The Tenth International Conference on Creating, Connecting and Collaborating through Computing (C5)
Institute for Creative Technologies, University of Southern California, CA, USA
January 18-20, 2012
IEEE Computer (to appear)

Orca: A Single-language Web Framework for Collaborative Development

Lauritz Thamsen*, Anton Gulenko*,
Michael Perscheid†, Robert Krahn†, and Robert Hirschfeld†
Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany
*{firstname.lastname}@student.hpi.uni-potsdam.de
†{firstname.lastname}@hpi.uni-potsdam.de

David A. Thomas
Bedarra Research Labs
Ontario, Canada
dave@bedarra.com

***Abstract*—**
In the last few years, the Web has been established as a platform for interactive applications. However, creating Web applications involves numerous challenges since the Web has been created to serve static content. In particular, the separation of the client- and the server-side, being only connected through the unidirectional Hypertext Transfer Protocol, forces developers to apply two programming languages including different libraries, conventions, and tools. Developers create expert knowledge by specializing on a few of all involved technologies. Consequently, the diverse knowledge of team members makes collaboration in Web development laboriously.

We present the *Orca* framework that allows developers to work collaboratively on client-server applications in a single object-oriented programming language. Based on the Smalltalk programming language, full access to existing libraries, and a bidirectional messaging abstraction, Orca provides a consistent environment that supports common idioms and patterns in client- and server-side code. It reduces expert knowledge and the number of development tools and, thus, facilitates the collaboration of Web developers.

***Keywords*-Collaborative Web Development, Orca, Smalltalk, JavaScript**

I. Introduction

From its beginnings, the Web has evolved from a system for serving static documents to a platform for deploying interactive applications. To support this evolution, numerous technologies have been developed to leverage the original architecture of the Web. As of today, servers provide services [1], clients run self-supporting systems and display interactive user interfaces [2], and polling idioms allow servers to send data spontaneously to clients [3].

However, the Web's distinction between the client- and the server-side renders Web development more complicated than necessary. For example, the communication between the two parts is often uniquely implemented for a specific application, since there is no broadly established way to expose Web application interfaces without exposing implementation details [4]. Further, developers combine numerous Web technologies such as scripting languages, frameworks, database mappers, communication protocols, and graphical markup languages. Developers need to remember syntactic and semantic differences as well as the functionality and interfaces of two standard libraries. These development practices force developers to mix functional, procedural, and declarative development styles in their application sources, which considerably reduces the readability of their implementations [5].

The duality of programming languages and the multitude of applied technologies unavoidably lead to distinct and heterogenous code bases, vocabularies, development practices, and, thereby, expert knowledge. For these reasons, development teams become inflexible and the collaboration while developing a single Web application becomes difficult.

We present the *Orca* framework that allows developers to work collaboratively on all aspects of Web applications in a single object-oriented language. Orca reduces the dominant difference between client and server by letting developers describe client parts, server functionality, and client-server communication in the language and development environment of the server. Our Smalltalk/Squeak [6] implementation automatically translates client-side parts of the Web application into readable JavaScript code. For these translated parts, it provides a client-side runtime, access to JavaScript libraries within Smalltalk, and transparent message passing between client and server. Thus, developers can express complete applications in a single programming language and teams are able to share not only a homogenous code base and tools, but also knowledge in a consistent vocabulary.

The contributions of this paper are as follows:

- A Web framework that allows to program both the client- and the server-side in a single object-oriented programming language and development environment (Section III, IV).
- A Smalltalk-to-JavaScript translator that automatically generates readable code and a client-side environment that permits access to language features and existing libraries of both languages (Section V-A).
- A bidirectional messaging abstraction that enables objects on clients and the server to communicate transparently (Section V-B).

The remainder of this paper is organized as follows. Section II illustrates the challenges of Web development in teams.

Section III introduces our Orca framework, while Section IV shows an example application. Section V describes Orca's implementation. Section VI evaluates our approach based on the example application. Section VII discusses related work, while Section VIII concludes this paper.

II. Challenges in Web Development

We describe the development of a chat application by a team of programmers to demonstrate why Web development today is more challenging than it should be. The chat application allows users to type in their names and to send notes to other participants. These interactions trigger dynamic updates of the browser's user interface.

The development team confronted with the task of implementing this application has to apply various technologies. They express the user interface using HTML, CSS, and JavaScript and implement the server functionality of chat message distribution in an additional language.

