D-Goldilocks: Automatic Redistribution of Remote Functionalities for Performance and Efficiency

Kijin An and Eli Tilevich
Software Innovations Lab
Dept. of Computer Science, Virginia Tech
Blacksburg, VA, United States
{ankijin,tilevich}@vt.edu

***Abstract*—Distributed applications enhance their execution by using remote resources. However, distributed execution incurs communication, synchronization, fault-handling, and security 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. In this paper, we present 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. To that end, our approach insources a remote functionality for local execution, splits it into separate functions to profile their performance, and determines the optimal redistribution based on a cost function. Redistribution strategies combine separate functions into single remotely invocable units. To automate all the required program transformations, our approach introduces a series of domain-specific automatic refactorings. We have concretely realized our approach as an analysis and automatic program transformation infrastructure for the important domain of full-stack JavaScript applications, and evaluated its value, utility, and performance on a series of real-world cross-platform mobile apps. Our evaluation results indicate that our approach can become a useful tool for software developers charged with the challenges of re-architecting distributed applications.**

***Keywords*-distributed applications; re-architecting; refactoring;**

I. Introduction

Distribution has become part and parcel of the majority of computing domains. By using remote resources, a distributed application can enhance its functionality or improve its quality of service. Sometimes distribution is inevitable, when certain resources can be accessed only remotely. In other cases, distribution is a choice: the same functionality can be executed by means of local or remote resources. Distributed execution is not free though—invoking remote functionality incurs communication, synchronization, and programming effort costs. These costs must be offset by the attendant execution enhancement for distribution to remain beneficial. Otherwise, distributed execution only incurs overheads, which hurt both performance and maintainability. Of course, if some functionality can be accessed only remotely, these overheads are justified and unavoidable. However, if distribution is introduced to improve an application's quality of service, it must be introduced at the right level of granularity, when opting to execute a functionality remotely over the network indeed improves application performance. Introducing *too much distribution* for the expected benefits is a known architectural problem, documented as the *Nano-Service Anti-Pattern* [1].

Distributed execution is often used to improve performance. For example, in mobile apps, as the computing resources of remote, cloud-based servers surpass those of mobile devices, a functionality can be executed faster remotely than locally. However, executing a remote cloud-based functionality requires passing parameters and receiving results over the network. Network communication significantly complicates the device/cloud performance equation. Transferring data across a network imposes latency and energy consumption costs. For low-latency, high-bandwidth networks, these costs are negligible. For limited networks, these costs can grow rapidly and unexpectedly, as operating over high-loss networks requires retransmission, which consumes additional battery power. Hence, the added overhead of network transfer burdens the mobile device's battery budgets, often negating the performance benefits of executing a local functionality at a remote cloud-based server [2].

To maximize the benefits of distribution, it has to follow *the Goldilocks Principle* [3], [4]: it cannot be introduced at a granularity that is too fine or too crude; it must be just right. In this paper, we study the problem of re-architecting distributed applications to adjust their distribution granularity as a means of improving performance and efficiency. Our approach comprises the following steps: (1) our novel domain-specific refactoring—*Client Insourcing*—automatically integrates a remote functionality with the local code, creating a semantically equivalent centralized application variant; (2) the variant's performance is profiled, with a cost function-based heuristic determining how to reshape the original distribution to improve performance and efficiency; (3) a series of automatic refactorings reshape and redistribute the original remote functionality into new remotely invoked units, whose aggregate latency and resource consumption are minimized thus boosting distribution benefits.

The contribution of this paper is four-fold:

1) An automatic redistribution approach for distributed ap-

plications that improves their performance and efficiency by reshaping their remote functionalities.
2) A set of domain-specific automatic refactorings for moving remote functionalities to the client as well as their reshaping and redistribution.
3) A cost function-based heuristic for identifying how to improve the performance of distributed applications by reshaping their remote functionality.
4) A reference implementation of our approach that concretely realizes our design and a systematic evaluation of our approach's value, utility, and performance.

The rest of this paper is structured as follows. Section II describes our approach for assessing and improving the utility of distributed functionality. Section III introduces the reference implementation of our approach. Section IV evaluates the development and performance aspects of our approach. Section V discusses the applicability and limitations of our approach. Section VI compares our approach to the related state of the art. Section VII presents concluding remarks and outlines future work directions.

II. Assessing and Improving the Utility of Distributed Functionality

We target distributed applications that comprise the client and server parts, communicating with each other by means of distribution middleware, such as the HTTPClient library or CORBA [5]. Our application domain are *full-stack JavaScript applications*, in which both the client and server parts are written in JavaScript; this domain is becoming increasingly widespread due to the popularity of Node.js and other server-side JavaScript frameworks. The client invokes server-side remote functionality, which executes corresponding code and returns back the results to the client. The client passes input parameters, and the server returns results. The middleware mechanism serializes and deserializes both the parameters and results to transfer them across the network and make them available for computation.

A. Motivating Example

Consider *Bookworm*[1], a book reader implemented as a full-stack JavaScript mobile app. In addition to enabling users to read books on their mobile devices, *Bookworm* has a feature that reports statistical information extracted from the text of the books. To that end, the app features a remote service that given a book title, analyzes its text and returns the results of this analysis. Because text processing is computationally intensive, it is commonly performed remotely at a powerful server rather than locally on a mobile device. The original implementation of this remote text analysis service runs all analysis tasks (e.g., overall length, punctuation percentage, unique vocabulary, etc.) in sequence, returning their results in bulk. For large books, waiting for all the tasks to complete before any results become available can degrade the user experience. Hence, a possible restructuring could separate the unit containing all sequentially performed analysis tasks into multiple asynchronous units, each of which immediately returning the computed results back to the client.

[1]https://github.com/davidwoodsandersen/Bookworm

As it turns out, the majority of the resulting remote executions for analysis tasks (e.g., overall length, punctuation percentage, etc.) are not computationally intensive. However, the client consumes additional resources to execute these tasks by performing multiple remote invocations. The only analysis task that involves heavy processing and takes a long time to execute is "extract unique vocabulary." To minimize the overall latency of invoking these text analysis tasks, they can be restructured into two remote services: one to invoke "extract unique vocabulary" and the other one to invoke the remaining analysis tasks in bulk.

