The Client Insourcing Refactoring and Its Applications to Optimizing and Enhancing Distributed Execution

Kijin An

Thesis proposal for

Doctor of Philosophy
in
Computer Science and Applications

Eli Tilevich, Virginia Tech, Chair
Godmar Back, Virginia Tech
Walter Binder, University of Lugano
Xun Jian, Virginia Tech
Francisco Servant, Virginia Tech

March 20, 2020
Blacksburg, Virginia

Keywords: Software Engineering, Reengineering, Web Apps, JavaScript, Mobile Apps, Program Analysis & Transformation, Middleware, Cloud Computing, Edge Computing

The Client Insourcing Refactoring and Its Applications to Optimizing and Enhancing Distributed Execution

Kijin An

(ABSTRACT)

Developers often need to re-engineer distributed applications to address changes in requirements, made only after deployment and usage. Much of the complexity of inspecting and evolving distributed applications lies in their distributed nature, while the majority of mature program analysis and transformation tools works only with centralized software. Inspired by business process re-engineering, in which remote operations can be insourced back in house to restructure and outsource anew, we bring an analogous approach to the re-engineering of distributed applications. We introduce Client Insourcing, a novel automatic refactoring that creates a semantically equivalent centralized version of a distributed application. This centralized version can then be inspected, modified, and redistributed to meet new requirements. We demonstrate the utility and value of Client Insourcing in optimizing and enhancing distributed execution to meet the changed requirements in performance, reliability, and security. We realize the reference implementation of Client Insourcing in the important domain of full-stack JavaScript applications, and apply our implementation to re-engineer mobile web applications. The ultimate goal of this dissertation research is to reduce the complexity of inspecting and evolving distributed applications in order to facilitate their re-engineering.

Contents

1 Introduction

Developers often need to enhance distributed applications to address requirement changes made only after deployment and usage. Re-engineering captures evolutionary modifications that range from maintenance tasks to architecture-level changes [1, 9]. A re-engineering effort can involve adding a major feature, protecting against security vulnerabilities, or removing performance bottlenecks. Modifying existing distributed applications requires complex program analysis and modification operations that are hard to perform and even harder to verify. One of the main causes of this complexity is the distributed execution model of distributed applications. In this model, a distributed application's execution flows across the separate address spaces of its client and server parts. All remote interactions are typically implemented by means of middleware libraries. As a result, the control flow of distributed applications can be highly complex, with their business and communication logic intermingled. That complexity hinders all tracing and debugging tasks. In addition, distributed execution over the network makes web applications vulnerable to partial failure and non-determinism.

Program analysis is central to software comprehension. The distributed application is predominated by dynamic languages, which defeat static analysis techniques. Hence, to comprehend programs written in dynamic languages, such as JavaScript, requires dynamic analysis. Software debugging hinges on the ability to repeat executions deterministically [35, 38]. However, many distributed applications are stateful, with certain client server interactions changing the server's state. It can be quite laborious and error-prone to restore the original state to be able to repeat a remote buggy operation [19, 31, 39]. All in all, it is the presence of both distribution and stateful execution that makes it so hard to trace and modify distributed applications.

In this work, we draw inspiration from business process re-engineering that can bring remote operations in-house via *Insourcing*. Once the insourced operations are redesigned and restructured, some of them can be outsourced anew. As argued above, the notion of local operations being easier to analyze and restructure than remote ones equally applies to distributed applications. Specifically, the approach presented herein first automatically transforms a distributed application, comprising a client communicating with a remote server, to run as a centralized program. The resulting centralized variant retains to a large degree the semantics of the original application, but replaces all remote operations with local ones. The centralized variant becomes easier to analyze and modify not only because it has no remote operations, but also because the majority of program analysis and transformation approaches and tools have been developed for centralized programs. After the centralized variant is modified to address the new requirements and the modifications have been verified, it is then redistributed again into a re-engineered distributed web application. Our target domain are distributed applications written entirely in JavaScript, both the client and server parts; such applications are referred to as *full-stack JavaScript applications*. We take advantage of the monolingual nature of such applications to streamline our implementation.

To achieve the stated goals of this research, we propose to (1) create an automated refactoring, *Client Insourcing* for the domain of full-stack JavaScript applications and apply this refactoring to (2) to localize and fix bugs in web applications, (3) enhance and optimize ill-conceived distributions of real-world apps by adjusting their distribution granularity, (4) reconcile the design-time and run-time mismatch in cloud-based applications.

1. **Creating the Client Insourcing Refactoring for Full-Stack JS Apps**

   The focal point of our approach is Client Insourcing, a new automatic refactoring that undoes distribution by gluing the local and remote parts of a distributed full-stack JS application together. Client Insourcing can precisely identify the functionality of HTTP middleware—irrespective of its API and in the presence of stateful operations—by combining program instrumentation, profiling, and fuzzing in a novel way. These HTTP middleware APIs tend to differ widely depending on their vendor: the JavaScript ecosystem features numerous dissimilar distribution frameworks and libraries. Rather than encoding domain-specific preconditions, our approach uses domain agnostic semantic to identify the entry/exit points of middleware functionality.

2. **Localizing and Fixing Defects in Web Applications**

   Localizing bugs in web applications is complicated by the potential presence of server/middleware misconfigurations and intermittent network connectivity. We introduce a novel debugging approach to localizing bugs in distributed web applications. The debugged application is converted to its semantically equivalent centralized version, thus separating the programmer-written code from configuration/environmental issues as suspected bug causes. The centralized version is then debugged to fix various bugs. Finally, based on the bug fixing changes of the centralized version, a patch is automatically generated to fix the original application source files.

3. **Correcting Ill-Conceived Distribution Granularity**

   Distributed applications enhance their execution by using remote resources. However, distributed execution incurs communication overheads, if these overheads are not offset by the yet larger execution enhancement, distribution becomes counterproductive. For maximum benefits, the distribution's granularity cannot be too fine or too crude; it must be just right. We introduce a novel approach to re-architecting distributed applications, whose distribution granularity has turned ill-conceived. To adjust the distribution of such applications, our approach automatically reshapes their remote invocations to reduce aggregate latency and resource consumption by means of a series of domain-specific automatic refactorings.