Although this server-side language can be chosen freely, the team does not use JavaScript for this task, because JavaScript has some problematic features [7]. For example, there are implicit global variable definitions, equality relationships that are not transitive, automatic type conversion for values that are neither equal nor identical, and four distinct function invocation patterns. If developers write JavaScript by hand, they can accidentally inject security threats [8]. Nevertheless, the team has to describe the client-side in JavaScript to allow the chat application to run in any browser. Further, the team directly applies the unidirectional Hypertext Transfer Protocol (HTTP) to transfer notes to the server.

Spreading implementations across these technologies renders the development of the example complicated and laborious. Developing Web applications using two different programming languages has several drawbacks as there are differences in syntax, semantics, object models, means of modularization, and standard libraries. The developers have to deal with this duality of languages either individually or by splitting up into client- and server-side teams. When they decide not to split up according to technologies, each developer has to constantly switch between languages. Splitting up, however, inevitably leads to expert knowledge as each programmer works with a subset of all involved technologies—the team becomes more dependent on individual developers.

Client- and server-side developers have to negotiate interfaces and data flows for most features. For example, adding author names to chat notes requires changes to the interface on the client-side, changes to the data model on the server-side and adaptions to the communication between both. Even with object interfaces, serialization formats, and server routes in place, both need to agree on the structure of transmitted content. Further, the developers mirror parts of the implementation across languages as the idea of notes is necessary both on the client and the server. This duplication violates principles of good software design [5].

The described development process is characteristic for Web development in general. First, the chosen application demonstrates all aspects of a typical Web application as it is interactive and collaborative [9]. Second, although teams apply popular Web frameworks as Ruby on Rails [10] and Django [11] that claim to alleviate Web development, these frameworks do not reduce the number of necessary technologies. Client-side developers are still using JavaScript, while the server-side language is called from within HTML templates, forcing client and server developers to mix up their implementations.

Even though development of Web applications is special in that it is tied to the Web's technology stack with JavaScript, HTML, and CSS on the client-side and HTTP for client-server communication, these technologies can be abstracted. A unified and object-oriented solution for Web development can render the direct application of the unidirectional communication protocol, graphical markups, and two programming languages unnecessary while providing full access to libraries on client and server.

III. The Orca Framework

The Orca Web framework strives to reduce the difficulties of conventional Web development. It allows the development of all aspects of Web applications in a single language to increase the consistency of implementations. Although using multiple languages can alleviate programming of applications if specific languages are chosen for their eligibility as, for example, Domain Specific Languages (DSLs), the languages of the Web are either general purpose as JavaScript or no longer applied as originally intended as HTML and CSS [12]. Therefore, Orca solely relies on the Squeak/Smalltalk language. Using Smalltalk avoids the aforementiond ambigious features of JavaScript.

Development in a single language should not restrict developers in expressing client-side programs. This is important since executing logic on client-side can increase interface responsiveness and reduce network reliance [13]. Therefore, Orca does not apply markup generation or wrappers for JavaScript libraries, but allows the expression of arbitrary client-side code in Smalltalk.

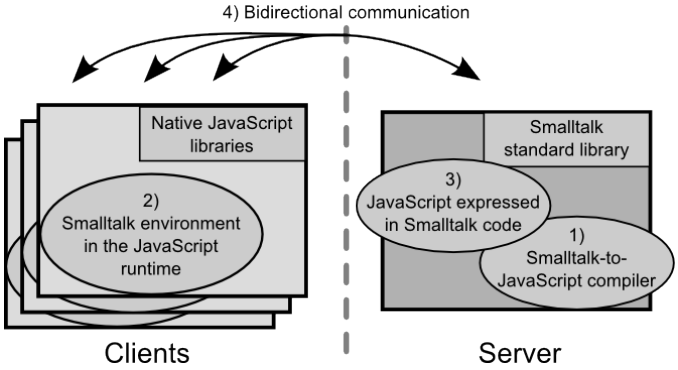

Figure 1. The main components of Orca as single-language Web development solution.

Four main components, as depicted in Figure 1, constitute the Orca Web framework and enable its single-language Web

development approach.

1) A compiler or interpreter that translates the client-side part to readable JavaScript code. This translator reflects semantic differences like variable scope, default values for initialization and returns, and provides access to literals and language features available in JavaScript. The generated JavaScript code maintains the chosen language's semantics as developers do not write JavaScript. A server-side translator allows caching of generated code for numerous clients.
2) A JavaScript runtime environment that emulates the language features that have no direct equivalent in JavaScript. This runtime provides the object model, access to available objects, and emulations of primitives or operators.
3) Mechanisms to express JavaScript features that have no equivalent in the server-side language. For these mechanisms, the compiler has to implement rules, while the client-side runtime implements semantics.
4) Abstractions for the communication between remote application parts. For a complete single-language programming experience, it enables bidirectional communication and remote invocations.

