Programming Language Design for Service-Oriented Systems

by

Salman Azeem Ahmad

M.S., Stanford University (2011)
B.S.E., Arizona State University (2008)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Author .....
Department of Electrical Engineering and Computer Science
August 21, 2014

Certified by .....
Sepandar D. Kamvar
Associate Professor
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chair, EECS Committee on Graduate Students

Programming Language Design for Service-Oriented Systems

by

Salman Azeem Ahmad

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Designing systems in a service-oriented manner, in which application features are decoupled and run as independently executing services over a network, is becoming more commonplace and popular. Service-oriented programming provides a natural way to model and manage many types of systems and allows software development teams to achieve operational flexibility, scalability, and reliability in a cost-effective manner. In particular, it has been used quite successfully for Web and mobile applications. However, building, deploying, and maintaining service-oriented systems is challenging and requires extensive planning, more effort during development, a detailed understanding of advanced networking techniques, and the use of complicated concurrent programming.

This thesis presents a new programming language called Silo. Silo integrates features that address key conceptual and pragmatic needs of service-oriented systems that, holistically, are not easily satisfied by existing languages. Broadly, these needs include: a unified distributed programming model, a simple yet efficient construct for concurrency, a familiar yet extensible syntax, and the ability to interoperate with a rich ecosystem of libraries and tools.

In this dissertation, I describe how Silo's features, constructs, and conventions satisfy these needs. Then, I present various compiler and runtime techniques used in Silo's implementation. Lastly, I provide a demonstration, through a variety of programming patterns and applications, of how Silo facilitates the design, implementation, and management of service-oriented systems.

Thesis Supervisor: Sepandar D. Kamvar
Title: Associate Professor

Dedicated to my parents,

Dr. Seema Munir and Dr. Jalil Ahmad

Acknowledgments

I am incredibly grateful to have had the support of so many brilliant and kindhearted individuals over the years. This work would have been *impossible* without them.

- My advisor, Sep Kamvar, for being a constant source of encouragement, inspiration, and wisdom. His sincere generosity and genuine care for others is overwhelming. As a professor, he is an unwavering advocate for all of his students and works tirelessly to support their best interests. I greatly value my relationship with Sep. He is a true role model and a great friend.

- My committee, Armando Solar-Lezama and Daniel Jackson, for their invaluable feedback and generous advice. Their brilliance was outmatched only by their kindness.

- My mentors, Scott Klemmer, Jeffrey Heer, Rob Miller, David Karger, and Randall Davis, for their unwavering support and insightful guidance.

- My friends, colleagues, and collaborators, Peter Woods, Laurel Woods, Ranjitha Kumar, and Jerry Talton, for constantly raising my spirits. The Social Computing group, Yonatan Cohen, Edward Faulkner, Wesam Manasra, Pranav Ramkrishnan, Kim Smith, and Jia Zhang, for constantly making me smile. The administrative staff at MIT, Kristina Bonikowski and Janet Fischer, for constantly saving my neck.

- My family. My loving parents, Seema Munir and Jalil Ahmad. My incredible sisters, Lubna Ahmad and Minal Ahmad. My kick-ass brother, Rizwan Ahmad. For always being there with endless love, care, compassion, and trust... and cookies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Service-Oriented Systems

The idea behind a service-oriented system is fairly straight forward – take a complex system, identity the key features and important units of functionality, and break them out as "sub-systems" or "services" that run concurrently and communicate with one another by passing messages. This approach has been proposed, used, and rediscovered under different names countless times in the past [89, 43, 87, 48].

A distinctive attribute of *modern* service-oriented systems is that they are distributed and run across many machines communicating over a network. As CPUs reach their limits in terms of sequential performance many applications are forced to leverage multiple machines to scale. Additionally, service-oriented systems align well with the trend towards cloud computing where application developers rent machines from hosting providers on a per-usage-basis [51, 52].

This thesis is particularly interested in systems that power applications deployed over the Web. These systems routinely have to handle a large number of concurrent requests, changing product requirements, low budgets, and rapid growth. Within this space, some of the largest and most successful software organizations in industry often discuss their use of service-oriented architectures and indicate that it was critical for their operational successes [23, 10, 101, 49]. The service-oriented approach for building these types systems is contrasted by multitiered systems (sometimes referred

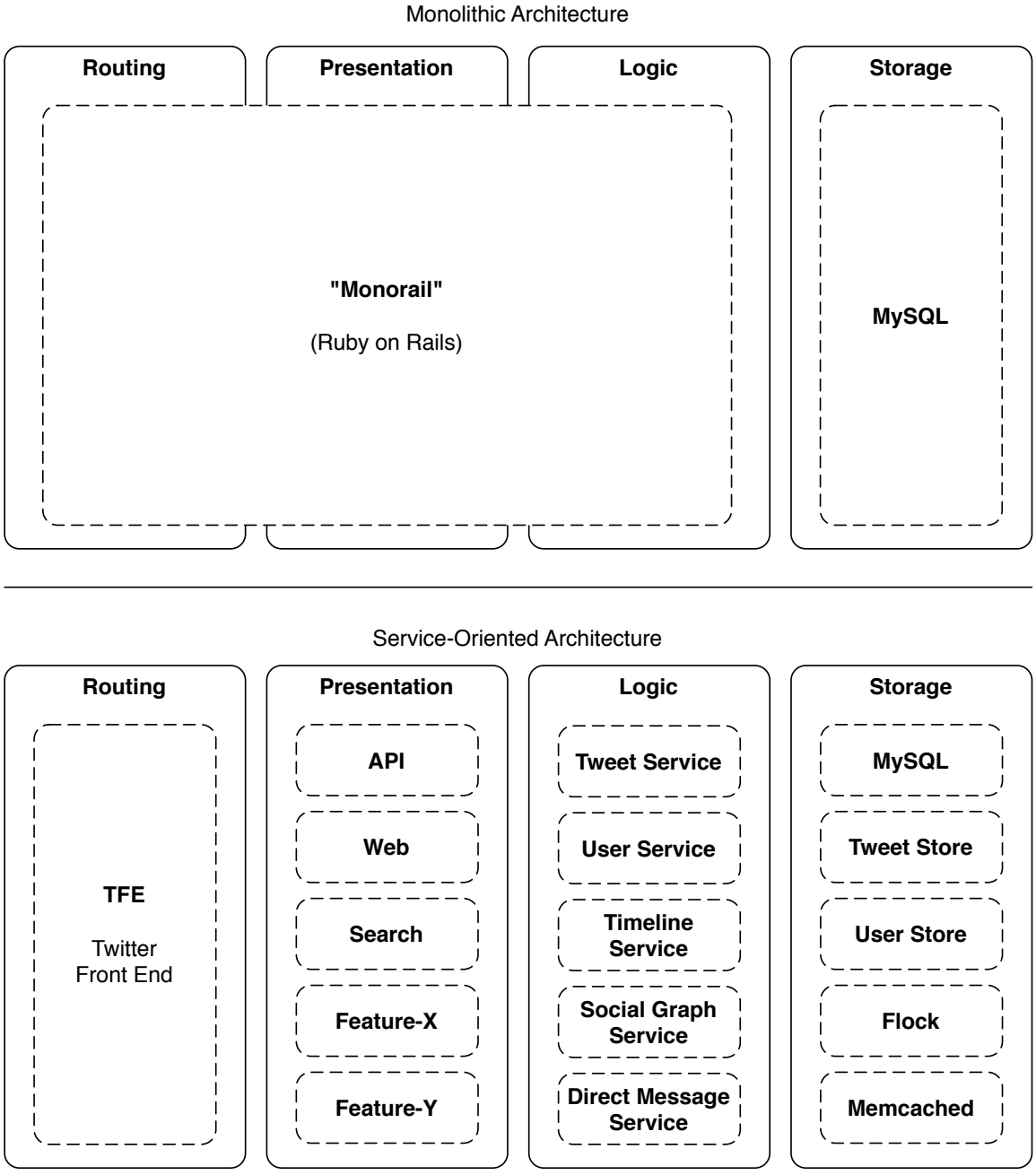

Figure 1-1: An illustration of how Twitter migrated from a monolithic architecture to service-oriented architecture. Adapted from [23].

to as monolithic systems) which organize software components into layers within a single process (as opposed to multiple processes). Figure 1-1 illustrates the differences between service-oriented architectures and multitiered architectures in practice.

1.1.1 Benefits

Building systems in a service-oriented manner confers many benefits.

- **Modularity** Many systems are naturally modeled in a service-oriented manner as they logically compose multiple independent sub-systems. For example, it is common for a modern application to have a Web site, a mobile API, an email sub-system that sends out email to users, etc. These types of systems are best implemented as multiple separate services running in parallel rather than packing all of the functionality into a single monolithic program. Also, from a management perspective, it provides a natural way for dividing software teams and assigning responsibilities.

- **Flexibility** Service-oriented systems allow development teams greater flexibility when dealing with operational challenges, coping with changing product requirements, and adopting newer technologies [10]. For example, suppose the developers of a streaming music site want to provide song recommendations to their users. Instead of having to make intrusive changes to their existing code base, the recommendation feature could be implemented as a separate service that runs on its own. As a new feature, the recommendation service will likely receive more changes and updates than the rest of the system. Making these updates is much easier in a service-oriented system (compared to a monolithic system) since only the recommendation service will need to be upgraded and the rest of the system continues with no downtime. Additionally, the developers do not need to worry about the hardware requirements or proactively provision new machines in anticipation of the feature going live. Instead, they can run the service on any existing box and make the service accessible to only

a small number of "beta-testers". As the developers gain confidence in the feature, they can move the recommendation service onto its own hardware. Lastly, the recommendation system will likely need a special purpose database that is optimized for numerical computations. Instead of feeling tempted to use their tried-and-true database of choice, developers are encouraged to explore new database solutions for the recommendation service since the service is independent of the rest of the system and thus allows the developers to be more tolerant of errors and failures.

- **Scalability** Service-oriented programming gives developers greater freedom to optimize the performance of their system and utilize computational resources more intelligently. For example, if an application's search feature takes a long time to process queries, the search service can be moved independently onto a machine with more memory and a faster CPU while the system's other services will remain running on more economical hardware. As another example, if the authentication service is a bottleneck in the system, the developers may choose to provision more machines that run the authentication service.

- **Reliability** Since services operate independently from one another, a failure or bug in one service does not crash the entire system [27]. Crashes, performance bottlenecks, and security concerns are isolated to a single service rather than spreading. For example, if the server running the search service for a shopping Web site has a power supply failure, users may not be able to search for new products but they can still log in to their account, track existing shipments, and write reviews.

1.1.2 Challenges

Despite the benefits of the service-oriented architectures, there are notable challenges.

- **Difficult to Build** Service oriented systems are notoriously hard to implement [23, 119]. Instead of having a single program, the functionality of service-

oriented systems is broken up across many programs and code bases. Additionally, developers need an in depth and advanced understanding of networking (technical details, techniques, libraries, etc.) and concurrency (thread pools, event loops, synchronization, race conditions, etc.). Moreover, developers are forced to write code in a manner that forgoes many of the comforts and conveniences of other programming paradigms. For example, invoking a service is not as straight forward as calling a method. The service call may be dropped by the network, arrive out of order, be corrupted, or be duplicated. Developers need to write robust code that handles these numerous sources of error.

- **Difficult to Manage** Service-oriented systems involve a lot of moving parts that are difficult to manage. To run a system you need to start up many different services, handle failures from multiple sources, and monitor the behavior of numerous processes. This works well when the development team is large and can divide responsibilities amongst many developers. But a small team that is growing will find it hard to remain agile.

- **Difficult to Change** The flexibility afforded to developers is balanced against ensuring appropriate technical integration between sub-systems. In other words, service-oriented systems are often rigid. Making changes that "cut across" many different services is quite challenging [23]. Additionally, once services are defined and established, it is hard to then break them up into smaller services or merge them into a new larger service. This makes service-orientation less attractive to early stage projects or companies where there is a lot of uncertainty or where budgets are tight.

A a rule of thumb, there general tradeoff is fairly clear: service-oriented architectures are powerful but difficult to use. Ultimately, what is needed is an approach that balances power with ease-of-use and lowers the technical barriers to entry.

1.2 Thesis Statement

The thesis of this dissertation is that new languages can simplify the design, implementation, and management of service-oriented systems and should support service-orientation as a first class programming paradigm. Most languages provide metaphors that abstract memory access, CPU control flow, and concurrency. This thesis argues that additional steps should be taken to abstract *communication* and *distribution* as well. Service-oriented systems are becoming increasingly relevant in the era of cloud-computing and the current technology trends indicate that service-oriented programming will continue to be important in the future.

1.3 Contributions

The core contribution of this thesis is a new programming language called Silo. Unlike most existing languages which are intended to coordinate the execution of a single operating system process running on a single machine, Silo is designed to model large systems that run across many machines connected by a network. To achieve this goal, this thesis presents the following contributions:

- **Design Goals** This thesis first identifies a core set of design goals that should govern the design of service-oriented programming languages (Chapter 2). These design goals address conceptual, technical, and pragmatic issues when building service-oriented systems.

- **Familiar Homoiconicity** Silo's novel syntax is easy-to-use and similar to popular languages that are familiar to most developers (Chapter 3). However, the syntax is also highly versatile and extensible. It allows developers to easily extend the language's core syntax with new constructs that can improve performance as well as developer productivity. This is particularly important for building service-oriented systems which tend to be diverse and could greatly benefit from custom language constructs.

- **Location Transparency** Silo's programming model is designed such that programs can be easily "broken apart" and moved between machines with ease (Chapter 4). Additionally, Silo's incorporates a novel communication mechanism that is extensible and can be used in virtually any environment from privately owned server clusters to the cloud. This greatly simplifies many of the challenges of service-oriented programming.

- **Efficient Coroutines** Building scalable I/O-bound systems is challenging. Developers are often forced to choose between the intuitiveness of blocking code and the performance of event-driven code. Silo provides a balance between these two approaches using coroutines and presents novel techniques for compiling coroutines on the Java virtual machine (Chapter 5).

Additionally, this dissertation contributes numerous examples of sample code, programming patterns, and use cases that demonstrate Silo's utility, usability, and efficiency. In particular, two case studies are presented that describe how Silo can be used during the design, implementation, and deployment of software systems.

- **Updating a Legacy System** I discuss how to migrate an existing Java-based application to Silo (Chapter 7).

- **Building a New System** I discuss how to build new systems from the ground up using Silo (Chapter 6).

1.4 Outline

Chapter 2 identifies the main design goals for a service-oriented programming language and, in particular, how these goals differ from those found in many existing languages. Chapter 3 introduces Silon, a data format and programming syntax that serves as the syntactic foundation of Silo. Chapter 4 introduces Silo, a new programming language for building service-oriented systems and discusses its design, its programming model, and how it implements many common tasks. Chapter 5 discusses Silo's implementation and includes various optimization and transformation

techniques used both during compilation and at runtime. Chapters 7 and 6 present case studies about building systems with Silo. Chapter 8 presents avenues for future work.

Chapter 2

Background

2.1 Why a New Programming Language?

Service-oriented programming is important but it can also be challenging. It makes sense to explore solutions for facilitating the creation of service-oriented systems. However, why a new programming language? Why Silo?

A key reason why I argue that a new programming language is not only warranted but overdue is that the primary programming protocols used in service-oriented programming do not have natural analogs in existing programming languages. Most existing languages are organized around features like functions, objects, closures, monads, etc. However, in service-oriented programming, the primary programming protocol is distributed and asynchronous message passing, the exact semantics of which are not naturally representable in most languages. Even so called "message-passing" languages like Smalltalk, Ruby, and Objective-C are quite different: messages are guaranteed to be delivered and processed synchronously [38, 34, 61]. As a result, service-oriented systems are awkward and verbose to write in many existing languages because the manner in which these systems are designed does not align with existing language features.

It is, of course, possible to implement asynchronous message passing libraries and distributed programming frameworks in existing languages (most programming languages are, after all, Turing complete). However, this misses an important point.

Languages do not just provide convenient programming features but also establish essential conventions and idioms. A good analogy to this is garbage collection. There are many ways to manage memory without garbage collectors (e.g. reference counting, smart pointers, private allocators). However, garbage collected languages greatly simplify development by ensuring that all code can interoperate with ease. Developers do not need to worry about the memory conventions of each function they call; rather, they can rest easy knowing that the language-wide conventions and semantics are guaranteed to be enforced. Likewise, with a service-oriented programming language, developers do not need to mentally switch back and forth between programming styles and constantly consider which features they can and cannot use.

In short, service-oriented programming represents a style of development that is not naturally or consistently expressible with existing languages.

2.2 Design Goals and Language Overview

Four core goals drove the design and development of Silo: distributed programming, efficient and simple concurrency, extensible syntax, and interoperability with a mature ecosystem. While *some* of these goals are shared with other programming languages, it is important to achieve *all* these goals simultaneously, without falling victim to design incompatibilities.

2.2.1 Distributed Programming Model

Description A hallmark feature of a modern service-oriented architecture is that different services run on different machines connected by a network. Thus a service-oriented programming language should make distributed programming easier. In particular, distributed programming should be unified throughout the design of the entire language and not just relegated to a simple RPC (remote procedure call) or networking library. The programming protocols used for local development should generalize and should be usable in a distributed environment as well. As a rule of thumb, a distributed programming model

should enable any part of an existing non-distributed program to be readily "broken off" and moved to another machine without any trouble. This ability is called "location transparency".

Silo's Approach Silo is built on the Actor model [45, 4]. Actors run concurrently, cannot share memory and can only communicate by passing messages, thus mirroring the realities of distributed systems. Silo's message passing capabilities are also extensible and allows developers to customize the protocols, encodings, and techniques used when sending messages between actors. Additionally, Silo's conventions, idioms, and standard libraries greatly discourage the use of mutable shared state. In fact, unless developers go to extraordinary lengths, values in Silo are always immutable, including core data structures like vectors and dictionaries. This makes it much easier to "break apart" an application since values can be passed to a function with the same semantics as being passed to another machine. This is in contrast to most imperative programming languages in which code is architected around large mutable in-memory object graphs. In these cases, sending an object to a remote machine is often of little use since the object is generally dependent on other objects in the graph and cannot be used in isolation. Furthermore, coordinating and synchronizing the object graph across many computers is difficult and in some cases impossible [19].

Silo's design goes to great lengths to ensure that all language features, conventions, and idioms are equally available and useful whether the system is run on a single machine or many machines. In particular, it avoids the trap in which a language originally designed for local development is retrofitted with distributed capabilities as this approach rarely works in practice [59]. In other words, Silo prevents developers from designing systems in a manner that is hard to distribute and makes an implicit promise to its users: if you can figure out how to model a system using Silo, Silo promises that the system can be split up and run across many machines with ease.

2.2.2 Efficient and Simple Concurrency

Description A service-oriented system consists of multiple services running independently from one another. Consequently, it should be no surprise that a service-oriented programming language should make concurrency easy and efficient.

Silo's Approach Silo provides an actor-based concurrency model that is optimized for message-passing concurrency.

Each actor executes on its own with its own stack and communicates by passing messages to other actors. Each actor processes a single message at a time and Silo discourages the use of shared memory between actors which eliminates many of the typical challenges of concurrent programming like race conditions and synchronization errors [84]. It also allows actors to be scheduled across many threads (or even different machines) to achieve greater parallelism.

Additionally, Silo is optimized for message-passing workloads as services spend a lot of time sending messages to one another. Just using threads for these types of workloads (i.e. processing each message on its own thread) is less than ideal because sending and receiving messages takes a long time and blocks the thread from processing other messages or doing other useful tasks [86, 126, 29]. As a result, many developers often handle messages in a non-blocking manner inside event loops which dispatch "events" to "handlers" as they occur. This reduces the number of threads and eliminates the overhead of context switches but it requires logic to be broken up across multiple callback functions leading to "callback hell" in which code is difficult to maintain [3, 125, 41].

Silo balances these two approaches using coroutines. A coroutine is a function that can pause and resume and can do so much more efficiently than a thread [25]. Silo actors run on coroutines (which are multiplexed across a fixed number of threads for parallelism) and whenever an actor blocks to receive a message it pauses the coroutine and frees the underlying thread to execute other coroutines instead. When an actor receives a message, its coroutine is resumed. This allows

straight-forward blocking code to reach the performance of non-blocking event-driven code.

2.2.3 Mature Ecosystem

Description Systems these days, especially the ones deployed for handling Web and mobile applications, are remarkably diverse. It is not uncommon for a Web service to incorporate different types of programming tasks including machine learning classification, database querying, image analysis, etc. Supporting these diverse needs in practice requires the ability to easily integrate with a mature ecosystem of existing library and tool.

Silo's Approach Silo compiles to JVM bytecode [63] and makes it trivially easy to interoperate with Java's rich ecosystem of libraries, frameworks, other languages, and tools.

2.2.4 Extensible and Familiar Syntax

Description Due to the diversity of service-oriented systems it is important for a language to incorporate an extensible and easy to use syntax. Different syntaxes are useful for different purposes. The code that is intuitive for matrix multiplication is different from the code for querying a database or declaring rules for a security policy. A service-oriented programming language must accept the inevitability that the language designers cannot predict the diverse needs of developers. Thus, the language should provide a syntax that is flexible and easy for developers to extend for their own purposes. As a rule of thumb, any special syntactic constructs that I, as a language designer, would personally want to implement to support Silo's goals of simplifying distributed or concurrent programming, should be possible to achieve without special support from the parser or the compiler. In other words, to keep myself honest I am only allowed to use language features that are also accessible to the end user.

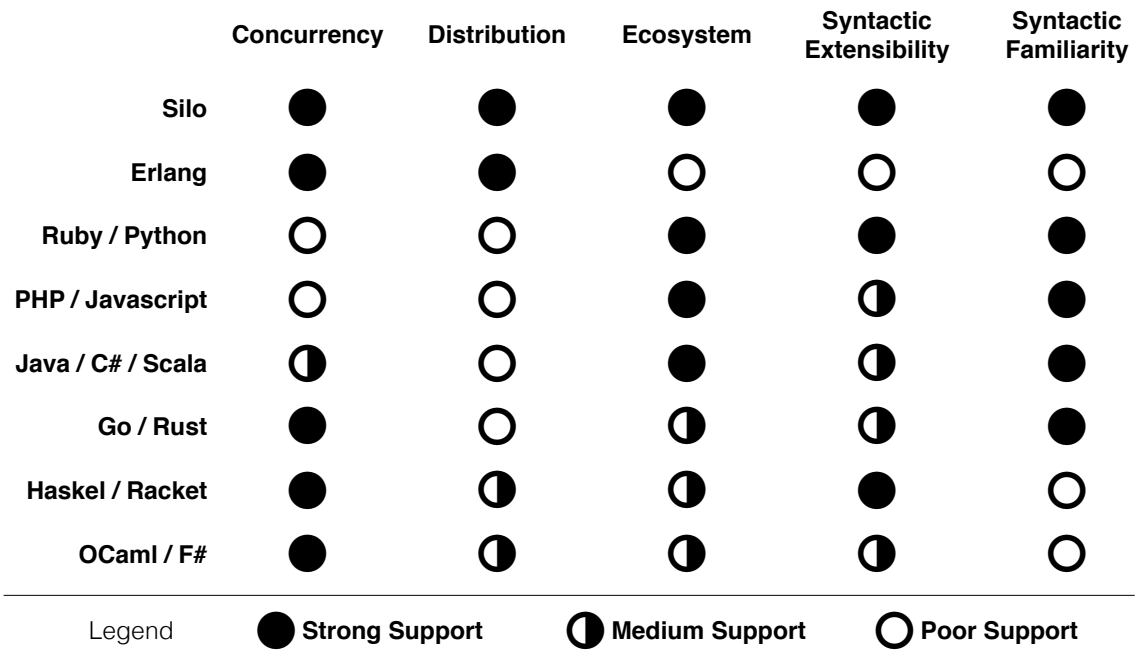

Figure 2-1: Silo's design goals and features compared to other languages.

Silo's Approach Silo achieves extensibility by introducing a syntactic notation that is homoiconic. Homoiconicity means that the language's syntax is represented using a built-in datatype of that same language [76]. This means that Silo programs can easily analyze, inspect, and change their own internal structure. This allows developers to easily extend the language with new constructs and syntactic forms.

2.2.5 Language Overview

While some of Silo's design goals are shared with other programming languages it is important that all of them be satisfied simultaneously, which turns out to be quite tricky in practice. For example, languages like Ruby provide meta programming capabilities to change the behavior and methods of objects at runtime which allows for flexible syntaxes and embedded DSLs. While this works great for scripting tasks, meta programming in this manner is not possible in a distributed environment since it relies on mutable shared state. Figure 2-1 highlights how Silo's design compares to

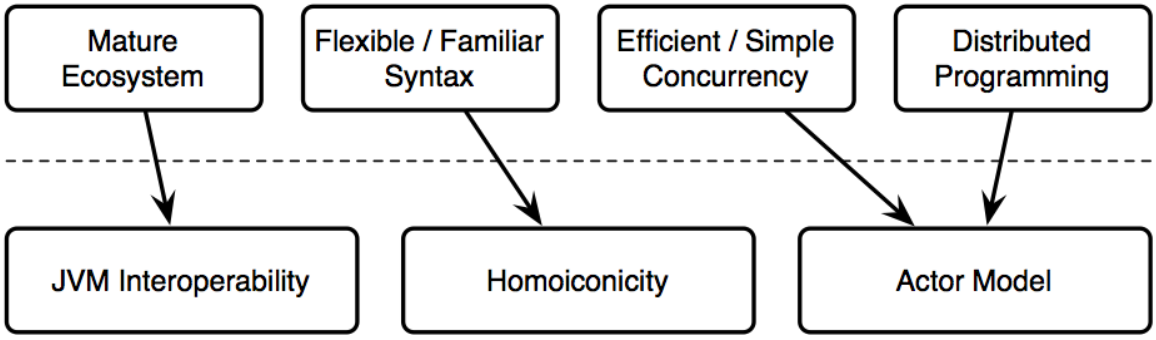

Figure 2-2: An overview of Silo's design.

other languages. This figure is not intended to be critical of any particular language; rather, it speaks to a broader point that service-oriented programming is an important niche that is being underserved and there is a great opportunity for new languages to be helpful.

Furthermore, a summary of Silo's design, features, and how they relate to the challenges and needs of service-oriented systems is shown in Figure 2-2.

2.2.6 Non-Goals

When designing a new programming language it is easy to get carried away and lose focus on a core set of features. Towards that end, it is important to be clear and upfront about non-goals of the language as well.

First, the language is a language. It is not a framework or a library. It attempts to provide a strong foundation to build larger more expressive abstractions but that is outside the scope of this thesis. In particular, Silo is not a Web framework. Web frameworks can be written in Silo but Silo does not support or encourage a particular style of Web development.

Second, Silo is envisioned primarily as a server-side language. With the growth of mobile and client-side applications there is a lot of interest in using a single unified language for both front-end and back-end development. While I can certainly appreciate the appeal of this approach it is not a goal for Silo. Silo is designed to make it easy to build network-driven server-side applications. If the language can be adopted

and used for client-side development, all the better! But it is not the goal of this line of research.

Third, Silo is not Java. It is true that Silo compiles to JVM bytecode but this does not mean that Silo is intended to be a Java-replacement. Moreover, this does not mean that Silo is an object-oriented programming language at all. Silo does include several ideas and features that are borrowed from traditionally object-oriented programming but it does not intend to provide all features like classes, inheritance, abstract classes, etc. If absolutely necessary, it is possible to access these features through Silo's interoperability with Java but writing code in this manner is unidiomatic and considered bad form; rather, developers are encourage to embrace Silo's conventions and idioms.

Chapter 3

Silon: A Familiar Homoiconic Data Format and Syntax

The syntax for Silo was heavily influenced by Lisp. However, Lisp's syntax, represented using S-Expressions, is often criticized as being hard to use for many developers. This is unfortunately because Lisp is homoiconic, in which a language's syntax is expressed using a builtin data structure of that same language, making it possible to process and manipulate application source code easily and enabling powerful extensibility and meta-programming capabilities. This chapter introduces Silon, a notation for programming syntaxes that balances the power of homoiconicity with the ease-of-use and familiarity of C-like languages. Silon is easily implementable, versatile, capable of expressing most programming constructs and is used in the Silo programming language.

3.1 Overview

Homoiconicity is a property of certain programming languages in which the language's syntax is represented using a builtin data type of the same language [76]. As a result, homoiconic programming languages are able to easily inspect and analyze their own internal structure (often referred to as an AST, abstract syntax tree) just as easily as they can manipulate an array or string. Having access to the AST enables powerful programming features like macros (special functions that extend a language's compiler by "expanding" code into a new form), meta programming (where a program is able to re-program itself), domain specific languages (implementing a language inside of another language), code analyzers (static error checkers, linters, etc.), and others [62, 118, 35].