4. **Reconciling the Design and Runtime Mismatch in Mobile and Edge Apps**

   Communicating Web Vessels A mobile web app's performance depends heavily on the underlying network and the client device's hardware capacity. Static distribution allocates some application functionality to run on the client and some on the server,

with the resulting allocation remaining fixed throughout the app's execution. If the available network and the client device mismatch those assumed during the static distribution, app responsiveness and energy efficiency can suffer. To address this problem, we propose Communicating Web Vessels (CWV), an adaptive redistribution framework that improves the responsiveness of full-stack JavaScript mobile apps. By monitoring the network conditions, CWV determines whether a dynamic redistribution would improve application performance. It then triggers a redistribution that moves server-side functionalities to the mobile client and vice versa. When CWV moves server-side functions and their reachable state to the client, they are invoked as regular local functions. The moved functionalities are accessed remotely again as soon as the network conditions improve.

Edge Insourcing Edge computing offers novel execution optimization opportunities, however it would be non-trivial to adapt existing client/cloud apps to take advantage of edge-based resources. We propose to extend the idea of Client Insourcing into *Edge Insourcing*, a new refactoring technique that can automate such adaptation, offering a unique technical perspective for solving this important problem of engineering edge applications.

Proposal Structure: The rest of this Ph.D. preliminary proposal is organized as follows. Section 2 presents the preliminary research we have accomplished. Section 3 discusses the remaining work. Section 4 compares the proposed work with the state of the art. Section 5 outlines the research schedule and presents concluding remarks.

2 Preliminary Work

2.1 Client Insourcing Refactoring: JS-RCI

Approach We follow the software engineering methodology of creating a novel domain-specific refactoring and demonstrating its value, applicability, and limitations through a series of case studies of enhancing distributed application execution. Our new domain-specific refactoring—Client Insourcing—is realized as a reference implementation, called JavaScript Remote Client Insourcing (JS-RCI) [5].

JS-RCI integrates advanced software engineering techniques, such as program synthesis, which identifies which application functionality satisfies a given remote execution's input and output relationship. JS-RCI also makes use of constraint solving to extract JavaScript server-side functions. The constraint solving functionality is based on declarative logic-based programming, which represents JavaScript program statements and their relationships to each other as logical facts and predicates. As a specific example, consider the relationship

that expresses the transitive dependency across the statements, executed between the entry and exit points of a remote service:

//Dependency Analysis to extract a function from server code
$\text{ExecutedStmts}(stmt_n, V^{u_{id}}_{unMar}, V^{u_{id}}_{Mar}) \leftarrow (\text{DataDep}(stmt_n, stmt_1) \wedge \text{Marshal}(stmt_1, v_1, V^{u_{id}}_{Mar}))$
$\wedge\ (\neg\text{DataDep}(stmt_n, stmt_2) \wedge \text{UnMarshal}(stmt_2, v_2, V^{u_{id}}_{unMar}))$

//the entry point at the server
$\text{UnMarshal}(stmt_1, v_{unMar}, V^{u_{id}}_{unMar}) \leftarrow$
$\text{Write}(stmt_1, v_{unMar}) \wedge \text{Ref}(v_{unMar}, V^{u_{id}}_{unMar})$

//the exit point at the server
$\text{Marshal}(stmt_1, v_{Mar}, V^{u_{id}}_{Mar}) \leftarrow$
$\text{Write}(stmt_1, v_{Mar}) \wedge \text{Ref}(v_{Mar}, V^{u_{id}}_{Mar})$

One of the key functions of a middleware system is *Marshaling* and *UnMarshaling* methods for handling parameters and return values. Marshaling (exit point in server) converts program values to a data format suitable for transmission; unmarshaling (entry point in server) reverses the process by converting the transmitted data format to regular program values. To determine such entry/exit points, JS-RCI instruments the app's source code statements, which implement HTTP commands, as well as applies advanced testing techniques (e.g., *fuzzing*). Specifically, to identify these points with minimal false-negatives, JS-RCI fuzzes the original HTTP traffic by padding the HTTP header and body data with random data. Since many of these testing techniques requires that the subject be executed repeatedly in a deterministic way, JS-RCI also makes it possible for stateful servers to be executed idempotently by automatically undoing their state changes. Overall process of JS-RCI is shown in Figure 1.

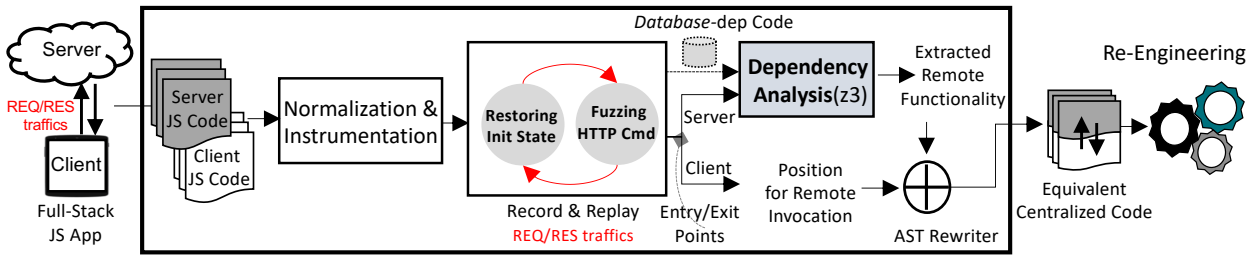


Figure 1: Client Insourcing Refactoring (JS-RCI)

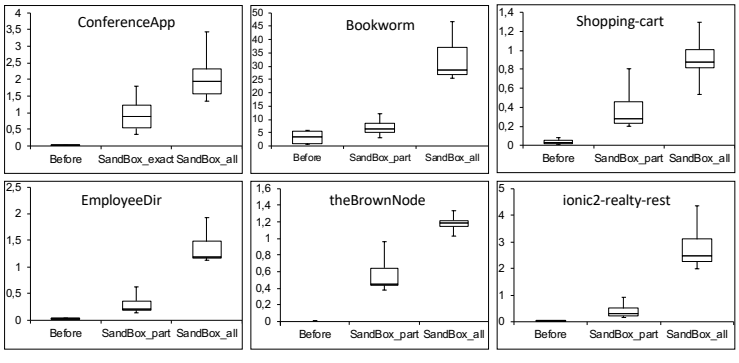
Result We have successfully applied JS-RCI to numerous real-world apps that use different middleware frameworks to implement their client/server (T1/T2) communication and different third-party libraries for database operations (T3) as shown in the Figure 2.

