poor configuration by considering the input JS features, hardware characteristics and the tunning configuration, which can reduce the measurement cost on the devices, and greatly accelerate the search process. Furthermore, we design the crowdsourcing module to collect data for continuously improving JSTuner. We evaluate JSTuner by applying it to 60 JS benchmarks on 3 representative mobiles. Experimental results show that JSTuner outperforms OpenTuner, the state-of-the-art autotuning framework, with an average improvement of 1.12x (up to 2.51x) across these test benchmarks.

In addition, our approach has been paired with multiple autotuning algorithms, resulting in improved performance compared to the original methods. Although our approach has not yet been tested with the latest autotuning systems, such as CompilerGym and SuperSonic, which primarily optimize for LLVM, and need a significant amount of engineering effort to modify them, we plan to integrate our method with these systems in the future research and expand our approach to other programming languages.

References

[1] TIOBE, “TIOBE Programming Community Index,” 2022. [Online]. Available: https://www.tiobe.com/tiobe-index/
[2] P. Mpeis, P. Petoumenos, and H. Leather, “Iterative compilation on mobile devices,” 2015, *arXiv:1511.02603*.
[3] Z. Wang and M. O’Boyle, “Machine learning in compiler optimization,” in *Proc. IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018.

[4] A. Rasch, R. Schulze, M. Steuwer, and S. Gorlatch, “Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF),” *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, pp. 1–26, 2021.
[5] P. M. Phothilimthana et al., “A flexible approach to autotuning multi-pass machine learning compilers,” in *Proc. 30th Int. Conf. Parallel Archit. Compilation Techn.*, 2021, pp. 1–16.
[6] H. Wang et al., “Automating reinforcement learning architecture design for code optimization,” in *Proc. 31st ACM SIGPLAN Int. Conf. Compiler Construction*, 2022, pp. 129–143.
[7] J. S. Acosta, A. Diavastos, and A. Gonzalez, “XFeatur: Hardware feature extraction for DNN auto-tuning,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2022, pp. 132–134.
[8] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang, “Lamda: Learning-assisted multi-stage autotuning for FPGA design closure,” in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, 2019, pp. 74–77.
[9] L. Almagor et al., “Finding effective compilation sequences,” *ACM SIGPLAN Notices*, vol. 39, no. 7, pp. 231–239, 2004.
[10] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano, “COBAYN: Compiler autotuning framework using bayesian networks,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, pp. 1–25, 2016.
[11] H. Jordan and P. Thoman, “A multi-objective auto-tuning framework for parallel codes,” in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–12.
[12] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, “Minimizing the cost of iterative compilation with active learning,” in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2017, pp. 245–256.
[13] J. Ansel et al., “OpenTuner: An extensible framework for program autotuning,” in *Proc. 23rd Int. Conf. Parallel Archit. Compilation*, 2014, pp. 303–316.
[14] A. Goel, V. Ruamviboonsuk, R. Netravali, and H. V. Madhyastha, “Rethinking client-side caching for the mobile web,” in *Proc. 22nd Int. Workshop Mobile Comput. Syst. Appl.*, 2021, pp. 112–118.
[15] R. E. Corporation, Angry birds. [Online]. Available: https://www.angrybirds.com/
[16] A. O. Community, “AR.js - Augmented reality on the web,” 2023. [Online]. Available: https://ar-js-org.github.io/AR.js-Docs/
[17] TensorFlow, “TensorFlow.js is a library for machine learning in JavaScript,” 2023. [Online]. Available: https://www.tensorflow.org/js/
[18] J. Mayes, “Make a smart webcam in JavaScript with a TensorFlow.js pre-trained machine learning model,” 2021. [Online]. Available: https://codelabs.developers.google.com/codelabs/tensorflowjs-object-detection#0
[19] Lorenz, “Voice controlled ToDo List: JavaScript speech recognition,”, 2020. [Online]. Available: https://dev.to/webdeasy/voice-controlled-todo-list-javascript-speech-recognition-11of
[20] V. J. engine, “Node.js,” 2022. [Online]. Available: https://nodejs.org/
[21] C. Cummins et al., “CompilerGym: Robust, performant compiler optimization environments for AI research,” in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2022, pp. 92–105.
[22] A. Haj-Ali et al., “AutoPhase: Juggling HLS phase orderings in random forests with deep reinforcement learning,” in *Proc. Mach. Learn. Syst.*, vol. 2, pp. 70–81, 2020.
[23] Mozilla, Pdf.js. 2022. [Online]. Available: https://mozilla.github.io/pdf.js/
[24] Box2d. 2022. [Online]. Available: https://github.com/hecht-software/box2dweb
[25] G. Koch et al., “Siamese neural networks for one-shot image recognition,” in *Proc. ICML Deep Learn. Workshop*, 2015, pp. 1–8.
[26] F. Pedregosa et al., “Scikit-learn: Machine learning in Python,” *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
[27] H. Mendoza et al., “Towards automatically-tuned deep neural networks,” in *AutoML: Methods, Sytems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., Berlin, Germany: Springer, 2018, pp. 141–156.