Unfortunately, the benefits of homoiconicity come at a cost. A notation that is good at representing an AST is rarely conducive as a syntax for programming. As an example, Lisp, perhaps the most famous homoiconic language, has a syntax that only uses nested list expressions called S-Expressions [100]. S-Expressions are perfect for representing an AST but require a large number of parentheses to represent most programming constructs. As a result, Lisp is commonly described (often humorously) as being difficult to use for many tasks [96]. On the other hand, most mainstream programming languages (C, Java, Python) are not homoiconic and use special parsers that transform source code into an AST before that AST is passed to the compiler. This allows language designers to provide special support for common programming constructs (control structures, function declarations, etc.) that are more familiar and easier to use. In general, many developers prefer C-like syntaxes to Lisp. As an example, Figure 3-1 shows a comparison between Lisp and Javascript.

I present Silon, a notation for homoiconic programming syntaxes that retains a resemblance to C-like languages. Silon allows developers to enjoy the benefits of homoiconicity while working with a syntax that they are familiar with and that is easy to use. In this chapter, I make the following contributions:

- Present a formal specification for Silon (Section 3.3). Notably, this specification

```
% Lisp
(define (add-if-all-numbers lst)
  (call/cc
   (lambda (exit)
     (let loop ((lst lst) (sum 0))
       (if (null? lst) sum
         (if (not (number? (car lst))) (exit #f)
           (+ (car lst) (loop (cdr lst)))))))))

// Javascript
function addIfAllNumbers(list) {
    var sum = 0
    for(i in list) {
        if(typeof(list[i]) == "number") {
            sum = sum + list[i]
        } else {
            return false
        }
    }
    return sum
}
```

Figure 3-1: A simple program in Lisp and Javascript. Code in Lisp dialects (including Scheme, Racket, and Clojure) is written using S-Expressions and is homoiconic. Code written in Javascript requires a special parser but is generally considered easier to use.

includes a thin layer of syntactic sugar (infix operator notation and decorative markings) that makes Silon easier to use than traditional homoiconic syntaxes while preserving simplicity.

- Demonstrate Silon's utility and ease-of-use through a series of examples and use cases as well as articulating common sources of confusion and frustration that developers should be aware of (Sections 3.4 and 3.5).

- Perform a comparative analysis between Silon and other notations that indicate that Silon is easy to use and is simple to implement (Section 3.6).

3.2 Design Goals

Silon primarily serves as the syntax for Silo[1], a programming language for building distributed service-oriented systems. To service the diverse needs of these types of systems Silon was designed to support programming syntaxes that balance three goals: expressivity, usability, and simplicity.

Expressivity Silon should support syntaxes that are expressive and versatile. Since the needs of large systems are diverse and constantly changing, designing a syntax with special constructs for all conceivable use cases is not practical. Instead the syntax should be inherently extensible so developers can customize it to their needs by creating their own syntactic constructs and language abstractions. For example, developers of database-centric Web applications should be able to create constructs that facilitate writing complex database queries. Additionally, a large part of building systems requires editing configuration files. As such, Silon should be able to function not only as a programming syntax but as a data format as well.

Usability Silon should be easy to use and support language syntaxes that are understandable, writable, and maintainable. As systems and codebases continue to grow, having usable programming language is of vital importance.

[1]The name "Silon" stands for "**Silo** Notation".

Simplicity Silon needs to be a simple syntax that is easy to parse, both for humans and for computers. This is important for building tools that analyze or process application code (e.g. documentation tools, package builders, static error checkers, syntax highlighters, IDEs). Additionally, since Silon is intended to be used as a data format as well as a programming syntax, it is important that Silon be simple so that parsers in other languages are easy to write.

Reconciling these three goals and finding a balance is challenging. For example, Ruby has an expressive and easy-to-use syntax but its grammar is far from simple. Likewise, CSS is easy-to-use and has a simple format, but it lacks the expressivity of a general purpose programming syntax.

3.3 Specification

The approach for creating Silon started by defining a *simple* and lightweight unicode data format (rather than a programming syntax) and slowly adding features to achieve *expressivity* and *usability*. Silon achieves expressivity through homoiconicity and achieves understandability through a thin layer of syntactic sugar, which comes at negligible cost to simplicity, thus balancing the three design goals.

3.3.1 Data Types

Silon is a built on the small set of atoms shown in Figure 3-2. Each of these atoms are easily supported in virtually all modern programming languages and are mostly self-explanatory. Nodes, however, deserve special attention.

Nodes are a tree-like structure that have a label and a list of children (Figure 3-3). Both the children and the label can be any other Silon atom, including another node. Syntactically, a node is represented using a balanced pair of open and closed parentheses. A node's label is placed immediately before the open parenthesis and its children are placed inside the parenthesis separated by either a newline character, comma, or semicolon. If a node's label is omitted then it is assumed to be `null`. Additionally, while the separator between the children can be omitted and replaced

Atom	Example
Null	`null`
Boolean	`true, false`
Integer	`2, 512, 1024`
Long	`2L, 512L, 1024L`
Float	`3.14159f`
Double	`3.14159, 2.71828d`
String	`“Hello, World”`
Symbol	`PI, System, printLine, user_name, x`
Node	`print(“Hello, World”)`

Figure 3-2: The builtin data types in Silon.

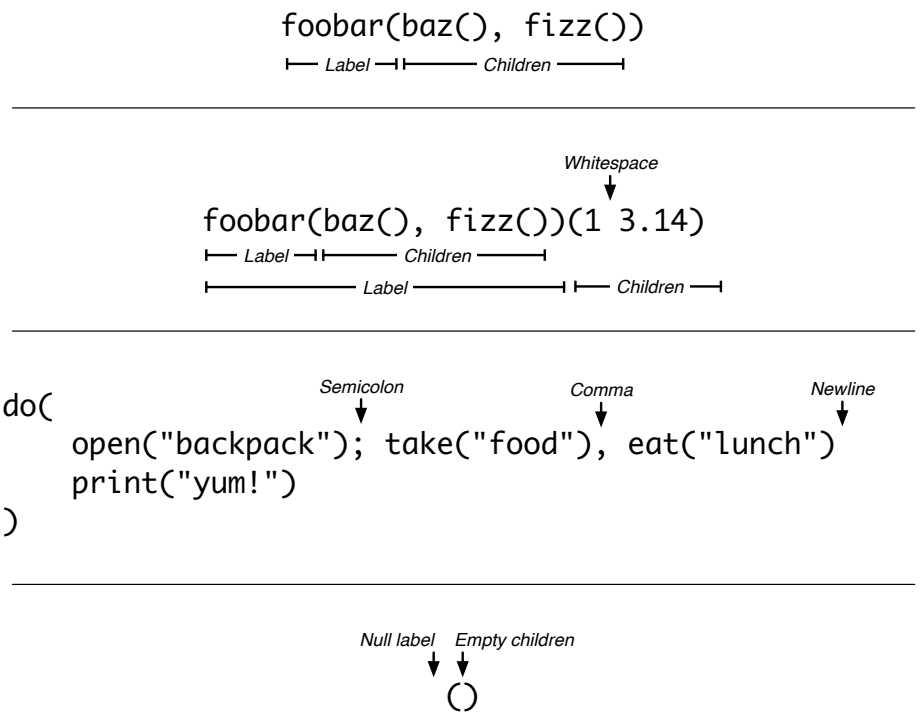

Figure 3-3: Nodes are a composite data type in Silon that have a single label and a list of children.

Sugar Form	Silon Expanded Form	S-Expressions
`x + y`	`+(x, y)`	`(+ x y)`
`a + b * c`	`+(a, *(b, c))`	`(+ a (* b c))`
`(a + b) * c`	`*((+(a, b)), c)`	`(* (+ a b) c)`
`a = b + c`	`=(a, +(b, c))`	`(= a (+ b, c))`
`{}`	`()`	`()`
`Console.print("Hi")`	`.(Console, print)("Hi")`	`((. Console print) "Hi")`

Figure 3-4: Silon supports infix operators and has syntactic sugar that parses curly braces as a node with a `null` label. The equivalent expressions are shown using S-Expressions as well.

with any whitespace character; however, this may change the way the expression is parsed. For example, `a(b() c())` is the same as `a(b(), b())` but `a(b ())` (note the spaces) is not the same as `a(b, ())`.

3.3.2 Syntactic Sugar

Two common criticisms leveled against Lisp is that it lacks infix operators and it is often hard to visually distinguish user code from language constructs because everything is represented using nested parentheses [32]. Silon addresses these concerns by introducing a thin layer of syntactic sugar.

Silon allows symbols that contain operators to be used with infix notation. Symbols are general-purpose identifiers. They are similar to strings but are not wrapped with quotes and therefore cannot contain whitespace characters (Figure 3-2). While symbols are mostly alphanumeric (e.g. variable names), they can also contain operators like `.` (dot), `+`, `-`, and `=`. However, a symbol containing an operator must *only* contain other operators. When an "operator-symbol" appears between two Silon expressions, it will be parsed as a node with the two expressions as its children and the operator as its label. For example, `5 + 5` will be parsed as `+(5, 5)`. The order of

operations in Silon are generally similar to most mainstream programming languages like C and Java. The precedence table is shown in Section 3.3.3 and additional examples of infix operators, along with a comparison to Lisp's S-Expressions, are shown in Figure 3-4.

Silon also parses curly braces as a node with a `null` label (Figure 3-4). Curly braces are used idiomatically to denote blocks of code similar to C-like languages. Curly braces function as decorative elements that make it easier to skim large chucks of code and distinguish blocks of code from language constructs (control structures, definitions, etc.).

Beyond these two forms of syntactic sugars, Silon does not contain any other special syntax exceptions that developers need to be aware of. I felt that this was a good balance of keeping Silon simple while also being able to serve as a syntax for a practical programming language.

3.3.3 Formal Grammar

A formal specification for Silon is shown in Figure 3-5 in BNF syntax. The precedence table for infix operators is determined by the starting prefix of the operator and described in Figure 3-6. All operators are left associative except for operators that contain an equals sign (`=`) which are right associative.

3.3.4 Implementation

The reference implementation of the Silon parser was built using Java and ANTLR (an LL(*) parser generator) [88]. This implementation is not particularly optimized for performance (conformance to the spec and understandability of the source code took precedence) but it is quite efficient for most practical applications and is able to parse a 8MB file in 3 seconds. Put in perspective, the parser can process the entire Silo standard library in around 500ms. All of the applications presented in this dissertation use this reference implementation.

```
 1.  silon-expression
 2.   : silon-expression operator-symbol silon-expression
 3.   | value;
 4.  value
 5.   : node | symbol | string | number
 6.   | 'true' | 'false' | 'null';
 7.  node
 8.   : '(' ')'
 9.   | value '(' ')'
10.   | '(' children ')'
11.   | value '(' children ')'
12.   | '{' children '}';
13.  children
14.   : silon-expression
15.   | silon-expression children
16.   | silon-expression terminator children;
17.  symbol: letter-symbol | operator-symbol;
18.  letter-symbol: letter | letter symbol;
19.  operator-symbol: operator | operator operator-symbol;
20.  string: '"' '"' | '"' chars '"';
21.  chars: char | char chars;
22.  char
23.   : Any Unicode character except '"' and '\'
24.   | '\"' | '\\' | '\b' | '\f' | '\n' | '\t' | '\r'
25.   | '\u' (0x0000 ... 0xFFFF);
26.  number: int | long | double | float;
27.  int: digits | '-' digits;
28.  long: int 'L';
29.  double: real | real 'd';
30.  float: real 'f';
31.  fraction: '.' digits;
32.  exponent: e digits;
33.  e: 'e' | 'e-' | 'e+' | 'E' | 'E-' | 'E+';
34.  digits: digit | digit digits;
35.  digit: '0' ... '9';
36.  real
37.   : int '.'
38.   | fraction
39.   | int fraction
40.   | int exponent
41.   | int fraction exponent;
42.  letter
43.   : 'a' ... 'z'
44.   | 'A' ... 'Z'
45.   | '$' | '_'
46.   | Unicode Categories Ll, Lu, Lo, Lt, Nl;
47.  operator
48.   : Ascii 0x21 ... 0x7E
49.      excluding letters, terminators, braces, and parenthesis
50.   | Unicode Categories Sm, So
51.      excluding braces and parentheisis;
52.  terminator
53.   : ',' | ';' | '\n' | '\t';
```

Figure 3-5: Silon's grammar in BNF.

Operator Prefix	Associativity
. ::	Left
| #	Left
!	Left
:	Left
=>	Left
* / %	Left
< <= > >= != ==	Left
&&	Left
||	Left
= |= :=	Right
(all other operators)	Left

Figure 3-6: The precedence table for Silon's infix operators.

3.4 Use Cases and Capabilities

This section discusses how many programming language constructs, features, and capabilities are naturally represented using Silon as a syntax. To illustrate this point, it includes many code fragments and describes language implementation details that are specific to the Silo programming language. While Silo and Silon were designed in tandem they are not exclusive to each other. Silo's semantics could be provided using a different syntactic representation and Silon could be used as a syntax for building other languages that are different from Silo. Thus, the examples in this section are illustrative and not prescriptive.

3.4.1 Common Language Constructs

Silon expressions are built by combining and nesting the builtin data types in different ways. The figure below shows a simple program that highlights some of the essential programming constructs like local variable assignment, control structures, functions, namespaces, and others.

```
i : int = 0
while(shouldContinue() {
    if(i % 2 == 0 {
        print("Tick")
    } else {
        print("Tock")
    })
    i = i + 1
})
```

3.4.2 Extensibility with Macros

An important attribute of Silon's homoiconicity is that programs can manipulate a structured representation of code. This allows developers to extend a compiler using macros that expand code of one form into another form. As a motivating example, a for-loop can be implemented by expanding code into a while loop and an if statement as shown below:

```
// A for loop like this...
for(i : int = 0; i < 5; i = i + 1 {
    println(i)
})

// ... can be re-written as a while loop
i : int = 0
while(i < 5 {
    println(i)
    i = i + 1
})
```

Thus, for-loops in Silo can be implemented using a macro. To be clear, this means that the for-loop in Silo is actually implemented in Silo itself and does not require

any special support from the parser, compiler, or runtime. The for-loop macro is shown below using the `transform` construct which simply "copies and pastes" code fragments into a code template. This is an easy way to implement many types of macros and is similar to the C-preprocessor.

```
transform(for(init, condition, end, body) {
    init
    while(condition {
        body
        end
    })
})
```

Interestingly, underneath the scenes, the `transform` construct is actually just a macro itself. Formally defined, A macro is any function that accepts one or more Silon expressions as input and returns a single Silon expression as output. The transform macro is simply a "helper macro" that facilitates a common use case, namely, taking multiple parameters and inserting them at certain placeholders within a template. However, macros are Turing complete and can perform any arbitrary computation as long as they return a valid Silon expression.

The canonical representation of a macro in Silo is a function that is annotated with the `macro` flag. For example, the implementation of the `while` macro is show below. The `while` macro is more barebones and low-level when compared to the for-loop because many of the higher-level syntactic constructs (for example, the `transform` macro) are implemented using the `while` macro and, thus, are not available for use.

```
function(
    name(while)
    macro
    inputs(condition, body)
    outputs(silo.lang.Node)
    {
```

```
        b : java.util.Vector = java.util.Vector()
        b#add(condition)
        b#add(body)
        b#add(silo.lang.Node(silo.lang.Symbol("break")))

        l : java.util.Vector = java.util.Vector()
        l#add(silo.lang.Node(silo.lang.Symbol("branch"), b))

        silo.lang.Node(silo.lang.Symbol("loop"), l)
    }
)
```

Transforming code from one form into another is a powerful abstraction; however, at some point, these transformations need to end and yield a canonical representation that the compiler understands and can compile into machine code. These representations are called "special forms". Special forms represent a low-level set of expressions that could otherwise not be implemented using a function or a macro. For example, the `branch` statement cannot be implemented using a function (since its arguments need to be lazily evaluated) or a macro (because a macro can only transform code into a new form which is not helpful if a "branch form" is not available). Thus `branch` is implemented as a special form inside the compiler itself. Using the branch special form, developers can now add more familiar constructs like if-else statements, switch statements, and pattern matching without support from the compiler. This allows the core compiler to remain simple as it only needs to be aware of a handful of special forms. As an example, Silo's compiler only implements thirty special forms. The exact number and purpose of these special forms will vary drastically between compilers but as a general practice, most compilers will typically include special forms so as to expose all the features and functionality of the underlying target platform or instruction set (x86, JVM bytecode, etc.).

As can be seen, macros are powerful and can be used by developers to add new syntactic constructs to a language without cooperation or special support from the

compiler. In fact, most common programming constructs are implemented as macros (`func`, `while`, `if`, and others).

3.4.3 Developing Macros

Silo includes many constructs to facilitate the development of macros. The `quote` construct accepts a single Silon expression and returns an expression that, when compiled and run, yields a literal representation of that same expression. For example, if a developer wanted to create an expression for the literal `a + b` they could use the following code:

```
a : int = 5
b : int = 10
println(a + b)
println(quote(a + b))
```

This code will first print out "10" since the compiler will actually execute the code. However, the second print statement will print out "+(a, b)" since the quote macro returns its argument exactly as it appears (unaltered and unevaluated), hence the name "quote". However, sometimes it is useful to insert an evaluated expression into a Silon literal. This can be done using the `escape` inside of a quoted expression. `escape` is not a special form; rather, the `quote` macro scans over its argument and whenever it finds a node labeled `escape` it allows the expression to be evaluated by the compiler. For example:

```
a : String = "foo"
b : String = "bar"
println(quote(a + escape(b)))
```

will print "a + bar" since the `b` variable is escaped and will be evaluated with respect to the surrounding context. As an interesting edge case, one may ask how to quote an expression that yields "a + escape(b)"? The following code fragment demonstrates this.

```
quote(a + escape(quote(escape))(b))
```

This code works because the quote macro searches for a *node* with a label called "escape". However, the symbol "escape" is treated like any other symbol. Thus, the code first escapes the literal symbol "escape" and then uses it as the label of a node that has a symbol "b" as its argument. In certain ways, it parallels the use of "double backslash" escape sequences in C string literals.

The quote macro is a useful construct at creating snippets of code. An excellent example of the quote macro in use is the `macro` macro. The `macro` macro is syntactic sugar that allows developers to declare a macro without having to resort to the the verbose `function` special form (as shown previously with `while`). A "max" macro can be defined as such:

```
macro(max(a, b) {
  // Code omitted.
})
```

The implementation of the `macro` macro is similarly straight forward:

```
function(
    name(macro)
    macro
    inputs(id : Object, body : Object)
    {
        quote(silo.core.func(escape(id), escape(body), Boolean.TRUE))
    }
)
```

An interesting take away from this example is the use of the fully qualified name "silo.core.func". Macros (like `func`) are functions and (unlike special forms) are organized into namespaces and packages to avoid name clashes. Using unqualified names can lead to awkward and difficult to debug bugs. For example, the following code may not do what the developer expects:

```
package(example.library)

func(println(a) {
    file.append(file.open("log.txt"), a)
})

macro(log(a) {
    quote(println(escape(a)))
})
```

In this example the developer created their own `println` function that appends to a log file. They also created a macro called `log` to inline calls to this println function. Thus, the following code:

```
package(example.app)
func(start {
  log("Hello, World!")
})
```

is expanded to:

```
package(example.app)
func(start {
  println("Hello, World!")
})
```

The problem is now perhaps more evident. The user code is not inside the "example.library" package and thus when the expanded form is compiled the `println` inside of `silo.core` (one of the default packages that is included automatically) will be called rather than `example.library.println`. To correct the issue, the macro developer needs to be mindful of this and use the fully qualified form:

```
package(example.library)

macro(log(a) {
    quote(example.library.println(escape(a)))
})
```

As an alternative, the Silo compiler provides a `quotecontext` special form which operates like the `quote` macro except it expands resolvable symbols into fully qualified names as if they were un-quoted function calls. This make it convenient for writing complex macros that expand into calls to many functions or recursively transform themselves into other macros. Unlike `quote`, `quotecontext` is a special form and not a macro because it requires cooperation from the compiler to expose the context in which the statement was used. Another way to implement the log macro is as follows:

```
package(example.library)

macro(log(a) {
    quotecontext(println(escape(a)))
})
```

One last source of confusion for macro developers is the topic of hygiene. This is perhaps best illustrated by means of an simple example:

```
macro(max(a, b) {
  quote(if(escape(a) > escape(b) {
    escape(a)
  } else {
    escape(b)
  }))
})
```

In Silo the `if` statement returns the last value of the branch it takes. As such, at first pass, this code snippet seems to be correct. However, there is a subtle issue:

```
a : int = 5
b : int = 4
println(max(a, b = b + 2))
```

Most would expect this code to print "6" when in reality it will print "8" because the "b = b + 2" will be evaluated twice. To avoid this issue the macro should assign both values to a temporary variable to the expression is only evaluated once:

```
macro(max(a, b) {
  quote({
      tempA = escape(a)
      tempB = escape(b)
      if(tempA > tempB {
        tempA
      } else {
        tempB
      })
  })
})
```

This updated version will work correctly. Our simple driver program will correctly print "6" but there is another issue:

```
tempA : int = 0
a : int = 5
b : int = 4
max(a, b = b + 2)
println(tempA)
```

The above code will print "5" instead of "0". The reason for this is that the macro squashed the local variable "tempA". This is problematic because it is a bug that is not visible in the user's code and thus hard to find. To avoid this issue, Silo includes

the “uniquesymbol” and “macrolocal” special forms which can be used to create identifiers that are guaranteed to not create naming conflicts. “uniquesymbol” will yield a symbol that is impossible for a user program to generate by embedding an infix operator into the symbol before a unique auto-increment counter. For example, a possible output of `uniquesymbol()` is “silo.core.unique.symbol:42”. This is impossible to appear in a user program because the parser would be expand this text into a Node expression: “:(.(.(.(silo, core), unique), symbol), 42)”. While “uniquesymbol” produces a unique identifier, “macrolocal” produces a re-usable hygienic identifier that can be used within a macro like a local variable; however, it is still guaranteed to be unique across different macros and even across different invocations of the same macro. A final, proper max macro therefore follows as such:

```
macro(max(a, b) {
  quote({
    macrolocal(a) = escape(a)
    macrolocal(b) = escape(b)
    if(macrolocal(a) > macrolocal(b) {
      macrolocal(a)
    } else {
      macrolocal(b)
    })
  })
})
```

3.4.4 Use Cases for Macros

Macros are a powerful language feature that have several applications and use cases. Many special features of existing programming languages can actually be easily implemented through macros without any changes to the compiler or language specification.

For example, C# has a `using` statement that coordinates access to unmanaged resources like file descriptors, hardware device contexts, fonts, etc. The using state-

ment in C# ensures that a special `Dispose` method is called to ensure that the system resources are released. In C# the `using` construct requires special support from the compiler but in Silo a `using` statement is a simple macro:

```
transform(using(expression, body) {
  macrolocal(a) = expression
  try({
      body
  } finally {
      if(macrolocal(a) {
          dispose(checkcast(macrolocal(a), Disposable))
      })
  })
})

// Example usage:
using(f : File = file.open("file.txt") {
    ...
})
```

Similarly, macros can be used to create convenient constructs that help eliminate small annoyances. For example, many managed programming languages like Java and C# throw exceptions when certain pre-conditions are not met, for example, if a file does not exist. While this makes it easy to track down errors at runtime it requires developers to insert try-catch statements that can clutter their code unnecessarily. To avoid this, an `ignore` macro simply re-writes user code to automatically ignore certain exceptions. It is a way for developers to tell the compiler that certain exceptions may be thrown but they do not care:

```
contents : String = null
ignore(FileNotFoundException {
    f : File = file.open("file.txt")
```

```
    contents = file.readAll(f)
})

if(contents == null {
    log("Empty file or file not found...")
    return
})
```

Macros can also approximate helpful features of languages such as built-in syntax for matrices like Matlab and named parameters like Smalltalk and Objective-C:

```
m : Matrix = matrix.create(int, 3 * 3 {
    1 0 0
    0 1 0
    0 0 1
})

m = matrix.subMatrix(left=0, right=2, top=0, bottom=1)
println(m)

// Outputs:
// 1 0 0
// 0 1 0
```

Lastly, macros can be used for more than mere syntactic sugar and copy-and-paste programming. Macros can extend the behavior of a compiler in many ways. For example, a simple macro that was developed early in Silo's implementation was the `todo` macro:

```
macro(todo(message : String) {
    throw(message)
})
```

This macro simply throws an exception saying that a certain task had not been completed yet. This is interesting because the exception will be thrown at *compile* time, not at runtime. This is a handy mechanism to statically detect that certain parts of a code base have not been written yet.

Taking this idea further, macros can implement any arbitrary logic. As a crazy and extreme example, a macro could post a message on Reddit[2] and ask users to implement certain functionality. Once a developer responds to the post, the macro returns the code to the compiler which can check if the code is valid or not. Example usage could look something like the code snippet below:

```
func(average(a : Vector) {
    askReddit("Hi Reddit, can someone write a
        function that computes the average of a vector?
        The name of the vector is \"a\". Thanks!"
    )
})
```

More practically, macros can be used to detect certain bugs, check to see if a code fragment performs side effects and optimize accordingly, check to see if functions are documented or tested, or even ensure that certain organizational naming conventions are obeyed.

3.4.5 Domain Specific Languages

Another major use case for macro are to support and implement hosted domain specific languages. The distinction between a DSL and an easy-to-use API is not clear but the moniker "DSL" is usually applied to cases that are more involved, unique, and focused in nature. For example, writing database queries, setting security policies in a declarative manner, or establishing HTTP routes that map URLs to executable logic. These advanced use cases are easily expressible in Silon in a familiar manner.

[2]Reddit is a popular online forum that has many sub-communities dedicated to specific topics, including programming.

As an example, the program below is a simple HTTP Web service that queries a school's database and returns a list of every student for a given teacher. This example provides a demonstration of Silon's flexibility in ways that are far more difficult (if not impossible) in other syntaxes and languages. Once again, these advanced constructs are all possible using macros and require no cooperation from the compiler itself.

```
get("/teacher/:id/students" {
    teacherId : String = map.get(params, "id")

    query : Query = db.sql(
        select(students.*)
        from(students => s)
        join(enrollment => e on d.student_id == s.id)
        where(
            // Notice how I can use the "teacherId" variable
            e.teacher_id == escape(teacherId)
        )
    )

    results : ResultSet = db.exec(query)
    return(json.stringify(results))
})
```

3.4.6 Data Formats

Silon is equally useful as a simple data format for information sent over a network as well as as an application or configuration file format. The latter is particularly interesting as large systems comprise many configuration files for various use cases: localization files, database configuration, test suite configuration, compiler configuration, documentation of patches notes, etc. Many systems use formats like JSON, XML, or YAML for these purposes since they are easily parsed and human readable.

Silon is just as capable in this respect but has the added benefit of being the same format as the application code. Not only does this preserve consistency across the code base but it also makes it easy to move application configuration properties between hard-coded constants and configuration files. The example below shows how to represent structured data using Silon in a manner that is similar to JSON or XML.

```
database(
    adapter => mysql
    encoding => utf8
    reconnect => false
    database => test
    pool => 5
    username => root
    password => ""
    socket => /tmp/mysql.sock
)
```

Recently, traditional data formats like XML and JSON are being co-opted as programming interfaces. For example, MongoDB uses JSON as an alternative to SQL [28, 2], Microsoft's WPF framework uses an extension to XML (called XAML) to build user interfaces [66, 77], and the Apache Ant build tool uses XML for common scripting tasks [8]. All of these tools chose to use an existing data format because parsers are either readily available or easily implementable in different languages, making interoperability simple. Unfortunately, the syntax for common tasks can be awkward as shown in the MongoDB query below:

```
// Instead of a more natural "quantity > 20" MongoDB requires:
db.inventory.find({"quantity": { "$gt": 20 } } )
```

However, Silon provides a best of both worlds. Like JSON and XML, Silon is a simple data format that is easily parsed and representable in most modern programming languages. However, Silon is more conducive as a programming syntax than JSON or XML.

3.5 Failure Cases

While Silon has certain positive properties, it is also important to realize Silon's limitations and potential sources of frustration.

3.5.1 Syntactic Gotchas

Since Silon is similar to C and C-like languages, developers may instinctively write code that is valid in C but not valid Silon. This can lead to simple mistakes that are easy to miss and hard to track down. In my experience using Silon, three concrete examples are troublesome.

1. **Return Statements**. Consider the following function:

   ```
   func(add(a : int, b : int => int) {
       // Note: the missing parenthesis with return
       return a + b
   })
   ```

   This code yields a compiler error. In Silo when the "return" statement appears on its own as a symbol (rather than as a node with parentheses) the compiler assumes that the developer is marking the end of a function and returning nothing (i.e. `null`). However, in this example, since `null` is an object reference and not an `int` the compiler raises an error.