We observed that Idempotent Execution (w/o IdEx) with its Record/Replay phases removes the *false-negatives* in the detected marshaling points for stateful servers. In contrast, Fuzzing (w/o Fuzz) removes *false-positives* for detecting marshaling points (Figure 3-(b)).

Subjects (T1,T2,T3)	$HTTP_{Methods}$	**Services**	$C\&P/M$ (ULOC)
recipebook (AngularJS, Express,MySQL)	GET	/recipes	22/45
	GET/PUT/DEL	/recipes:id	72/172
	POST	/ingts	25/48
	GET/PUT/DEL	/ingts:id	74/207
	POST	/directions	26/57
	GET/PUT/DEL	/directions:id	60/130
DonutShop (Ajax, Express,knex)	GET/POST	/donuts	22/88
	GET/POST/DEL	/donuts:id	29/155
	GET/POST	/employee	20/71
	GET/POST/DEL	/employee:id	29/138
	GET/POST	/shops	16/83
	GET/DEL	/shops:id	19/128
res-postgresql (axios,restify, Postgres)	GET/POST	/user	22/71
	GET/PUT/DEL	/user	40/120
shopping-cart (Angular2,Express)	GET/POST/DEL	/cart-items	79/130
realty_rest (Angular2,Express)	GET/POST/DEL	/prprts/favs	34/73
	POST	/prprts/likes	291/304
	GET	/prprts	284/297
	GET	/prprts:id	287/300
	GET	/brokers	86/99
	GET	/brokers:id	90/103
med-chem (fetch,koa.js ,knex)	GET	/hbone	9K/9K
	GET	/molecular	9K/9K
BrownNode (AngularJS ,Express)	GET	/u/search	37/65
	GET	/u/search/id	36/64
Bookworm (JQuery, Express)	GET	/ladywithpet	394/409
	GET	/thedea	394/409
	GET	/theredroom	394/409
	GET	/thegift	394/409
	GET	/wallpaper	394/409
	GET	/offshore	394/409
	GET	/bigtripup	394/409
	GET	/amont	394/409
ConfApp (Angular2 ,Express)	GET	/findpeakers	13/66
	GET	/findSpeaker	15/68
	GET	/findSessions	43/117
	GET	/findSession	46/119
Emp_Dir (Angular2 ,Express)	GET	/employees	22/44
	GET	/employees/id	38/60
Total	**61**		**24K/26K**

Figure 2: Effort Saved by Client Insourcing

Client Insourcing creates a redistributable application variant that can be *refactored* and *enhanced* using any state-of-the-practice program transformation tools and then *distributed anew* using any state-of-the-art ditribution tools. We applied two JavaScript refactoring tools on our centralized variants: Node-SandBox for security enhancements and extremeJS [52] for redistribution. We measure the additional execution time incurred by sandboxing only a subset of the remote functionality vs. the entire original remote functionality. This comparison highlights the importance of isolating only the code that needs to be sandboxed. The observed differences in execution time between these two versions are quite striking, clearly showing that sandboxing the entire server part is impractical (Figure 3-(a)).



(a) Redistribution with Sandboxing

Subject Apps	State -less	Data base	All	w/o Fuzz	w/o IdEx
BNode	✓	X	2/2	0/2	2/2
Bworm	✓	X	8/8	0/8	8/8
ConfApp	✓	X	4/4	4/4	4/4
EmpDir	✓	X	2/2	2/2	2/2
shopping	X	X	3/3	3/3	0/3
realty	X	X	8/8	6/8	2/8
recipe	X	✓	13/13	13/13	0/13
Donut	X	✓	14/14	14/14	0/14
r-post	X	✓	5/5	5/5	0/5
m-chem	✓	✓	2/2	2/2	2/2
Total			**100%**	**80%**	**32%**

(b) Correctness affected by Search Methods

Figure 3: JS-RCI's Value of Redistribution and Correctness

2.2 Debugging Distributed Applications: CanDoR

Approach One application of Client Insourcing is to facilitate the debugging of distributed apps. The underlying idea is to be able to debug the semantically equivalent centralized variant of a faulty app (e.g., performance bottlenecks, memory leak). Our tool CanDoR [4] creates a centralized app variant that can be debugged by means of the numerous existing state-of-art debugging tools for centralized JavaScript programs. This refactoring preserves the app's business logic, while significantly simplifying its control flow. Rather than spanning across two JavaScript engines (client and server), the resulting centralized apps require only one engine (Figure 4). Once the programmer fixes the bug in the centralized variant version with these tools, CanDoR applies the resulting fixes to the actual client and server parts of the original distributed application. To that end, CanDoR automatically generates input scripts for GNU Diff, which executes these scripts against the source files of the original full-stack JavaScript application by using GNU patch. For certain types of bugs, this debugging approach proves to be effective and reducing the required debugging efforts.

```
//equivalent Centralized App
var users=[]; //insourced
function getObjsInArray(obj,
     array){...} //Insourced(
     buggy iterations)
function getUsers(searchUser)
     {...} //Insourced

function users_search(i){
 var out=getUsers(i);
 return out;
} //Insourced

//$.ajax({url: '/users/search
     ',...}) is replaced by
var input = {fName:...};
var output=users_search(input);
$('#results').text(JSON.
     stringify(output));
```

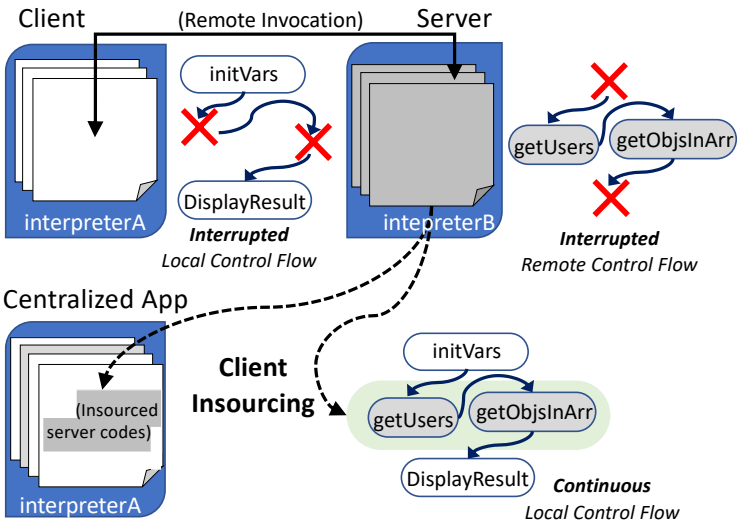


Figure 4: Continuous Control Flow of Distributed Codes constructed by Client Insourcing

Result CanDoR reduces the complexity of debugging distributed apps. It makes it possible to debug the app's business logic locally, so the app can be executed repeatedly, with the debugging output examined in place. As a result, the required debugging effort decreases. For example, CanDoR reduces the time taken to execute a debugged functionality by more than **90%** on average. Given that debugging typically involves repeated execution, having much faster subjects to debug improves the the debugging process's efficiency.