To determine the optimal structuring and distribution of the text analysis tasks would require profiling their execution under different inputs. Hence, the remote analysis services need to be both restructured and redistributed, a non-trivial re-engineering task. The approach presented herein systematically identifies what an optimal distribution is for a given optimization criteria and presents automated program transformations that eliminate much of the engineering complexity of redistribution.

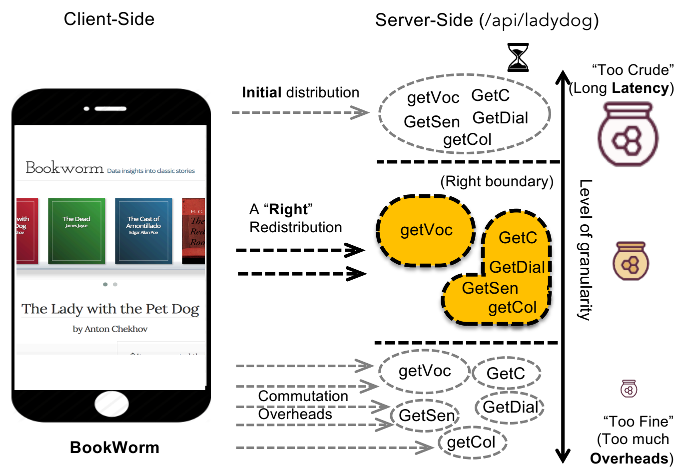

Fig. 1. Motivating Distributed Mobile App `Bookworm`

B. Distribution Execution Cost Function

In the motivating example above, we show how redistribution can optimize remote services. Rather than trying to determine redistribution optimization opportunities through trial-and-error, distributed app developers need intuitive numerical models that can inform them about the actual cost/benefit ratio of remote services. To that end, we formulate the *distribution execution cost function*.

Problem Formulation: Our goal is to determine which functional distribution from the client's standpoint would minimize the following cost of distributed execution function:

$$C^{Dist_Exec}(r_i) = \alpha \cdot latency(r_i) + (1-\alpha) \cdot \sum resource(r_i).$$

Intuitively, executing a functionality remotely reduces the computational load on the client at the cost of the delay, measured as the remote invocation's latency, and the local resources consumed to make this invocation. Typically it is distribution middleware that consumes these additional resources, including computation, memory, and energy. The invocation latency measures the expected deterioration in the user experience—the more time the user has to wait for a remote functionality to complete, the less satisfying their experience will be. Hence, the distribution cost is the sum of the expected deterioration in the user experience and the amount of additional local resources consumed to invoke the remote functionality. Hence, the optimization objective is to identify the most favorable remote utility/client cost ratio. Our optimization strategy strives to determine the level of granularity for remote services that maximizes this ratio.

Consider a long-running bulk remote service r. Since invoking this service takes a long time, even with low client-side resource utilization, the client cost may not be as favorable. One can break up this long-running service r into a collection of smaller services $r_1^{'}, \ldots, r_k^{'}$. Assume that these smaller services are not inter-dependent and now can be invoked asynchronously. First of all, the original computational work being offloaded to the server or the *utility* for r remains unchanged by this redistribution. However, the combined latency of invoking these smaller services would decrease, but the consumed client resources would increase due to multiple invocations, so the resulting cost (or utility/client cost ratio) may not decrease. A more optimal redistribution in this scenario may be to combine some of these smaller remote services into one to decrease the resources that the client would consume to invoke them. Hence, the cost of distribution function is defined as the sum of the normalized execution latency and client-consumed resources required to invoke a remote service. It is the weight factor α that normalizes the latency and resource consumption terms.

Problem Solution Outline: We estimate an optimal distribution for a remote service by minimizing the cost of invoking the service's constituent functionalities. To that end, these functionalities can be invoked individually or in bulk in different combinations. The following two operations express the required program restructurings:

- $[r_1^{'}, \ldots, r_k^{'}]$=$partition(r)$: partitions a remote service r into k independent parts, each of which becomes an individually invocable remote functionality.
- R_h=$batch([r_0, \ldots, r_n])$: batches n remote functionalities into a larger remote service R_h.

Notice that the *batch* operation may be applied multiple times to different remote services to achieve the required service combinations. D-GOLDILOCKS implement a divide & conquer algorithm that by means of *partition* and *batch* identifies a distribution that minimizes the C^{Dist} function.

C. Client Insourcing to Restructure Remote Services

The *partition* and *batch* operations work with regular JavaScript functions rather than remote services. Hence, to transform remote services into local functions, we introduce a domain-specific automated refactoring—*Client Insourcing*, which given a remote functionality invoked via middleware, integrates that functionality with the client code. Client Insourcing undoes the current distribution r, thus creating a centralized variant r_{local} that can be redistributed differently. Section III-A describes the technical details of Client Insourcing.

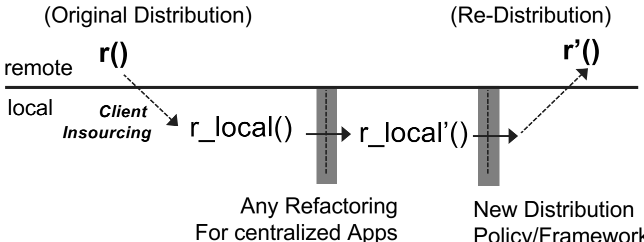

Fig. 2. Restructuring a remote service r into r' remotely with Client Insourcing

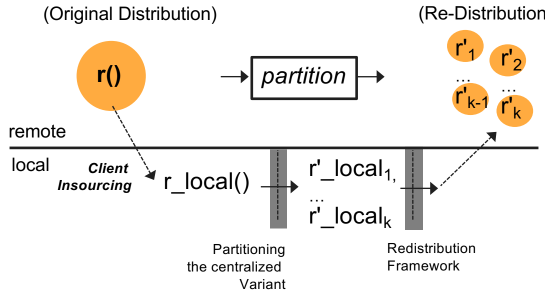

Fig. 3. Partitioning remote service r via Client Insourcing

D. Partitioning Insourced Functions

Based on the ability of Client Insourcing for restructuring, our approach can partition a remote service into multiple remote parts (Figure 4). Assume a remote function r consists of some distinct functionalities. To identify independently invocable parts, we can perform Client Insourcing on r, making it a centralized function r_local. Then, we can apply program analysis and refactoring for centralized apps to split the r_local into k distinct functions $r'_local_1, \cdots, r'_local_k$. Finally, k distinct functions can be distributed becoming $r_1^{'}, \cdots, r_k^{'}$.