2. **Trailing Parenthesis** Most common constructs in Silon have a closing parenthesis after the closing brace (e.g. while-loops, if-statements, functions). Often times a developer's "muscle memory" will take over and they will instinctively close the parenthesis before typing the braces leading to strange compilation errors. As an example:

   ```
   // This is not valid. Note that the braces come after the parenthesis.
   if(user == "bob") {
   ```

```
    print("Bob logged in")
}
```

3. **Type Declarations** Consider the following code snippet:

   ```
   // Note: not "s : String"
   String s = "Hello, World!"
   ```

   Instead of using Silo's "colon-syntax" for types, this code is using traditional C/C++ syntax where the type name comes before the identifier. This example is particularly nefarious because there is no compilation error at all. The compiler will parse `String` as a reference of the string class and then parse a local variable assignment. Since the type is omitted, the Silo compiler will assume that the `s` variable is an `Object` (i.e. the root type in the type system), which could lead to weird runtime behavior or compilation errors that appear at a distance from the offending line.

Developers should keep these shortcomings in mind when using Silon. Some of them are easily mitigated while others are more tricky. For example, many modern text editors and IDEs automatically insert the closing braces or parentheses for you, drastically reducing the likelihood of misplacing the closing parenthesis in a Silon expression. Moreover, linters, warnings, in addition to well worded compilation and error messages can go a long way at helping developers identify the potential root cause of an error.

3.5.2 Dangers of Macros

Macros are a powerful language feature and like many powerful features allow developers ample opportunities to shoot themselves in the foot.

Developing macros can be challenging. Macros in Silo are Turing complete and thus infinite loops (and other degenerate behaviors) are possible and can be hard to track down. For example, macro `A` may expand into a call to macro `B` (as opposed

to calling it directly) and `B` expands back into a call to `A` leading to infinite recursion. This is particularly troublesome since stack traces are completely useless in this situation.

Not only can macros be hard to implement, they can also be hard to use. Many bugs can be introduced because users did not use a macro in the way the macro was intended to be used. This can lead to bugs that are far harder to track and find because the bug is not in user code but buried in third party code. For example, a macro developer could have been negligent and accidentally overwritten the local variable `i` that the calling code was also using. This is especially troublesome with "heavy-weight" macros that expand code multiple times.

Lastly, macro usage can be taken to extremes. Since building DSLs is simple in Silo developers can feel the urge to make a DSL for everything. This is problematic because it reduces the consistency between third party libraries. Every library is used differently, exposes a radically different API, and incorporates different programming models and usage paradigms. While presenting a custom and hand-crafted API is well intentioned, it can also make code hard to read if every library introduces new syntactic constructs that need to be learned and mastered independently. Developers of third-party libraries need to resist the urge of creating exotic APIs and instead be mindful of language-wide conventions, idioms, and syntax.

In light of these issue, the general rule of thumb is to use macros sparingly and for specific and narrow purposes. If a particular problem can be solved using a function, then the use of a function is preferred over a macro. If a macro is needed, the macro should be kept as simple as possible. Functions are generally easier to understand, use, debug, and standardize than custom syntaxes created with macros.

That said, when using macros, developers should also make use of debugging tools provided by Silo. In particular, the `macroexpand` special form takes a Silo expression and recursively expand all macros until the expressions is transformed contains only special forms. Additionally, the `macroexpandonce` special form only performs a single iteration of macro-expansion rather than fully expanding the entire expression. Together these two special forms provide developers some tools when

working with macros.

3.6 Comparative Evaluation

3.6.1 Other Syntaxes

To evaluate Silon's usability, I compare code samples written in different syntaxes. While code preference is highly subjective, developers overall are more comfortable and familiar with C-like syntaxes, like Javascript [116]. Figure 3-7 shows the binary search algorithm implemented in Javascript, Silo, and Scheme (a Lisp dialect). Looking at these code examples, Javascript is far similar to Silo than it is to Lisp.

In fact, the Silo and Javascript implementations are nearly identical with the exception of the placement of the closing parentheses. However, even this is likely not problematic for most developers given the emerging popularity event-driven Javascript code which encourages passing functions as arguments. An example from the popular jQuery library is shown below:

```
jQuery("p").click(function() {
    jQuery(this).hide();
}); // Note the brace inside the parenthesis
```

Judging from the similarities between Silo and popular existing languages (like Javascript), it would seem that developers would be familiar with and comfortable using Silo for development tasks.

3.6.2 Grammatical Complexity

To evaluate Silon's simplicity (one of its design goals), I compare the number of BNF rules needed to implement a parser for Silon to a variety of different languages and data formats (Figure 3-8). The number of rules for every language was taken from the language's formal specification or from the source code of its reference implementation. As can be seen, Silon's grammar ranks as one of the simplest along with JSON

```
// Silon
func(binarySearch(nums, target) {
    helper = fn(helper, lo, hi {
        if(hi < lo return(-1))
        guess = (hi + lo) / 2

        if(nums(guess) > target {
            return(helper(lo, guess - 1))
        } else(nums(guess) < check) {
            return(helper(guess + 1, hi))
        })

        guess
    })
    helper(helper, 0, arraylength(nums))
})

// Javascript
function binarySearch(nums, target) {
    helper = function(lo, hi) {
        if(hi < lo) { return(-1) }
        var guess = (hi + lo) / 2

        if(nums[guess] > target) {
            return helper(lo, guess - 1)
        } else(nums[guess] < check) {
            return helper(guess + 1, hi)
        }

        return guess
    }
    helper(0, nums.length)
}

% Scheme
(define (binary-search value vector)
  (let helper ((low 0) (high (- (vector-length vector) 1)))
    (if (< high low)
        #f
        (let ((middle (quotient (+ low high) 2)))
          (cond ((> (vector-ref vector middle) value)
                 (helper low (- middle 1)))
                ((< (vector-ref vector middle) value)
                 (helper (+ middle 1) high))
                (else middle))))))
```

Figure 3-7: A comparison of binary search implementations in Silon, Javascript, and Scheme.

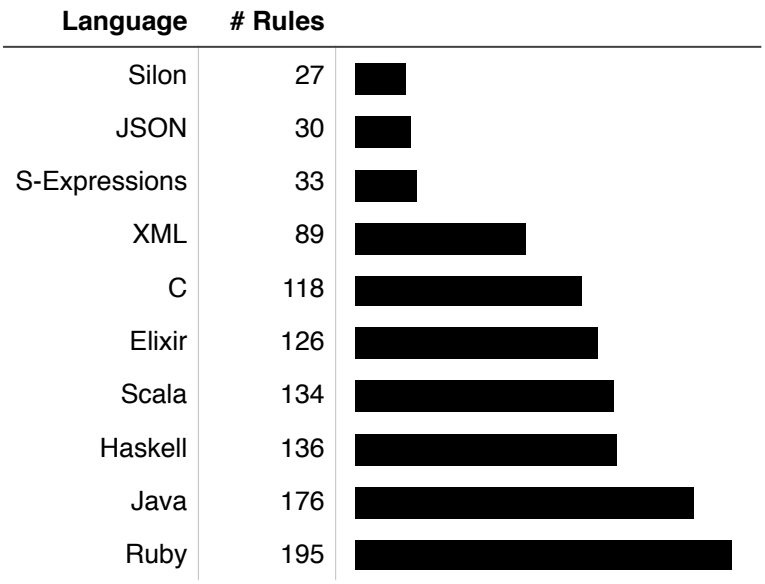

Figure 3-8: The number of rules needed to implement a parser of various languages using BNF.

and S-Expressions. My hope is that this will help encourage its use and adoption in many environments, especially in the design of future programming languages.

3.7 Related Work

3.7.1 S-Expressions and Lisp Dialects

Silon is most directly related to S-Expressions, a list-based notation that is famously used in Lisp dialects [100]. S-Expressions, however, are often criticized as a programming syntax because understanding the structure of a program is difficult due to the overabundance of parentheses. Silon avoids this problem by using a different notation and incorporating a thin layer of syntactic sugar.

The *Readable S-Expression Project* is an effort that attempts to improve the usability of S-Expressions as a syntax for Lisp dialects [32]. Notable examples of these improvements include "c-expressions", which allow infix operator notations if the expressions are wrapped with curly braces, "n-expressions", which allow the name of a

function to be moved before the parenthesis during an invocation, and "t-expressions", which make parentheses optional and uses indentation to distinguish function arguments. Together, these extensions to S-Expressions are called "Sweet Expressions".

Silon and Sweet Expressions aim to address similar issues but take different approaches. Sweet Expressions are designed specifically as a compatible drop-in replacement for Lisp dialects whereas Silon is intended as an entirely new format and can thus take greater syntactic liberties. As an example, common programming expressions like `System.out.println("Hello, " + user.name)` and `std::cout << "Bye!";` are not possible with Sweet Expressions but are possible with Silon.

Beyond the *Readable S-Expression Project*, certain Lisp dialects introduce their own syntactic sugar for special purposes. As an example, Clojure has built-in support for array and dictionary data types [46].

3.7.2 Other Homoiconic Programming Syntaxes

Many homoiconic programming languages have been proposed over the years. Classic examples include TCL and certain shell interpreters like Bash [85, 95]. These languages are string-based which greatly simplifies exposing their internal structure to developers. However, being string based, it is often difficult to infer a program's structured representation making it hard to use for complex programming tasks. Thus, these languages do not meet our usability design goal.

More recent languages like Julia and Elixir are also homoiconic but take a different approach from Silon. They have grammars with special syntactic support for common constructs conditional statements, modules, loops, and others [15, 92]. As a result, while Julia and Elixir technically expose an AST to users, the overall syntax of the language is less generalizable and the representation of the AST is more complex to process. Thus, languages like Julia and Elixir do not meet our design goals for simplicity (it is harder to parse these languages; see the Section 3.6.2) and versatility (it is difficult for these languages to be used as a data format).

3.7.3 Meta-Programming and Lazy Evaluation

There are ways to achieve expressivity other than homoiconicity. Languages like Ruby and Haskell can craft extremely flexible APIs using features like anonymous functions, late binding, runtime evaluation, invocation forwarding, and lazy evaluation [34, 47]. These meta-programming techniques were not chosen for Silon as it did not satisfy our need for simplicity and versatility. Building a dynamic language runtime or a lazy-evaluation programming language is a heavy-weight solution for many of Silon's use cases (e.g. application file formats). Additionally, Ruby-esque meta programming relies on techniques which change the behavior and methods of objects to allow for flexible syntaxes and embedded DSLs. While this works well for scripting tasks, this type of meta programming is not possible in a distributed environment since it makes liberal use of side effects and shared mutable state which is difficult to coordinate across multiple machines. Lastly, meta-programming techniques only address some of the benefits of homoiconicity. Other use cases like compiler extensions, static error checkers, and linters are not possible through these meta-programming techniques.

Chapter 4

Silo: A Service-Oriented Programming Language

Service-oriented systems, in which application features run as independently executing services over a network, are notoriously hard to build and manage. However, service-oriented architectures are becoming increasingly important, especially for Web and mobile applications, to achieve high scalability and reliability in a cost-effective manner. This chapter introduces Silo, a new programming language that facilitates the design, implementation, and maintenance of service-oriented systems. Using Silo, developers are able to model a complex distributed system as a single program and easily express many service-oriented patterns directly in the language without relying on additional infrastructure, ad-hoc conventions, or intrusive and convoluted programming styles to "glue" the system together.

4.1 Overview

Service-oriented systems decompose application components into separate and independent services that run together in a networked environment. This approach has many benefits and is becoming increasingly popular in industry; however, it is also particularly challenging (see Section 1.1).

A key contributing factor to these challenges is the design of existing languages which are designed to only coordinate the execution of a single machine. They encourage designs and offer enticing features that either do not work or complicate distributed programming. This mismatch between distributed programming and existing languages makes engineering systems challenging. For example, many languages make heavy use of memory side effects when building applications. It is quite common to write functions that have no return value in languages such as Java, Ruby, Python, and C; instead, these functions are called to either change some global shared state in the program or mutate one or more of its arguments. In both of these instances, the program's *correctness* depends on consistent and shared memory, which is not possible to achieve across multiple machines [19]. On the other side of the language spectrum, many functional languages like Haskel, OCaml, and Clojure sidestep this issue by encouraging referential transparency, immutable values, and a minimization of shared state. However, most functional languages, like their imperative counter parts, still do not have constructs that naturally model distributed systems[1] and have to rely on low-level and platform-specific functionality like threads, processes, message queues, and sockets which often undermine nice high-level language abstractions and require in-depth technical skill to use effectively.

To address these challenges, I present Silo, a new service-oriented programming language. Silo introduces a simple programming model that aligns well with distributed programming. It allows applications to be easily migrate from running on a single machine to running across many and provides an extensible message passing mechanism that abstracts communication in distributed environments. This chapter

[1]With the exception of Erlang, which we address separately.

presents the following contributions:

- I explain the idea and importance of location transparency, in which parts of a program are able to be moved from one machine to another without major changes, and why it is critical for service oriented systems (Section 4.2).

- I describe Silo's core features, idioms, and conventions and how they address common programming tasks without sacrificing location transparency (Section 4.3). Interestingly, Silo is an imperative language but is still able to achieve location transparency, whereas most attempts in the past pursued a functional approach.

- I describe Silo's concurrency and message-passing capabilities (Section 4.4). Silo also presents a novel idea called polymorphic delegation which allows developers to unobtrusively extend the message-passing capabilities to work in virtually any computing environment.

- I present a performance evaluation that demonstrates how Silo can reach the performance of Java and outperform many other languages commonly used in Web application development (Section 4.6).

- I identify a series of patterns that routinely appear in service-oriented systems and describe how to express those patterns directly in Silo without relying on external functionality (Section 4.7). The implementations of these patterns are concise, reusable, and approach the performance of existing best-in-class solutions.

4.2 Location Transparency

Achieving location transparency means that a program or system can continue to operate correctly regardless of the physical location of a particular resource or software component. Practically speaking, this means that parts of an application can be "split

off" and run as a separate OS process or on another machine without major changes to the source code.

4.2.1 Needs of Modern Systems

Location transparency is particularly helpful in the evolution of service-oriented systems. A key challenge when architecting such systems is determining which functionality to expose as a separate service. In some cases, it may be obvious (e.g. the mobile API); however, others are less obvious (e.g. a logging service) and others only become clear after building an initial version of the system and deploying it to real users. For example, it may turn out that malicious users are spamming the search functionality and reducing the overall performance of the system for other users. In response, the development team may want to deploy its search functionality as a separate service with more hardware resources to handle the load as well as deploying a new monitoring service that tracks the behavior of users on the site to help identify bad citizens. Unfortunately, at this point it may be too late as these types of changes may require fundamentally re-designing the entire system. There are plenty of stories of organizations (big and small) struggling to keep their products up and running in the amidst of unprecedented growth, traffic load, and new feature requests [40, 68, 7, 13, 5].

What is ultimately needed is an agile mechanism by which developers can build an initial version of a system quickly and progressively "mold it" as the product requirements change and as pain points are discovered. This is where location transparent programming would be particularly valuable. Instead of having to build an initial version and then start over from scratch, a new language would enable the system to naturally evolve and adapt to new requirements, features, and unforeseen usage patterns. Developers can make technical design decisions just-in-time rather than attempting to be clairvoyant and predict future needs.

4.2.2 Actor Model

Silo is built on the foundations laid by the Actor model, a theoretical model of concurrent computation that emphasizes asynchronous execution and unbounded nondeterminism [45, 4]. An actor is a fundamental unit that encapsulates processing, storage, and communication. Informally, each actor has an address and a mailbox. Actors communicate by placing messages in a mailbox at a certain address. In the actor model, only the owner of a mailbox can read the messages but any actor can place a message in any mailbox. Furthermore, message passing is the only way that actors can communicate as there is no shared memory. In fact, the only way for an actor to discover the address of another actor is if that address is sent as a message, similar to a return address on an envelope.

A unique aspect of the actor model is that all communication is asynchronous and best effort, meaning that messages can be lost, arrive out of order, and mixed in with messages from other actors. While these lack of guarantees may makes the actor model seem spartan (or perhaps useless) it accurately captures the realities of distributed systems and provides a theoretical framework by which researchers can explore approaches and mechanisms to solve practical issues. Additionally, at a practical level, virtually all communication mechanisms, from hardware protocols like SPI to high level networking protocols like HTTP to social protocols like snail mail, can be easy expressed by the actor model. Thus by adopting the actor model's semantics, Silo can provide a unified interface that properly abstracts virtually any communication mechanism. Additionally, developers can rest easily knowing that Silo's programming protocols can be used in many computational environments and do not forgo relying on platform-specific affordances like XPC on Mac OS X or Amazons Simple Queue Service on AWS [9, 6].

Actor also serves as an excellent language abstraction for achieving location transparency. Since actors execute independently from one another and because the message passing semantics can be preserved in a distributed environment, moving an actor from one machine to another is not a problem. Thus in a sense, an actor-based

language forces developers to build applications in a manner that can be readily split apart. This is especially true if the language enforces immutability and requires the use of actors to encapsulate shared mutable state. Additionally, since actors exist as a language-level construct, refactoring an application can be easy since it requires no cooperation with external infrastructure like sockets, port mapping, DNS entries, load balancers, firewall settings, etc.

4.3 Core Features

At its core, Silo is a fairly simple procedural programming language that compiles to JVM bytecode. It should be familiar to most programmers comfortable with C-like languages. However, Silo was designed specifically to ensure location transparency: all language features, idioms, and programming protocols that work on a local single machine program work in a distributed environment as well.

In a way, Silo represents a compromise between imperative and functional programming paradigms. Functional languages are often touted for their ability to easily parallelize (and perhaps by extension distribute) due to referential transparency, minimization of shared state, and immutable value types. Silo adopts many of the ideas of functional languages but avoids becoming a functional language as many developers are more familiar with an imperative programming model. At the same time, designing Silo requires a lot of discipline to avoid adding features from other imperative programming languages that do not work in a distributed system and finding appropriate replacement for those features. This compromise, I feel, strikes a nice balance that facilitates not only development of service-oriented system but is hopefully inspirational to future distributed programming languages.

4.3.1 Basic Usage

Silo's base syntax uses Silon, a homoiconic data format and programming syntax discussed in Chapter 3. Despite being homoiconic, Silon should appear familiar to users of C-like programming languages. Silo supports most common language fea-

```
func(binarySearch(nums : Vector, target : int => int) {
    helper : Function = fn(helper, lo, hi {
        if(hi < lo return(-1))

        guess : int = (hi + lo) / 2
        value : int = vector.get(nums, guess)

        if(value > target {
            return(helper(lo, guess - 1))
        } else(value < check) {
            return(helper(guess + 1, hi))
        })

        return(guess)
    })
    helper(helper, 0, vector.length(nums))
})
```

Figure 4-1: Binary search written in Silo with types.

```
package(linkedlist)
type(LinkedList {
    head : int
    tail : LinkedList
})

func(create(values ... => LinkedList) {
    list : LinkedList = null
    for(i : int = vector.length(values); i >= 0; i = i - 1 {
        value : int = vector.get(values, i)
        list = LinkedList(value, list)
    })
    return(list)
})

func(insert(list : LinkedList, index : int, value : int => LinkedList) {
    if(index < 0 {
        throw("Out of bounds")
    } else(index == 0) {
        return(LinkedList(value, list))
    } else {
        return LinkedList(
            list.head,
            insert(list.tail, index - 1, value)
        )
    })
})

l = linkedlist.create(1, 3, 4)  // (1, 3, 4)
l = l | linkedlist.insert(1, 2) // (1, 2, 3, 4)
```

Figure 4-2: A linked-list data structure written in Silo.

tures: standard primitive data types, if-statements, while loops, for loops, exceptions, functions, anonymous functions, closures, static typing, packages, and structures.

A simple binary search program is shown in Figure 4-1 and a linked list data structure is shown in Figure 4-2. These examples not only introduce many of the common language constructs but also highlight some of the the naming and coding conventions as well. There are some unique aspects to the Silo in the code examples that are worth mentioning.

1. The binary search example makes uses of a y-combinator on line 16 in conjunction with the anonymous function. This is necessary because the anonymous function is recursive and the variable that it is assigned to is not bound at the time the function is defined. In many languages with closures (e.g. Javascript) this would not be necessary because closures are mutable and as long as the variable is bound by the time the function is *called* (as opposed to the time by which the function is *defined*) it will work as intended. However, in Silo, closures are immutable and thus the reference to the function must be explicitly passed in. This step is only necessary for recursive (or mutually recursive) *anonymous* functions. Function defined with the `func` statement are forward declared automatically and thus their identifier is statically known to the compiler and can be used.

2. The implementation of the linked list may seem strange to many Java or C programmers. This is because types in Silo are immutable by default. Thus the `insert` function has to copy the list's "prefix" when inserting a new element, hence the perhaps unusual implementation.

3. The linked-list program makes use of Silo's pipe operator on line 30. The pipe operator takes the output of the left expression and inserts it as the first argument of the right expression. Thus, `a | b()` is the same as `b(a)`. Silo is not an object-oriented language and thus the common practice of "chaining" methods together is not possible because there is no "receiver". Instead, developers can use the pipe operator to achieve similar results in a more generalizable manner.

```java
public class Account {
    private String id;
    private double balance;
    public Account(String id, double balance) {
        this.id = id;
        this.balance = balance;
    }
    public transfer(Account a, int amount) {
        a.withdraw(amount);
        this.deposit(amount);
    }
    // Other methods ...
}
```

Figure 4-3: A simple banking application in Java. Java, like many languages, encourages memory side effects and often updates method parameters "in place".

4. The linked-list example demonstrates important Silo naming conventions regarding packages. Notice how the package names are used to make code more readable in instance like `linkedlist.create(...)`. Additionally, notice how data types are Pascal cased (`LinkedList`) whereas packages and camel cased.

4.3.2 Immutability

Silo balances imperative and functional programming by encouraging and simplifying the idiomatic use of immutable value types (like functional programming) but also providing developers the ability to use local variables (like imperative programming).

Imperative programs encourage liberal use of side effects and mutable shared state, which is difficult to coordinate in a distributed system and thus makes it hard to achieve location transparency. As an example, consider a simple banking program that allows money to be deposited and withdrawn from accounts. The most straight forward way of modeling this program in an object-oriented language like Java is to create an `Account` class with the relevant methods as shown in Figure 4-3. As can be seen, the methods update the object "in-place". Now imagine that we need to change this program to run in a distributed environment and that an `Account` object is sent over a network from Machine A to Machine B. Machine B performs a deposit

on the account but now we have a problem: this change is not visible by Machine A since both machines have disconnected memory and the entire program and API would need to be re-architected, perhaps in a client-server model. Thus, in terms of location transparency, the imperative programming model failed us. It encouraged us to pursue a design that does not scale as we run on multiple machines. This challenge is what leads many developers to advocate the use of functional programming. Unlike imperative programming, functional programming encourages the use of value types and little to no shared state. Thus, the problem identified above would not be possible since the "functional" way of solving the problem would be much easier to distribute from the start. While functional programming certainly has its benefits, abandoning imperative programming completely seems a bit drastic.

One way of thinking about side effects is considering the visibility of the effect. Local side effects (e.g. to local variables on the stack) is not a problem because no one else can see it. A function can change local variables at will without impacting location transparency. Furthermore, shared state (e.g. data on the heap) is also not bad if that data never changes and is immutable. This is the balance that Silo makes. All data types in Silo are immutable. Passing an argument to a function assumes pass-by-value semantics and changes to the argument in a function are not visible outside the function. If function wants to mutate an argument it must return an updated "copy" of that argument. However, inside a function, the program is free to use and mutate local variables as usual. Additionally, Silo provides syntactic capabilities to simplify manipulating local variables in a way that retains the feel of an imperative program as shown in Figure 4-4.

To mitigate the performance overhead of using immutable types, Silo performs the following optimizations:

- **Reference Counting** Values are not copied until they are mutated. Whenever the value is assigned to a new identifier, its reference count is incremented. If a value that is mutated has a reference count less than 1, then it is not copied and can be safely updated in place. Reference counting does incur a runtime performance penalty but it does not seem to be significant in practice (see

```
type(Game {
    title : String
    platform : String
})

a : Game = Game("Sonic", "Genesis")
b : Game = a

// Field assignment is possible ...
b.title = "Tetris"
b.platform = "TI-83"

// ... but implicitly creates a copy ...
println(a)      // Game("Sonic", "Genesis")
println(b)      // Game("Tetris", "TI-83")

// Data structures are also immutable
// Changes a vectors requires re-assigning it
x : Vector = vector.create(1, 2, 3)

// This prints [1, 2, 3]
vector.push(x, 4)
println(x)

// This prints [1, 2, 3, 4]
x = vector.push(x, 4)
println(x)

// Silo provides the pipe operator to make it easier to
// work with immutable types that are passed into functions
x | vector.get(1)       // 2
x |= vector.push(5)     // [1, 2, 3, 4, 5]
```

Figure 4-4: By default, all values in Silo are immutable. Silo includes syntax that makes manipulated immutable types easier.

Section 4.6). Nevertheless, I look forward to the inclusion of values types in the JVM (currently planned for Java 9) which allow the Silo compiler more opportunities for optimization [56].

- **Persistent Data Structures** Silo's implementation of common data structures like vectors and maps are persistent (see Figure 4-4). They minimize the amount of copying and provide all essential operations in the same time complexity as their non-persistent counterparts (e.g. access a map takes constant time) [33, 14].

In cases where it is appropriate and necessary, Silo does provided mutable shared state in the form of actors. Actors are discussed in the next section but essentially encapsulate shared and mutable state in a manner that does not violate location transparency and encourages designs that are inherently distributable. Towards this end, it is important to emphasize that *all* Silo types are immutable, including closures and fibers (discussed later, fibers are cooperatively scheduled threads). This means that a program can pause a thread and then resume it multiple times from the same point. Taking this even further, a system could start running a thread on one machine, pause it, serialize it, send it over the network to another machine, and continue executing there. This opens up exciting and interesting possibilities.

4.3.3 Polymorphism

Silo provides polymorphism in the form of a feature called "traits", which are similar to Java interfaces. A "trait" allows developers to describe a set of methods that need to be implemented. Unlike Java interfaces (prior to Java 8), traits can have default implementations of methods. An example of a trait is shown below:

```
package(json)
trait(Json {
    stringify(this => String)
})
```

A type implements a trait during its declaration as shown below:

```
type(Car {
    make : String
    model : String
    year : int
} json.Json {
    stringify(this => String) {
        // The Map class also implements the JSON trait, so we
        // are going to re-use its json stringify implementation
        json.stringify(map.create(
            "make", this.make
            "model", this.model
            "year", this.year
        ))
    }
})
```

Trait methods are called just like a function; however, they are dispatched based on the first argument to the function. Unlike traditional object-oriented programming languages, traits explicitly require the receiver or "this" to be the first argument to the trait method. Also, it is important to note that unlike classes in Java and C#, traits do not form a namespace in Silo. Their methods are "mixed in" to the package in which the trait is defined. This was done in the spirit of keeping language features orthogonal. Silo already has support for packages so there was no need for traits to serve the same purpose. An example is shown below:

```
// Example Usage:
car : Car = Car("Ford", "Model T", 1908)
println(json.stringify(car))
```

There are two nice properties of a trait. First, it allows Silo to achieve polymorphism without having to rely on heavy-weight features like objects or sub-typing.

Second, unlike many single-inheritance and single-dispatch object-oriented languages, traits do not pollute a value's namespace. For example, the `Person` type can implement a `stringify` method for the `Json` trait as well as a `stringify` method for the `Xml` trait without those method names clashing. Moreover, it is able to do this in a way that avoids multiple inheritance and the diamond problem [17].

4.3.4 Macros

Silo supports and makes heavy use of macros. Unlike C/C++, macros in Silo are hygienic and have full access to the programs syntax tree (see Chapter 3 for more details). A simple example macro is shown below:

```
// A for-loop...
for(i : int = 0; i < 5; i = i + 1 {
    println(i)
})

// ... is expanded into this
i : int = 0
while(i < 5 {
    println(i)
    i = i + 1
})

// ... which is expanded into this
i : int = 0
loop(
    branch(i < 5 {
        println(i)
        i = i + 1
    } {
```

```
        break
    })
)
```

Macros provide an elegant approach to extensibility in a manner that does not require mutable closures. For example, consider the following `transaction` method in Ruby.

```
# Defining a transaction method
def transaction(&block)
    begin
        start_transaction()
        block.call()
        commit()
    rescue Error => e
        rollback()
    end
end

# Using the method
transaction do
    ...
end
```

This transaction method is versatile, reusable, and higher-order method that accepts a Ruby `block` (the code marked by `do ... end`, which is a closure) and invokes it in between calls to `begin` and `commit`. Thus, it ensures that user-specified code is executed correctly while also providing an elegant syntactic representation. Moreover, note how the block can reference and use variables within lexical scope. Silo cannot implement this functionality with its higher-order functions because closures are immutable and changes to lexical scoped variables will not be preserved.