In Orca, developers apply the same syntax, semantics, object model, and standard library to express both client and server, while utilizing existing JavaScript functionality. Instead of writing HTML and CSS, they apply JavaScript widget libraries. Rather than applying HTTP directly to request data from server routes, remote invocations are similar to the language's invocation mechanisms, including addressing the language's entities instead of URLs.

The Orca framework further incorporates deployment facilities as, for example, a Web server to allow developers to concentrate on programming instead on configuring a server's dispatch, request handling, and caching.

IV. Orca By Example

We present an example application to give an intuition of developing Web applications with Orca.

A simple chat application, as shown in Figure 2, allows receiving and sending chat notes. The server-side of the chat's implementation distributes such notes among participants, while the browser displays them to users. Instead of describing the complete implementation, we will focus on parts of the interface and the client-server communication.

First, we create a new Orca application, set the application's name, define necessary JavaScript libraries, and declare which classes are required on the client-side in addition to certain classes of Smalltalk's standard library and the application class itself (Listing 1). When an application class has been created and configured, it is immediately available on the application server.

The `ChatWindow` is one of three classes that constitute the implementation of this chat example as shown in Figure 3. Instances of the `ChatWindow` class represent an arbitrary number of clients and display notes, while a single `ChatHub` instance distributes notes among all chat participants. Both the `ChatWindow` instances and the `ChatHub` singleton use `ChatNote` objects. That is, the model for notes is necessary on both the client- and the server-side.

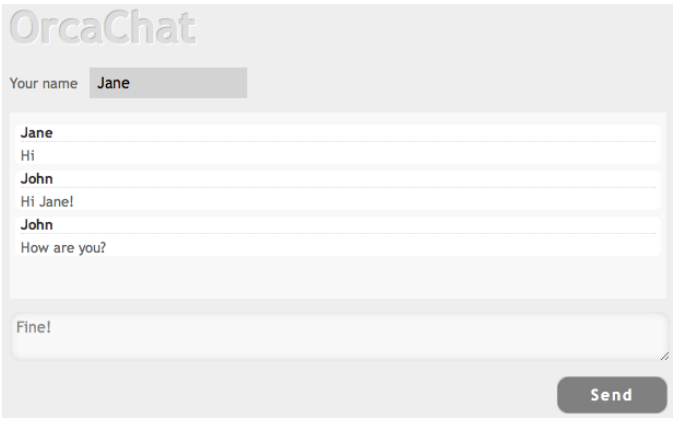

Figure 2. The chat participants share notes through a central server.

```
ChatWindow class>>#requiredClasses

    ↑ { ChatNote. }
```

Listing 1. The client-side requires the classes for chat windows and notes.

We use the application's initialization to setup the desired user interface including the chat's button (Listing 2).

The button gets a label and a callback. Orca comes with predefined user interface elements, although arbitrary HTML tags can be used. Orca's class for interface elements is called `OrcaWidget` and its initialization shows how to create an arbitrary HTML element (Listing 3).

The `Js` class provides access to global JavaScript objects on the client-side and in this case to the `Document` object.

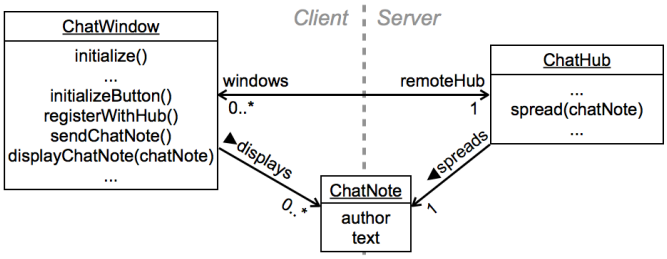

Figure 3. The chat's implementation includes objects for the client's window, the server's message distribution and for chat notes.

```
ChatWindow>>#initializeButton

    self sendButton:
        (OrcaSubmitButton new
            text: 'Submit';
            onClickDo: [ self sendChatNote ])
```

Listing 2. The initialization of the chat window creates a button.

```
OrcaWidget>>#initialize

    self
        node: (Js Document createElement
            value: self class htmlTag);
        children: OrderedCollection new.
```

Listing 3. The Orca widget initialization creates document nodes.

```
ChatWindow>>#sendChatNote
    | chatNote|
    chatNote := ChatNote
        text: messageInput text
        author: nameInput text.
    self remoteHub spread: chatNote.
```

Listing 4. The client sends a chat note to the server.

This object provides the `createElement()` function that creates a new HTML element with a supplied HTML tag. Since an object's function can be either retrieved or evaluated in JavaScript, we use the message `value:` to explicitly evaluate retrieved functions. We have chosen this message selector since JavaScript functions and Smalltalk blocks are conceptually similar and `value:` is used to evaluate Smalltalk blocks.