E. Batching Remote Invocations (BRI)

For networking environments with large bandwidth and high latency, it may be advantageous to batch multiple remote invocations into a single one to reduce the aggregate latency. The research literature describes several approaches that implement this optimization. The Data Transfer Object (DTO) and Remote Façade [6] design patterns aggregate individual services. Remote Façade exposes multiple fine-grained services via a coarse-grained remote interface. DTO serves as a bulk object for transferring parameters and results of

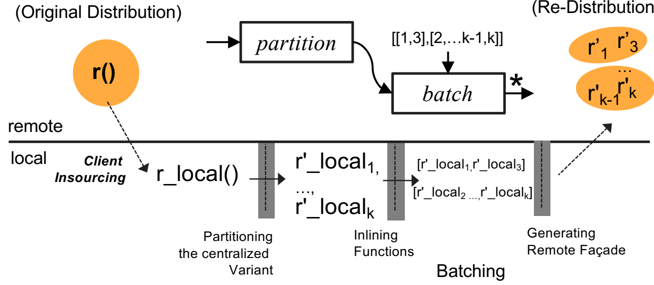

Fig. 4. Partitioning/Batching remote service r

remote service invocations. Remote Batch Invocations (RBI) provides language support for creating DTOs for combining the invocation of arbitrary remote services, which can also be intermixed with local operations [7].

Our approach batches fine-grained distributions by automatically generating Remote Façades. First, the small remote functionality is insourced to become local. Then, the resulting independent local functions are inlined into a single function using the *Inline Function* refactoring. That single function then becomes a new unit of distribution.

Combining to the $partition$ operation, our approach can arbitrarily generate Remote Façades for the k distinct functions $r'_local_1, \cdots, r'_local_k$. For an instance, a subset $\{r'_local_1, r'_local_3\}$ and are inlined into a single function $r'_local_{1_3}$, which is then distributed into r'_{1_3}. In essence, a combination of these automated refactorings creates a distributed execution that would be similar to the result of implementing the Remote Façade optimization by hand.

III. Reference Implementation: D-Goldilocks

We concretely realized our approach as a series of automated refactorings. These refactorings both migrate functionalities between different hosts and also restructure the granularity of remote service invocations Figure 5 shows our overall automated refactoring approach.

A. Client Insourcing Refactoring

Client Insourcing is a domain-specific[2] refactoring that automatically integrates a remote functionality with the client code. Because our application domain is JavaScript applications, and JavaScript is known to defeat static analysis techniques, Client Insourcing includes a dynamic analysis phase to identify the exact boundaries of the server functionality to insource. The programmer is only required to annotate the invocation points of the remote functionality to insource. These points correspond to the locations in the client code, at which remote invocation parameters are serialized to be transferred across the network, and the remote service's results are unserialized to be used in the subsequent program steps. Intuitively, Client Insourcing identifies serialization/unserialization points in the client code is to detect the entry/exit execution points of the remote functionality to insource. To that end, Client Insourcing extracts the passed parameters and the return results in the recorded HTTP traffics.

Algorithm 1: Client Insourcing Refactoring

1 Client_Insourcing (C_{src}, S_{src});
Input : C_{src}:client code, S_{src}:server code
Output: $C_{src}^{insourced}$: Centralized App for C_{src}, S_{src}
2 S_{n_src}=normalize(S_{src});
```
/* add Instrumenting code(Jalangi2/JS code) */
```
3 [C_{src}^{inst},$S_{n_src}^{inst}$]=addInstrument(C_{src},S_{n_src});
```
/* Instrumenting the server/client code to
   identify entry and exit points          */
```
4 Log=remoteExecution(C_{src}^{inst}, $C_{n_src}^{inst}$);
```
/* Loading JavaScript program rules         */
```
5 $Rule_{nodejs}$=loadNodejsProgramRules();
```
/* Generate facts for server code           */
```
6 $Fact_{S_{src}}$=genProgramFact(S_{n_src});
7 $Model_{S_{src}}$ = $Rule_{nodejs}$ + $Fact_{S_{n_src}}$;
```
/* Check dependency for candidate points    */
/* Query dep. stmts for entry/exit          */
```
8 $Stmts_{entry}$=queryDepStmt($Log.P_{entry}$,$Model_{S_{src}}$);
9 $Stmts_{exit}$=queryDepStmt($Log.P_{exit}$,$Model_{S_{src}}$);
```
/* Cutting dependent JS statements          */
```
10 $Stmts_{dep}$=$Stmts_{exit} - Stmts_{entry}$;
```
/* Make a regular ftn with adaptation       */
```
11 f_{local}=compGen($Stmts_{dep}$, $Log.(P_{entry}, P_{exit})$);
/* Add the f_{local} with Logged Position */
12 $C_{src}^{insourced}$=compAdder(f_{local},$Log.pos$,C_{src});
/* Return insourced version of C_{src} */
13 return [$C_{src}^{insourced}$, f_{local}];

The design Client Insourcing of follows that of other declarative program analysis frameworks [8]–[10] that analyze JavaScript using the z3 SMT solver. To analyze server code written by means of the Node.js framework, Client Insourcing defines its own sets of z3 rules for Node.js and that of facts for subject programs. The profiled parameters and return results are added as new z3 facts to be able to reason about the entry and exit points of the remote execution. Client Insourcing generates local functions by solving constraint problems with z3. First, Client Insourcing checks the dependency between the entry and exit point candidates to locate the correct pair of points. Next, it finds a subset of dependent statements for the exit point in the server part and the subset of the dependent statements from the entry point. Client Insourcing generates local functions by differencing the entry and exit sets. Finally, the generated functions change the call structure of the client code into regular local calls by using the recorded insertion points. Algorithm 1 shows the overall refactoring procedure.