However, Silo's macros allow developers to achieve similar results without sacrificing immutability by creating a transaction macro.

```
transform(transaction(body : Node) {
    try({
        startTransaction()
        body   // User-provided code is inserted here
        commit()
    } catch(e : Exception) {
        rollback()
    })
})

transaction({
    ...
})
```

4.3.5 Java Interoperability

Silo compiles to Java Virtual Machine bytecode, can run on any standard JVM and take advantage of the JVM's high performance JIT compiler, garbage collector, and large ecosystem of libraries and tools. This is an approach taken by many other new languages like Clojure, JRuby, Scala, and Groovy [46, 79, 80, 109]. Furthermore, interoperating with Java in Silo is straight forward.

Silo is always compiled to JVM bytecode and, unlike JRuby, does not have an intermediate "interpreter" mode. However, the Silo runtime and tool chain supports both on-the-fly compilation (source files are compiled in memory and immediately executed) as well as ahead-of-time (source files are compiled to JVM "class" files and written to disk). On-the-fly compilation is better suited for scripting tasks whereas ahead-of-time compilation is preferred for deploying large applications where the compilation time is significant or for times in which developers do not wish to disclose

the underlying source code.

While Silo's use of Java is useful, all of the standard libraries use Silo-specific APIs and are designed in a manner that is Java-agnostic. This means that Silo, architecturally, could easily target platforms like LLVM, CLR, and others provided that the compiler and runtime are ported as well. While this may seem daunting, the reality is that most of the compiler and runtime is actually implemented in Silo itself, which may make porting simpler. That said, this approach would limit the ability to interoperate with other Silo libraries that depend on platform specific features; for example, an application that uses the Java Servlet API (instead of Silo's HTTP API) cannot run on a version of Silo that targets the CLR.

Additionally, Java code called through Silo is not subject to the Silo's immutability properties. For example, instead of using Silo's built in persistent vector, a developer could use Java's mutable ArrayList class instead. Using Java interoperability to get around Silo immutability conventions is considered unidiomatic and bad form. Developers who choose to subvert Silo's conventions do so at their own risk, just like users of functional languages like Haskel give up many guarantees when using a foreign function interface.

Calling Java constructors and instantiating an object does not require a `new` keyword; rather, types are "called" directly:

```
alias(ArrayList, java.util.ArrayList)

data : ArrayList = ArrayList()
num : java.math.BigDecimal = java.math.BigDecimal(42)
```

Note that the `ArrayList` did not need to be fully qualified like `BigDecimal` because the ArrayList identifier was aliased to avoid constant repetition.

Accessing static fields and calling static methods is similarly straight forward. In fact, the syntax is identical to Java:

```
// Calling a static method
num : int = Math.min(0, Math.PI)
```

```
// Reading a static field
separator : String = java.io.File.pathSeparator

// Writing a static field
ExampleUserClass.defaultName = "Unknown User"
```

Calling instance methods (both virtual methods and interface methods) in Silo is slightly different than Java and uses the # operator. This reason for this is two fold. First, it makes it very clear which calls are instance methods and which are static methods. Second, it discourages developers from relying on Java classes in favor of Silo's traits. That said, the syntax is fairly straight forward:

```
// Simple method call
System.out#println("Hello, World!")

// Interface method call
map : Map = HashMap()
map#put("Hello", "World")

// Method chaining
"Hello, World!"#toLowerCase()#substring(0, 5)#equals("hello")
```

Silo also supports Java arrays, although the syntax is somewhat awkward. Developers are encouraged to use Silo builtin data structures (vectors and maps) but arrays can be used for performance critical code and for interoperability:

```
// Create an empty array
fib : array(int) = arraynew(int, 5)

// Read and write an array
fib(0) = 0; fib(1) = 1
```

```
fib(2) = fib(0) + fib(1)

// Get length of an array
for(i : int = 0; i < arraylength(fib); i = i + 1 {
    println(i)
})

// Multi-Dimensional Array. A 3 x 3 int matrix
ticTacToe : array(array(String)) = arraynew(String, 3, 3)
ticTacToe(1)(1) = "x"
```

Creating a new Java class in Silo is done with the `defineclass` special form. This special form is intended to be wrapped by macros that present a more user-friendly interface. As a result, usage of `defineclass` may seem unnecessarily low level and verbose:

```
defineclass(
    name(Car)
    field(
        name(name)
        type(String)
        modifiers(private)
        default("")
    )
    constructor(
        inputs(name : String)
        modifiers(public) {
            this.name = name
        }
    )
    method(
```

```
        name(getName)
        outputs(String)
        modifiers(public) {
            return(this.name)
        }
    )
)
```

Silo can also be embedded into existing Java programs and called from Java. To do so, the host Java program needs to create an instance of the `silo.lang.Runtime` class. To call a function, developers call the `spawn` function and either pass in the fully qualified name of a function or an instance of the `Function` class (which serves as a function pointer in Silo) along with the function's arguments. The `spawn` method creates and returns an `Actor` object that represents the execution of the function (actors are discussed in depth later in the paper). The output of the function can be retrieved by invoking the `await` method to wait until the Actor finishes. As an example, the following Java code calls the Silo `print` function:

```
Runtime rt = new Runtime();

// Pass a fully qualified name of the function as a String
Actor a = rt.spawn("silo.core.print", "Hello, World!");
a.await();

// Silo functions are compiled as Java classes. A class
// reference can be passed in lieu of a String. A class
// Object in Java is retrieved using the ".class" pseudo property
Actor a = rt.spawn(silo.core.print.class, "Hello, World!");
a.await();
```

Along these lines, note that Silo packages are identical to Java packages. Java

packages can be imported into Silo programs just as easily as Silo packages are imported into Java programs.

4.3.6 Notable Omissions

A major challenge when designing Silo was determining language features that should not be included. Some features were avoided in the name of location transparency but others were also avoided in the name of simplicity and to ensure the orthogonality of the core language feature set. As the language evolves some of these features may be incorporated, but for the time being developers should be aware certain features are not supported:

- **Object Oriented Programming** Silo notably does not have a builtin notion of classes (aside from classes used through Java). There were two reasons for this. First, object-oriented programming is generally valuable to coordinate and encapsulate side effects and mutable state. Since Silo minimizes mutable state there is little benefit for actual objects. Second, in terms of building systems, objects and actors (discussed in the next section) serve similar roles as both represent "active" entities and have "personalities" that model different parts of the system. Since Silo is built around the idea of an actor (for the purposes of location transparency and distribution) it seemed superfluous to also include objects. Furthermore, it unnecessarily complicates the application design process for developers as it forks the programming protocols of the language by forcing developers to answer the question: "should I use an actor or an object?"

- **Sub-Typing and Inheritance** Silo types cannot be inherited or sub-typed. This feature was omitted to avoid overly complex APIs with rigid type hierarchies that are hard to learn. Using composition and traits addresses the most common use cases for sub typing. Developers are always able to create custom Java classes and use Silo like they would use Java. However, this is considered unidiomatic and bad form.

- **Generics** Silo currently has no support for generics or templates. Unlike the other omitted features, I would very much like to see Silo incorporate a more powerful type system that employs generics. However, subjectively, I have not seen an implementation of generics that was not overly complex or forced developers to have a deep understanding and appreciation of type systems. Implementing generics, therefore, is left for a future version.

4.4 Concurrency and Communication

4.4.1 Actors and Fibers

Actor are foundational constructs in Silo that are unified throughout the language and are used to model concurrent and distributed programming.

In many ways, actors can be thought of as a thread or an operating system process as they represent an independent and autonomous unit of execution. To create a new actor, a Silo program will call the `spawn` function and provide a reference to some function where that actor should start executing. The `spawn` function will return an `Address` to the new actor just like the `fork` Unix system call returns a `pid` (process id). However, unlike Unix processes, actors provide a built-in mechanism for communication via message passing. An actor can send a message to an `Address` using the `send` function. The message will be sent asynchronously and the `send` function will return immediately. The message will be enqueued on a mailbox belonging to the receiving actor, who can retrieve the message at anytime using the `read` function. Any Silo value can be sent as a message from integers to maps. A simple "ping-pong" program using actors is shown in Figure 4-5. The program will simply send the same message back and forth between two actors until one of the actors sends a "stop" message. Notice how message passing is one-way and asynchronous — the actors must explicitly block and wait for a reply message. This is different from the call-and-return semantics of functions which are two way (a return value is given) and synchronous (the calling code does not continue until the function is complete).

```
func(pong {
    pinger : Address = actor.read()
    while(true {
        m = actor.read()
        if(m | instanceof(Map) {
            s : Address = map.get(m, "sender")
            actor.send(s, "Pong")
        })
    })
})

func(ping(ponger: Address, main : Address) {
    repeat(10 {
        actor.send(ponger, map.create(
            "sender" actor.self()
            "payload" "Ping"
        ))
        actor.read()
    })
    actor.send(main, "Done!")
})

// Main starting point...
actor.spawn(ping
    actor.spawn(pong)
    actor.self()
)

actor.read()
```

Figure 4-5: Ping-pong program using actors.

However, unlike threads, actors in Silo are incredibly light weight. They require less than half a kilobyte of memory overhead (compared to threads which have many megabytes of overhead) and do not put pressure on the operating system kernel scheduler. Developers are free to `spawn` as many actors as they desire. In fact, it is easily possible to create millions of actors on a single machine. This is because actors in Silo are not directly backed by threads; rather, they execute on a construct called a fiber (also known as a coroutine) [104]. Fibers, unlike threads, do not exhibit pre-emptive multitasking. Instead, they are cooperatively scheduled and must choose to give up the CPU before another fiber can run. Since fibers run at the language-level, they can generally context switch faster and consume far less memory than threads, which require large statically-allocated stacks and a trip through the kernel to context switch. When a fiber is scheduled to execute, it will be placed in a queue and executed by the first available system thread. Thus, it is not guaranteed that an actor will always be executed by the same thread. In fact, it is extremely likely for an actor to execute on multiple threads over the course of its lifetime.

The question still remains, "when do fibers yield?". Fibers yield when a program attempts to retrieve a message from its actor's mailbox and the mailbox is empty. Instead of waiting idly until a message arrives, the fiber will cooperatively yield and allow another fiber to execute. When a message is enqueued onto an actor's mailbox, that actor's fiber will be scheduled to resume. This allows a large number of threads to execute on a small number of threads.

4.4.2 Message Passing Semantics

Silo's actor API consists of following operations: `spawn`, `send`, `read`, `peek`, `skip`, `count`, `self`, and `yield`. This API is quite different from may other actor implementations (for example, Erlang) and aligns well with an imperative programming model. The execution behavior of actors is shown in Figure 4-6.

An actor encapsulates the following information:

- Address: The address of the actor. The address must be unique.

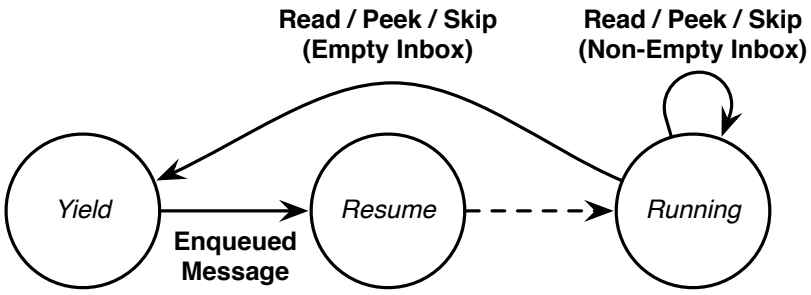

Figure 4-6: The execution semantics of Silo's actors.

- Mailbox: A queue of messages sent to the actor's address. This queue is unbounded. This is private and can only be accessed by the owning actor. The one obvious exception is that other actors can place messages at the end of a mailbox owned by another actor when sending messages.
- Skipped Queue: A queue of messages that have been "skipped" by the actor. This queue is unbounded. This is private and can only be accessed by the owning actor.

A description of the actor API is shown in Figure 4-7 and is discussed below. With the exception of `spawn` and `send`, the API can only be used to access and manipulate the mailbox belonging to the current actor. In other words, an actor cannot access the mailbox or skipped queue of another actor.

- `spawn(start : Function, args : Vector => Address)` will create a new actor and return its address. The newly created actor will begin execution in the specified function with the given arguments.
- `send(address : Address, message : Object => boolean)` will attempt to deliver a message to the actor at the specified address. The message will be placed at the end of the target actor's mailbox. Actors can send messages to themselves. The `send` function is asynchronous and returns immediately. Programs must explicitly wait for a reply message to implement call-and-return semantics. The `boolean` return value indicates if a *known* error took place. However, sending a message is *never* guaranteed. Thus, just because `send`

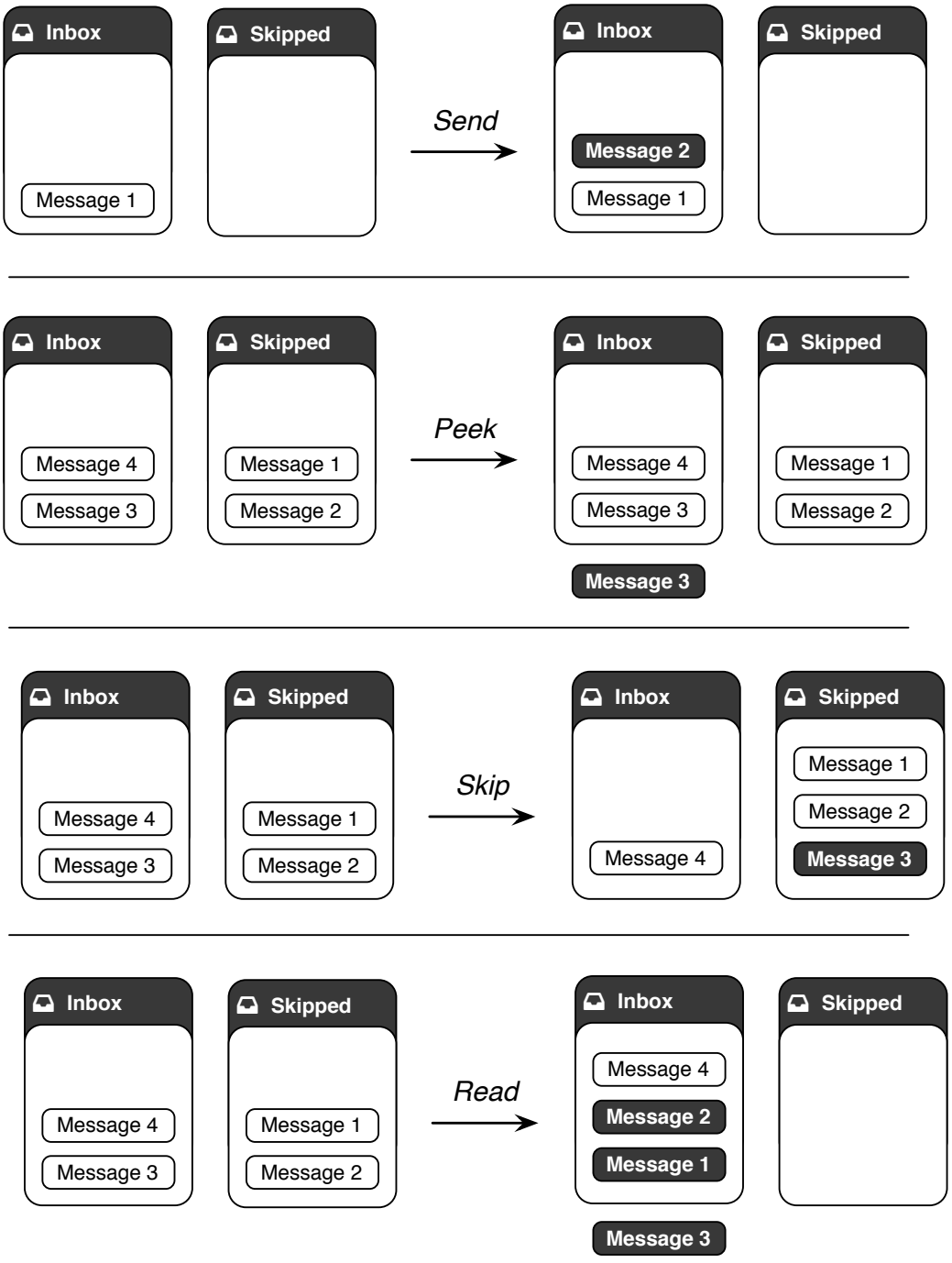

Figure 4-7: A visual illustration of the actor API.

returns `true` does not mean that no error took place. There are still numerous opportunities for errors: the message may be lost during delivery, the actor may crash before the message is delivered, the actor may be out of memory and not be able to accept the message onto its inbox, the actor may receive the message but crash before it is able to process it, or the actor may simply choose the completely ignore it, etc. Developers need to design protocols that account for this behavior, for example, incorporating timeouts. The nice thing about this is that once established, these protocols will work locally and also remotely.

- `read(void => Object)` will dequeue a message from mailbox of the current actor. If the mailbox is empty, this function will block and cause the underlying fiber to yield control to another fiber. All skipped messages will be returned to the mailbox in their original order.

- `peek(void => Object)` will access and return the next message from the mailbox. However, the message will not be removed from the mailbox. If the mailbox is empty this function will block and cause the underlying fiber to yield control to another fiber.

- `skip(void)` will skip and return the next message from the mailbox. The message will be removed from the mailbox and placed in the skipped queue. If the mailbox is empty this function will block and cause the underlying fiber to yield control to another fiber. The `skip` function can be used in conjunction with the `peek` function to implement a "selective-receive" functionality where an actor temporarily ignores messages until the desired message appears. This is particularly useful for abstracting the actor API in a composable manner.

- `count(void => int)` will return the number of messages in the current actor's mailbox. This is useful for determining if another actor function will cause execution to block.

- `self(void)` will return the address of the calling actor.

- `yield(void)` will force the current actor to yield and give another actor a chance to execute. An actor that calls `yield` will immediately be scheduled to execute. This is useful for functions that perform long running computations and want to be a "good citizens" and allow other actors a chance to execute.

4.4.3 Abstracting the Actor API and Selective Receive

The actor API is often too low-level for most day-to-day programming tasks. Message-based protocols are often messy, verbose, error-prone, and hard to document. A cleaner approach is to abstract the low-level message passing details of a protocol inside a function.

Let us consider the implementation of the `sleep` function. The sleep message protocol could not be more straight forward: an actor sends a message to the system and the system sends back a reply message after a certain amount of time. A naive (and wrong) implementation could look like this:

```
func(sleep(ms : long) {
    actor.send(silo.core.system, SleepMessage(ms))
    actor.read()
})
```

The problem with this code is that it is susceptible to false positive. What if another message arrives before the timeout? To correct this, the sleep function should wait for a particular message before returning:

```
func(sleep(ms : long) {
    actor.send(silo.core.system, SleepMessage(ms))
    while(true {
        message : Object = actor.read()
        if(instanceof(message, WakeUpMessage) {
            return
        })
```

```
    })
})
```

This is better, but there is a subtle bug. Consider the following code snippet:

```
actor.send(silo.core.system, SleepMessage(0))
sleep(50)
```

The sleep function will receive a spurious `WakeUpMessage` and likely not wait the full 50ms. To avoid this, the sleep protocol is enhanced with a unique identifier. Clients send a `SleepMessage` with a timeout duration along with a string identifier. When the system replies, it will include this identifier so that the client can distinguish messages:

```
func(sleep(ms : long) {
    // Generate a unique identifier
    id : String = uuid.create()
    actor.send(silo.core.system, SleepMessage(id, ms))

    while(true {
        message : Object = actor.read()
        if(instanceof(message, WakeUpMessage) {
            if(checkcast(message, WakeUpMessage).id == id {
                return
            })
        })
    })
})
```

This is an improvement, but there is one last issue. The sleep function currently throws away all the messages that it ignores. This is really bad because some of those messages may have been important and it means that the `sleep` function is not

composable. To avoid this issue, Silo programs often use a technique called "selective receive" as demonstrated in the final (working) version of sleep below:

```
func(sleep(ms : long) {
    ...
    while(true {
        // Take a look at the message but do not read it
        message : Object = actor.peek()
        if(instanceof(message, WakeUpMessage) {
            if(checkcast(message, WakeUpMessage).id == id {
                // We found the message we want.
                // Read it so it is removed from the mailbox
                actor.read()
                return
            })
        })

        // If this is not the message we want, skip it. It will be
        // placed back on the mailbox when we finally do "read" a message
        actor.skip()
    })
})
```

This sleep function is now correct, composable, and safe to use just like any other function. The ability to skip messages means that messages can be multiplexed over a single mailbox and that a single actor can concurrently communicate with multiple entities.

Since selective receiving is so common, Silo includes the `await` macro that makes selective receive more aesthetically pleasing. The await macro allows developers to specify which messages they are interested in using pattern matching. Messages that do not match are implicitly skipped. Example usage of `await` is shown below.

```
func(sleep(ms : long) {
    id : String = uuid.create()
    actor.send(silo.core.system, SleepMessage(id, ms))

    actor.await(
        WakeUpMessage(escape(id)) {
            return
        }
    )
})
```

The await macro allows developers to quickly match incoming messages easily. In the above example, the program is waiting for a message of type `WakeUpMessage` in which the first field is the same as the local variable, `id`. In many ways, this pattern matching is similar to product types found in many other functional programming languages. However, the pattern matching in Silo cannot implement all logic and sometimes developer will have to fall back to lower-level primitives. Nevertheless, for most use cases, the await macro greatly simplifies application source code.

4.4.4 Programming with Actors

Actors in Silo are the only way to model shared mutable state and the only way to share state between actors is by passing messages (recall that all values in Silo are immutable). For example, a "registry" actor is shown in Figure 4-8. This actor creates a local hash map and then enters into a loop where it processes incoming messages. Others actors can get access to this hash map by sending certain messages to the registry. Notice how the registry actor models a server. It encapsulates a resource (the hash map) and listens for requests to access that resource. However, the program does not need to worry about low level concepts like sockets, ports, and networking protocols. Also, notice how the registry requires messages to contain a return address to the actor that sent the message. The sending actor can access its address by

```
// Message Types
type(GetRequest {
    clientToken : String
    sender : Address
    key : String
})
type(PutRequest {
    clientToken : String
    sender : Address
    key : String
    value : Object
})
type(Resp {
    clientToken : String
    value : Object
})

// Actor Service Body
func(registry {
    data : Map = map.create()
    while(true {
        actor.await(message
            GetRequest {
                r : Resp = Resp(message.clientToken, map.get(data, message.key))
                actor.send(message.sender, r)
            } PutRequest {
                data = map.put(data, message.key, message.value)
                r : Resp = Resp(message.clientToken, message.value)
                actor.send(message.sender, r)
            } else {
                // Ignore
            }
        )
    })
})

// Client Library Function
func(getValue(registry : Address, key : String => Object) {
    token : String = uuid.create()
    actor.send(registry, GetRequest(token, actor.self(), key))
    actor.await(response
        // We only want to match the token. We don't care about the value
        Resp(escape(token), _) {
            return(response.value)
        }
    )
})
```

Figure 4-8: Modeling shared mutable state in Silo.

calling `actor.self`. Without the sender's address, the registry would not know to which address the reply should be sent. Programming with actors forces developers to think about how their program would operating in a distributed environment. More advanced examples of actor use cases are shown in Section 4.7.

Beyond mutable state, it is also important to realize that *all* side effects must be coordinated through actors. This includes I/O and system calls as well. For example, to read a file, an actor will send a request to a special "system" actor who will read the file from disk and send a reply message to the calling actor with the file contents. Another example is the `sleep` function. To wait for a certain amount of time, an actor will send a message to the same "system" actor and ask it to send it a message after a certain amount of time has passed. The actor will then block and wait until that message arrives.

Lastly, actors are not only useful for modeling logical services. They can also be used as a general construct for concurrency. For example, downloading a list of images can be easily done in parallel using actors.

```
func(downloadAll(urls : Vector) {
    uniqueid : String = uuid.create()
    address : Address = actor.self()
    foreach(url in urls {
        actor.spawn(fn({
            content : String = http.download(url)
            actor.send(address, map.create(
                "id" uniqueid
                "content" content
            ))
        }))
    })
    // Wait for all messages to be downloaded
    repeat(vector.length(urls) {
        actor.await(Map("id" escape(uniqueid)))
```

```
    })

    // Continue
    ...
})
```

4.4.5 Concurrency Models

Silo supports a variety of different concurrency models: hybrid-preemptive (actors), preemptive (pinning), and cooperative (fibers).

An issue with the default concurrency behavior (hybrid-preemptive) of actors is that fairness is not guaranteed and a new actor is given a chance to run only when another actor blocks on the actor API. As such, an actor performing a long-running computation (for example, matrix manipulation or calling a Java method) could accidentally (or maliciously) starve others.

To avoid this, Silo supports "thread pinning". Normally actors are multiplexed across a small number of shared threads. With thread pinning an actor is given its own dedicate thread that will be preemptively scheduled by the operating system. This will ensure that long running operations do not block the system from making progress and also give an actor a higher priority to execute.

Silo also allows fibers to be used on their own. Fibers are first class citizens in Silo and an actor could create a fiber and run it. The fiber will continue to execute until it yields or returns. A running fiber can even use the actor API to access the mailbox of the enclosing actor. If the running fiber blocks, it will block the entire actor. Thus, fibers can be used to model concurrent execution flows within a single actor. Moreover, since fibers are first class values, they are also immutable. This means that they can be resumed multiple times from the same starting point and also can be serialized and saved to disk to be resumed later. An example of using fibers is shown below.

```
// Creating a fiber from an anonymous function
```

```
adder : Fiber = fiber.create(fn(a : int
    while(true {
        // yield is a two-way street. It will yield a value to the
        // calling code but also allow the calling code to pass in
        // a new value
        increment : int = fiber.yield(a)
        a = a + increment
    })
))

// Fibers are immutable. This code prints 5
fiber.resume(adder, 5)
fiber.resume(adder, 5)
println(adder.value)

// This code prints 10
adder = fiber.resume(adder, 5)
adder = fiber.resume(adder, 5)
println(adder.value)

// This code is more succinct
adder |= fiber.resume(5)
adder |= fiber.resume(5)
println(adder.value)
```

4.4.6 Polymorphic Delegation

An important promise of Silo's actor implementation is that it enables applications to achieve location transparency. While it is true that an actor-based system can *architecturally* be split across multiple machines, there are other practical considerations that are also important. Notably, how are messages sent remotely? Do they

use a custom TCP protocol or a higher-level protocol like HTTP? These decisions are best left to developers since each system has its own unique requirements and use cases. This is especially true with the trend toward cloud hosting environments where development teams may choose to leverage proprietary messaging solutions that are scalable and cost effective [51, 6].

Instead of forcing developers to use a particular technology for remote message passing, Silo includes a feature called polymorphic delegation that allows user-level code to customize how messages are passed. As mentioned before, each actor is identified by an `Address` value. However, the `Address` type in Silo is not concrete data type; rather, it is a polymorphic type. The `Address` trait shown below.

```
trait(Address {
    method(actualAddress(this, currentPayload) => LocalAddress)
    method(actualPayload(this, currentPayload) => Object)
})
```

When `actor.send` is called it will invoke `actualAddress` and `actualPayload` on the polymorphic `Address` value. This allows user-defined code to "route" the message's "actual payload" through a delegate actor (living at the "actual address") that will send the message to its final destination. The call to `actualPayload` is useful for wrapping the actual message with other useful information that the delegate may need, for example the desired recipient. The delegate actor is free to perform any arbitrary computation to deliver the message to the intended destination. It can even choose to ignore the message as a form of congestion control.

Figure 4-9 illustrates how delegation can be used to send a message to an actor running on a remote system. In this figure, the message is sent to an RPC actor which talks to another RPC actor running on another machine. The exact transport mechanism is up to the RPC service — it can use TCP, HTTP, Amazon's SQS, Apple's XPC, or even carrier pigeons.