Revisiting our button's initialization, the `onClickDo:` message receives a Smalltalk block as callback and sets it as an element's property. The callback is executed on every click on the button and invokes the `sendChatNote` method (Listing 4).

To understand how the note is transferred to the server and to all clients, we need to know what the `remoteHub` object is. On initialization, the client-side of the chat registers with the server-side (Listing 5). The `asRemote` message creates a proxy object for the `ChatHub` class of the server-side. This client-side proxy object forwards the `uniqueInstance` message to the class on the server-side. The invoked method returns a proxy for the singleton to the client. The method further sends the client's chat window to the chat hub object. This way, the hub receives a proxy for this client-side object and can send messages to it as in the `spread:` method (Listing 6) which executes whenever the button's `sendChatNote` is clicked.

Since clients may be no longer available, the server might receive an exception on the attempt to send a note to a client. In this example, we use this exception to inform all participants that the unreachable client left the chat.

When a note reaches a client, the client creates a new

```
ChatWindow>>#registerWithHub

    self remoteHub:
        ChatHub asRemote uniqueInstance.
    self remoteHub registerWindow: self.
```

Listing 5. The client-side initialization connects client and server.

```
ChatHub>>#spread: aChatNote
    self registeredChatWindows do:
        [ :each |
            [ each
                performForked:
                    #displayChatNote:
                with: aChatNote ]
                    on: ClientTimedOut
                    do: [ :ex | self
                        informParticipantsAbout:
                         ex timedOutClient ]].
```

Listing 6. The server spreads chat notes among clients.

```
ChatWindow>>displayChatNote: aChatNote

    | noteWidget |
    noteWidget := ChatNote for: aChatNote.
    self add: noteWidget.
```

Listing 7. The chat window displays received notes.

element for the message and adds it to the display (Listing 7).

The presented implementation of a chat application with Orca demonstrates our single-language Web development approach. The complete application—including the client-side and client-server communication—is expressed solely in Smalltalk. This is possible through a Smalltalk-to-JavaScript translator, a custom client-side environment, and a bidirectional remote messaging abstraction.

V. Implementation

Orca is a single-language Web framework implemented in Smalltalk/Squeak.

This section shows how Orca's compiler translates Smalltalk code to JavaScript, how its custom client-side environment executes the translated code, and how it makes all JavaScript features available in Smalltalk. The section further describes how Orca enables client-server communication through Smalltalk messages.

A. Expressing Client-side Functionality in Smalltalk

1) Translating Smalltalk to JavaScript: Orca's Smalltalk-to-JavaScript translator generates the client-side of an Orca application. Besides syntactic translation, it takes semantic differences between both languages into account.

Variable declarations are translated directly, but each variable is initialized to the default value `nil` instead of `undefined`. If there is no explicit return from a method, a return statement is added that returns the receiver, consistent with Smalltalk semantics.

Orca's compiler translates message sends to function calls. A mapping from Smalltalk message selectors to JavaScript function names avoids collisions of Smalltalk messages and instance variables because there is no different scope for functions and properties and no overloading in JavaScript.

The compiler translates Smalltalk literals to calls to global functions with the primitive JavaScript equivalent as argument. These functions produce Smalltalk-typed objects for the JavaScript values without adapting objects of JavaScript's standard library.

The compiler also takes care of Smalltalk cascades. As there is no similar structure in JavaScript, it compiles them into equivalent statements.

JavaScript code translated with Orca does not apply JavaScript control structures directly. Each message send is translated to a function call; including Smalltalk messages as `ifTrue:ifFalse:`. The compiler translates necessary parts of Smalltalk's standard library to allow calling these functions. This way, Orca allows translation of arbitrary Smalltalk without any type analysis. It translates the code directly and produces a JavaScript version with the exact call structure.

The compiler applies JavaScript code conventions and preserves comments to alleviate debugging the client-side with the browser's tools.

In the future, we would like to use type analysis to enable the compiler to use JavaScript control structures for performance and readability reasons.

2) Running Smalltalk in JavaScript: The translation of our compiler depends on features of Smalltalk that are not present in JavaScript. Therefore, Orca provides a Smalltalk-like environment for the JavaScript runtime.

Although Smalltalk and JavaScript are both dynamically typed and object-oriented programming languages, they are based on two distinct object models. JavaScript objects are prototypes, whereas Smalltalk objects are instances of classes. Orca applies JavaScript's prototypical inheritance to emulate Smalltalk's class system on the client.