B. Partitioning a Function into Individually Invoked Functions

To partition a JavaScript function into individually invoked functions, D-GOLDILOCKS first applies static analysis to determine the dependencies of the function-to-partition. These dependencies comprise all references to global references and the

[2]its application domain are full-stack JavaScript applications

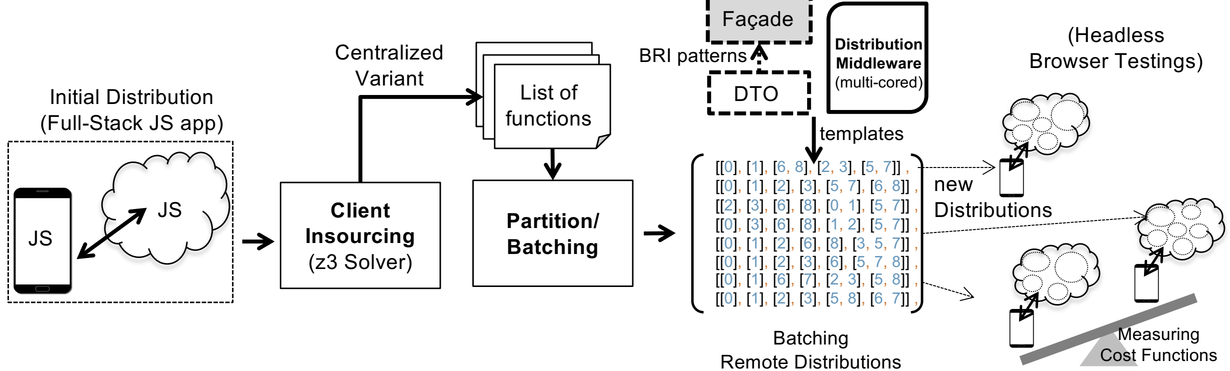

Fig. 5. Process for D-GOLDILOCKS

```
//original Client:app.js
$scope.getLadyWithPetDog =
    function() {...
  $http.get('/api/ladypet').then(
      function(response){
     var text = response.data; ...
    });/*remote invocation*/ }
//original Server:server.js
function getSenAvg(array){...};
function getVoca(str){...};
...
app.get('/api/ladypet',
   function(req, res){...});
```

```
//after Client Insourcing:app.js
//Insourced remote functions
function getSenAvg(array){...};
function getVoca(str){...};
...
function ladypet_local(){
//invoke every subtasks
...};
$scope.getLadyWithPetDog =
    function() {...
    //from remote to local
    var text = ladypet_local();
...}
```

```
//after Redistribution:index.html
<!DOCTYPE html>
<script src="./app.js">
 ...
 ClientDTO.b_param = BATCH_PARAM;
//Batched Invocations:
 getSenAvg=ClientDTO(getSenAvg);
 getVoca=ClientDTO(getVoca);
 ...
</script>
```

Fig. 6. Redistribution Steps

invocations of other functions. To that end, D-GOLDILOCKS traverses the function's control-flow graph[3] in the depth-first order. Then a greedy algorithm is applied to determine the maximum number of partitions, each of which is an independently invocable function. The algorithm strives to produce the highest number of candidate partitions, with the following exceptions: 1) mutually dependent partitions as indicated by the original function's call graph or 2) partitions that share global variables. Such candidate partitions are merged into a single one.

C. Batching Remote Invocations

D-GOLDILOCKS automatically generates a client-side DTO and remote Façade stubs for batching the small remote services. The actual Remote Façade function, invoking the original services, becomes the new entry point of the remote execution. The client DTO stub accumulates the remote invocations of the fine-grained services at the client before transferring them in bulk to the remote Façade function; the `BATCH_PARAM` parameter to the batch specification becomes the number of service invocations to accumulate. The remote Façade function sequentially (or synchronously) invokes the bundled services and returns their execution results combined into a single value in bulk. For the following specification, D-GOLDILOCKS generates a remote Façade `f1name_f2name` with the concatenated function names of the original fine-grained services `f1name` and `f2name` (Figure 7).

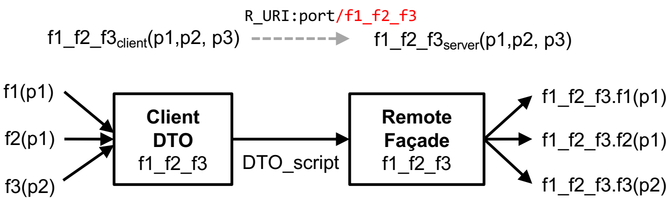

Fig. 7. Batching Invocation of f1,f2, and f3 by means of Client DTO and Remote Façade

D. Redistribution Steps

The three code snippets in Figure 6 show the original client and server parts, their centralized insourced version, and a redistributed client part.

Client Insourcing automatically transforms a client server distributed interaction into a centralized counterpart, moving server functions to the client and replacing all middleware invocations with local function calls. When insourcing a remote functionality, all its dependent server-side code has to be copied to the client. That code may be scattered around multiple functions and standalone declarations. Each insourced remote functionality is placed in a single function, added to the client codebase. Client Insourcing is similar to the *Extract*

[3]https://github.com/wala/JS_WALA

Function refactoring. Both refactorings create a newly named function and the call sites to invoke it. Client Insourcing differs in moving the extracted code from the server to the client and replacing middleware functionality with local calls. The middle column of Figure 6 shows the centralized variant produced by Client Insourcing the code in the left column.

This centralized variant is used for profiling and redistribution. In this example, two of the original server functions are batched into a single function, invoked in the same remote roundtrip. The batching operation is implemented via the Data Transfer Object (DTO) pattern on the client and the Façade pattern on the server.

E. Distribution Framework: Transforming Local Functions into Remote Services

D-GOLDILOCKS implements a framework for seamlessly transforming local JavaScript functions into remote services. D-GOLDILOCKS maintains a list of local (insourced) functions that implement the business requirements. D-GOLDILOCKS uses the mustache.js framework[4] to generate different client/server combinations of the insourced functions. The resulting client and server parts communicate with each other by means of Ajax and the Express.js middleware, as the majority of our subject apps already use this middleware. For the server to explicitly handle concurrent executions, it is enhanced with a multi-core engine[5] for the node.js. These frameworks introduce the required distribution with minimal changes. The newly redistributed functions only need to unmarshal their parameters and marshal their results.