Note that the `actualAddress` method must return a value of type `LocalAddress`. A `LocalAddress` is a concrete type that implements the `Address` trait and is used

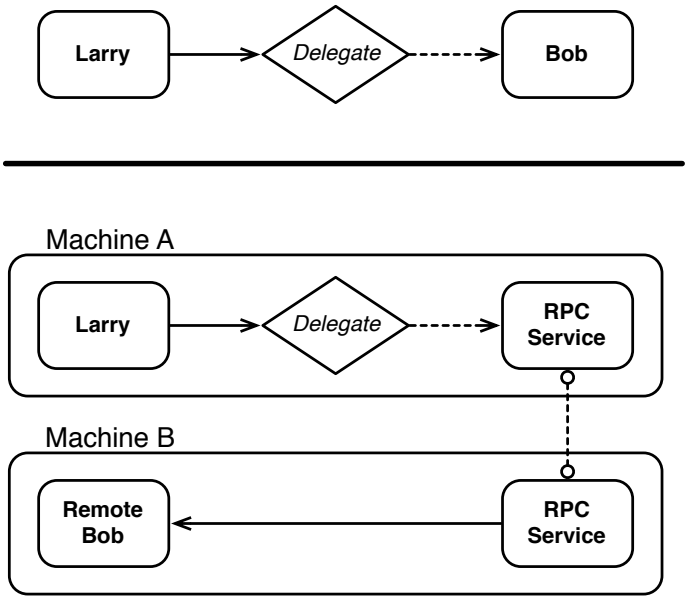

Figure 4-9: Polymorphic delegation allows user code to customize how messages are delivered. The top example does nothing and simply allows the message to be sent to another local actor. The bottom example routes the message through an RPC service to an actor running on another machine.

to reference an actor running in the current local process. It is a built into the Silo runtime and is what is returned by "normal" calls to `spawn`. Calling `actualAddress` or `actualPayload` on a `LocalAddress` will simply yield the same value.

An example program that demonstrates polymorphic delegation for remote message passing is shown below:

```
// The cluster package is a built-in remote message passing library.
// It uses a custom TCP protocol built. Developers are free to create
// their own alternatives.
c = cluster.join(map.create(
    "port" env("port")
    "certificate" ...
    "known-hosts" ...
))

// Spawns a key-value registry actor on the machine running at "127.0.0.1:8002"
```

```
// "a" is actually an instance of ClusterAddress which will route message
// through the cluster ("c") for delivery.
r : Address = cluster.spawn(c, ":8002", registry.start)

// The registry client library can still be used unchanged even through the
// registry is running on a different machine. The same messaging protocol can
// be used as the client library's implementation is location agnostic.
registry.set(r, "hello", "world")
println(registry.get(r, "hello"))
```

The example above also communicates the importance of a core convention in Silo: library code should avoid calling `spawn`. For example, suppose the registry library provided a function called `registry.spawn` which uses `silo.core.spawn` to create a new registry actor and returns its `Address`. The reason why this is bad is it prevents the developer from specifying where that actor should run. Instead of using `silo.core.spawn` the developer may have wanted to use `cluster.spawn` instead. Thus, the library made a decision that should have been left to the developer. The convention in Silo is to create a function called "start" which is where newly created actors should begin. It is up to the developer to determine how this function should be called. If the library does need to call `spawn` internally (for example, perhaps it needs to return a more complex value rather than just an `Address`) it should provide some mechanism by which developers can customize which "`spawn`" is called. A good way of doing this is to accept a function as an argument or use a macro of some sort. An example is shown below:

```
// Tells the registry library to spawn a new registry
// using the default spawn function
registry.spawn(silo.core.spawn)

// Tells the registry library to spawn a new registry
// using the cluster API by passing in an anonymous function
```

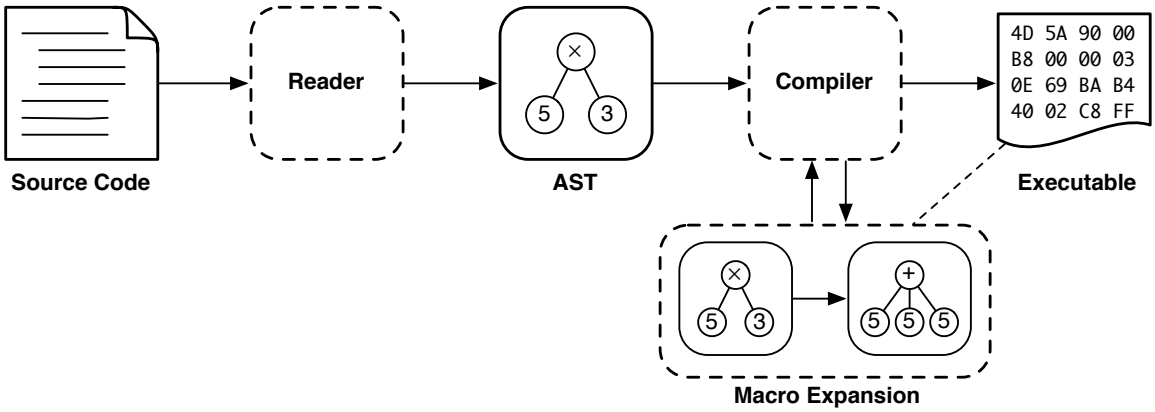

Figure 4-10: The Silo compilation pipeline.

```
registry.spawn(fn(a : Function, b : Vector => Address {
    cluster.spawn(c, ":8002", a, b)
}))
```

Lastly, polymorphic delegation is useful for more than just remote message passing and can be used in many scenarios:

1. Test cases where messages are artificially delayed or intentionally dropped
2. Implementing transparent encryption of messages
3. Client-side load balancing
4. Congestion control
5. Logging and debugging

4.5 Implementation

4.5.1 Compilation Pipeline

Silo compiles to JVM bytecode. An overview of Silo's compilation pipeline is show in Figure 4-10. Of note, Silo's compiler performs a "macro-expansion" step where it invokes user-specified macros to transform syntax into a lower-level representation before the core compiler processes it. The lowest-level constructs in Silo are called

```
access          declare         package
alias           function        quotecontext
array           import          return
arraylength     instanceof      throw
arraynew        invoke          try
block           invokevirtual   uniquesymbol
branch          loop            <math-ops>
break           monitorenter    <relational-ops>
checkcast       monitorexit     <literals>
```

Figure 4-11: The list of special forms in Silo.

"special forms". Special forms provide access to all functionality that is not possible by other means (functions, other macros, etc.). The number of special forms in Silo is quite limited and shown in Figure 4-11. Most of Silo's implementation is written in Silo itself as macros or as bootstrapped runtime APIs. This is an important property since it keeps the core compiler small, simple, and easy to manage. Moreover, porting the compiler to another platform (for example, Microsoft's CLR, LLVM, or x86) only requires re-implementing a handful of special forms.

A major challenge for Silo's compiler is implementing fibers, which, as aforementioned, are used as an alternative to threads by Silo's actor implementation. Implementing fibers on the JVM is particularly tricky since the JVM does not provide any mechanism for pausing and resuming execution (aside from threads). The technical details of fibers are outside the scope of this chapter and are discussed in-depth in Chapter 5.

4.5.2 Runtime Architecture

An overview of Silo's runtime architecture is shown in Figure 4-12. This runtime architecture is provided by the `silo.lang.Runtime` class. Silo's runtime uses two

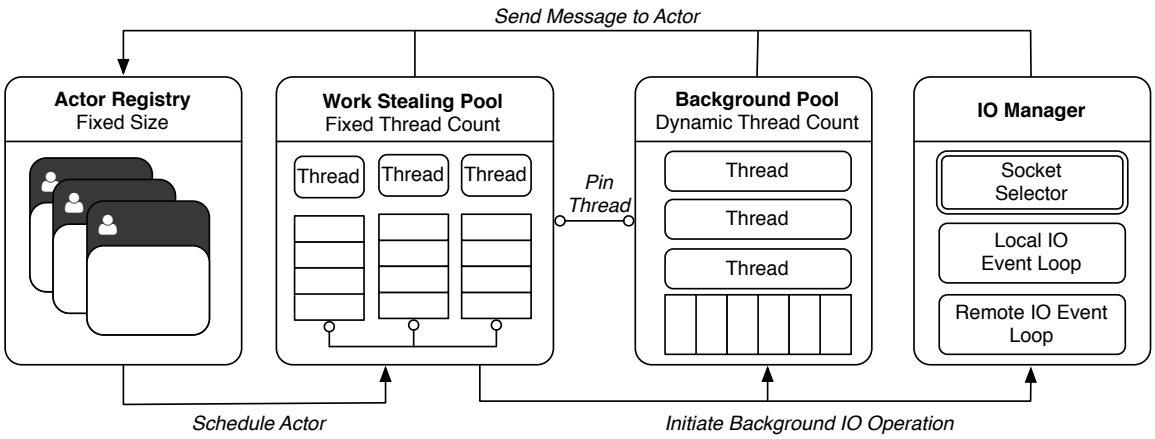

Figure 4-12: Silo's runtime architecture.

thread pools: one for executing actors and the other for executing background tasks. The actor thread pool uses a work-stealing scheduler [82, 16] and results in huge performance improvements for actor-based programs compared to standard thread pools (see Section 5.5). The background thread pool, however, is a standard thread pool that is used (1) to pin an actor to a thread to ensure that it is not swapped out by the runtime when waiting for a message to arrive or (2) to execute long running computations that would block the actor thread-pool. Notably, the background thread pool is also leveraged to run a standard `java.util.Timer` object which is used to implement the Silo `sleep` function (since call `Thread.sleep` would block the work-stealing thread pool).

4.5.3 Other Features

Traits

A trait in Silo is compiled to Java interface as well as an abstract class that implements that interface. The abstract class allows for default implementations of certain trait methods. Furthermore, the method names in the interface are mangled to ensure that no naming conflicts occur. For example, the `stringify` method on the `Json` trait is internally renamed to `silo$trait$Json$stringy`.

Immutable Data Types

Silo data types are compiled as a Java class with private fields. They extend the `ReferenceCounted` class to gain access to the `release` and `retain` methods along with an internal counter. Whenever a value is assigned to a new variable, the `retain` method is invoked. Additionally, whenever a value is passed into a function, it is `retained` before the function call and then `released` after the call. At the end of a function, all local variables are `released`; Silo currently does not perform liveness analysis to minimize the number of variables that are `released` at the end of the function.

Accessing a field (e.g. `user.name`) is compiled to a "getter" method. Mutating a field (e.g. `user.name = "Bob"`) is compiled to a "setter" method. This setter method checks the reference count. If the count is one, the setter method updates the field and then returns the same object (i.e. returns `this`). If the count is not one, the setter clones `this`, sets the field, and returns the new value. Values are clones using Java's `Object.clone` method. This method is a JIT-intrinsic which means that the JVM will use platform-specific capabilities to improve the performance of the copying, for example, by using x86's SSE vectorization instructions. In short, `Object.clone` is quite fast.

Nested mutations use a special "getter" method called "access-for-mutation". For example, `user.friend.name = "Bob"` is compiled to:
`user = user.accessForMutation_friend().mutate_name("Bob")`. Access-for-mutation will check the internal reference count, perform a shallow copy as necessary, and update its reference counter to zero. The reason why the reference count is set to zero (instead of one) is that the reference count will be incremented the moment the value is assigned back to a variable (i.e. the "`user = ...`" in the previous example code). This allows the compiler to mutate nested structures without having to deeply copy the entire hierarchy.

Benchmark (ms)	Java (v1.8)	Silo (v0.1)	JS (V8 v3.8.9)	JRuby (v1.7.10)	Ruby (v2.0)	Erlang (R16B03)
Mandelbrot	23,811	20,354	34,627	125,322	323,141	155,860
Fibonacci	4,238	4,097	16,363	23,416	182,756	43,649
Parse Int	7,712	9,809	12,831	24,768	66,053	12,430
Binary Tree	2,283	3,312	30,104	65,674	357,731	2,557
Vector	12,059	18,944	21,994	17,854	39,484	246,245
Immutable Vector	16,101	1,658	18,159	19,430	67,307	246,245
Dictionary	5,756	6,213	16,378	26,831	121,788	23,756
Immutable Dictionary	48,362	1,144	562,171	92,309	916,200	23,756
Total	**120,322**	**65,531**	**712,627**	**395,604**	**2,074,460**	**754,498**
Total (no immutable)	**55,859**	**62,729**	**132,297**	**283,865**	**1,090,953**	**484,497**

Figure 4-13: Silo's performance compared to Java, Javascript, JRuby, and Ruby.

Persistent Collections

The reference implementation of Silo uses Clojure's implementation of persistent data structures. Clojure is a Lisp-dialect that compiles to JVM bytecode [46].

4.6 Performance Evaluation

To evaluate Silo's performance, I conduct a fairly standard set of benchmarks that stress different computational tasks: execution performance, memory allocation, recursion, etc. I compare Silo's performance to Java, Javascript, Ruby, JRuby, and Erlang. Java was chosen to illustrate an upper bound of the type of performance that Silo *could* hope to achieve (recall that Silo compiles to JVM bytecode). Javascript, Ruby, and JRuby were chosen as they are mature languages that have recently undergone signifiant performance optimizations and are popular choices in the Web application domain. JRuby was included to provide a reference of the type of performance possible with another JVM-based language. Lastly, Erlang was chosen as it is perhaps the most famous actor-based programming language. A description of the benchmarks is shown below.

- **Mandelbrot** Generate a Mandelbrot set. This benchmark stresses basic math performance and nested loops.

- **Fibonacci** Compute progressively larger fibonacci numbers. Dynamic programming must *not* be used (i.e. no memoizing results) and recursive implementations *must* be used (i.e. no iterative algorithms). This stresses the performance of recursion.

- **Parse Int** Parse a string representation of an integer value. This benchmarks stresses basic string processing and is also a common task in practice.

- **Binary Tree** Allocate and then deallocate progressively larger binary trees. This benchmark stresses memory allocation, management, and garbage collection.

- **Vector** Push a large number of strings on a vector and then pop them all off. This benchmark stresses the performance of the standard vector implementation in the language. The standard vector container must be used (i.e. no special hand-rolled implementations). This also includes an immutable version of the test where the implementation must make "copies" of the vector.

- **Dictionary** Assign a large number of key in a dictionary data structure and then remove them. This benchmark stresses the performance of the standard dictionary implementation in the language. The standard dictionary container must be used (i.e. no special hand-rolled implementations). This also includes an immutable version of the test where the implementation must make "copies" of the dictionary.

The raw results are shown in Figure 4-13. The *exact* numbers are of perhaps little interest but there are some interesting observations that can be made.

- Silo reaches the performance of Java and achieves much greater performance than the other contenders. This is particularly notable because Silo provides a rich set of abstractions (actors, coroutines) that are not directly supported by the JVM and could easily negatively impact performance. Nevertheless, Silo's reference implementation is able to achieve good performance thanks to several

compilation optimizations and runtime techniques. Chapter 5 provides more technical details on these optimizations.

- The binary tree, vector, and map benchmarks demonstrate that an imperative language can enforce immutability without sacrificing performance. While the Silo's emphasis on immutability causes it to achieve less performance than Java, the difference is not as drastic as one might think. Silo's reference-counting optimizations and use of persistent data structures seem to perform well and, in particular, are able to far outstrip languages like Ruby and Javascript, both of which use highly optimized mutable hash map implementation written in hand-rolled C. It also bear repeating that in specific instances where performance is critical, Silo developers can always fall back onto mutable data structures in tight loops. Developer must do so, however, at their own risk.

- JRuby's performance is quite different from Silo and illustrates an important point: just because a language compiles to JVM bytecode does not means that it is automatically fast. Language design choices, compiler optimizations, and idioms make a huge difference. Silo's semantics, design, and implementation appear to align well with the capabilities of the JVM.

- Erlang's performance on these CPU-benchmarks also lags behind Silo and Java. This does not come as a surprise since Erlang was designed for message passing and programming telephone switches rather than raw CPU performance. Nevertheless, CPU performance is still important for many applications, especially large service-oriented systems. A comparison between Erlang and Silo for message-passing and I/O performance, which is just as important for service-oriented systems, is shown in Section 5.5.

4.7 Common Patterns

4.7.1 Basic Networking (HTTP, TCP)

At the core of any service-oriented system is networking. Silo's common library consists of high performance support for many protocols including HTTP, TCP, UDP, SMTP, and others. Moreover, Silo's networking stack is integrated into its actor implementation which allows for high-performance non-blocking networking using simple blocking APIs.

Silo's high performance networking is a combination of (1) its scalable actor implementation as well as (2) its ability to access high quality and optimized Java libraries. For example, Silo's HTTP library makes heavy use of Java's `nio` package which includes zero-copy buffers between the kernel and user space.

An example of an HTTP server is shown in the code below.

```
1.  import(silo.net.http)
2.
3.  func(handler(r : connection.Request, c : connection.Connection) {
4.      m : connection.HttpContentMessage = connection.readAll(c)
5.      connection.writeAll(c, 200, null, "Hello, World!")
6.  })
7.
8.  func(simpleServer() {
9.      println("Silo - Running on port 8000")
10.
11.     options : Map = map.create()
12.     options = map.set(options, "port", Integer(8000))
13.
14.     s : server.Server = server.build(handler, options)
15.     server.start(s)
16.
17.     // Block forever
18.     while(true {
19.         actor.read()
20.     })
```

```
21. })
22.
23. simpleServer()
```

4.7.2 Fan-Out and Fan-In

A common task in many servers is to make many requests to back-end services in parallel and wait for them to come back. This is called "fan-out" and "fan-in". The example below is a simple actor service that waits for user account requests and fetches all the information in parallel. This example demonstrates a powerful abstraction in Silo called `Futures`. A `Future` is a construct that allows an actor to spawn another actor to execute code in parallel and provides an API to wait for the return value to be returned back. It is a re-usable construct that allows message-passing code to appear like "call-and-return" code.

```
1.  type(AccountInfoRequest {
2.      id : String
3.      sender : Address
4.  })
5.
6.  type(AccountInfoResponse {
7.      id : String
8.      info : Map
9.      image : String
10.     friends : Vector
11. })
12.
13. func(start() {
14.     // The server will loop forever and wait for message
15.
16.     loop({
17.         actor.await(message
18.             AccountInfoRequest(_) {
19.                 // Fetch the user's account information. Set a 200ms QoS timeout
20.                 info : Future = future.spawn(user.getBasicInfo(message.id), 200)
```

```
21.
22.                     // Fetch the user's profile picture. Set a 200ms QoS timeout
23.                     image : Future =  future.spawn(assets.getProfilePictureUrl(message.id), 200)
24.
25.                     // Fetch the user's friends. Set a 200ms QoS timeout
26.                     friends : Future = future.spawn(social.getFriends(message.id), 200)
27.
28.                     // Wait for the futures to be satisfied
29.                     info |= future.wait()
30.                     image |= future.wait()
31.                     friends |= future.wait()
32.
33.                     actor.send(message.sender, AccountInfoResponse(
34.                         message.id
35.                         info.value
36.                         image.value
37.                         friends.value
38.                     ))
39.                 } else {
40.                     // Ignore message
41.                 }
42.             )
43.         })
44.     })
```

Keep in mind that it is possible for users to create more powerful constructs on top of `Future`; for example, a `doInParallel(...)` macro. However, in this example, I avoid doing this to demonstrate the `Future` API.

Another example of fan-out is when an actor send the same message to many instances of the same back-end service and returns the result from the quickest service. If the services are running on different machines, this avoids having to wait for a strangling machine before returning a result. The code is shown below:

```
1.  // The variables "a", "b", and "c" are back-end services.
2.  // This function will return the response from the quickest service.
3.  func(startService(a : Address, b : Address, c : Address) {
```

```
4.      actor.await(request
5.          SomeRequest {
6.              aa : Future = future.spawn(actor.send(a, r))
7.              bb : Future = future.spawn(actor.send(b, r))
8.              cc : Future = future.spawn(actor.send(c, r))
9.
10.             // future.await will return the first
11.             // future that is satisfied
12.             output : Future = future.await(aa, bb, cc)
13.
14.             // Create a response from the first response
15.             response : SomeResponse = buildResponse(output.value)
16.             actor.send(request.sender, response)
17.         } else {
18.             // This "else" block is really important, even if it is empty
19.             // This actor will recieve many messages from the "late"
20.             // services that would otherwise fill up the current
21.             // actor's mailbox. This else block throws those messages
22.             // away and clean up the mailbox.
23.         }
24.     )
25. })
```

4.7.3 Client-Server

A common pattern shown in many examples is an actor that runs in an infinite loop and waits for certain messages to appear. Moreover, most of the examples provide helper functions that send and receive messages so client code does need to be concerned with low-level message passing.

This pattern is so common that Silo ships with a macro that provides an easier to use API. All developers need to do is specify a set of functions and the macro will automatically create message definitions for each of those functions, define an actor that dispatches those messages to the correct function, and emit client-side functions that handle the low level message passing. An example is shown below.

```
server.create(
    name(math)

    options(
        timeout(200)
        defaultSpawn(silo.core.spawn)
    )

    actions(
        add(a : int, b : int => int) {
            return(a + b)
        }

        subtract(a : int, b : int => int) {
            return(a - b)
        }

        // Etc...
    )
)

// The server.create macro creates a "Service" type automatically.
// It also create a "start" function which spawns the services.
// The start function accepts an optional argument that allows
// the user to specify which "spawn" function to use. The default
// is to use "silo.core.spawn".
s : math.Service = math.start()

// It also exposes client functions.
println(math.add(s, 5 , 5))

// Prints 0
println(math.sub(s, 5 , 5))
```

4.7.4 Load Balancing

Distributing load across actors is often needed in practice as a single instance of a service cannot handle the load. Often times this requires special external infrastructure from a third party that runs along side application code. However, users can naturally represent load balancing directly in Silo without any external dependencies. A simple reusable example is shown below:

```
1.  // Note how the function allows the developer to specify which "spawn"
2.  // function to use. Thus, staying true to the convention that library
3.  // code shoud never call spawn.
4.  func(loadBalance(f : Function, spawner : Function, replication : int) {
5.      servers : Vector = vector.create()
6.
7.      repeat(replication {
8.          servers = vector.push(servers, spawner(f))
9.      })
10.
11.     while(true {
12.         message = actor.read()
13.         i : int = Math.random() * replication + 1
14.         a : String = vector.get(servers, i) | checkcast(String)
15.
16.         actor.send(a, message)
17.     })
18. })
19.
20. // Spawn 5 instances of "someservice" using the standard "silo.core.spawn"
21. a : Address = spawn(loadBalance(someservice.start, silo.core.spawn, 5))
22.
23. // Can interact with back-end actors transparently through the loadbalancer
24. actor.send(a, "FooBar")
```

4.7.5 Monitoring and Fault Tolerance

A major challenge with distributed programming is handling errors. Programs can crash, hardware failures can occur, and the network may become partitioned. In many programming languages, if an error or unhandled exception takes place, the program simply crashes and the operating system shuts down the process. In service-oriented systems, this approach is not practical as the system needs to keep on running — just relying on exceptions is not a sufficient form of error handling.

Silo provides monitoring and error handling capabilities in the form of "supervisors". Each actor can call a special function called `spawnSupervisor` to create a supervisor. A supervisor is just like any other actor, however, the Silo runtime will send the supervisor a message when its corresponding actor either finishes (returns cleanly) or crashes (an unhandled exception propagates). This allows the supervisor to perform any background maintenance tasks (for example, sending a recurring heart beat message to let other actors know that it is still alive) or cleanup tasks (sending a "dead signal" to ignore others of the crash).

An actor can create as many supervisors as it desires. However, the link between an actor and its supervisors is not bi-directional. If a supervisor crashes, the actor will not be notified. Furthermore, if a supervisor crashes, it will not be resumed automatically. Thus, it is important to keep the logic of a supervisor as simple as possible. However, the supervisor is guaranteed to be in the operating system process as its corresponding actor and is *guaranteed* to receive the "exit" message from the runtime. Keep in mind, however, that it is still possible, and in fact common, for both the supervisor and the actor to crash at the same time (e.g. a power failure) and protocols should be designed according (i.e. perhaps a protocol should not rely on a "did finish" message since it may never arrive and rely on timeouts with a "heart beat" signal instead). Lastly, it is worth knowing that a supervisor to create supervisors of its own. However, in practice, it is perhaps better to keep supervisors as simply as possible and create "supervisor trees" at the application level.

The program below is a simple demonstration of supervisors. The application

consists of a simple "ping-pong" actor, which we have seen previously, and a registry actor, which acts as a "DNS service" that allows actors to lookup the "physical" address of another actor using a descriptive name. The program make uses of a supervisor that automatically restarts the ping-pong service in the event that it crashes.

```
1.  func(pingPongSupervisor {
2.      actor.await(message
3.          silo.core.ExitMessage {
4.              // Restart the the ping-pong service
5.              spawn(startPingPong)
6.
7.              // Close the supervisor, a new one will be created
8.              return()
9.          } else {
10.             // Ignore
11.         }
12.     )
13. })
14.
15. func(startPingPong {
16.     // Create the pingpong service
17.     spawnSuperVisor(pingPongSupervisor)
18.
19.     // Register the ping-pong actor with the registry
20.     registry.set("ping-pong", actor.self())
21.
22.     // Start running pingpong service
23.     pingpong.start()
24. })
```

4.8 Related Work

Programming Languages

Perhaps the most spiritually similar language to Silo is Erlang [11]. Many core aspects of Silo's design were heavily inspired by Erlang, including the design of the actor

implementation. A key difference is that Erlang is a functional languages and compiles to BEAM bytecode while Silo is imperative and compiles to JVM bytecode. Moreover, Erlang does not have Silo's polymorphic delegation capability and running Erlang code on multiple machines requires using a "distributed Erlang system" which is hard to customize and deploy in modern cloud-based hosting environment. Additionally, Silo's imperative actor API is different from Erlang's which is tightly coupled with other functional programming features like pattern matching, product types, and guards.

Functional programming languages like Haskell, Racket, and OCaml are often touted for the ease at which compiler can parallelize code, which is analogous to Silo's location transparency [47, 93, 53]. Unlike these languages, Silo provides an imperative model which is more familiar to most developers [116].

Many distributed programming languages have been introduced. Early languages and systems like Argus and CLU attempted to coordinate consistent memory across multiple machines [64, 65]. Silo does not attempt to guarantee consistent or coordinated memory abstraction and rather focuses on providing a message passing API. Languages like Salsa and Axum are other interesting examples of more recent distributed programming languages [122, 72]. However, they take quite different approaches to Silo by featuring different message passing semantics, not encouraging immutability when create sequential programs, and lacking Silo's polymorphic delegation capabilities.

Silo joins a large number of existing languages that compile to JVM bytecode including Scala, Clojure, JRuby, Groovy, Kotlin, and Ceylon [80, 46, 79, 109, 55, 97]. However, unlike many of these languages, Silo does not attempt to be a "better Java" and provides a drastically different programming model that is location transparent.

Clojure's emphasis on immutability and use of persistent data structures was inspirational to Silo. In fact, the reference implementation of Silo uses Clojure's implementation of persistent data structures. However, Clojure does not provide actors or lightweight concurrency constructs like Silo. Instead, Clojure requires developers to write code in continuation passing style for high-concurrency and use software

transactional memory (STM) for coordinating shared mutable state [103, 99]. STM is a powerful tool however it is hard to guarantee transactional memory in a distributed environment. That said, future work for could explore implementing Clojure-style STM on top of Silo's actor implementation in a manner than works in a distributed environment.

Rust and Go are both modern imperative programming languages that were designed for concurrency [36, 90]. Like Silo, both language include lightweight constructs for concurrency that are multiplexed across a small number of OS threads. However, these languages focus on multi-core programming rather than distributed programming. As a result, they still encourage shared state and their message passing semantics offer guarantees that are hard to provide in a networked environment. That said, I continue to be pleased with Rust's design and while it does allow shared mutable state it encourages developers to use immutable values by default.

Languages like Ur/Web, Orc, and Links also address similar issues as Silo as they aim to facilitate building Web applications [20, 60, 26]. Silo takes a more barebones approach and focuses on tools for building scalable reliable backend services that can be consumed from any client-side application or framework. Ur, Orc, and Links, on the other hand, offer a "full-stack" development experience where the front-end and back-end are both described in the same program.

Lastly, languages like Ruby, Python, and Javascript (Node.js) are commonly used in practice to implement service-oriented systems [34, 121, 57]. The dynamic nature of these languages make them ideal for building many small services that communicate with one another. However, as scripting languages, they often require external infrastructure to implement complex systems. As an example, implementing a load balancer in Ruby is not practical. Instead, multiple Ruby processes are run behind a special-purpose load balancer (like HAProxy) and monitored by operating system utilities in case a process crashes.

Frameworks and Software Applications

Many frameworks have been proposed that solve similar problems to Silo that deserve mention. Thrift is a cross language RPC library that allows RPC stubs to be generated between many languages [105]. Similarly, ZeroMQ is an efficient RPC library that allows messages to be sent between different nodes in a network [50]. Netty, Node, and LibAsync are event-driven networking libraries that allow program to achieve greater scalability, performance, and throughput than standard blocking APIs [113, 57, 30]. Akka is a famous actor framework that includes a remote message passing capability [120]. Additionally, message queues like RabbitMQ and ActiveMQ run as separate processes that broker messages sent between different programs allow systems to be designed in a loosely coupled manner [91, 112].