After the initialization of the Smalltalk standard library, the runtime creates objects that are part of every Smalltalk runtime. For example, it creates globally accessible singletons like `nil, true, false`.

Besides the object model and literally accessible objects, some programming concepts of Smalltalk have to be emulated in JavaScript. For example Smalltalk's *doesNotUnderstand* mechanism or non-local method returns of blocks are required to maintain the semantics of translated code. The implementation of Smalltalk's *doesNotUnderstand* concept relies on the compiler's collection of used message names. These message names are collected to create a new root class of the class hierarchy that provides default implementations for all actually sent messages. For Smalltalk's non-local method returns, Orca uses JavaScript exceptions to return the block's home context.

Orca emulates required primitives of Squeak's virtual machines that have no software implementation.

3) Expressing JavaScript in Smalltalk: For complete single-language development, developers need to be able to express any client-side programs in Smalltalk.

We achieve that by mapping JavaScript operators and values to equivalent Smalltalk messages and objects. Further, functions are mapped to the native interface of Smalltalk blocks.

Other aspects of the JavaScript language require an explicit representation in the Smalltalk environment. For example, Orca allows assignment and testing of object slots by using explicit JavaScript-only objects. Messages that conform to Smalltalk's conventions for accessing instance variables are used as getters and setters of object slots. Other explicitly implemented JavaScript features include creating plain JavaScript objects through the `new` operator or literals.

These mechanisms enable developers to take advantage of existing JavaScript functionality instead of writing and maintaining wrappers for libraries. For example, the browser interface, which is used to build and manipulate the structure of the application's Web interface, is accessible.

Objects on the client-side have two representations to achieve interoperability with library code. There is one representation for Orca's Smalltalk environment in JavaScript and one for the original environment. Upon entering Orca's Smalltalk environment within JavaScript, JavaScript objects are automatically boxed to resemble Smalltalk equivalents of their JavaScript equivalents.

B. Expressing Client-server Communication in Smalltalk

With Orca, objects on client and server interact through messages. Orca implements transparent message passing as in Distributed Smalltalk [14] to allow code to invoke local and remote methods.

When messages address remote objects, Orca forwards them transparently. The framework serializes objects and creates requests. However, remote messages still require local receivers. Orca provides proxies for objects that are not part of the local address space. Client-side proxies hold an identifier for an actual object of the server, whereas server-side proxies additionally contain a reference to a certain client. *Remote Object Maps* resolve these identifiers on client and server. Remote messages to such proxy objects transfer arguments and return values. Orca's remote messaging transfers numbers, strings, characters, booleans, and the pseudo-variable `nil` by value. The default option for instances of other classes is to transfer a reference instead of an actual object. That is, a proxy is generated transparently on such messages and is supplied as parameter. Orca further lets developers specify that objects should be passed by values despite being none of the mentioned types. Instances of classes that return true on `copyOnSend` messages are passed by value and will, therefore, be locally available in the receiver's environment. Since remote messages are orders of magnitude more expensive than local messages, developers might apply this copy mechanism as optimization. When an object is passed by value, the object's class has to be available in the target environment.

Like local sends, remote sends are synchronous by default. However, Orca also provides message sends that do not wait for remote answers, but directly receive Smalltalk's `nil` value. These *one-way sends* are available through the `performForked:withArgs:` message that expects a message selector and arguments.

Further, clients or the server might no longer be available, but still be referenced. In this case, the sender receives an exception after a timeout.

Orca's message passing mechanism is built on top of a bidirectional abstraction layer that applies long polling. In addition to allowing server-side sends at anytime our abstraction layer handles allocation of answers. This way, Orca alleviates the implementation of control flows that request further information on requests without responding to the first request immediately.

In the future, we will address security concerns as clients can currently send any messages to all global Smalltalk objects.

VI. Evaluation

This section evaluates whether Orca alleviates Web development by discussing its impact on the development process of the chat application. Further, we explain certain compiler optimizations that reduce the execution time of Orca's generated client-side code significantly.

A. Development of Orca Applications

With Orca, teams design and implement the application in a single development and runtime environment. The developers use a more consistent tool chain, including a single code browser, search facilities, test runner, and refactoring tools. A single source code management system contains the team's code. There is considerably less distinction of client and server programming—both apply the same language, standard library, and the same conventions. Section II discusses that this is not the case with traditional Web development. The team has to cope with multiple sources and differences between both applied programming languages.

Development solely in Smalltalk increases readability and understandability of the system as the listings in Section IV illustrate. The whole team is able and encouraged to understand all parts of the implementation.