[28] Complexity-report. 2016. [Online]. Available: https://github.com/escomplex/complexity-report
[29] I. C. Coverage, Javascript test coverage made simple. 2019. [Online]. Available: https://istanbul.js.org/
[30] C. M. Bishop, M. Christopher, and M. Nasser, *Pattern Recognition and Machine Learning*, New York, USA: Springer, 2006.
[31] B. F. Manly and J. A. N. Alberto, *Multivariate Statistical Methods: A Primer*. Boca Raton, FL, USA: CRC Press, 2016.
[32] S. Hemminger et al., “Network emulation with NetEm,” in *Proc. Linux Conf. AU*, 2005, pp. 18–23.
[33] JetStream2. 2022. [Online]. Available: https://browserbench.org/JetStream/
[34] G. D’Angelo and F. Palmieri, “GGA: A modified genetic algorithm with gradient-based local search for solving constrained optimization problems,” *Inf. Sci.*, vol. 547, pp. 136–162, 2021.
[35] D. M. Chickering, “Optimal structure identification with greedy search,” *J. Mach. Learn. Res.*, vol. 3, no. Nov, pp. 507–554, 2002.
[36] Z. B. Zabinsky et al., “Random search algorithms,” Dept. Ind. Syst. Eng., Univ. Washington, USA, 2009.
[37] L. Yuan et al., “Using machine learning to optimize web interactions on heterogeneous mobile systems,” *IEEE Access*, vol. 7, pp. 139394–139408, 2019.
[38] W. Ertel, “On the definition of speedup,” in *Proc. Int. Conf. Parallel Archit. Lang. Europe*, 1994, pp. 289–300.
[39] X. Jiang et al., “MNN: A universal and efficient inference engine,” in *Proc. Mach. Learn. Syst.*, 2020, pp. 1–13.
[40] S. Fouladi et al., “From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 475–488.
[41] D. J. Beutel et al., “Flower: A friendly federated learning research framework,” 2020, *arXiv: 2007.14390*.
[42] S. K. Shivakumar, “Mobile web performance optimization,” in *Modern Web Performance Optimization*. Berlin, Germany: Springer, 2020, pp. 79–103.
[43] Closure Compiler. 2022. [Online]. Available: https://developers.google.com/closure/compiler/
[44] T. Hartley, F. S. Zakkak, A. Nisbet, C. Kotselidis, and M. Luján, “Just-in-time compilation on arm—a closer look at call-site code consistency,” *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, pp. 1–23, 2022.
[45] E. Wen and J. Shen, “DSPBooster: Offloading unmodified mobile applications to DSPs for power-performance optimal execution,” in *Proc. IEEE 46th Annu. Comput. Softw. Appl. Conf.*, 2022, pp. 614–623.
[46] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “A general path-based representation for predicting program properties,” *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.
[47] R. S. Malik, J. Patra, and M. Pradel, “Nl2type: Inferring Javascript function types from natural language information,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 304–315.
[48] M. Pradel, G. Gousios, J. Liu, and S. Chandra, “Typewriter: Neural type prediction with search-based validation,” in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 209–220.
[49] H.-J. Jeong, I. Jeong, and S.-M. Moon, “Dynamic offloading of web application execution using snapshot,” *ACM Trans. Web*, vol. 14, no. 4, pp. 1–24, 2020.
[50] A. Sivakumar et al., “Nutshell: Scalable whittled proxy execution for low-latency web over cellular networks,” in *Proc. 23rd Annu. Int. Conf. Mobile Comput. Netw.*, 2017, pp. 448–461.
[51] Nexe. 2022. [Online]. Available: https://github.com/nexe/nexe
[52] T. Chen et al., “{TVM }: An automated { End-to-End} optimizing compiler for deep learning,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 578–594.
[53] S. Kaufman et al., “A learned performance model for tensor processing units,” in *Proc. Mach. Learn. Syst.*, vol. 3, pp. 387–400, 2021.
[54] J. Ansel et al., “Petabricks: A language and compiler for algorithmic choice,” *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 38–49, 2009.
[55] R. Poli, J. Kennedy, and T. Blackwell, “Particle swarm optimization,” *Swarm Intell.*, vol. 1, no. 1, pp. 33–57, 2007.
[56] P. Zhang, J. Fang, C. Yang, C. Huang, T. Tang, and Z. Wang, “Optimizing streaming parallelism on heterogeneous many-core architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 8, pp. 1878–1896, Aug. 2020.
[57] J. Ren et al., “Camel: Smart, adaptive energy optimization for mobile web interactions,” in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 119–128.
[58] C. Feng et al., “Wi-Learner: Towards one-shot learning for cross-domain Wi-Fi based gesture recognition,” in *Proc. ACM Interactive Mobile Wearable Ubiquitous Technol.*, vol. 6, no. 3, pp. 1–27, 2022.

Jie Ren (Member, IEEE) received the PhD degree in computer system architecture from Northwest University, Xi’an, China, in 2017. He is an associate professor with the School of Computer Science at Shaanxi Normal University, Xi’an, China. His research mainly focuses on mobile system optimisation and runtime scheduling.

Ling Gao is a professor with the Department of Computer Science at Northwest University, Xi'an, China. His research interests include privacy protection and edge computing.

Zheng Wang (Member, IEEE) is chair professor of Intelligent Software Technology with the School of Computing at the University of Leeds and a turing fellow with the Alan Turing Institute. His research focuses on accelerating large-scale deep learning models, program optimisation, applied machine learning and system security.