To demonstrate the value of CanDoR, we removed 11 performance bottlenecks from both the original subjects and their centralized variants. As it turns out, the bottleneck removals improved the performance of both versions (distributed and centralized) of each subject. Figure 5 summarizes the observed performance improvements. For the original distributed subjects (ΔT_{dist}), the improvements range between 29.5% and 2.0%. For their centralized variants (ΔT_{cent}), the improvements range between 34.8% and 1.6%. We also applied a linear regression analysis to compute how closely ΔT_{dist} and ΔT_{cent} correlate with each other, resulting in $\Delta T_{cent} = \mathbf{1.0089} * \Delta T_{dist} + \mathbf{1.556}$. This equation shows that ΔT_{cent} and ΔT_{dist} are almost perfectly correlated, so centralized variants can indeed serve as reliable and convenient proxies for an important class of performance debugging and optimization tasks.

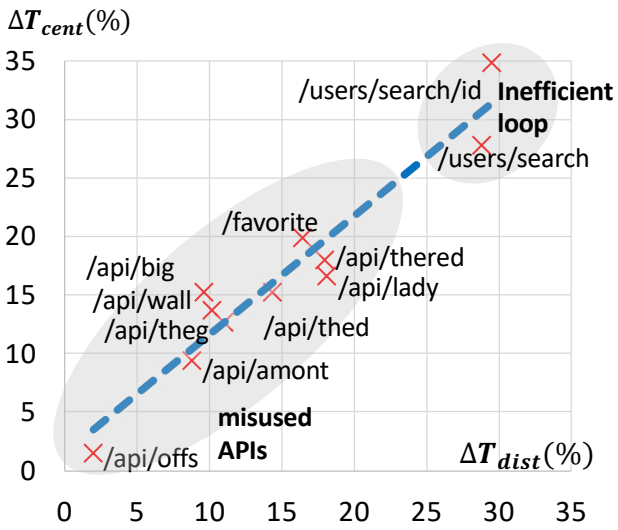


Figure 5: Regression Test for ΔT_{dist} vs. ΔT_{cent}

2.3 Optimizing Remote Execution Granularity: D-Goldilocks

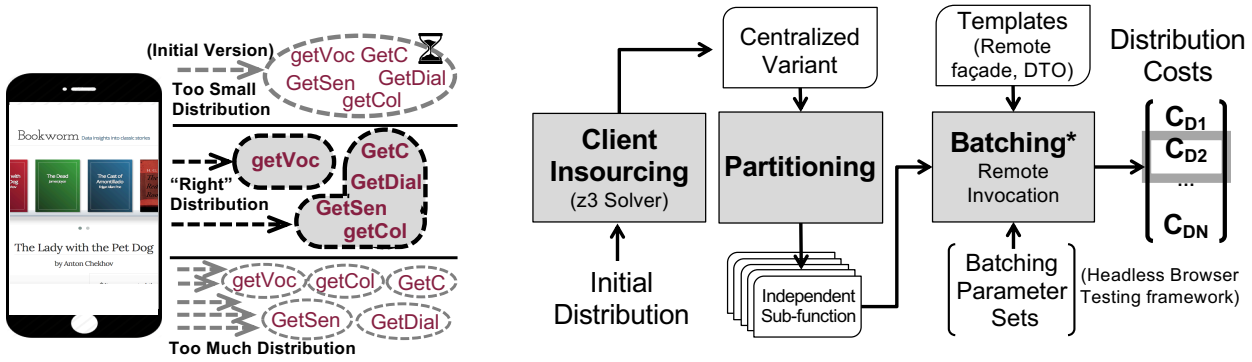
Approach Another application of Client Insourcing is restructuring the distribution of web apps to maximize their performance and efficiency. Our approach, named D-Goldilocks [6], follows *the Goldilocks Principle* [13, 22] to maximize the benefits of distribution. That is, the granularity of a remote service should not be too fine or too crude; it must be just right. For instance, the granularity of *Bookworm* is too crude as shown in Figure 6-(a).

D-Goldilocks re-architects distributed web apps to adjust their distribution granularity as a means of improving performance and efficiency. To that end, using a semantically equivalent centralized application variant, D-Goldilocks profiles the variant's performance, with the following cost function $\mathcal{C}(r)$, determining how to reshape the original distribution r to improve performance and efficiency.

$$\mathcal{C}(r) = \alpha \cdot latency(r) + (1 - \alpha) \cdot \sum resource(r).$$

Extant automatic refactorings (e.g., *partition* and *batch*) are then applied to reshape and redistribute the original remote functionality into a service with a changed granularity. In the end, D-Goldilocks identifies a distribution that minimizes the $\mathcal{C}$ function (Figure 6-(b)).

Result D-Goldilocks saves programming effort by automatically reshaping the granularity of the remote functionalities of web apps. To to determine what the optimal combination