Orca's Smalltalk-to-JavaScript compiler translates arbitrary Smalltalk code. It produces valid code that relies on our emulation of the Smalltalk environment to execute on the client. Our research team and five experienced JavaScript and Smalltalk developers evaluated the readability of the resulting JavaScript code. It is formatted according to JavaScript coding conventions, includes the original comments, and keeps the structure of the Smalltalk code. The generated code can be debugged in its actual JavaScript runtime environment and insights can be easily applied to the Smalltalk source code.

Orca reduces the amount of source code and prevents duplications between client and server. The chat implementation showed that developers do not need to write or call any serialization routines and that no code has to be mirrored across languages. They do not create HTTP requests on the client-side, dispatch them on the server-side, and specify responses. Developers express the communication on the level of abstraction of Smalltalk. That is, already existing object interfaces are used instead of ad-hoc interfaces defined by name-value pairs, serialization formats and server routes.

For larger and more complex applications, we think that development might be impeded by the implicit nature of proxy-based message forwarding. Orca, therefore, exposes its bidirectional communication layer to application developers and comes with simpler messaging abstractions as, for example, remote block evaluation.

To summarize, the development of Web applications with Orca avoids many of the problems of traditional Web development. The development team can focus on writing homogenous application code in a single development environment. There is no context switching between technologies and, thus, likely less expert knowledge.

B. Execution of Orca Applications

Server-side logic, client-server communication, and client-side functionality constitute Orca applications. The server-side of Orca applications executes as fast as regular Smalltalk, while remote *one-way sends* can transfer parameters as value and, thereby, in an equal number of HTTP requests as used manually. However, the generated client-side of Orca applications is expected to be slower than manual implementations.

To measure the performance of the generated code, we implemented the `ackermann(3, 4)` [15] function with Orca and compared its execution time to that of native JavaScript. We chose this recursive algorithm, because its execution includes arithmetical operations, conditional control structures, and numerous method calls. The experiment was conducted with an Intel Core i7 processor, 8 GB of main memory, the operating system Windows 7 and the Google Chrome browser version 15.0.874.121. We used JavaScript's `Date().getTime()` function and averaged 50 runs.

The plain JavaScript version took 0.4 milliseconds, while the Orca version was finished within 207 milliseconds.

Orca's generated version runs orders of magnitude slower than the JavaScript implementation. The main performance loss results from the compiler's preservation of the characteristics of the original Smalltalk code. That is, message sends express arithmetic operations and control flow. Further, the implementation of our custom environment wraps each message send into two functions. For these reasons, the addition of two numbers, for example, results in a callstack of at least eight invokations in addition to the actual + statement, while an `ifTrue:` statement requires nine calls and, further, creates a short-living BlockContext object.

A number of optimizations can improve Orca's compiler. The first improvement should be to remove one of the functions wrapped around each method call. Experiments showed a performance gain of 15%.

Further, performance can be increased through using JavaScript operators for arithmetic operations and control flow. We could apply similar techniques as used by optimizing compilers of polymorphic code [16]. The compiler would detect variables that are likely of a certain type through

static analysis of message sends. For such variables, it could generate type checks as well as JavaScript control structure and operators. On successful type checks, the JavaScript engine would execute operators directly, while the original polymorphic implementation would be executed otherwise. The overhead created by the type checks is considerably low compared to the performance gain we encountered.

We modified the generated code for the `ackermann(3, 4)` function to measure the proposed optimizations. All optimizations resulted in code that ran only five times slower than our manual JavaScript implementation of the example.

An issue with generating optimized code is reduced readability. Such compilation should, therefore, only be feasible for production versions that developers will less likely debug. Another problem might pose the increased code size, but as Orca transports code compressed, loading a web page should still be sufficiently fast.

In conclusion, we demonstrated that the measured low performance is not inherent to our approach. In the future, we will reduce the performance penalties through the proposed optimizations and will experiment with different type analysis tools such as TypeHarvester [17].

VII. Related Work

1) Server-centric Web Frameworks: Numerous Web frameworks support the development of database-driven, server-centric Web applications [18]. Such frameworks as, for example, Ruby on Rails, Django, and Seaside [19] run on the server-side and generate HTML for the browser. Ruby on Rails and Django both rely on HTML templates and inline calls to server-side functionality. That is, such template-based frameworks continue to build user interfaces with HTML pages and mix different programming paradigms and languages in their source files, which reduces readability and maintainability. In contrast, Seaside provides an embedded DSL for HTML components and, therefore, can be considered as a single-language approach to server-centric Web application development. All three examples have in common that developers express client-side logic either directly in JavaScript or through JavaScript wrappers. In contrast, Orca allows expression of client-side functionality in the server-side language including facilitating JavaScript widget libraries.