IV. Evaluation

Our evaluation seeks answers to the following questions:

- **RQ1:—Value:** How much programmer effort is saved by D-GOLDILOCKS's automatic redistribution operations?
- **RQ2:—Cost Model Correctness:** How applying the partition and batch operations affect the distributed execution's "latency" and "consumed resources" attributes?
- **RQ3:—Utility of Cost Model for Redistribution:** How useful is the cost function for guiding redistribution decisions?
- **RQ4:—Energy Consumption:** What is the effect of redistribution on the amount of energy consumed by the client?

A. Evaluation Setup

1) Dataset: Our evaluation subjects are real-world full stack distributed mobile JavaScript applications and benchmarks from the *extremeJS* [11]–[13]. *extremeJS* built remote services over their distributing framework focusing on JavaScript offloading. We tested their remote functionalities only changing the server middleware from their distributing framework (V8 within a C++ app) into Express.js.

2) Latency and CPU Utilization of Remote Services: We profile remote services under standard loads in terms of the latency and resources for the client. We use a V8 profiler[6], which supports the line-by-line performance profiling of JavaScript programs. D-GOLDILOCKS injects probes into the instrumented source code and collects samples, which contain the execution times (L in Table I) and CPU utilization levels for each block. By summing these CPU levels, We calculate the resource consumed by a remote execution ($\sum T_{cpu}$ in Table I). Computationally intensive benchmarks with remote functionalities always exhibit a high latency. We ran our measurements over headless browser testing frameworks[7] to emulate a real world's web client applications. The remote server is hosted by DELL-OPTIPLEX5050 and we execute remote services over a stable WiFi network.

TABLE I
SUBJECT REMOTE SERVICES

Remote Serv	L(ms)	$\sum T_{cpu}$	f_{CI}^{LOC}	f^{decl}	f_{sub}^{ind}	$\|D\|$
/api/ladypet	77.38	337	394	9	8	1.6M
/api/thedea	164.62	695	394	9	8	1.6M
/api/thered	42.96	370	394	9	8	1.6M
/api/thegift	37.69	390	394	9	8	1.6M
/api/bigtrip	42.11	304	394	9	8	1.6M
/api/offshore	30.82	400	394	9	8	1.6M
/api/wallpaper	56.2	396	394	9	8	1.6M
/api/thecask	20.6	432	394	9	8	1.6M
/string-fasta	29.85	328	38	5	2	76
/cflow-rec	35.43	326	49	4	3	245
/prprty/brokers	20.64	323	379	3	3	1516
/prprty/brokerId	15.62	332	382	3	3	1528

B. Evaluating Software Engineering Value

1) Programmer Effort Saved: To answer **RQ1**, we estimate the value of D-GOLDILOCKS automatically generating JavaScript code. As an automated refactoring, Client Insourcing saves programmer effort required to move remote functionality to the client, so it can be invoked via local function calls. We count the number of uncommented lines of JavaScript code (LOC) that need to be edited by hand to perform the refactoring. Notice that Client Insourcing transformations involve two phases: generating local functions and replacing middleware invocations with local calls. The local functions are generated by copying the server-side code, which becomes the body of new client-side functions, whose parameters and returns values are automatically inferred from the corresponding server entry/exit points (f_{CI}^{LOC}). The original middleware functionality is replaced with local calls. For instance, Client Insourcing the `/api/ladypet` remote service generates a local function of 394 ULOCs. Another D-GOLDILOCKS's operation is partitioning an insourced function f^{LOC_CI} into smaller individually invoked functions $f_{CI_1}^{LOC} ... f_{CI_n}^{LOC}$. For each subject, we report the number of the resulting functions f_{sub}^{ind}. The final D-GOLDILOCKS's operation is batching individually invoked

[4]https://github.com/janl/mustache.js
[5]http://learnboost.github.io/cluster/
[6]https://github.com/node-inspector/v8-profiler
[7]https://github.com/GoogleChrome/puppeteer, https://github.com/jsdom/jsdom