(a) Motivating Example *Bookworm* (b) Process of D-Goldilocks

Figure 6: D-Goldilocks for Optimizing Remote Execution Granularity

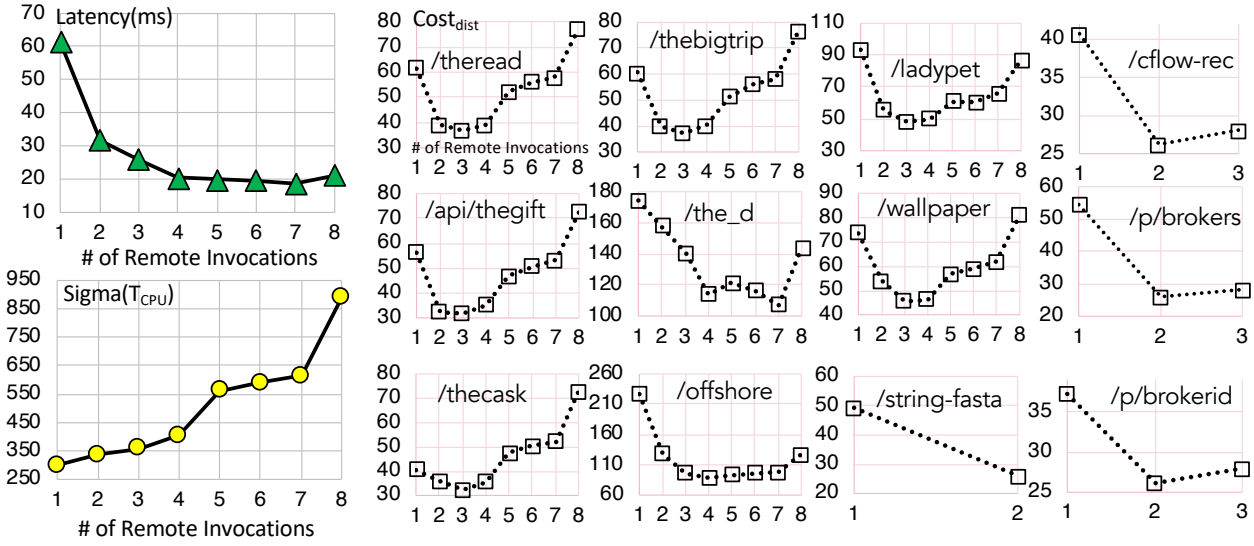


Figure 7: Number of Remote Invocations vs. Cost functions for Subject Remote Executions

of functions is, D-Goldilocks generates all possible combinations of individually invoked functions. For some subjects, D-Goldilocks can save almost 1.6×10^6 lines of code required to discover the optimal granularity. Splitting a singe long-running remote function into a small number of asynchronously invoked parts decreases both the aggregate latency and cost. However, as the number of partitions grows, so does the cost, due to the increasing overhead of invoking multiple remote functions (Figure 7).

3 Remaining Work

The remaining work of this dissertation will apply Client Insourcing to adapt distributed applications for their execution environments at runtime and to transform two-tier cloud-based applications to take advantage of edge computing resources. We describe these two remaining work thrusts in turn next.

3.1 Adapting Web Execution at Runtime: CWV

Approach To achieve the best performance for all combinations of client and server devices and network connections, a web app would have to be distributed in a variety of versions. My ongoing research is concerned with adaptively redistributing functionalities of a full-stack mobile JavaScript app to optimize its performance and energy consumption. We have been working on a new framework, Communicating Web Vessels (CWV). In a CWV-enabled web app, the client dynamically instruments and monitors app's distributed execution for "$/service\_r$". In response to a deterioration in network conditions, the client can request that the server-side JavaScript code and program state of a remote service be moved to the client, so the service can be invoked locally as a regular function call ($insource($"$/service\_r$"$)$). When the network conditions become favorable, the moved service starts to be invoked remotely again ($revert(r)$). Invoking remote services locally or remotely helps achieve the best possible response time of the web app under the current network conditions. The CWV's communication protocol defines how and when an app switches between the Original, Insourcing/Reverting, and Local modes described in Figure 8.

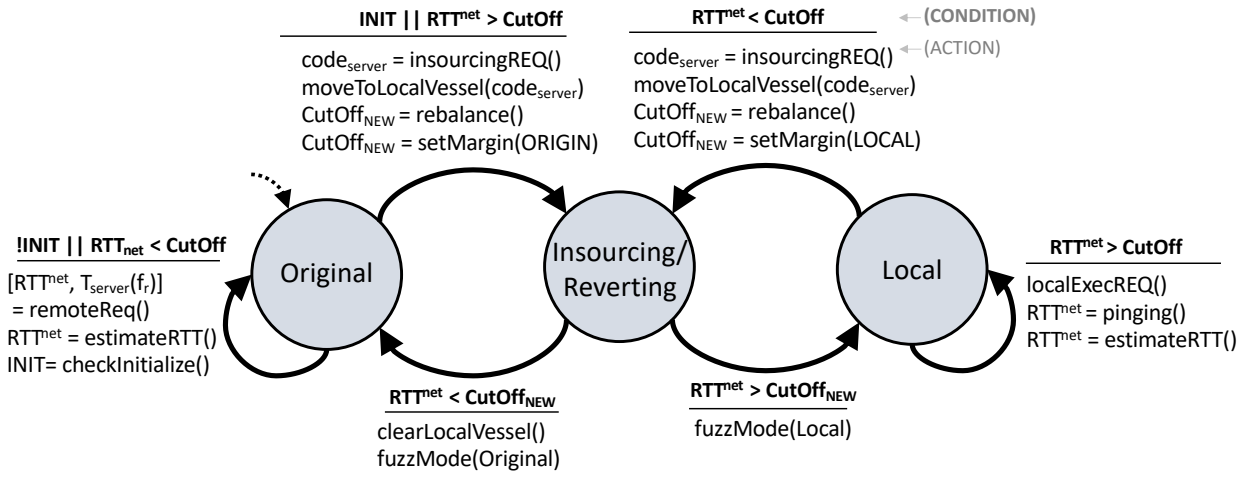
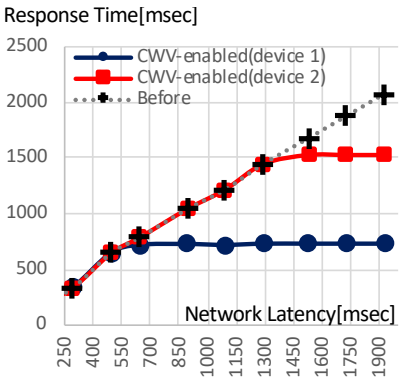


Figure 8: CWV Protocols for transitioning modes

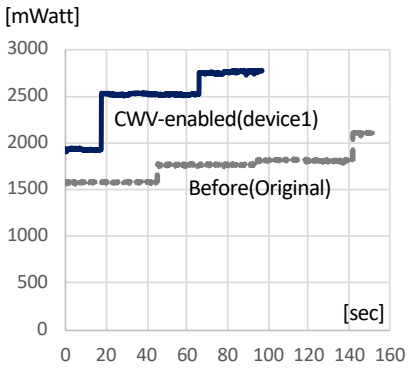
Result so far Preliminary results for CWV show that it effectively adapts web apps for dissimilar networks and two different mobile devices. CWV offers a low performance overhead,

Serv	τ^{D1}_{cutoff}	τ^{D2}_{cutoff}
/ladypet	$176ms$	$421ms$
/thedea	$1120ms$	$2332ms$
/offshore	$619ms$	$1528ms$
/amondo	$194ms$	$465ms$
/thered	$158ms$	$424ms$
/thegift	$97ms$	$120ms$
/bigtrip	$146ms$	$224ms$
/thecask	$90ms$	$102ms$
/fastaRepeat	$656ms$	$1424ms$
/fannkuch	$2576ms$	$4982ms$
/spectral	$1896ms$	$4873ms$
/Donuts:id	$0.71ms$	$2.2ms$
/dirs/:id	$0.75ms$	$2.1ms$

(a) Cutoff Network Latency for device 1 and device 2.