2) Single-language Web Frameworks: GWT is a framework for single-language Web development in Java based on a Java-to-JavaScript compiler, emulation of the Java Runtime, and a widget library. Similar components resemble the Python-based single-language Web framework Pyjamas [20]. Both generate browser-specific client-side code and, thereby, relieve developers from ensuring interoperability across multiple browsers. GWT and Pyjamas allow embedding JavaScript code directly into the host language. Pyjamas code can inline JavaScript code, while GWT's JavaScript Native Interface (JSNI) only allows implementation of whole JavaScript methods due to Java's static type system. However, JavaScript calls to existing libraries still need to be wrapped for complete single-language development. In contrast, Orca enables developers to express JavaScript within Smalltalk. Both GWT and Pyjamas do not incorporate bidirectional remote invocations as Orca does. In comparison of all three approaches, GWT provides the most complete tool support, while Orca provides the most integrated single-language development approach.

3) HOP: The HOP programming language [21] is a Scheme-dialect for developing multimedia Web applications. It is based on two execution strata; one executes JavaScript on the client-side, while the other executes Scheme on the server-side. HOP provides a Scheme-to-JavaScript compiler [22], syntax to escape inline between both strata, and bidirectional communication through an event loop and remote function invocation. In contrast to Orca's translator, HOP's compiler is optimized for performance. We experienced HOP's generated JavaScript to be unreadable and, therefore, difficult to debug. While HOP is build on the functional programming language Scheme, Orca is based on Smalltalk's object-oriented and class-based development style.

4) Amber: Amber [23] is a self-contained Smalltalk implementation in JavaScript. It includes a Smalltalk-to-JavaScript compiler, inline calls to JavaScript functionality, and syntax to use JavaScript objects from within Smalltalk. For performance reasons, Amber tries to use native JavaScript values where possible, making it necessary to implement considerable parts of the standard library in the compiler. In contrast to Orca, Amber is a pure client-side environment. It can not directly be used to implement Web applications that require server functionality and client-server interactions.

5) Lively Kernel: The Lively Kernel [24] is an open and self-supporting JavaScript environment that runs completely in the browser. The Morphic user interface environment [25] and the absence of separation between design-time and run-time allow rapid application development through direct composition and immediate feedback, while its wiki-like deployment [26] supports collaborative development efforts. In contrast to Orca, Lively Kernel development happens in JavaScript. Furthermore, although the Lively Kernel is currently deployed with server-side JavaScript through Node.js [27], the client-side and the server-side are less integrated than their counterparts in Orca.

6) Dart: The Dart Programming language [28] was developed as language for usage on both clients and servers. Dart aims on better support for modularity and developer collaboration by providing classes, interfaces, optional types, libraries, and tools. To our most recent knowledge, a Dart program is compiled to JavaScript and then executed in a Web browser. As a general purpose language, Dart does not include support for client-server communication. Compared to Orca, the Dart language aims on unifying client- and server programming, while Orca is a framework that further incorporates client-server communication, Web server infrastructure, and integration with existing JavaScript libraries.

7) Native Client: Native Client [29] is a sandbox for multi-threaded execution of untrusted x86 native code in browsers. The browser extension runs such code in its own address space, but provides interfaces for side effects to the JavaScript

environment. Since Native Client is not supported by all major browsers, Native Client programs do not run as ubiquitously as JavaScript and, therefore, Orca programs. Further, Native Client is not a Web development framework but a code execution sandbox.

VIII. Conclusion

With Orca, development teams can work collaboratively on all aspects of Web applications. Our framework enables development of the client and server in the Smalltalk programming language, provides access to existing functionality of both languages, and alleviates client-server communication. Orca, thereby, unifies client- and server-resident parts and interaction between them.

Developers neither program in two distinct environments nor specialize on certain parts of Web applications. Instead of only gaining expert knowledge on the subset of an application, the developer and their team share a homogenous, single-language code base, a consistent vocabulary of idioms and patterns, and development tools for implementing, testing and refactoring.

In the future, we want Orca's compiler to be able to generate the client-side in two different versions with one of which is optimized for readability while the other aims on performance through direct application of JavaScript operators and control structures. Furthermore, we will restrict remote messaging to certain global Smalltalk objects to address security concerns.

Nevertheless, the Orca framework already permits the construction of complete Web applications within Smalltalk and, thereby, enables development teams to work with a single development solution.

IX. Acknowledgments

We would like to thank Hauke Klement, Lars Wassermann, Robert Strobl, Sebastian Woinar, and Stephan Eckardt for their valuable contributions to Orca. We would also like to express thanks to Fabian Bornhofen, Thomas Bünger, and Eugenia Gabrielova for their comments on drafts of this paper.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*, 1st ed. Springer, 2010.