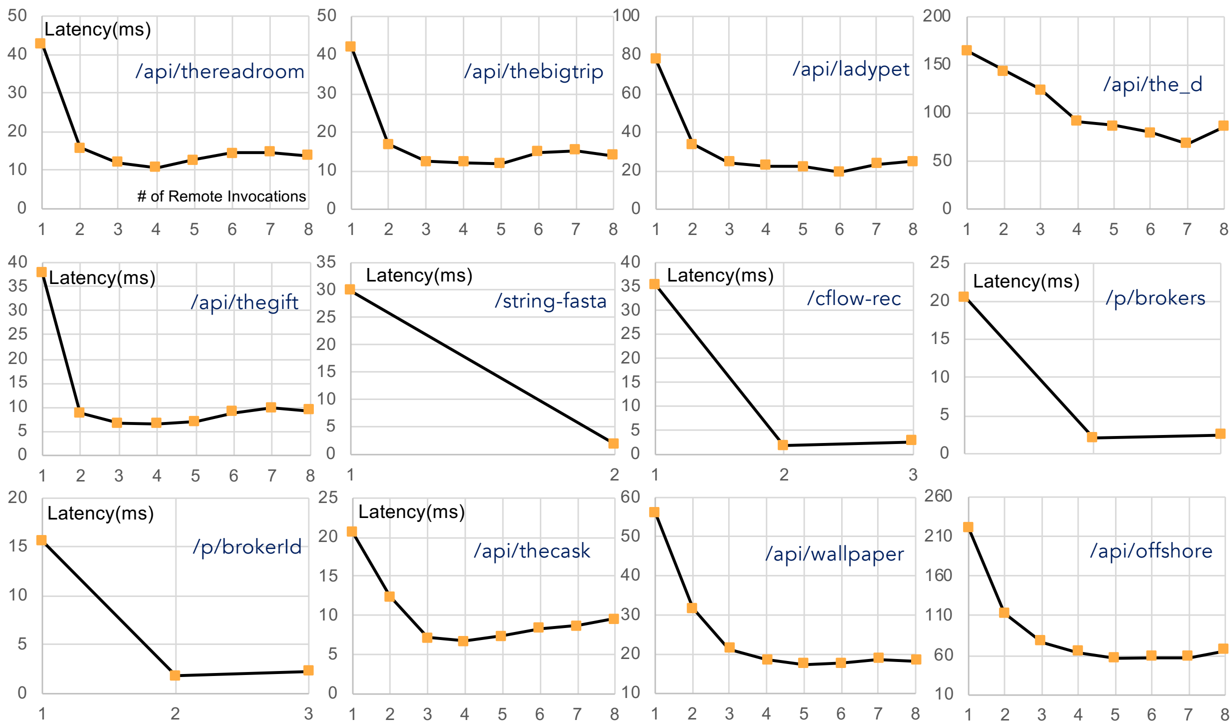

Fig. 8. Latency(ms) versus the number of Remote Invocations

functions into a larger function. To be able to determine what the optimal combination of function is, D-GOLDILOCKS generates all possible combinations of individually invoked functions. Hence,we estimate the saved manual programming effort, $|D|$, as the product of f_{CI}^{LOC} and all possible combinations of f_{sub}^{ind}. Because of the combinatorial explosion, the values of $|D|$ tend to be too large for any reasonable manual treatment. For example, $|D|$ for `/api/ladypet` is $394 \times 4,139 \cong 1.6 \times 10^6$ ULOCs.

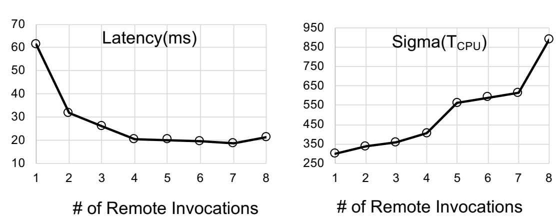

Fig. 9. Scales between Latency and CPU Usage

2) Refactoring Impact: D-GOLDILOCKS redistributes a remote functionality by insourcing it, partitioning it into parts, and batching these parts into new individually invoked remote functionalities. In this experiment, we assess the actual performance impact of the number of batched parts on the resulting invocation latency and consumed resource (**RQ2**). Figures 8 and 9 show the observed metrics for our experimental subject applications. The larger the number of new remote functionalities, the smaller is the aggregate average latency incurred by invoking them. The latency drops precipitously as the number of functionalities start growing, but than flattens due to the additional overhead of multiple remote invocations. Whereas, the overhead or the CPU usage proportionally increases with the number of new remote functionalities.

3) Utility of Redistribution Cost Model: To answer **RQ3**, we applied our cost function to different redistribution scenarios of our subjects. We empirically determined the required normalizing factor for the latency and sum of CPU usages terms by scaling the observed latency/CPU usage ratios across all measurements (See Figure 9).

$$C^{Dist_Exec}(r_i) = \alpha \cdot L(r_i) + (1-\alpha) \cdot \sum T_{cpu}(r_i)$$

, where $\alpha = \overline{L}/\overline{\Sigma T_{cpu}} = \mathbf{0.9281}$.

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 10).

Figure 11 shows how two the optimal distributions of `/api/theread` and `/api/thegift` bring the distributed execution cost down to the minimums. Recall that the task of getting unique vocabulary (`getVocabulary`) was relatively computationally intensive, as compared to other tasks.

The optimal distribution comprises three individually invoked remote services, extracted by partitioning `getVocabulary` into smallest possible functions and then batching them to minimize the aggregate latency and CPU utilization.

C. Evaluating Performance and Energy Consumption

To answer **RQ4**, we measure the amount of energy consumed by a mobile device to execute remote services over