(b) Network Latency versus Response Time



(c) Trace for Power profiles (100 times of remote executions)

Figure 9: CWV's Redistribution Adaptivity for different Devices and Network Conditions

so CWV-enabled apps improve their responsiveness by adapting to the current runtime conditions (Figures 9-(a) and (b)). Table demonstrates that the *cutoff network latency* of Device 2 (τ^{D2}_{cutoff}) is always larger than that of Device 1 (τ^{D1}_{cutoff}) to minimize the mode switching overhead. Also, these switching points reflect the device processing capacity: Device 1 is about two times faster than Device 2 for running JavaScript code on a CWV-enabled client. CWV's runtime takes into account the client device's processing capacity in addition to the current network conditions.

CWV-enabled web apps can also improve their energy efficiency by as much as **10%** for poor network conditions (Figure 9-(c)). As the longer total execution time still causes larger overall energy consumption even though the device switches into the low power mode during the idle states.

3.2 Edge Insourcing

Cloud-based execution makes it possible to take advantage of superior remote computing resources, thereby improving performance. Consider how cloud computing is used to train machine learning (ML) models on sensor data collected by mobile devices. Mobile and IoT devices feature multiple sensors, used to collect various types of data. This data then is passed as input for training ML models. The resulting models can then be used to optimize and specialize the execution of mobile devices. Since ML model training is a computationally intensive task, taking advantage of powerful cloud computing resources can improve performance. However, as the number and variety of mobile and IoT devices has drastically increased, so has the volume of resulting sensor data, referred to as *sensor data deluge* [47]. It is due to this development, transferring all the collected sensor data to a cloud-based server for processing becomes disadvantageous due to the network transmission bottlenecks. As a solution to this problem, edge computing makes it possible to train ML

models locally, so the collected sensor data would not need to be transmitted to the cloud.

To take advantage of edge processing resources, edge applications can be developed from scratch, an expensive software development task. Another more efficient approach is to modify existing cloud-based applications to integrate edge-based processing. In other words, 2-tier (cloud-device) applications would need to be transformed into 3-tier (cloud-edge-device) applications. This transformation can be quite laborious and intellectually tiresome to perform by hand. To facilitate this transformation, we will create a new domain-specific refactoring, *Edge Insourcing*. This refactoring will take advantage of Client Insourcing but will also introduce its own edge-specific transformations.

Adapting Cloud-Based Code for the Edge: The computing resources of the cloud and the edge differ widely. While cloud resources are powerful and abundant, edge resources are limited and scarce. Hence, cloud-based code cannot be directly insourced to the edge, but it must be properly adapted to consume fewer edge-based resources. To that end, the proposed research will insource only those server-based functionalities that are determined to be failure-resistant. We notice that much of server-based processing handles various erroneous inputs and failed operations. The code responsible for such processing would remain cloud-based, but the rest of server-based functionality can be insourced to executed on edge-based devices. In those cases in which edge-based execution fails, the required failure handling will be offloaded to the cloud.

Formally Verifying the Correctness of Edge Insourcing: Recently the program verification community has made significant inroads in formally verifying the correctness of distributed systems [23, 27, 45, 53]. By leveraging this work, we plan to prove that Edge Insourcing preserves application semantics. The proof strategy would require modeling both distributed and insourced executions and comparing their respective output.

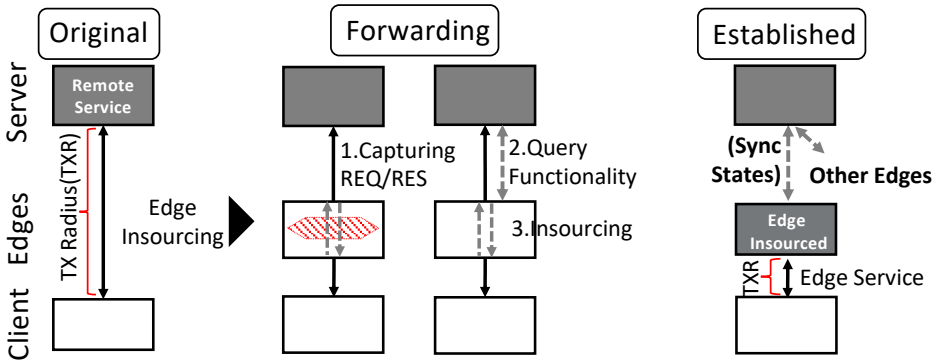


Figure 10: Distributed Architecture for Edge Insourcing

In the envisioned distributed architecture, edge nodes would serve as processing and forwarding proxies for cloud nodes (Figure 10). Initially, all the processing functionality will be hosted by the cloud nodes. An edge node would be able to issue a special insourcing request to a cloud node, receiving some program code and state to execute locally. The

state will be synchronized between the edge and cloud nodes based on a relaxed consistency model, as allowed by different application scenarios.

4 Related Work

Improving various aspects of distributed application execution has been the target of numerous prior research efforts. Since our work applies novel software engineering approaches, techniques, and tools, we limit our coverage of the related state of the art to this area.

Our reference implementation of Client Insourcing, relates to advanced program analysis techniques for JavaScript, due to its target domain—cross-platform mobile applications. Our approach follows declarative program analysis frameworks that statically analyze JavaScript code by means of a constraints solver [20, 30, 34, 49]. The JavaScript language constructs for programming event-based applications that wait for dispatches events or message asynchronously. Some advanced static analysis approaches apply formal reasoning for *callback* and *promises* of web apps based on a calculus [36, 37]. Existing dynamic analysis tools [30, 43] have a scalability problem to analyze entire JavaScript program. Dynamic symbolic execution (DSE) symbolically executes a JavaScript program by applying concrete input values [41] for a faster analysis. For more advanced DSE, MultiSE [44] uses a value summary in Jalangi2 to effectively generate testing input values of a JavaScript program to speed up dynamic symbolic execution.