[2] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, "Web Browser as an Application Platform," in *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications*, ser. SEAA '08. IEEE Computer Society, September 2008, pp. 293–302.

[3] D. Crane and P. McCarthy, *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*, 1st ed. Apress, 2008.

[4] T. Mikkonen and A. Taivalsaari, "The Mashware Challenge: Bridging the Gap between Web Development and Software Engineering," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER '10. ACM, November 2010, pp. 245–250.

[5] ——, "Web Applications - Spaghetti Code for the 21st Century," in *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, ser. SERA '08. IEEE Computer Society, August 2008, pp. 319–328.

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '97. ACM, October 1997, pp. 318–326.

[7] D. Crockford, *JavaScript: The Good Parts*, 1st ed. O'Reilly Media, 2008.

[8] C. Yue and H. Wang, "Characterizing Insecure JavaScript Practices on the Web," in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW '09. ACM, April 2009, pp. 961–970.

[9] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari, "Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. ACM, March 2011, pp. 800–807.

[10] S. Ruby, D. Thomas, and D. Hansson, *Agile Web Development with Rails*, 3rd ed. Pragmatic Bookshelf, 2009.

[11] A. Holovaty and J. Kaplan-Moss, *The Definitive Guide to Django: Web Development Done Right*, 2nd ed. Apress, 2009.

[12] M. Jazayeri, "Some Trends in Web Application Development," in *Proceedings of the 2007 Future of Software Engineering*, ser. FOSE '07. IEEE Computer Society, May 2007, pp. 199–213.

[13] J. Kuuskeri and T. Mikkonen, "Partitioning Web Applications Between the Server and the Client," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. ACM, March 2009, pp. 647–652.

[14] J. K. Bennett, "The Design and Implementation of Distributed Smalltalk," in *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications*, ser. OOPSLA '87. ACM, January 1987, pp. 318–330.

[15] W. Ackermann, "Zum Hilbertschen Aufbau der reellen Zahlen," *Mathematische Annalen*, vol. 99, pp. 118–133, 1928.

[16] D. Ungar, R. B. Smith, C. Chambers, and U. Hölzle, "Object, Message, and Performance: How they Coexist in Self," *Computer*, vol. 25, pp. 53–64, October 1992.

[17] M. Haupt, M. Perscheid, and R. Hirschfeld, "Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages," in *Proceedings of the 25th Symposium on Applied Computing*, ser. SAC '11. ACM, March 2011, pp. 1282–1289.

[18] I. Vosloo and D. G. Kourie, "Server-centric Web frameworks: An Overview," *ACM Computing Surveys*, vol. 40, pp. 4:1–4:33, May 2008.

[19] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside: A Flexible Environment for Building Dynamic Web Applications," *IEEE Software*, vol. 24, no. 5, pp. 56–63, September 2007.

[20] Leighton, Luke K. C., "Pyjamas Book," http://pyjs.org/book/output/Bookreader.html, 2009, retrieved December 12th 2011.

[21] M. Serrano, "Programming Web Multimedia Applications with Hop," in *Proceedings of the 15th International Conference on Multimedia*, ser. MULTIMEDIA '07. ACM, September 2007, pp. 1001–1004.

[22] F. Loitsch and M. Serrano, "Hop Client-Side Compilation," in *Draft Proceedings of the 8th Symposium on Trends in Functional Languages*, ser. TFP '08, 2008, pp. 141–158.

[23] N. Petton, "Amber Documentation," http://amber-lang.net/documentation.html, 2011, retrieved December 16th 2011.

[24] D. Ingalls, "The Lively Kernel: Just for Fun, Let's Take JavaScript Seriously," in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS '08. ACM, July 2008, pp. 9:1–9:1.

[25] J. H. Maloney and R. B. Smith, "Directness and Liveness in the Morphic User Interface Construction Environment," in *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, ser. UIST '95. ACM, December 1995, pp. 21–28.

[26] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz, " Lively Wiki - A Development Environment for Creating and Sharing Active Web Content," in *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*, ser. WikiSym '09. ACM, October 2009, pp. 9:1–9:10.

[27] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *IEEE Internet Computing*, vol. 14, pp. 80–83, November 2010.

[28] The Dart Team, "Dart Programming Language Specification," http://www.dartlang.org/docs/spec/dartLangSpec.pdf, 2011, retrieved December 16th 2011 (Draft Version 0.06).

[29] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, ser. SP '09. IEEE Computer Society, May 2009, pp. 79–93.