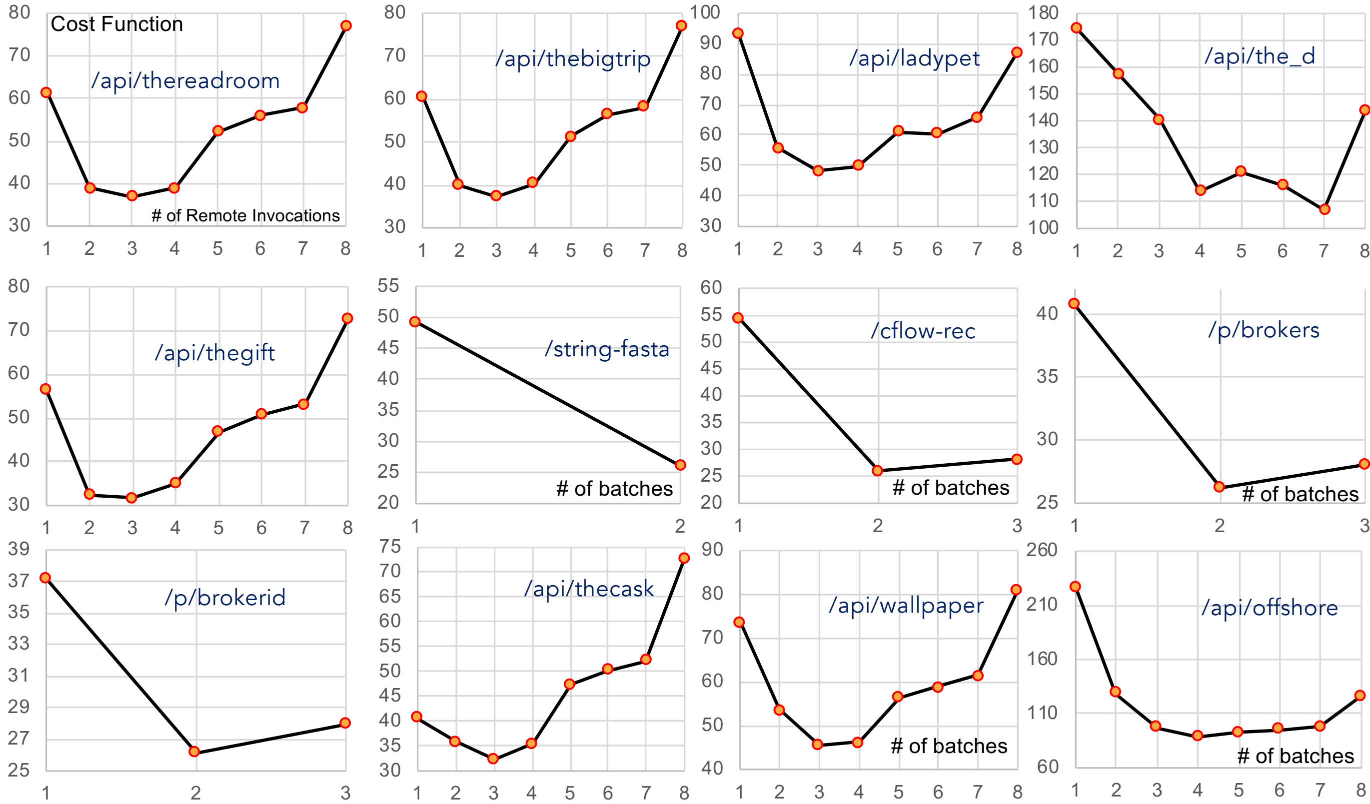

Fig. 10. Cost Functions versus the number of Remote Invocations

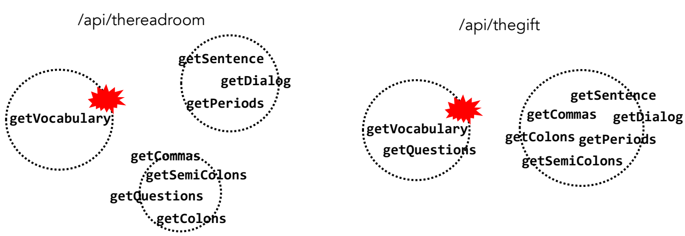

Fig. 11. Examples of Optimal Distributions for `/api/thereadroom` and `/api/thegift`

a stable WiFi network. The client device, QISKIW-L24-HUAWEI, runs Android Marshmallow, and the remote server is hosted by DELL-OPTIPLEX5050. We use PowerTutor [14], a model-based energy profiler for mobile apps, to estimate energy consumption (EC).

We report on the energy amounts consumed in three deployments of the *Bookworm* application: (1) the original distributed execution, $EC_{original_dist}$, (2) the best distributed execution achieved via redistribution, EC_{best_dist}, and (3) the worst distributed execution achieved via redistribution, EC_{worst_dist}. Figure 11 shows the best distribution (2 total remote invocations). The worst distribution makes 8 remote invocations, while the original version makes 1 remote invocation. As it turns out, the original distribution consumes the lowest amount of energy, $EC_{original_dist}$=8.4mJ, with the best distribution not far behind, EC_{best_dist}=13.4mJ. The worst distribution is an energy guzzler, consuming 6 times as much energy as the original version, EC_{worst_dist}=47.4mJ.

V. DISCUSSION

Our experimental results are subject to both internal and external threats to validity. Our approach also has applicability constraints. We discuss these and other issues in turn next.

A. Internal Validity

To redistribute our subject applications, we use the Express, JQuery, and Cluster frameworks. The way these frameworks introduce distribution may certainly affect the performance of the resulting distributed applications. By using these frameworks in a black-box fashion, we have no control over how they implement their remote execution features. Since our redistribution phase starts from a centralized JavaScript variant, any other distribution frameworks can be used in place instead, possibly resulting in differently performing distributed applications. Nevertheless, these differences would be unlikely to change the overall performance profile of the redistributed applications. As our measurements show, the performance latency of remote invocations is dominated by network communication and the server's computational load. The choice of distribution middleware would have a marginal impact on the performance of these functionalities.

As our units of distribution, we use existing functions. Another possibility would be to consider splitting existing functions into smaller units that can be distributed independently. Measuring the performance of and redistributing code

at the level of granularity of existing functions certainly have impacted our performance results. Nevertheless, in this work we aim at a fully automatic approach to determining which distribution would be optimal. It would be impossible to automate the process of breaking up existing functions into meaningful constituent blocks.

B. External Validity

D-GOLDILOCKS makes all redistribution decisions based on the obtained performance characteristics of the application, whose remote functionality has been insourced. Our implementation relies on the V8 profiler to measure the performance of such applications. It is possible that other profilers could show different performance numbers, thus affecting D-GOLDILOCKS's redistribution recommendations. Network connectivity can also affect our experimental results. All our experiments were conduced over a stable WiFi network connection. Operating over limited unstable networks would incur higher energy consumption overheads. D-GOLDILOCKS applies a cost function to decide whether a given distribution needs to be fine-tuned. One may disagree with this heuristic and choose a different one, particularly well-suited for certain application domains. Even if one completely rejects the validity of our decision-making heuristic, our overall redistribution approach still has value. The ability to reshape centralized functionality before redistributing the result is a new promising approach to optimize the execution of distributed applications.

C. Applicability and Limitations

Distributed execution is always a result of certain architectural decisions. D-GOLDILOCKS makes it possible for developers to revisit these decisions, without resorting to prohibitively expensive manual code modifications. Instead, D-GOLDILOCKS relies on domain-specific and general refactoring transformations. Hence, developers who use D-GOLDILOCKS are still required to understand the original distributed application's architecture. As is usually the case, D-GOLDILOCKS eliminates much of the *accidental* rather than *essential* complexity of architecting distributed applications [15].

All our evaluation examples are full-stack JavaScript applications. However, conceptually our approach is quite general and should be applicable to any distributed application domain. However, other domains may require additional engineering effort. Although full-stack JavaScript applications have become extremely popular, to redistribute JavaScript code to execution platforms that use a different programming language, one may be able to apply language-to-language translation, an approach whose success can differ widely depending on the source and target languages.

VI. Related Work

Our approach to redistributing full-stack JavaScript applications relies on the dynamic analysis of JavaScript programs, middleware design, automated software transformation, and domain-specific refactoring. We briefly describe the most closely related state of the art next.

Dynamic analysis approaches have been applied to ascertain the properties of JavaScript programs for various purposes. An empirical study identifies the factors that affect the performance of JavaScript programs by using browser profiling engines [16]. Dynamic analysis is also used to detect Just-In-Time(JIT)-unfriendly JavaScript code, suggesting a refactoring to improve performance. JITProf [17] measures how prevalent JIT-unfriendly code is, with the goal of helping developers detect such code locations. DLint [18] proposes a dynamic checker for bad coding practices in JavaScript, as based on formal descriptions. JSweeter [19] detects performance bottlenecks that are related to the type mutation of the V8 engine.