Most of these frameworks are heavy weight and programs using them typically need to be architected around the frameworks' conventions and idioms. That said, in most cases, they offer complimentary functionality to Silo and this paper does not intend to argue that one is better than the other. Silo's polymorphic delegation mechanism allows these existing frameworks to be "dropped in" without changing the rest of the Silo program. For example, Silo ships with a default distributed messaging capability based on Netty and TCP. However, if a developer wants to use their organization's existing deployment of RabbitMQ, they can do so by using a RabbitMQ adapter. As implementations of networking libraries improve, so can Silo.

Chapter 5

Implementing Coroutines in Silo

Languages that compile to Java Virtual Machine (JVM) bytecode rarely support coroutines because they are hard to implement efficiently given the constraints of the JVM. This is problematic for building scalable I/O-bound concurrent systems, like Web services, in which coroutines provide an elegant balance between the intuitiveness of blocking code, where a separate thread processes each request, and performance of event-driven code, where an event loop dispatches I/O events as they occur to callback functions. This chapter introduces compilation techniques for implementing high performance coroutines as a first class citizen in Silo, a JVM-based language for building service-oriented systems. Silo's coroutines require no changes to the JVM, are suitable to enable by default in new programming languages, and match or beat the performance of languages specifically designed to support coroutines.

5.1 Overview

Coroutines are functions that can be suspended and resumed at pre-defined locations. They are powerful primitives and have been used to implement many high level programming constructs like iterators, infinite lists, and exceptions. Recently, there has been renewed interest in coroutines for implementing cooperatively scheduled tasks, commonly referred to as "fibers". Fibers can be thought of as "lightweight threads" and are particularly useful for implementing high concurrency servers.

Unlike many programming languages, Java does not support coroutines. Moreover, the Java Virtual Machine enforces constraints to memory usage and control flow that make it impossible to use existing techniques for implementing coroutines [63, 37, 22, 75, 42, 117]. This is unfortunate for two reasons (1) Java is perhaps the most widely used language for implementing Web services and (2) many languages are now compiling to JVM bytecode to gain access to a high performance VM and a rich ecosystem of libraries and tools.

I present a suite of compilation techniques for supporting coroutines on the JVM that is suitable for use in new languages that compile to JVM bytecode. I implement these techniques in Silo, an actor-based programming language that uses coroutines to back the execution of a large number of concurrent actors onto a fixed number of operating system threads. In this chapter, I make the follow contributions:

- I describe a method for representing coroutines in JVM bytecode. The interesting properties of the method are that it works on any JVM without modification, is high performance, does not incur runtime penalty for function invocations that do not suspend, and does not generate unduly large methods.

- I present an empirical evaluation of Silo against other programming languages, Web servers, as well as other approaches for implementing JVM coroutines. Silo's coroutine implementation matches or exceeds languages with dedicated support for coroutines in micro-benchmarks. Furthermore, Silo is able to handily outperform Web servers written in popular languages for "Web development" and matches the performance of the Nginx Web server [114].

5.2 Background

5.2.1 Scalable I/O Architectures

Coroutines have received recent interest for facilitating the implementation of the scalable Web architectures [90, 11, 36]. Handling concurrency efficiently is of utmost importance for Web services since they need to handle a large number of requests that can arrive at the same time. There are two main approaches for implementing these types of servers in practice.

- **Forking** Many servers create multiple threads to handle requests simultaneously. When a message is received, the server will create a new thread (or fork a new process) to handle the request and then return to waiting in a loop for the next message. This type of server is often called a "forking server", making reference to the `fork` Unix system call. One problem with this approach is that many network requests are short lived (e.g. sending a small image) and the time spent creating a new thread is often more than the time spent processing the request. To address this, many forking servers will create and maintain a pool of thread (or processes) upfront instead of creating and destroying a new thread for every request [69]. This is called a "pre-forking server". Furthermore, to prevent threads from sitting around doing nothing, some pre-forking servers will create sophisticated thread pools that automatically destroy threads that are idle for a long time or dynamically add more threads as needed [69, 113]. Technical minutia aside, the most important aspect of multithreaded servers is that they rely on the operating system kernel's scheduler to handle concurrency. Web servers like Apache's httpd and Microsoft's IIS are examples of pre-forking servers [111, 71].

- **Asynchronous Event Loops** Instead of using a separate thread to process each request, some servers use a single thread running in an event loop instead. Event loop servers tell the operating system to notify them when certain events take place, for example, a new connection being created or new data arriving

over an existing connection. However, special care must be taken to not perform long running operations when processing an event since it will block the entire event-loop and drastically reduce performance. Luckily, most servers do not perform long running computations; rather, they spend most of their time doing I/O operations (e.g. talking to a database, reading a file from disk, writing data to the network, etc). Many of these tasks can also be implemented in an event-driven manner and merged into the server's event loop. For example, instead of querying a database and waiting for the results, the server can send the query to the database over a TCP connection and then tell the operating system to notify the server when new data arrives from the TCP connection (i.e. the results have been returned). In the mean time, the server will return to the event-loop and can be processing other events that have taken place, for example, perhaps a new client connected and wants to download a particular HTML file. This style of development is called "non-blocking" or "asynchronous" since the server never blocks on an operation and always returns to the event loop as soon as possible. By processing requests in a piecemeal and overlapping manner the server is able to handle many requests with just a single thread. As an enhancement, many event loop servers run multiple event loops (each on their own thread) to take advantage of multicore processors. Web servers like Nginx, Lighttpd, as well as the Node.js HTTP server are examples of event-driven servers [114, 54, 57].

Event loop servers can be far more scalable than forking servers [86, 126, 1, 31, 98]. Each event loop thread can handle many client connections whereas forking servers can only handle a single connection per thread. Moreover, since most requests are I/O bound, forking servers end up creating a lot of threads that literally do nothing and simply wait for I/O operations to complete. These idle threads take up valuable system memory (for example, every Java thread is allocated with a 1MB stack by default), puts pressure on the kernel's scheduler (especially in cases when multiple threads "wake up" at the same time), and reduces overall system performance.

On the other hand, forking servers are far easier to implement and maintain. To implement event loop servers, developers need to manually manage the concurrent

```
// Non-Blocking / Event-Driven
myDatabase.startTransaction().success(function() {
    function abort() {
        myDatabase.abortTransaction()
    }
    function process(urls) {
        var url = urls[0]

        myWebService.get("SOME_URL").success(function(data) {
            myDatabase.insert(data).success(function() {
                if(urls.length == 1) {
                    myDatabase.commitTransaction()
                    ... continue ...
                } else {
                    process(urls.slice(1))
                }
            }).error(function(error) {
                abort()
            })
        }).error(error) {
            abort()
        }
    }
    process(urls)
}).error(function(error) {
    console.log(error)
})

// Blocking Code
myDatabase.startTransaction()

try {
    for(url : urls) {
        data = myWebService.get("SOME_URL")
        myDatabase.insert(data)
    }

    myDatabase.commitTransaction()
} catch(error) {
    myDatabase.abortTransaction()
}
```

Figure 5-1: Blocking vs non-blocking code in Javascript. Blocking code is often much more intuitive and easier to understand.

control flow for every request. In practice, this means that straight forward application logic needs written in continuation passing style and broken up across callback functions leading to "callback hell" in which code is hard to understand and maintain [3, 74]. This is particularly problematic for application servers (as opposed to static standalone HTTP file servers) since application code is likely to change more frequently. Figure 5-1 shows an example of easy to understand blocking code (used with a forking server) compared to non-blocking code (used with an asynchronous server).

Coroutines provide an elegant balance between ease of use and performance. Coroutines are functions that can pause and resume at predefined locations. Since they are a language-level construct and do not require support from the operating system kernel, coroutines can pause and resume much faster than threads. Furthermore, they are often implemented with stacks that dynamically grow and thus not only consume far less memory than threads but are generally faster to create as well. However, the true beauty of coroutines is that they *have* a stack that provide automatic stack management [3]. This means that developers can write code "normally" instead of in a continuation passing style.

The only caveat is that coroutines are not preemptive scheduled by the kernel; rather, they must be scheduled manually by the application itself. However, for servers this is hardly an issue as a natural "context-switching point" readily exists: I/O operations. A server could wrap all major I/O functions such that they initiate the I/O in a non-blocking manner and then pause the current coroutine and return to some sort of an event loop. When the event loops gets notified that the I/O operation has completed, it will find and resume corresponding coroutine. Until that time, the event-loop can resume other coroutines that are waiting to execute. In this way, developers get the best of both worlds. A single operating system thread can now handle many clients and application code is written in a straight-forward blocking manner.

5.2.2 Continuations

Coroutines are closely related to the concept of a "continuation", which is a descriptor of how and where a computation should "continue from" in order to complete. Conceptually, a continuation should include a location to resume execution (e.g. a function pointer or the address to an instruction) and any relevant data that is needed by the computation. Most of the time, continuations are created manually by the developer and then passed between between functions to be resumed at a future point in time. This style of programming is called "continuation passing style". The problem with continuation passing style is that it is a manual process. In fact, it is often called "manual stack management" because it forces developers to save information that a stack trace would naturally contain [3]. Language features like closures are somewhat helpful since they eliminate certain types of book-keeping but still require code to be structured in a somewhat awkward way since you cannot return in continuation passing style.

Coroutines offer an alternative to continuation passing style that offers "automatic stack management". There are many ways to implement coroutines, but one of the more efficient approaches allocates a large chunk of memory on the heap, which serves as a makeshift stack. To resume a coroutine, the processor's stack frame register is pointed to this heap-allocated stack and the program counter is set to the next instruction after where the coroutine paused execution. To pause, the state of the processor's registers are saved to the same heap-allocated stack (i.e. a continuation) and the processor's stack frame register and program counter are reset. This approach is fast (pauses and resumes in constant time) and does not require a trip through the kernel (which can serve as a bottleneck). Unfortunately, coroutines are not supported on the JVM.

5.2.3 Java Virtual Machine

The Java Virtual Machine (JVM) is a virtual machine that runs inside of a process and executes JVM bytecode. Java programs are first compiled to JVM bytecode

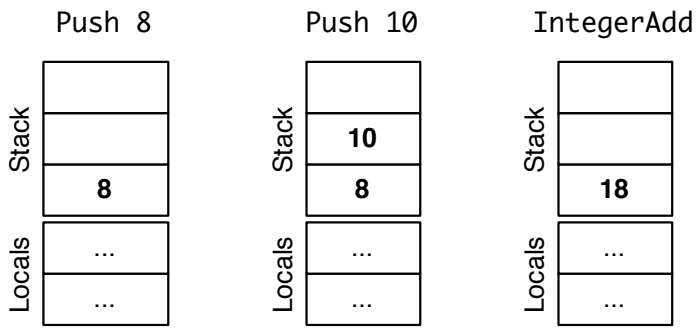

Figure 5-2: The JVM is a stack-based virtual machine. It operates by pushing operands onto a stack, popping them off, and pushing the results.

which, in turn, is executed by the JVM. As Java has grown, the JVM has gotten progressively more refined, performant, secure and stable and is one of the most advanced virtual machines in existence today [63]. As a consequence, many new programming languages elect to compile to JVM bytecode instead of native machine code to gain access to the JVM's highly optimized just-in-time (JIT) compiler (which compiles bytecode to machine code on the fly), high performance garbage collector, as well as interoperability with a huge ecosystem of libraries and frameworks.

The JVM is a stack-based virtual machine. The stack itself is divided into two parts: the operand stack and local variables, as shown in Figure 5-2. All JVM instructions work by popping operands from the operand stack and pushing back results. This includes local variables — to use a local variable, the variable must first be copied to the operand stack. Figure 5-2 shows an example of how the JVM adds two numbers. The JVM instruction set is somewhat large and contains support for basic arithmetic (addition, subtraction, etc.), simple control flow (conditional branch statements, and goto statements), storing and reading local variables, manipulating composite data types (arrays and objects), invoking methods, and others. While the JVM offers many high-level language features like virtual dispatch, exceptions, and monitors, the JVM spec does not have any support for coroutines. Moreover, many common techniques for implementing coroutines are not possible either of because of enforced requirements set by JVM or practical implementation details:

1. Control statements can only jump to instructions within the same method.

The only way to jump to a far off location is to throw an exception or to use an `invoke` instruction, which will transfer control to another method after allocating a new stack frame to hold the target method's local variables and operands. This means that is not possible to trick the JVM into using a heap-allocated stack and must always use the stack of a JVM thread.

2. A method can only access locations in the stack that are part of its frame and cannot access the stack of the calling method. Moreover, the only way to manipulate the stack *trace* is to call another method (which will add a frame to the trace) or return from a method (which will remove a frame from the trace). This means that it is not possible to "mimic" a heap allocated stack trace in a utility function that dynamically saves the state of the thread's stack and remove all the frames from the stack trace.

3. The JVM sets a maximum size of a single method to 64KB. This means that it is not possible to compile entire applications as a single method and co-opt control statements to implement functions. Furthermore, the JVM's high performance JIT compiler will not attempt to compile methods that are greater than 8KB. Thus, this approach is likely not performant [123].

4. A cornerstone feature of many JVMs (including the Oracle's HotSpot JVM and OpenJDK) is its tracing JIT compiler. The JIT works by identify code paths are that commonly executed and then translating them to native code. In the process, the JIT can perform many optimizations based on runtime profiling information like biasing against uncommon branch statements and inlining methods. A common approach for implementing coroutines is using a "trampoline" in which the caller of a method returns control to a trampoline that calls the method on the caller's behalf. When the callee finish, it also returns to the trampoline which calls the caller again, but this time with the return value [37]. This approach is useful for implementing coroutines because the trampoline can record a stack trace as the program executes. However, this approach yields terrible performance because of the high overhead (calling a

method is now four times as expensive) and it prevents the JVM's JIT from inlining method calls [18].

This discussion is not meant to be critical of the JVM. In fact, each of these restrictions is done for good measure — its makes the JVM much more secure, able to apply aggressive JIT optimizations, and far easier to parallelize. However, they do represent challenges that need to be overcome when implementing coroutines.

5.3 Continuation Passing Transform

To implement coroutines in Silo, the Silo compiler performs a continuation passing transform on all Silo code. The engineering details of this transformation cannot be taken lightly since all Silo code is internally converted to continuation passing style to ensure that any function can be paused and resumed. In short, this means that non-pausing code should not incur a significant runtime performance penalty.

The Silo compiler transforms every Silo function such that the stack can "unwind" and then "rewind". The intuition of this transformation is shown in Figure 5-3.

Each function is passed a hidden parameter of type `ExecutionContext`. This parameter is pass-by-reference and contains a boolean field that indicates if the current execution is yielding. Whenever a function calls another function, it passes the context variable as an argument. When the callee function returns, the caller checks to see if the `ExecutionContext` is in a yielding state. If it is not, it allows execution to continue as normal. If it is, the caller will pack the local variables and operand stack into a data structure called an `ExecutionFrame`, record the index of the call site where execution yielded into the `ExecutionFrame`, save the `ExecutionFrame` to the `ExecutionContext`, and return a dummy value. This same behavior will be performed on every function in the stack trace until the entire stack is unwound. To ensure that `ExecutionFrames` are inserted into the correct location into the `ExecutionContext's` "frames" array, all call sites are wrapped with `beginCall` and `endCall` which increments and decrements an internal stack pointer field.

```
// Original Code
public void foo(int a) {
    int b = bar(a);
    int c = baz(b);
}

// Transformed Pseudo Code
public void foo(ExecutionContext ctx, int a) {
    // Hoist and initialize variables.
    int b = 0; int c = 0;
    // Jump to the point where execution paused.
    // Program counter is -1 initially.
    switch(ctx.programCounter) {
        case 0: goto RESTORE_SITE_0;
        case 1: goto RESTORE_SITE_1;
    }

    goto CALL_SITE_0;
RESTORE_SITE_0:
    <restore-operand-stack>
CALL_SITE_0:
    ctx.beginCall();
    b = bar(ctx, a);
    switch(ctx.endCall() {
        case RESUMING:
            <restore-local-variables>;
            goto END_CALL_SITE_0;
        case CAPTURING:
            <store-operand-stack>;
            <store-local-variables>;
            ctx.setProgramCounter(0);
            return;
        case YIELDING:
            return;
        case RUNNING:
            // Fall through
    })
END_CALL_SITE_0:

// Similar code for "int c = baz(b)"
...
}
```

Figure 5-3: The Silo compiler transforms code so that the execution stack can "unwind" and then be "rewound".

To resume execution, the stack trace is artificially re-created. At the start of each Silo function, the compiler inserts a jump table that inspects the current `ExecutionFrame`'s call site index. If the index is "-1" that means that the function is being called for the first time and never yielded. Otherwise, it will jump to the call site that caused execution to yield. The function will then push dummy values onto the stack (to appease the JVM, see Section 5.2.3) and then call the function again. This continues all the way "down" until the stack trace has been re-created. Note, however, that no stack information has actually been restored. This happens after the callee function returns. The caller inspects the ExecutionContext (just like with a normal call) and if the context is yielding again (perhaps a function "downstream" resumed by yielded again) it simply returns a dummy value. In this case there is no need to save the local variables or operand stack because this information is already saved. However, if the context is not yielding, the function will check to see if a ExecutionFrame is present and restore the contents of the local variables and stack accordingly. Thus, in this way, functions delay restoring their stacks until absolutely necessary.

This approach has many benefits.

1. It preserves the JVM's call stack which is helpful for debugging (many continuation passing transforms lose all stack information making debugging hard [102]), allows functions to return after resuming, and does not get in the way of the JVM's JIT optimizations (unlike trampolining transformations, which does).

2. There is minimal overhead for functions that do not yield. This can be seen by Silo's performance results in Section 4.6 where Silo is able to achieve comparable performance with Java despite the fact that it is written in continuation passing style. In fact, the only real overhead Silo incurs is checking the the "yielding" field on the `ExecutionContext` which is a single branch instruction. Moreover, most of the time a call site will either always yield or never yield and the JIT compiler optimizes this type of code using runtime profiling and branch prediction [12].

```
public class ExecutionFrame {
    public int programCounter = -1;
    public Object[] locals;
    public Object[] stack;
}
```

Figure 5-4: Silo needs to store local variables as well as the operand stack when program execution is pausing. One way to do that is to create a stack frame which holds values in dynamically sized arrays.

```
public class Frame_58b1e3ea_IDLJL extends ExecutionFrame {
    public int stack_0;
    public double stack_1;
    public Object stack_2;

    public long local_1;
    public Object local_2;
}
```

Figure 5-5: Silo reifies custom stack frame objects that have fields corresponding to state of the operand stack and local variables at the call site where execution yields. An example is object is shown here and corresponds to a call site where the stack contains an `int`, a `double`, and an `Object` and there are two local variables, one `long` and one `Object`.

3. Resuming execution is also fast. The jump table inserted at the top of each function jumps to the call site in constant time (i.e. it is not a linear-time switch statement) and, again, is a single instruction that can be optimized by the JVM since call sites that yield are likely to yield again.

4. This approach avoids using exceptions for non-local returns. Since we need to "unwind" the stack to a fixed point, an idea that comes to mind is to throw a special exception that propagates up the stack. This approach avoids using exceptions since they can be up to 20 times slower than normal returns [73].

5.4 Optimizations

5.4.1 Custom Stack Frames

A technical detail that was glossed over in the previous sections was *how* the operand stack and local variables are saved. A potential solution is shown in Figure 5-4 in which `Object` arrays are used. The `Object` class is the root class in the Java language and can store any Java value and thus Object arrays appear to be a good general-purpose container for storing arbitrary data.

However, there are three major performance drawbacks to using object arrays. First, when restoring the stack, all values need to be type casted since the original type information is lost when using an Object array. Second, primitive types not only need to be casted, but also boxed and unboxed since Java primitive types (`int`, `double`, etc.) cannot be directly assigned to an `Object`. Third, array accesses in the JVM are subject to bound checks for security reasons and these bound checks impact performance [63, 94].

To get around these issues, Silo creates a custom ExecutionFrame for every call site. Instead of using an array, this custom object has separate fields for each operand and local variable. This eliminates the need to box primitives and the use of arrays. An example of a custom ExecutionFrame is show in Figure 5-5.

An obvious concern is that creating a custom class for *every* call site will result in a bunch of otherwise unnecessary classes. To avoid this, the Silo compiler will re-use ExecutionFrames between call sites as possible. For example, calls to `foo(int, int)` and `bar(int, int)` are likely eligible to share the same ExecutionFrame since the contents of the operand stack (i.e. two ints) are the same. There are some edge cases (for example, `sum(0, foo(1, 1))` is different from just `foo(1,1)`) but this drastically reduces the number of ExecutionFrames as the number of unique call sites is actually quite small in practice [106].

5.4.2 Minimizing Method Size

A major drawback to Silo's continuation passing transformation is that it inserts a large number of instructions for each call site. Under most circumstances, there is a staggering 60-to-1 ratio from a "normal" call and a transformed call. Moreover, this ratio can grow without bound depending on the level of nested function calls

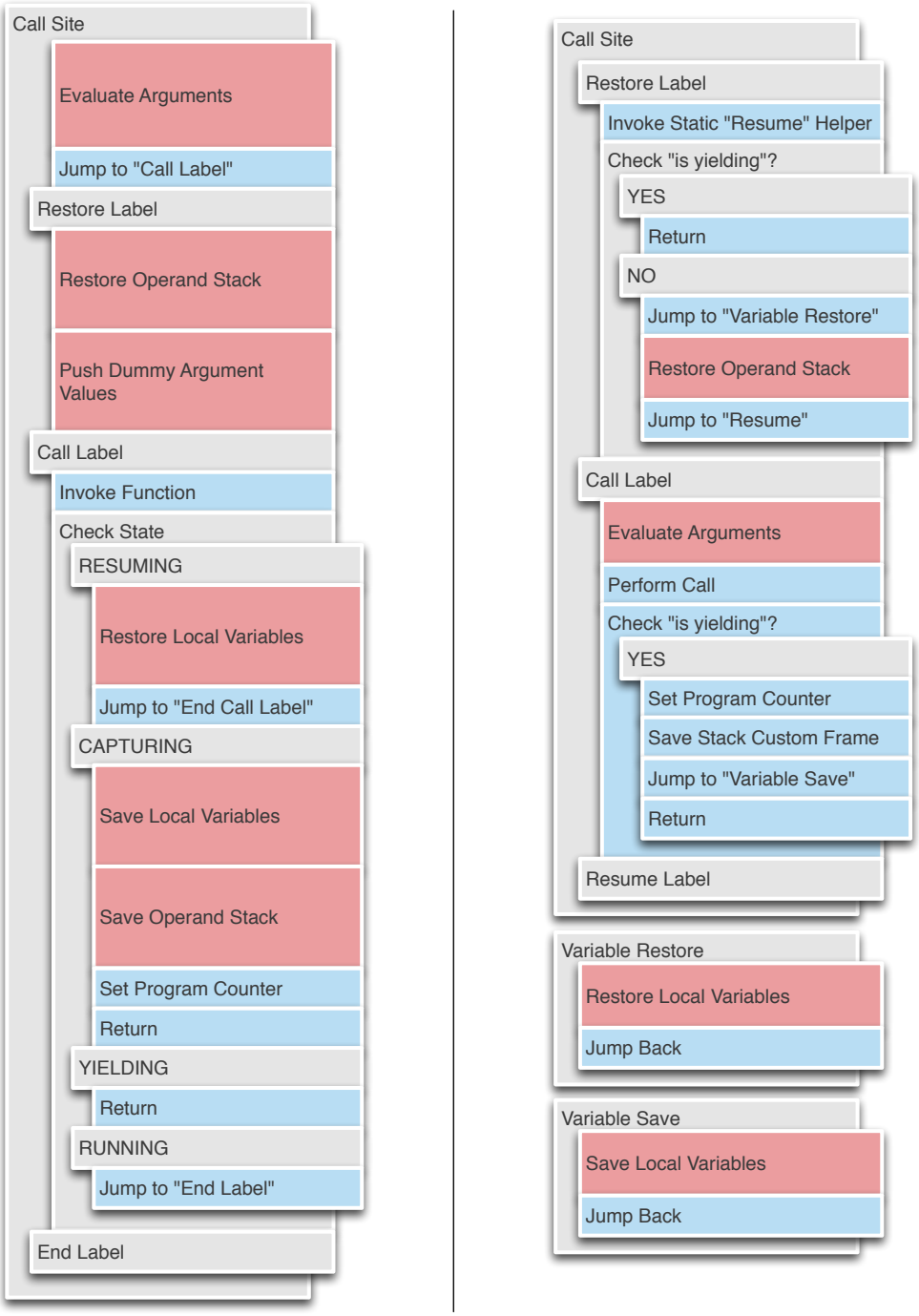

Figure 5-6: Silo attempts to reduce the code size of each call site. The figure on the left is the original call site and the right is the call site after code reduction. The reduced code size is a constant number of instructions (in blue) per call site with the exception of the operand stack, which grows depending on usage (shown in red).

and number of local variables. This is problematic for three reasons. (1) The JVM enforces a maximum method size of 64KB. (2) Most JVM implementations will disable the JIT compiler on methods that are greater than 8KB, which has a huge impact on performance [63, 123]. (3) During JIT compilation, one of the most powerful optimizations the JVM can do is inline method calls; however, on most common architecture, the JVM will only inline methods that are less than 325 bytes [78].

To demonstrate the performance impact, I ran a simple experiment that artificially inflated the size of a Silo function by inserting dummy instructions that would never get executed at runtime. The function that could be inlined finished in 1046ms, the function that could be JITed but not inlined finished in 6128ms, and the function that could not be JITed at all finished in 149218ms.

In addition to the severe performance impact, the code size problem is also quite common in practice. The 60-to-1 ratio means that Silo functions, effectively, need to be kept under 130 *bytes* in size to enable JITing and under just 5 *bytes* to enable inlining. To put this in perspective, if we take the Java standard library and apply the 60-to-1 ratio, 5% of all methods would be ineligible for JITing and 73% would be ineligible for inlining. Currently 0% of methods are ineligible for JITing and only 0.12% of methods are ineligible for inlining.

To avoid this issue, Silo reduces the size of a transformed method in several ways and can get the ratio down to approximately 10-to-1, a significant improvement. With this new ratio, all methods in the Java standard library are eligible for JITing and only 28% are ineligible for inlining. An illustration of the final transformation with all of the following improvements is shown in Figure 5-6. These improvements are discussed below.

- The instructions for saving and restoring local variables are abstracted and the execution flow of the program will jump to a certain label to save (or restore) the local variables and then jump back. Thus the cost of saving local variables is amortized across all call sites in the function. Unfortunately, there is no way to jump and return in the JVM. The `jsr` instruction would allow this behavior but is all but deprecated and most JVM implementations will disable the JIT

for methods that use `jsr` since the bytecode is hard to optimize quickly. To get around this, Silo's compiler will insert instructions to jump to the "local subroutine" and then, instead of "returning", the execution flow will jump to the start of the method and re-use the jump table to get back to the correct location. Thus, by simply jumping two times, the compiler can emulate the `jsr` instruction.

- The custom `ExecutionFrames` are compiled with a helper static method that accepts all of the operands and saves them to the appropriate fields. Thus, instead of emitting instructions to save each operand individually, the Silo compiler can insert a static method call which will save the operand stack in a single instruction.

- The instructions for pushing dummy values onto the operand stack are eliminated by splitting the call site into two call sites: one for normal execution and the other for resuming. Moreover, the call site for resuming is moved into a separate helper utility method so the Silo compiler does not need to insert instructions for pushing dummy parameter values. The reason for pushing dummy values in the first place was to appease the JVM verifier. The JVM requires that the operand stack is always consistent at any given instruction offset regardless of control flow. For example, the following JVM pseudo-bytecode is not valid because depending on the value of `c`, the operand stack at the `print` call is different — in one instance the operand stack will have a single string operand and in the other the stack will have a single integer operand:

```
boolean c = ...;
if(c) {
    <push_string "Hello">
} else {
    <push_int 1>
}
print()
```

To avoid this, the Silo compiler pushes dummy values on the stack before resuming to ensure that the operand stack is consistent after the call returns. However, by splitting the call site into two parts, this is no longer necessary.

Lastly, Silo allows developers to annotate functions as non-yielding to prevent the compiler from transforming call sites. This drastically reduces code size as it eliminates the overhead of the continuation passing transformation. In this case the size of Silo functions is comparable to the size of Java method generated by `javac`.

5.4.3 Hybrid Trampolining

Resuming a long stack trace can be expensive since each function along the trace must be called. To avoid this, Silo loads ExecutionFrames on-demand. It starts by resuming only the top-most ExecutionFrame by calling the relevant method dynamically. When this function returns, the Silo runtime checks to see if the ExecutionContext is yielding or running. If it is yielding, then the runtime does not do anything further and can execute another coroutines instead. However, if the function is not yielding, the runtime will then load and invoke the next ExecutionFrame so that execution can continue.