In JavaScript, choosing one programming implementation over another can make a significant difference performance effect on the overall application [17, 18]. An approach presented in [42] empirically identified reappearing patterns of inefficient JavaScript programs in open source community, so common performance bottlenecks can be automatically detected and fixed by using software engineering techniques.

Some prior approaches automatically change the locality of execution in existing application, a highly complex process, as centralized and distributed execution models differ from each other in terms of their respective latency, concurrency, and failure modes [51]. Researchers and practitioners have greatly studied to facilitate the task of rendering local functionalities remote to take advantage of remote resources. For example, offloading local functionality to the cloud has also been supported as an automated refactoring technique [24, 25, 52, 54].

Client Insourcing can be seen as a variant of program synthesis [8, 10, 11, 12, 14, 15, 21, 46], an active research area concerned with producing a program that satisfies a given set of input/output relationships. CodeCarbonReply and Scalpel [7, 48] integrate portions of a C/C++ program's source in another C/C++ program by leveraging advanced program analysis techniques. The programmer's effort is only limited to annotate the code regions to integrate, and then the tool automatically adapts the receiving application's code to work seamlessly with the moved functionality. Client Insourcing belongs to a category of refactor-

ing transformations that change the locality of application components for various reasons. One prominent direction in this research is *application partitioning*, which is an automated program transformation that transforms a centralized application into its distributed counterpart [7, 32, 33, 48, 50]. Another approach that leverages compiler-based techniques is the ZØ compiler [16], which automatically partitions CSharp programs into distributed multi-tier applications by applying scalable zero-knowledge proofs of knowledge, with the goal of preserving user privacy.

Several middleware-based approach has been proposed to reduce the costs of invoking remote functionalities. APE [40] is an annotation based middleware service for continuously-running mobile (CRM) applications. APE defers remote invocations until some other applications switch the device's state to network activation. Similarly, to reduce the overhead of HTTP communication, events for HTTP requests in Android apps are automatically are bundled into a single batched network transmission [28, 29]. The e-ADAM middleware [26] optimizes energy consumption by dynamically changing various aspects of data transmission scheme. My approach can also adapt the realities of executing full-stack JavaScript mobile apps to optimize the remote execution with a cost function.

5 Schedule and Conclusion

The preliminary work is published in conferences, such as the *Web Conference* (WWW), *IEEE International Conference on Software Analysis, Evolution & Reengineering* (SANER), and the *International Conference on Web Engineering* (ICWE). Many people provided helpful feedback on my dissertation topic presented in two doctoral Symposium papers in ICWE and WWW. I also contributed to two projects, unrelated to my dissertation research, whose results were published. I am the first author in a paper in MobileSoft and the second author both in the *ACM SIGPLAN International Conference on Generative Programming: Concepts & Experiences* (GPCE) and the *Journal of Computer Language* (Table 1).

No.	Paper	Conference	Area	Authorship
1.	*Client Insourcing* [5]	Web Conference 2020 (**19%, 217/1129**)	Web Engineering	1^{st} author
2.	*D-Goldilocks* [6]	SANER 2020 (**21%, 42/199**)	Software Engineering	1^{st} author
3.	*CanDoR* [4]	ICWE 2019 (**25%, 26/106**)	Web Engineering	1^{st} author
4.	Project1-paper1 [3]	MobileSoft 2018	Software Engineering	1^{st} author
5.	Project2-paper1 [32]	GPCE 2018	Software Engineering	2^{nd} author
6.	Project2-paper2 [33]	Journal Of Computer Language 2020	Software Engineering	2^{nd} author
7.	PhD Symposium1 [1]	ICWE 2019	Web Engineering	1^{st} author
8.	PhD Symposium2 [2]	Web Conference 2020	Web Engineering	1^{st} author
Ongoing	*Comm Web Vessels*	Submitted to ICDCS 2020	Distributed Systems	1^{st} author
Ongoing	*Edge Insourcing*	To be Submitted	Distributed Systems	1^{st} author

Table 1: Publication Records in PhD Course

We intend to target distributed systems or software engineering venues to publish the remaining work. Figure 11 outlines the proposed schedule for the remaining work outlined in this document.