As communication overhead can be critical in mobile application execution, middleware-based approaches optimize the network overhead in distributed mobile applications. To reduce the number of network requests, repeated HTTP request are detected and bundled [20]. Middleware e-ADAM [21] optimizes energy consumption by using middleware functionality that uses data communication, encoding, and compression. APE [22] is an annotation-based middleware service for continuously-running mobile (CRM) applications. APE distributed execution is deferred until other applications cause the device to switch into the network activation state. These approaches optimize various performance aspects of distributed mobile execution by reducing the network communication overheads. In contract, our approach considers the entire distributed application architecture, reshaping both the client and server parts and restructuring the communication functionality between them.

Recently, several techniques have been introduced that can automatically integrate portions of a program's source code into another program. Systems that include CodeCarbonReply and Scalpel [23], [24] support this functionality for C/C++ programs. Similarly to the D-GOLDILOCKS workflow, the programmer starts by annotating the code regions to integrate, and then a program transformation tool automatically adapts the receiving application's code to work with the transferred functionality. If the client and server parts of the same application are treated as separate programs, D-GOLDILOCKS can be also seen as a tool that integrates server-side code with the client code. However, another important feature of D-GOLDILOCKS is automatically replacing middleware functionality connecting the client and server parts with direct function calls. Finally, D-GOLDILOCKS focuses on improving performance by reshaping the granularity of remote functionality.

The research community has devoted a considerable effort to automate the program transformations required to render local functionality remote to the rest of the computation [25]–[27], an architectural change that is the opposite to that of *Client Insourcing*. The client-to-remote architecture evolution has even been supported as a refactoring transformation called *cloud refactoring*. Specifically, energy efficiency one of the foremost design and implementation concerns for mobile

apps. Currently, moving energy intensive functionality to the cloud has been promoted as a way to reduce the energy consumption. However, accessing a remote component incurs network communication and middleware processing costs, which can offset the energy savings provided by executing the component at a remote server. As confirmed by our evaluation, by eliminating any remote communication and removing middleware functionality [28]. As we discovered in this work, the realities of modern software development often require *both* types of architectural transformations, in which the locality of application functionality can be changed at will, from local to remote and vice versa.

VII. Conclusion and Future Work

We plan to continue exploring the range of applicability of our approach. Two additional performance aspect we want to explore are multiple clients/servers and dissimilar network connectivity. In addition, the ability to insource remote functionality can facilitate the execution of other software evolution and maintenance tasks that we plan to investigate.

We have presented an automated approach that makes it possible to revisit the architectural decisions made when introducing distributed execution to improve performance. Oftentimes these decisions are based on assumptions that fail to hold in realistic deployments. However, the current state of the art lacks systematic treatments both to identify the inefficiencies of distributed architectures and the automated tools that eliminate much of the tedious and error-prone manual effort required to redistribute such applications. As distributed execution has become required for the majority of computing applications, our approach can help improve the utility derived by distribution and can enjoy wide applicability in helping solve realistic performance problems.

Acknowledgment

The authors would like to thank Zheng "Jason" Song and the anonymous reviewers, whose insightful comments helped improve the technical content of this paper. This research is supported by the NSF through the grants # 1650540 and 1717065.

References

[1] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and detection of SOA antipatterns," in *International Conference on Service-Oriented Computing*. Springer, 2012, pp. 1–16.

[2] E. Tilevich and Y. Kwon, "Cloud-based execution to improve mobile application energy efficiency," *Computer*, vol. 47, no. 1, pp. 75–77, Jan 2014.

[3] L. Hatton, "Reexamining the fault density component size connection," *IEEE software*, vol. 14, no. 2, pp. 89–97, 1997.

[4] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on software engineering*, vol. 25, no. 5, pp. 675–689, 1999.

[5] J. Siegel and D. Frantz, *CORBA 3 fundamentals and programming*. John Wiley & Sons New York, NY, USA:, 2000, vol. 2.

[6] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook, "Remote batch invocation for compositional object services," in *ECOOP 2009 – Object-Oriented Programming*, 2009, pp. 595–617.

[8] B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, 2005.

[9] C. Sung, M. Kusano, N. Sinha, and C. 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*, ser. FSE 2016, 2016, pp. 447–459.

[10] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic symbolic testing of JavaScript web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 449–459.

[11] realty rest, https://github.com/ccoenraets/ionic2-realty-rest, 2018.

[12] ExtremeJS, https://github.com/wangxd18/extremejs, 2018.

[13] Bookworm, https://github.com/davidwoodsandersen/Bookworm, 2019.

[14] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010, pp. 105–114.

[15] F. Brooks Jr, "No silver bullet essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[16] M. Selakovic and M. Pradel, "Performance issues and optimizations in JavaScript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 61–72.

[17] L. Gong, M. Pradel, and K. Sen, "JITProf: Pinpointing JIT-unfriendly JavaScript code," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 357–368.

[18] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically checking bad coding practices in JavaScript," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, pp. 94–105.

[19] X. Xiao, S. Han, C. Zhang, and D. Zhang, "Uncovering JavaScript performance code smells relevant to type mutations," in *Asian Symposium on Programming Languages and Systems*. Springer, 2015, pp. 335–355.

[20] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of Android applications," in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 121–130.

[21] Y.-W. Kwon and E. 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)*. IEEE, 2014, pp. 106–115.

[22] N. Nikzad, O. Chipara, and W. G. Griswold, "APE: an annotation language and middleware for energy-efficient mobile application development," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 515–526.

[23] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 95–105.

[24] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, pp. 257–269.

[25] Y.-W. Kwon and E. Tilevich, "Power-efficient and fault-tolerant distributed mobile execution," ser. ICDCS '13. IEEE, 2013.

[26] X. Liu, M. Yu, Y. Ma, G. Huang, H. Mei, and Y. Liu, "i-Jacob: An internetware-oriented approach to optimizing computation-intensive mobile web browsing," *ACM Trans. Internet Technol.*, vol. 18, no. 2, pp. 14:1–14:23, Mar. 2018.

[27] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang, "Refactoring Android Java code for on-demand computation offloading," *SIGPLAN Not.*, vol. 47, no. 10, pp. 233–248, Oct. 2012.

[28] Y.-W. Kwon and E. Tilevich, "Cloud refactoring: Automated transitioning to cloud-based services," *Automated Software Engineering*, vol. 21, no. 3, pp. 345–372, Sep. 2014.