This approach may seem similar to the "trampoline" discussed earlier in Section 5.2.3. As previously mentioned, trampolining is far from ideal because jumping back and forth is a lot of overhead. To minimize this impact, Silo's runtime uses an exponential back-off technique to load progressively more frames each time execution returns to the trampoline while the ExecutionContext is not yielding. In other words, the runtime will first resume 1 frame, then 2, then 4, then 8, and so on. This approach is called "hybrid trampolining" and allows Silo to resume deep stack traces more efficiently.

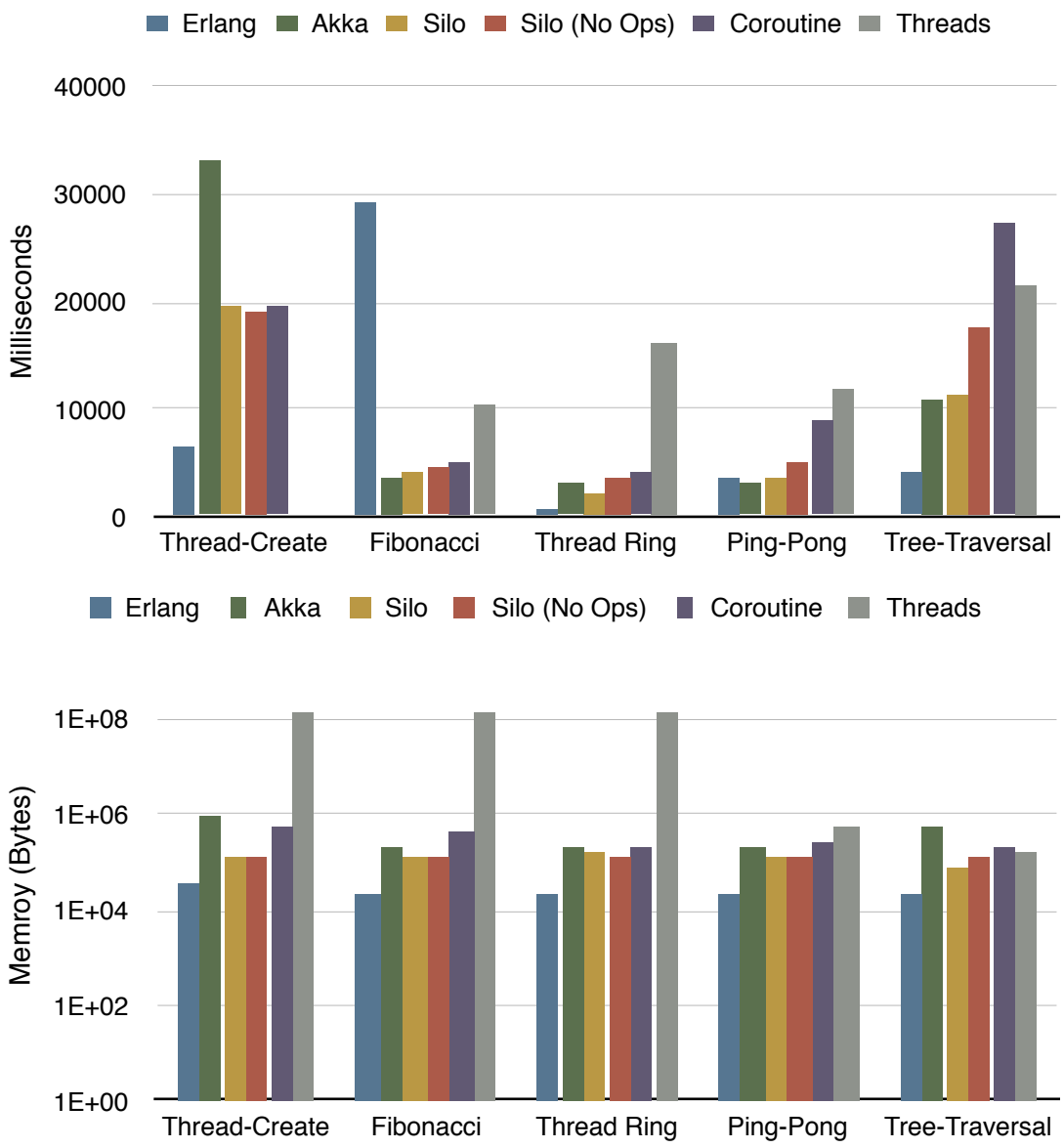

Figure 5-7: Silo's performance (top) and memory consumption (bottom) on message passing tasks compared to Akka, Erlang, Java (Threads), and the Java coroutine library. Note the Thread was unable to complete the "thread-create" benchmark. Also, note the the memory consumption is log scale.

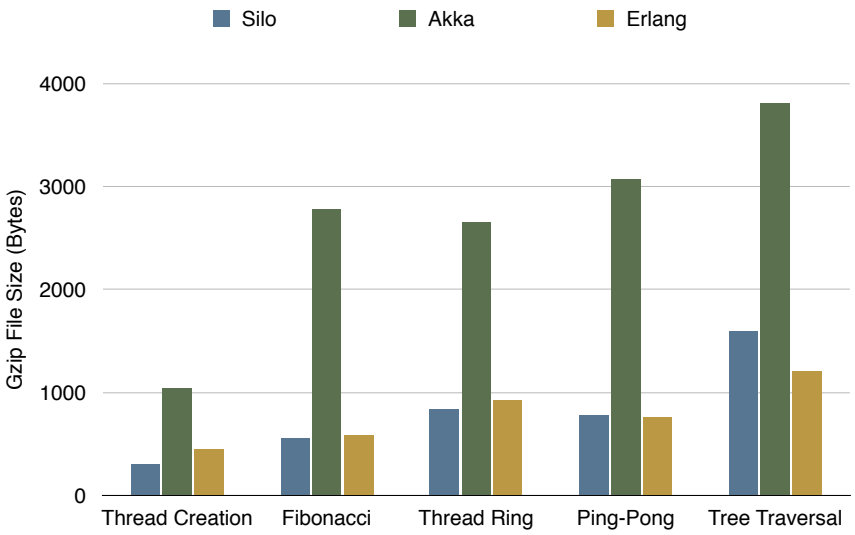

Figure 5-8: A comparison of the code size of various concurrent benchmark programs written in Silo, Akka (Java) and Erlang.

5.5 Evaluation

To evaluate the performance of Silo's coroutines I compare it against Erlang, the gold standard in concurrent programming, and Akka, arguably the most popular and highest performance actor framework written for the JVM, and actual threads. Additionally, I benchmark Silo with and without the optimizations described in this section to see if they make a difference in practice. Lastly, I include an existing JVM coroutine library that uses exceptions to unwind the stack to compare Silo's approach to existing JVM-based solutions. The benchmarks perform a variety of message passing tasks that stress different runtime behaviors.

1. **Thread Creation** Measures how long it takes to create and spawn a large number of actors. This test measures how fast it takes to "spin up" an actor.

2. **Fibonacci** Implements an "infinite" list of fibonacci numbers. Whenever the fibonacci actor receives a message, it replies with the next fibonacci number. This test measures runtime performance when two actors are sending messages back and forth between one another.

3. **Thread Ring** A "ring" of actors is created in which each actor passes a message to the next actor and I measure how long it takes to send a message around the ring. This test measures runtime performance when many actors are sending messages but always to the same actor.

4. **Randomized Ping Pong** A single actor randomly picks an actor from a pool and sends it a "ping" message and waits for a "pong" message in reply. This repeats a large number of times. This test measures performance when a system has a "one-to-many" broadcast configuration.

5. **Tree Traversal** One actor stores a tree-like database and exposes an API that allows clients to query for the children of a particular node. The client recursively traverses this tree in a depth-first manner until all nodes are reached. This test measures the performance when an actor has a deep stack because of recursion.

The runtime performance of these experiments along with the code size of the implementations are show in Figure 5-7. The code size is determined by taking the source code and compressing it with only the "level one" compression of gzip (Figure 5-8). Level one compression is the least compression that gzip will perform and is useful to simply strip whitespace and consolidate large redundant phrases (e.g. Java import statements). Looking at the results, there are a couple of interesting observations.

1. Silo is able to beat Erlang and Akka in certain tests. This is impressive because Erlang has special support for actors and concurrent programming, unlike the JVM which provides no support. Nevertheless, Silo performs well. Additionally, Akka is a Java framework that requires code to be written with manual continuation passing style. Despite this, Silo's automatic continuation passing transformation is still able to out perform manual hand-rolled continuation passing style of the Akka code in certain circumstances. This is likely because the Silo compiler is able to make transformations that developers would nor-

mally want to avoid because it requires too much effort or is unmaintainable in the long run.

2. Threads perform, by far, the worst. This finding corroborates the intuition that threads are too heavy-weight for massively concurrent systems. Threads are slower to context switch in virtually all tests because their context switches must be coordinate by the OS kernel, thus creating a concurrent bottleneck. Moreover, threads consume orders of magnitude more memory that all other implementations. For Web services, the memory consumption is likely to be more concerning. Slow context switching is bad but its degrades performance gracefully. However, once enough memory is allocated, performance will sharply drop: the system will start paging and struggle to perform routine tasks. As an example, the threaded implementation was unable to finish the Thread Creation benchmark on the benchmarking system. It was simply unable to create the necessary number of threads and the process was forcefully shutdown by the OS and runtime.

3. Erlang's performance is impressive and it certainly has earned its reputation as the gold standard for concurrent programming. However, it is important to realize that these benchmarks stress the performance of context switching between different tasks. Real applications will involve actual computations as well. The fibonacci benchmark highlights this issue. While Erlang is able to context-switch fast (as seen by other benchmarks), the moment it needs to perform computational tasks (even a task as simple as fibonacci) it quickly lags behind Java's high performance JIT compiler. This can also be seen when looking at the performance results in Figure 4-13. Thus, just because Erlang is able to pass messages quickly does not mean it is suitable for all aspects of system development.

4. Silo's optimizations make a major difference on the benchmarks. In many cases the optimizations move Silo from last place into first place. The tree traversal is particularly interesting and highlights the importance of Silo's hybrid tram-

polining. Unfortunately, on the tree traversal benchmark Silo is unable to reach the performance of Erlang. This is most likely because Erlang has support for tail-call eliminations which the JVM does not provide. In some cases, however, the optimizations cause Silo's performance to degrade. This is seen in the Thread-Create benchmark. The reason for this is that the custom stack frames force the JVM verifier to check for the exists of more classes at runtime than before.

5. The exception-based Java coroutine library exhibits fairly low performance. This should not be surprising and it confirms our concern that exceptions may be too slow for use. Nevertheless, the coroutine library is still able to outperform threads, which was its main goal to begin with. Additionally, while its runtime performance was not stellar, its memory consumption is far better than threads.

6. The memory consumption graphs show that Java uses far more memory than Erlang. This should not be cause of too much concern. Java is known to be a memory-hungry language that trades off high memory consumption for fast execution performance. Java tends to allocate large chucks of memory and waits until most memory is consumed until the garbage collector kicks in. In short, high memory usage is common for Java and is not much of a concern in practice.

7. When looking at the code size, Silo is comparable to Erlang and much more concise than the Akka code. This is the result of two factors. First, Java as a language tends to be a bit more verbose (although, the fact that the code is run through gzip mitigates this). Second, Akka requires code to be written in continuation passing style manually, Silo's compiler is able to perform this transformation automatically. Moreover, it would appear that Silo's automatic transformation is just as efficient, if not more. Thus Silo makes it both easier and more efficient to write code.

Lastly, to evaluate Silo's performance in real-world applications, I compare an HTTP server implemented in Silo to other languages. Silo performs well and the results are shown in Figure 5-9.

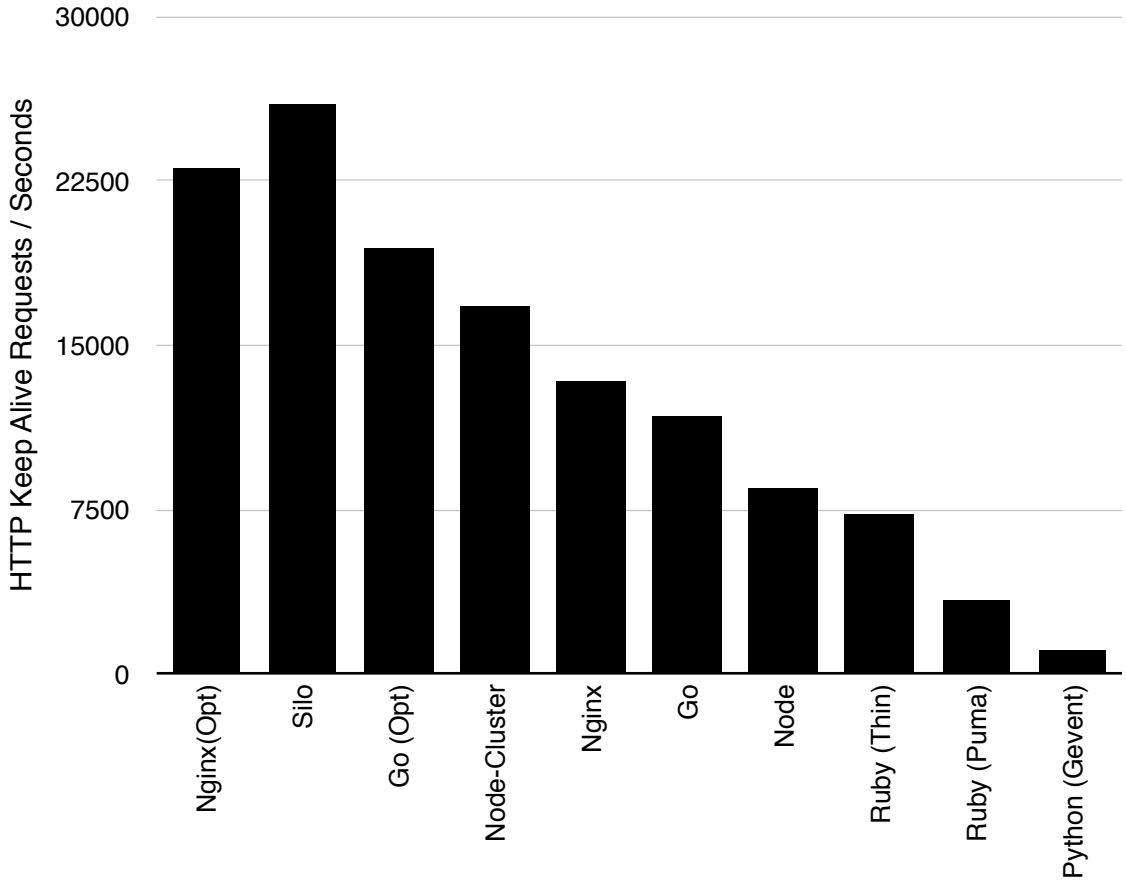

Figure 5-9: HTTP performance in Silo compared to many other languages.

5.6 Related Work

Coroutine frameworks in Java exist and, like Silo, convert JVM bytecode into a continuation passing style. Examples include Kilim, Matthias' Continuations Library, Rife, and Javaflow [106, 67, 115, 110]. These frameworks work by pre-processing JVM bytecode either as a separate build step or at runtime using a Java agent to "instrument" classes as they are loaded by the JVM. While the approaches of these frameworks are conceptually similar to Silo, Silo makes different engineering decisions: it does not use exceptions to unwind the stack, it uses a hybrid trampoline to speed up resuming, it includes a transformation style that reduce the overall code size, and it avoids capturing the stack until absolutely necessary. These improvements were necessary for Silo since, unlike the other approaches, Silo is a language and not a library and needs to provide a construct that is more general purpose and works well for many use cases without manual programmer intervention or tweaking.

Java's cousin, C#, has support for "async" methods that can pause and resume. C# async methods are not implemented at the VM-level but rather at the compiler level. The C# compiler will transform methods into continuation passing style but requires special syntax to call async methods and incurs a heavy runtime performance penalty [70, 108].

Approaches have been suggested for modifying the JVM to support continuations [81, 107]. However, none of these mechanisms have been adopted by the JVM spec nor by any mainstream implementation of the JVM. Moreover, while these approaches potentially offer much in terms of greater performance, it reduces the portability of JVM code and may come at the expense of stability. Most notably, it means that code can not run on mobile platforms like Android's Dalvik or Oracle's Mobile Application Development Framework [39, 83].

Many other languages provide efficient implementations for coroutines including Go, Haskell, Python (generators), Stackless Python, Racket, and Ruby (fibers) [90, 47, 121, 117, 93, 34]. Of these, Go and Haskell deserve special mention for addressing the same underlying challenge as Silo — handling I/O efficiently. Like Silo, Go's

coroutines (named "goroutines") implicitly yield whenever I/O takes place as does the Mio framework written in Haskell [124]. The key distinction between Go, Haskell, and Silo is that Silo targets the JVM and allows developers easy interoperability with Java library.

It is worth mentioning key alternatives to coroutines. The main aim of coroutines is to facilitate continuation passing style and allow programs to represent concurrent logic without relying on heavy-weight operating system threads. Frameworks and languages like Node.js, Akka, Netty, Haskell, Scala provide APIs and conventions to facilitate "normal" continuation passing programming. Node is built on Javascript and make heavy use of lexical closures. Moreover, many Javascript frameworks provide "promises" — a construct that allows closures to be "chained together" to facilitate many programming patterns [24]. Likewise, Java frameworks like Akka and Netty provide non-blocking APIs built around anonymous classes and futures. Lastly, Haskell's Monads and Scala delimited continuations allow developers to chain callback functions together in a manner similar to promises. These approaches are far simpler to implement than Silo coroutines but can come at the expense of performance and usability

Lastly, many low level techniques for implementing and using coroutines have been proposed over the years [22, 75, 42, 117]. However, most of these are not applicable on the JVM because of restrictions to memory access and control flow.

Chapter 6

Case Study: Building a Real-Time Multiplayer Game

Many new languages, despite incorporating interesting theoretical properties and novel features, are not always useful in practice. Developers are often wary of new languages because it is not always obvious how to use that language in real-world scenarios. This chapter introduces a case study of using Silo to implement a real-time online multiplayer game. The game demonstrates solutions to relevant technical challenges that are commonly found in applications today, including real-time browser communication, HTTP APIs, template rendering, dependency management, and workflow processing.

6.1 Application Overview

To demonstrate Silo's utility for building real-world applications, I created a real-time online multiplayer game called CardStack. CardStack is a version of the popular card game Speed.

In Speed, the objective of the game is to get rid of your cards as fast as possible. Each player is dealt 20 cards of which only 5 can appear in the players hand at a time. Two cards are placed in between the two players forming the start of two "piles". Players can place a card from their hand onto one of the piles if their card is immediately one greater or less than the top card of the pile. Once a player plays a card successfully he can draw from his reserve pile such that he has a maximum of 5 cards in his hand. Most importantly, there are no turns in this game. Players can play cards as fast as they can and whoever can get rid of their cards first wins. Hence the name, "Speed".

CardStack is a browser-based game. Players create a new game and share a unique link to that game with a friend they wish to challenge. Once both players connect, the game begins. A player can select one of their cards using the number keys 1 through 5, re-arrange their cards using the up and down arrows, attempt to place a card onto one of the piles using the left and right keys, and draw new cards using tab. If a player cannot make a move they can hit the space bar which issues a request to the other player. If the other player accepts the request, two random cards are placed onto the middle piles. All moves are seen in real-time by both players making the game fast paced and exciting. A screenshot of CardStack is show in Figure 6-1.

CardStack was chosen as a demo application for Silo because it has several characteristics that are commonly found in real-world use-cases.

- The game is played inside of a standard Web browser, like many apps today.
- The game takes place in real-time and requires clients to be is constant communication with the server, which is increasingly common in modern applications.
- The game logic is not inherently trivial and requires more than the simple

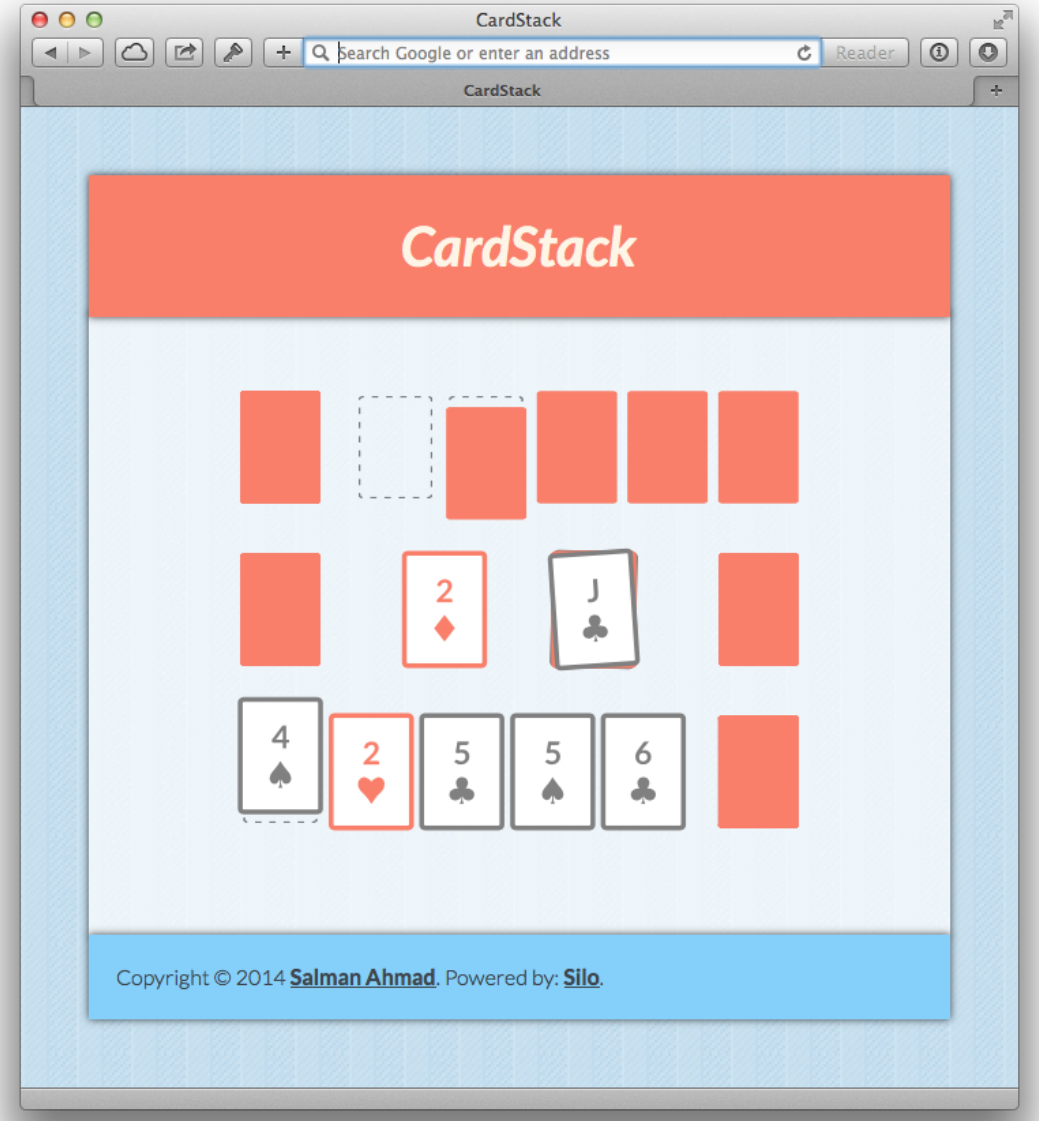

Figure 6-1: A screen shot of CardStack, a multiplayer game written in Silo.

CRUD (create-read-update-delete) logic that is found in most Web applications like blogs and forums.

- The game requires library support for common tasks like processing JSON, parsing HTTP requests, and rendering HTML templates.

In this chapter, I first describe how to architect and organize an application like CardStack using Silo. In the process I describe how many of Silo's features proved useful during CardStack's implementation. Then, I discuss how a game like Silo is deployed in different environments from privately owned dedicated servers to cloud hosting environments like Heroku [44]. Lastly, I reflect on how Silo compares to other languages that I am familiar with and avenues for future work to improve Silo's short comings.

6.2 System Architecture

The CardStack system consists of five types of services: the start up service, a registry service that allows other services to refer to each other using proper names, a "front-end" service which handles HTTP traffic, a "game-manager" service which allows games to be created and manages their life cycle, and finally game "instance" services which are created on-demand and represent individual games. A high level overview of the system architecture is show in Figure 6-2.

6.2.1 Start Up Service

The start up service is fairly straight forward — it is the main entry point of the application that is automatically created and executed by the Silo runtime. The start up service simply spawns the other services and then blocks indefinitely. Many Silo systems will have a start up service that serves a similar function. In some instances the start up service will be more complex, for example, it could connect the machine it is running on to a cluster of machines and spawn the services accordingly.

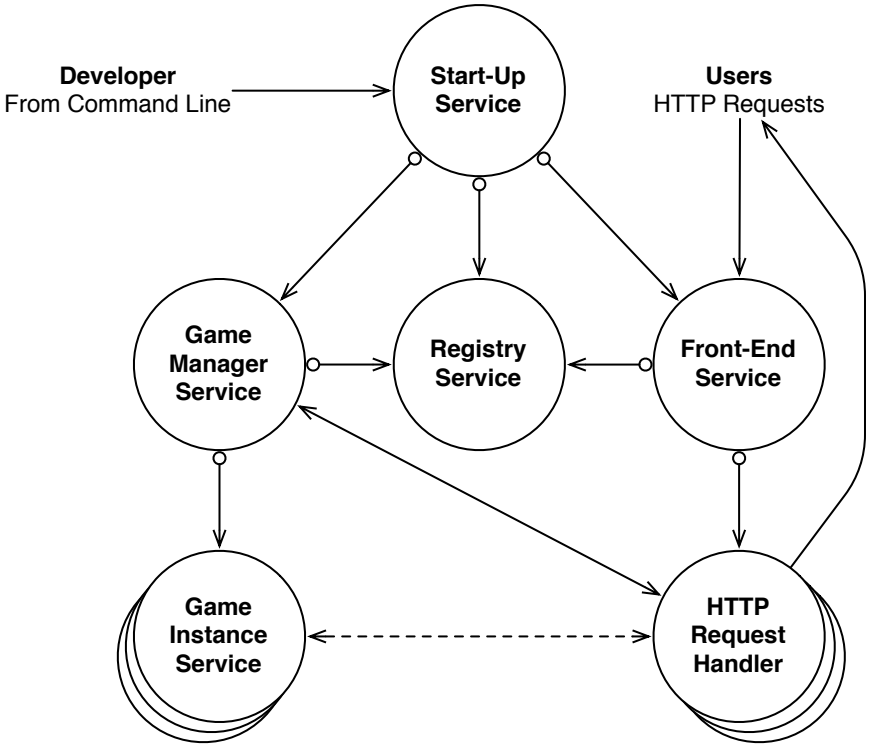

Figure 6-2: CardStack's architecture.

6.2.2 Front-End Service

The front-end service manages all HTTP traffic. It uses Silo's built-in HTTP networking library to open a socket and register a callback function that is used to process HTTP connections. Silo's HTTP library will spawn a new actor that calls the callback function for each HTTP connection. Since each connection is handled by a different actor, developers are free to make blocking calls inside the callback function without blocking the entire server. In fact, CardStack's front-end service can handle thousands of concurrent connections on modest hardware thanks to Silo's efficient implementation of coroutines.

The front-end service, like many popular Web frameworks today, establishes a series of "routes". When an HTTP request access a URI that matches a known route, an associated function is invoked to process that request. Routes are specified using regular expressions and a string to match against the request's HTTP "verb" (e.g. `GET`, `POST`, etc.). Normally, setting up and configuring these routes can be

cumbersome and verbose. However, thanks to Silo's support for macros, I was able to create a special syntactic construct that greatly facilitate managing CardStack's routes.

If no route matches, the front-end server falls back on attempting to serve a static file relative to a "root" location on the file system. Serving static files uses the `sendfile` system call (if available) to send file data without copying data between kernel-space and user-space. Many requests to the front-end service either require server-side templates to be rendered or JSON payloads to be processed. Views are rendered using Mustache templates [21] and JSON payloads are processed using the Java `org.json` library. CardStack's ability to leverage existing Java libraries greatly facilitate implementation.

The front-end service often passes requests onto "back-end" services. For example the `/game/action` and `/game/stream` routes send messages to the game manager and game instances services and wait for a reply from these services that is forwarded back to the client.

6.2.3 Game Manager Service

The game manager service maintains an internal list of active games. The front-end service uses the game manager to create a new game or determine if a game exists or is still active. The game manager also monitors all active games and automatically shuts them down after a certain timeout threshold, which is set to 20 minutes of inactivity.