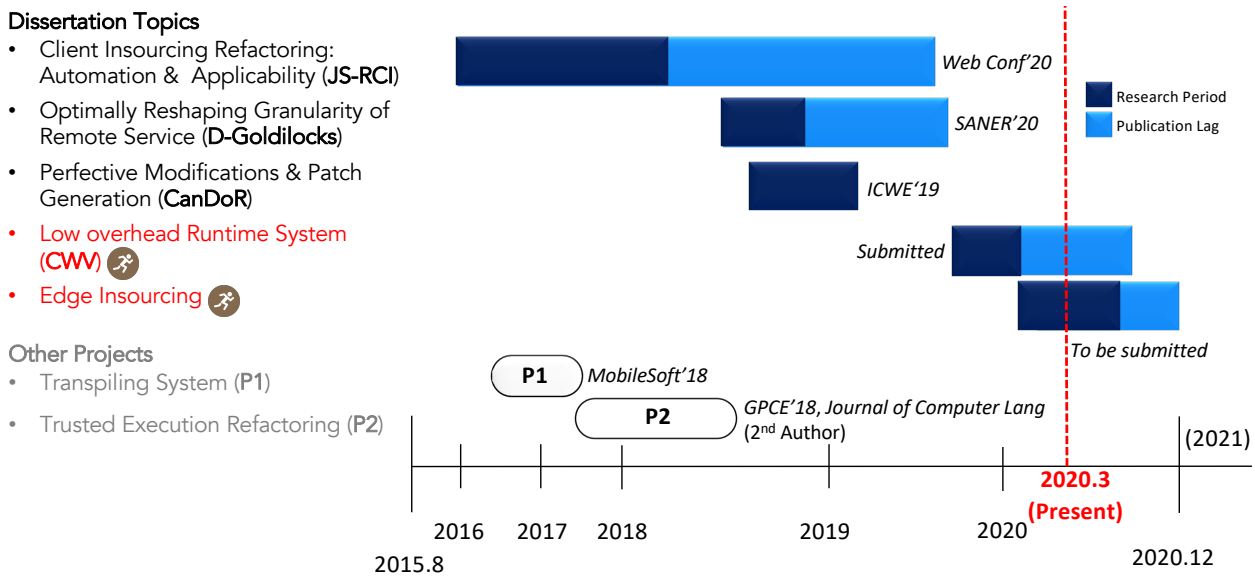


Figure 11: Proposed Work Schedule

The ever-changing realities of modern distributed apps put new obstacles on the road of providing a satisfactory user experience. In particular, modern users expect distributed applications to be responsive, reliable, and energy efficient. Distributed application developers need powerful approaches, techniques, and tools that allow them to reach these objectives on time and under budget. This dissertation research innovates in the software engineering space to create novel automated reengineering approaches that enhance and optimize distributed execution. In particular, we introduce a novel domain-specific refactoring, supported by state-of-the-art program analysis and transformation techniques, and apply this refactoring to address some of the most salient problems of distributed execution. This research contributes to the development of development approaches, techniques, and tools with novel designs, new technologies, and strong potential for practical adoption.

Bibliography

[1] Kijin An. Facilitating the evolutionary modifications in distributed apps via automated refactoring. In *Web Engineering*, pages 548–553. Springer International Publishing, 2019.

[2] Kijin An. Enhancing web app execution with automated reengineering. In *Proceedings of the Web Conference 2020*, 2020.

[3] Kijin An, Na Meng, and Eli Tilevich. Automatic inference of translation rules for native cross-platform mobile applications. In *Proceedings of the 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems(MobileSoft) 2018*, 2018.

[4] Kijin An and Eli Tilevich. Catch & release: An approach to debugging distributed full-stack JavaScript applications. In *Web Engineering*, pages 459–473, 2019.

[5] Kijin An and Eli Tilevich. Client insourcing: Bringing ops in-house for seamless re-engineering of full-stack javascript applications. In *Proceedings of the Web Conference 2020*, 2020.

[6] Kijin An and Eli Tilevich. D-goldilocks: Automatic redistribution of remote functionalities for performance and efficiency. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering(SANER)*, 2020.

[7] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, New York, NY, USA, 2015. ACM.

[8] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. In *ACM SIGPLAN Notices*, volume 52, pages 95–110. ACM, 2017.

[9] Eric J Byrne. A conceptual foundation for software re-engineering. In *Proceedings of the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.

[10] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, pages 1297–1305, 2016.

[11] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *ACM SIGPLAN Notices*, volume 53, pages 420–435. ACM, 2018.

[12] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *ACM SIGPLAN Notices*, volume 52, pages 422–436. ACM, 2017.

[13] Norman E Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on software engineering*, 25(5):675–689, 1999.

[14] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pages 229–239. ACM, 2015.

[15] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In *International Conference on Computer Aided Verification*, pages 454–471. Springer, 2016.

[16] Matthew Fredrikson and Benjamin Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 909–924, San Diego, CA, 2014. USENIX Association.

[17] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, 2015.

[18] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 94–105, 2015.

[19] Marco Guarnieri, Petar Tsankov, Tristan Buchs, Mohammad Torabi Dashti, and David Basin. Test execution checkpointing for web applications. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 203–214. ACM, 2017.

[20] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security Symposium*, volume 10, pages 78–85, Montreal, Canada, 2009. USENIX.

[21] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[22] Les Hatton. Reexamining the fault density component size connection. *IEEE software*, 14(2):89–97, 1997.

[23] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[24] Michael Hilton, Arpit Christi, Danny Dig, Michał Moskal, Sebastian Burckhardt, and Nikolai Tillmann. Refactoring local to cloud data types for mobile apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, pages 83–92. ACM, 2014.

[25] Young-Woo Kwon and Eli Tilevich. Cloud refactoring: automated transitioning to cloud-based services. *Automated Software Engineering*, 21(3):345–372, Sep 2014.

[26] Young-Woo Kwon and Eli Tilevich. Configurable and adaptive middleware for energy-efficient distributed mobile computing. In *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services (MobiCASE)*, pages 106–115. IEEE, 2014.

[27] Mohsen Lesani, Christian J Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. *ACM SIGPLAN Notices*, 51(1):357–370, 2016.

[28] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. An empirical study of the energy consumption of Android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 121–130. IEEE, 2014.

[29] Ding Li, Yingjun Lyu, Jiaping Gui, and William GJ Halfond. Automated energy optimization of HTTP requests for mobile applications. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 249–260. IEEE, 2016.

[30] Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, 2014.

[31] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410. IEEE, 2017.

[32] Yin Liu, Kijin An, and Eli Tilevich. RT-Trust: Automated refactoring for trusted execution under real-time constraints. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, pages 175–187. ACM, 2018.

[33] Yin Liu, Kijin An, and Eli Tilevich. Rt-trust: Automated refactoring for different trusted execution environments under real-time constraints. *Journal of Computer Languages*, page 100939, 2019.

[34] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005.

[35] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM.

[36] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about javascript promises. *Proc. ACM Program. Lang.*, 1(OOPSLA):86:1–86:24, October 2017.

[37] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 505–519. ACM, 2015.

[38] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[39] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 496–499, New York, NY, USA, 2011. ACM.

[40] Nima Nikzad, Octav Chipara, and William G Griswold. Ape: an annotation language and middleware for energy-efficient mobile application development. In *Proceedings of the 36th International Conference on Software Engineering*, pages 515–526. ACM, 2014.

[41] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[42] M. Selakovic and M. Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[43] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498. ACM, 2013.

[44] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853. ACM, 2015.

[45] Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

[46] Jiasi Shen and Martin C Rinard. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–285. ACM, 2019.

[47] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[48] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. Codecarboncopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 95–105. ACM, 2017.

[49] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. Static dom event dependency analysis for testing web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 447–459. ACM, 2016.

[50] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP 2002 — Object-Oriented Programming*, pages 178–204. Springer, 2002.

[51] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *International Workshop on Mobile Object Systems*, pages 49–64. Springer, 1996.

[52] Xudong Wang, Xuanzhe Liu, Ying Zhang, and Gang Huang. Migration and execution of javascript applications between mobile devices and cloud. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 83–84. ACM, 2012.

[53] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[54] Young woo Kwon and Eli Tilevich. Power-efficient and fault-tolerant distributed mobile execution. ICDCS '13. IEEE, 2013.