6.2.4 Game Instance Service

The game instance service models individual games. Thanks to Silo's concurrency model, each game is run as a separate actor which greatly facilitate implementation and making changes. The game instance service is essentially a state machine that moves to different states in response to incoming actions. This allowed the game logic to be represented in a natural manner rather than having to re-structure the

application around the server's HTTP event loop (like you would in Node.js, for example).

CardStack implements the real-time features using HTTP long polling and a log-oriented architecture. Every game action, for example selecting a card, is assigned a sequence number and appended to a log. The game clients connect to the game instance service (through the front-end service) and request all log entries from a certain starting sequence number. The instance service replies back to the client (again, through the front-end service). However, if the client requests a log entry that hasn't occurred yet, the game instance manager adds the client to a waiting list. Note that this means the front-end service will need to "hold on" to the HTTP connection for a long time, hence the name "long polling". However, once again, thanks to Silo's actor implementation, this does not mean that the front-end service is incapable of handling other requests in the mean time. Once a new entry is available, the game instance appends it to the log and notifies the front-end service, which sends the data back to all waiting clients and immediately closes the connection. The client processes the log entires, increments its sequence number and immediately issue a new request for new entires.

The game instance service maintains the log and other game state in memory and receives actions from the front-end service. Since CardStack is real-time, the game instance service stores all data in memory as using a database would likely introduce unnecessary latency.

CardStack's game logic is implemented as a "dumb client" application in which the clients simply send messages to the server and respond to messages as they are appended to the log. All of the actual game logic and rules are implemented and coordinated from the server. This is quite common for most online multiplayer games.

6.2.5 Client Side Code

While CardStack's game logic takes place on the server, it does incorporate a fair amount of client-side code for rendering the UI, playing animations, and handling input. CardStack's client-side code make heavy use of Javascript and SVG and this

code is located across various Javascript files, image files, and HTML template files that are served as static files by the front-end service.

Javascript is used for capturing user-input, sending these inputs to the server, and responding to actions that are appended to the game's log. The CardStack's core Javascript functionality is thus fairly simple and consists of only around 300 lines of code.

The game is rendered almost entirely using SVG, a vector-based image format. The game board is a pre-rendered SVG image made in Adobe Illustrator and has placeholders with unique identifiers for important locations on the board. For example, the board has a hidden placeholder box labeled `player-card-active-1` which is where the player's first card should be positioned. During game play, the client-side Javascript will place cards and render animations relative to the locations indicated by these placeholders. As such, the client logic is void of any hard-coded positional constants and the look, feel, and placement of game assets is dictated by the pre-rendered SVG board image.

6.3 Code Review

6.3.1 Directory Organization

CardStack's repository's organization is shown below.

- `app.silo` This file is the main entry point of the application. This file includes all of the other Silo source files so to compile and run CardStack, all that is needed is to execute `silo app.silo` from the command line.
- `cardstack/` This directory holds all Silo source files for CardStack. All of CardStack's constructs live in the `cardstack` package and the general convention is to organize source files according to the package that they contain.
  - `cardstack/front-end.silo` Contains the implementation of the front-end service.

  - `cardstack/manager.silo` Contains the implementation of the game manager service.
  - `cardstack/game.silo` Contains the implementation of the game instance service.
  - `cardstack/util.silo` Contains random helper functions and types.

- `lib/` This directory hold all dependencies that CardStack relies on. In this case, two jar files exists: the standard Java JSON library and a Java template library called Mustache. The `silo` command line utility automatically includes the `lib` directory as part of the `CLASSPATH` when it executes a Silo program so no further work is needed by the developer beyond just copying the necessary jar files into this location. As Silo matures, more advanced build tools will likely be created so that developers do not need to manually download jar files but rather use a system that downloads dependencies automatically.

- `views/` This directory holds all Mustache templates that CardStack needs to render. Following the convention found in many other Web frameworks, the `views` directory contains a special `_default.html` file which is a layout template that is used to maintain consistency across all pages on the site. Other files, for example `index.html` are templates for individual pages that are inserted into a layout template to yield to final HTML content that is sent back to the client.

- `static/` This directory holds all static assets like images, JavaScript files, CSS stylesheets, etc. All files that are placed in the `static` directory are publicly accessible.

6.3.2 Code Patterns

CardStack's codebase is organized around Silo packages. This is different from many mainstream languages that organize code around types or objects. In CardStack, each service lives in its own package. For example, all of the functions, types, and macros for the front-end service are contained within the `cardstack.frontend` package. A

notable exception is the `cardstack.util` package which does not logically contain a service but rather a set of useful functions.

All service packages, by convention, have a function called `start` which enters the service's main execution loop in which the service waits for and processes messages as they arrive. The structure of the loops generally looks like this:

```
1.  type(Request {
2.      sender : String
3.      header : String
4.      body : Object
5.  })
6.
7.  type(Response {
8.      success : boolean
9.      header : String
10.     body : Object
11. })
12.
13. func(start() {
14.     ...
15.     while(true {
16.         message : Object = actor.read()
17.         if(message | instanceof(Request) {
18.             request : Request = message | checkcast(Request)
19.             if(request.header == "game.new" {
20.                 ...
21.             } else {
22.                 // Ignore
23.             })
24.         })
25.     })
```

```
26. })
```

Note that the server loop always calls `actor.read` instead of selectively waiting for particular types of messages. This is so the actor's inbox does not fill up. During the course of execution, it is common for old messages to be left behind (e.g. timeout messages from old timers) and it is important to clean these messages up. This can lead to undesirable situations because it would mean that pre-existing messages on the inbox are lost. Thus, the `start` function is usually not invoked directly but `spawned` as a new actor. In CardStack, all of the service's `start` functions are `spawned` in the `app.silo` source file.

The majority of the CardStack resides in the `cardstack.game` package which implements the actual game logic. The game is modeled as an event loop that processes game actions as they occur. A portion of the main game loop is shown below:

```
1.  type(Game {
2.      log : Vector
3.      sequenceNumber : int
4.      startingDeck : Vector
5.      ...
6.  })
7.
8.  func(create(null => Game) {
9.      // Creates a new game
10.     ...
11. })
12.
13. func(dispatchAction(game : Game, request : Request => Game) {
14.     // Process the request
15.     // Return a new game with the updated state
16.     ...
17. })
```

```
18.
19.  func(start() {
20.      println("CardStack - Starting Game (" + actor.self() + ")")
21.      game : Game = create()
22.      ...
23.      while(true {
24.          message : Object = actor.read()
25.          if(message | instanceof(Request) {
26.              request : Request = message | checkcast(Request)
27.
28.              if(request.header == "game.action" {
29.                  game = dispatchAction(game, request)
30.              } else(...) {
31.                  ...
32.              } else {
33.                  // Ignore
34.              })
35.          })
36.      })
37.      ...
38.  })
```

There are a couple of important things to note about the code.

- First, all of the game state is encapsulated in a custom structure of type `Game`. Since types in Silo are immutable, the functions that implement certain actions cannot mutate the `Game` instance directly, but rather return a copy of the game with the necessary modifications. Hence, the code above has to re-assign the `game` variable to the return value from each function call. This approach is different from traditional object-oriented programming which encourages mutating objects.

- Second, the code is surprisingly void of synchronization primitives. Despite the fact that the game is real-time and multiple games can be taking place concurrently, each game instance is implicitly free of race conditions and other unpleasantries of multi-threaded programming because the semantics of actors in Silo dictate that they process a single message at a time. Thus, synchronization does not need to be explicitly coordinated by user-level code. Instead, developers can write code in a straight forward and natural manner without interspersing application logic with awkward synchronization or control flow constructs that are common to concurrent programming in other languages.

Lastly, CardStack makes use of a couple of macros that facilitate recurring tasks. The best example of this is processing HTTP requests. A snippet of the main processing loop from the front-end service is shown below:

```
1.  func(httpHandler(r : Request, c : Connection, options : Map) {
2.      ...
3.      is(r, "post", '/game/([\w-]*)/action' {
4.          try({
5.              pattern : Pattern = Pattern.compile('/game/([\w-]*)/action')
6.              matcher : Matcher = pattern#matcher(r.uri)#matches()
7.              gameId : String = matcher#group(1)
8.
9.              request : Map = fromJson(body)
10.             actor.send(gameId, Request(actor.self(), "game.action", request))
11.             response : Response = actor.read() | checkcast(Response)
12.
13.             if(response.success {
14.                 connection.writeAll(c, 200, toJson(map.create("success" true)))
15.             } else {
16.                 connection.writeAll(c, 500, toJson(map.create("success" false)))
17.             })
```

```
18.          } catch(e : Exception) {
19.              connection.writeAll(c, 400, null, "HTTP 400 Bad Request.")
20.          })
21.      })
22.      ...
23. })
```

In this code snippet, the `is` construct is a simple macro that makes handling HTTP requests drastically more natural. Its implementation is shown below:

```
1.  transform(is(request, m, path, body) {
2.      if(request.method#equalsIgnoreCase(m) &&
3.         Pattern.matches(path, request.uri) {
4.          body
5.          return()
6.      })
7.  })
```

Macros like `is` are quite common in Silo applications. They are simple syntactic constructs that are easy to understand, implement, and use. They make code more readable, maintainable, and succinct. Small constructs like `is` are perfect examples of good use cases for macros. While macros are extremely powerful, they can often be hard to debug. Thus, developers are encouraged to keep macros simple and straight forward, like CardStack's `is`.

6.4 Deploying

CardStack has been deployed in two production environments: using dedicated private servers and a cloud-based application platform. The term "cloud-based" may be confusing. By "dedicated private server" I mean a box that you have complete control over. In this case, it was a physical box running in a server closet but it could just as easily been a virtualized server running on Amazon's or Rackspace's

cloud. By "cloud-based application platform" I mean a service similar to Heroku [44] in which developers do not have access to the actual box or underlying OS but rather are given higher-level interfaces and tools for creating and managing applications. Generally, these services require developer to simply upload their code, specify certain parameters such as the number and types of machine instances to use, and the platform will take care of the rest. While these services are far more restrictive (for example, Heroku does not allow application code to open arbitrary ports and applications cannot save data directly to the file system) they tend to be easier to use and offer different price points that are attractive to many smaller organizations as opposed to the full-blown cloud computing solutions from the likes of Amazon.

6.4.1 Dedicated Private Servers

CardStack was first deployed on a privately owned server that was provisioned in a typical networking server closet. The box had a clean installation of Ubuntu, a popular Linux-based operating system. Installing Silo required no special packages or system-wide configuration. The standard Silo distribution package was simply copied to the server and the JRE (Java Runtime Environment) was downloaded from Oracle's Web site.

The distribution package contains a `jar` file with all of Silo's runtime environment, standard libraries, and dependencies. The distribution package also contains a helpful `silo.sh` file (symlinked as just `silo`) which is a convenience script and "front-end" into the `jar` file (using a small start-up shell script is common in most JVM languages to configure JVM parameters like the `CLASSPATH`).

The JRE directory was simply copied into the Silo distribution directory as opposed to performing a system-wide installation using the OS's package manager. By default the `silo.sh` script will attempt to use a system-wide installation of Java but if none is found, it will also see if a JRE is present locally within the distribution directory. This is quite convenient because it allow the entirety of Silo's installation to be cleanly located within a single directory with no other system-level dependencies, which is particularly helpful when deploying service-oriented systems which often

requires provisioning and configuring multiple machines.

To run a Silo program, usually developers will execute `silo some-file.silo` on the command line. While this works well for development and testing, it can be awkward to use in production. Most of the time developers connect to production machines remotely using SSH. The problem is that if developers run `silo file-some.silo` and then close the SSH connection, the child-processes of the SSH process will be terminated by the OS, which includes the Silo program. There are many work arounds to this but most of them require platform-specific functionality or heavy-weight solutions like using the `screen` command.

Silo provides a feature out of the box for running programs in the background such that they do not terminate when the SSH connection is closed and developers can re-connect to the running program if they need to. All that is needed is to run `silo service start some-file.silo`. The command will terminate immediately but will leave the Silo programming running in the background. It redirect's the program's output and stores its process identifier to a "service directory" that is named after the source file, for example `some-file.service`. To "connect" to the running program and see its output, developers can run `silo service monitor some-file.silo`. Finally, to stop the service, developers can run `silo service stop some-file.silo`. This small convenience takes away a lot of the pain when managing long-living systems running on multiple remote machines.

6.4.2 Cloud-Based Servers

CardStack was also successfully deployed on Heroku, a cloud-based application platform. Heroku uses a Git-based workflow in which developers push their code to a repository hosted on Heroku's servers. When code is pushed, Heroku automatically configures and runs the application. Developers are given a small but useful set of controls to customize the behavior of their application including how many "servers" to run the application.

Heroku supports a wide variety of different programming languages and frameworks, including Ruby, Javascript (Node), Python, Java, and others. Unfortunately,

it does not have explicit support for Silo. However, it is possible to embed the Silo runtime and thus run Silo applications from inside another a "thin" Java application. Thus, I exploit Heroku's support for Java to run CardStack on their platform. This also means that Silo can be easily used on any other platform and hosting environments that also support Java. Given Java's wide spread adoption and appeal this opens many options for Silo developers. Moreover, this also relevant to Silo's core design philosophy which was adamant about providing an efficient and easy to use concurrency model without altering the JVM. If Silo did require changes to the JVM it would be unlikely that Silo applications could be run on platforms like Heroku, which tend to use standard installations of Java and the JVM.

Running a Silo application on Heroku is fairly straight forward. All application dependencies are managed using the Maven build tool, which is common for Java-based applications and simple to configure for Silo. In fact, future versions of Silo will likely interoperate with Maven with ease just like build tools in other JVM-based languages including Lein for Clojure, SBT for Scala, and Graddle for Groovy. Heroku uses the concept for a `Procfile` that dictates which "processes" are created. The `Procfile` for a Silo application looks like this:

```
web:    java $JAVA_OPTS                                \
            -cp target/classes:target/dependency/* \
             silo.lang.Main app.silo
```

Once this `Procfile` was created, CardStack was pushed to Heroku and the application ran without issue.

6.5 Comparative Evaluation and Lessons Learned

When I reflect on my experience with Silo, especially compared to other languages and frameworks with which I am familiar, several thoughts come to mind.

6.5.1 Positive Outcomes

Overall, my experience with Silo, despite being a new language, was mostly positive.

- **Few External Dependencies**. CardStack did not require any external or system-level infrastructure. All of the application's logic is cleanly self-contained within the application's source code and there is a single point of entry. In particularly, CardStack did not require a standalone Web server like Nginx or Apache like many Ruby, Python, and Javascript applications. Additionally, CardStack does not need to reply on external OS features like cronjobs to perform recurring clean up tasks. All aspects of the system were modeled directly in the application.

- **Light Weight Frameworks**. Many frameworks in languages like Ruby or Python tend to be "heavy-weight" and require applications to be written according to framework-level conventions or idioms. In Silo I did not feel the need to have a heavy-weight Web framework. Despite being fairly complex, CardStack did not require special purpose libraries for things like real-time communication, HTTP handling, or recurring tasks. Which is quite common in together languages and frameworks. For example, in Ruby, real-time communication will typically require the use of Faye, HTTP handling will require a Web framework like Rails or Sinatra, and recurring tasks will require a job queue like Resque.

- **Simple Build Process**. Silo feels similar to a scripting language and does not require complex build tools. To compile and run CardStack, only a single command is needed: `silo app.silo`.

- **Performance**. When building CardStack, I never felt held back because of Silo's performance. This is especially true with Silo's string processing (for rendering HTML templates) and networking handling. Most of the time, applications written in languages like Ruby, Python, and Javascript use a separate server like Nginx or Apache to host static files because the application servers

written in, for example, Ruby are rarely performant. With Silo, this was not necessary. Static files were hosted directly by CardStack's application server written in Silo and performance was never a reason. In fact, Silo's HTTP performance often matches that of Nginx and Apache and uses best practices like the `sendfile` system call to minimizing buffer copying.

- **Easy Installation**. Installing Silo and running CardStack on multiple machines was simple. No system-wide installation or configuration was necessary and no other dependencies were needed beyond a default installation of any mainstream operating system (yes, this includes Windows). In particular, it is important to note that Silo also does not require the typical dev tools (like `gcc` and development header files) which are commonly required by languages like Python and Ruby to install libraries that use C extensions to achieve reasonable performance.

- **Java Interoperability**. Silo's ability to interoperate with Java libraries was essential for CardStack. In particular, CardStack uses the standard Java libraries for JSON and Mustache templates that would have been time consuming and difficult to re-implement.

6.5.2 Negative Outcomes

There were other points where Silo appeared in need for improvement.

- **Type System**. A common occurrence in CardStack was heavy use of the `checkcast` special form which casts one type to another. Many of these instances would be avoidable with a more powerful type system the incorporates generics. This is useful not only for re-usable data structures like Vector and Map but also utility functions and descriptive APIs.

- **Slow Compilation**. While Silo's runtime performance was quite fast, its compilation time leaves much to be desired. This was particularly annoying during

CardStack's development when the application was frequently stopped and re-run. The slow compilation time is mostly attributable to little effort spent optimizing the compiler and its reliance on Java reflection. Moreover, an unnecessarily large amount of time is actually spent in the parser, which was implemented using a standard parser generator rather being hand coded like most languages.

- **Library Support**. While Silo's interoperability was particularly useful, the library support are less than ideal. Java libraries have APIs that feel unidiomatic.

- **Immutability**. Silo's reliance on immutability has many beneficial properties but it does require writing code in a manner that I, who previously was most familiar with object-oriented programming languages, found awkward. Thinking in a "functional" manner requires subtly different approaches to many programming tasks and takes some getting used to. For example, the game instance service has many functions that manipulate the game's state. In an object-oriented programming language, these function would likely accept an argument of type `Game` that is mutated directly. In Silo, the approach was to re-write these function to accept an argument of type `Game` and return a new `Game` as output. While Silo's approach in retrospect is obvious (and in fact has many beneficial properties like being able to "roll-back" to a previous game state in the event of an exception) it did require me to make a mental shift in how I would naturally approach the problem.

Chapter 7

Case Study: Renovating a Legacy System

Software systems are subject to many corrosive forces over time and in need of regular maintenance including applying software updates, upgrades and changes to hardware configuration, and responding to bug fixes and feature requests. Developers often need to be mindful of not only the upfront development cost of a system but also the maintenance costs over time. This chapter presents a case study of how Silo was used to renovate a long running legacy system. Silo provides elegant ways of addressing the existing issues of this system while also providing mechanisms that would have prevented many of these issues to begin with.

7.1 Application Overview

WeFeelFine is an interactive art piece that continually crawls the Web for the phrases that begin with "I feel" [58]. These phrases are inserted into a database, exposed by an API, and visualized in a playful app running in the browser as a Java applet.

Ever since it was first launched in 2006, WeFeelFine has been subjected to many corrosive forces and slowly but surely many parts of the system's backend began to fail. To breath new life into the project, WeFeelFine's backend infrastructure was re-implemented in Silo. This chapter describes the approach taken with Silo.

7.2 Previous Architecture

WeFeelFine's original architecture consistent of many different components. A critical part was the crawler that was implemented as a series of Perl scripts that executed periodically as cronjobs. These crawlers would extract information from the Web, insert relational data into a MySQL database, and save images to an NFS volume. The MySQL data was queried by a series of Java servlets running on Apache Tomcat as part of the API. Lastly, an Apache httpd server was used to serve static files as well as the images for the API.

7.3 Issues

While WeFeelFine's architecture began simple and clean, it evolved to be more complex and eventually began to start breaking.

First, as the site grew, a single machine was unable to keep up with the load. Thus, multiple Apache servers were deployed to handle the traffic. However, the images from the crawl proved too large to replicate on each box. Thus, an NFS server was introduced and mounted by all Web servers. While this would work in theory it required careful operating system configuration to mount and use the NFS volume. Additionally, network lag and failures would often cause the NFS clients to get disconnected, which would cause the site to temporarily go down and require

manual intervention. Lastly, to ensure proper security, the NFS volume was hosted on a private subnet on virtualized network adapters which required configuration changes to low-level parts of the OS network stack. Keeping the NFS infrastructure up, running, and healthy required constant maintenance and changes in many different places.

Second, as the data set of the crawl grew, it became necessary to shard the MySQL database into multiple different tables. To do this, a new table was created for each month. As can be imagined, this required major changes to the code. Moreover, since the database was used in multiple places (by the crawler, by the Java servlets, and by various utility scripts) significant updates were needed in many different locations. Additionally, new infrastructure was necessary to automatically create the new tables at specific times. This was done using cronjobs.

Third, eventually WeFeelFine needed to be moved from one data center to another. This caused many failures. Much of the system depended on features of the old infrastructure that were no longer available, for example, load balancing capabilities, network address names, DNS entries, virtualized private LANs, etc. Moving the system cause many parts of the the system to simply stop working. A huge amount of time was necessary to rebuild this infrastructure and this proved difficult since key information was scattered across various configuration files buried in many system locations. Moreover, the system was moved onto new hardware running an updated operating system. This caused problems with the crawlers since they relied on an outdated version of Perl that was no longer available. Additionally, many of the cronjobs stopped working because of subtle OS differences and many of the deployment scripts needed to be re-written.

While many of these issues could have been prevent with better planning, the fact of the matter is that these issues are quite common in practice.

7.4 Updated Architecture

WeFeelFine' infrastructure was re-created in Silo as part of an effort to fix many of the outstanding issues.

The following services were created:

- **Start-Up**. The service that starts all of the other services and the main starting point of the system.
- **Static Web Service**. Hosts the static applet Web site HTML files. Multiple instances of the Static Web Service run in parallel.
- **API Service**. Hosts the HTTP API that was previously running under Tomcat. However, unlike the Tomcat version, the images are hosted directly from the API service rather than being sent by Apache. Many instances of the API Service run in parallel.
- **Load Balancer**. Distributes traffic on several instances of the API service and static Web service.
- **Crawler**. Periodically "wakes up" and creates and monitors many instances of crawlers.
- **Feelings Service**. Expose an interface for creating and querying feelings. The feelings service is used by the crawlers for inserting new feelings as well as the API service. The feelings service uses MySQL behind the scenes but abstracts all MySQL operations behind a unified interface that can be shared by the crawler and the API.
- **Image Service**. Exposes an interface saving and reading raw image files. This service is used primarily by the feelings service but was split off as its own service to handle the shear size of the image data set as well as performing image manipulation tasks like thumbnail generation, compression, and cropping.

7.5 Discussion

The main issues with the old version of WeFeelFine was that it had many moving parts that were glued together and that were hard to change. This is actually a major challenge with many software system. The updated version with Silo not only addressed the current issues but in many ways would have prevented them from happening.

With Silo, all of the components of WeFeelFine lived inside of a single program and did not rely on external infrastructure like cronjobs, NFS volumes, and multiple standalone servers from Tomcat to Apache httpd. Moreover, the all of these components were tied together in an adaptive manner. The old architecture "hard wired" many parts of the system (for example, the IP address of the machines) which made migrating the application difficult. However, the Silo version could be moved to another machine and data center as easily as copying the application source and running it. There are still some configuration changes that would be necessary but these configuration properties exist in a single place within the application source tree rather than being scattered in various locations within the operating system.

Silo also allowed WeFeelFine to perform higher-level abstraction of re-usable functionality. As a result, it facilitates making changes to the system and greater flexibility with decision making. For example, the Feelings service still uses MySQL like before, however it exposes a much higher API than what MySQL offers. Not only does this make it easier for clients (e.g. the API service and the crawler service) to use but it also gives the freedom to switch to another storage engine down the road. Moreover, changes made to the MySQL installation (e.g. sharding tables) can be performed by just updating the feelings service and not having to update every part of the system that accesses MySQL. Similarly, the image service has the flexibility of choosing the right storage technology for images, be that NFS, a key-value store, or a NoSQL database. The flexibility and re-usability of services also allows a piece-meal approach to software development. For example, the updated version of the crawler for WeFeelFine still uses the old Perl code base by forking a new process Perl process when

it is run. While this code will eventually be ported over to Silo natively, doing so would have been somewhat time consuming. Thus, for the time being, the crawlers re-use portions of the old Perl code. Once the new implementation is ready, it can be "dropped" in place of the Perl code without changes anywhere else in the system because the external interface provided by the crawler has not changed.

Lastly, Silo afforded great flexibility during the development process. For example, during the implementation of the Feelings services, it became obvious that much of its complexity involved handling the large number of images in the system. Thus, at that point, the Feelings service was split into two: the Feelings service and the Image service. The ability to easily break up complex services with ease not only makes initial development easier, but it also discourages "hacky" solutions and facilitates maintenance down the road.

By reducing the number of moving parts of the system, encapsulating complex multi-machine systems into a single program, and providing greater flexibility and opportunities for abstractions, a service-oriented language like Silo can reduce the amount of cruft that is built up by a system over time.

Chapter 8

Future Work

While Silo represents and usable and working language there are numerous avenues for future work.

8.1 Language Enhancements

Silo's initial design emphasized simplicity and orthogonality of language features. As a result, certain features were avoided until the language was more mature. Of note, generic programming capabilities and a richer type system are notably missing. Silo's type system provides little protection and expressivity when creating APIs. There are many ideas being explored with languages like Haskell and Scala that are particularly interesting for inclusion with Silo. In particular, it would be interesting to be able to create a type system that provides strong type safety in the presence of unknown messages that arrive on an actor's inbox.

8.2 Development Tools

Beyond the Silo command line tools (the compiler, REPL, and simple package manager) and a handful of text editor plugins, there is minimal tooling support for Silo. While things like providing plugin for commons IDEs are naturally needed, there are other unsolved research questions that need to be tackled, especially in the design of

debuggers. Most debuggers work well with programs executing on a single machine and that have simple control flows. However, Silo runs on multiple machines and the control sequence leading to a bug can be hard to determine due to pervasiveness of asynchronous message passing. Providing a mechanism to more easily identify and reproduce bugs in a distributed program is currently an open research question.

Additionally, Silo programming model requires new thinking around testing. Simulating failures, network lag, temporary crashes, and byzantine faults are interesting ideas to explore in the design of future test frameworks. Additionally, exploring ways to perform automated testing on a distributed system could be very valuable in practice.

8.3 Static Analysis

Silo offers a fundamentally different programming model compared to many mainstream languages. As a result, it opens the door to many types of errors and defects that developers are currently uncommon and not well known. Exploring static analysis techniques for identifying and fixing bugs could be hugely beneficial. Many existing static analysis techniques could be extended to address the semantics of Silo's unique runtime model. In particular, it would be interesting to see how static analysis techniques can account for the possibility of a remote machine crashing and coming back online, messages being indefinitely delayed, and unbounded non-determinism of Silo's actor semantics. Moreover, as privacy and network security continue to be major points of interest, it would be existing to explore and be able to prove security properties of protocols and programs written in Silo.

8.4 Use Cases

This thesis focused on Silo for the creation of, primarily, Web-based systems. However, Silo distribution and concurrency models are useful in many other domains as well. In particular, it would be interesting to explore how Silo could fare as a language

for massively parallel hardware like stream processors on GPUs. The actor semantics should still be preserved in that environment and performance concerns could be addressed using JNI and compiling a Silo-based DSL to Cilk or OpenCL code.

It would also be interesting to see how Silo fares as a language for client-side development. Many of the challenges of server-side programming are shared with GUI programming as well, for example, managing event-loops. It would be interesting to explore for to create an actor-based UI library. This would open exciting opportunities where a single Silo program can model a server-side service that supports client-side apps running on the user's local machine. Along these lines, it would be interesting to explore compiling Silo to platforms other than the JVM to allow interoperability with native UI frameworks. For example, compiling Silo to CLR on Windows, LLVM for iOS and OS X, and Dalvik for Android.

Lastly, Silo could work really well as a research prototyping language for new network protocols.

Chapter 9

Conclusion

Service oriented programming has become increasingly relevant in recent years and all indications show that it will be even more important in the future. In fact, I would go so far as to say that service-oriented programming will eventually replace object-oriented programming as the dominant programming paradigm.

In this dissertation we have explored the benefits and challenges of service-oriented systems and seen how a new programming language, Silo, can streamline and facilitate the design and implementation of such systems. Silo takes what I feel is a sensible approach to language design and balances solutions to theoretical challenges while addressing pragmatic needs of software developers. It provides a unified distributed programming model, an easy-to-use construct for handling high concurrent workloads, an extensible syntax, and the ability to natively interoperate with a rich ecosystem of libraries and tools.

Over the years programming languages have focused on different themes: resource management, concurrency, typing disciplines, expressivity, safety, etc. It is my hope that distributed programming is soon added to that list. More than anything, Silo provides a metaphor for distributed programming that emphasizes location transparency and the ability to seamlessly execute a single logical program across many independent computational devices. As multicore and streaming processors become critical for achieving performance, and as compute clusters become essential for delivering scalability and reliability, and as end-users' computing environments become

more heterogeneous and distributed, such metaphors are essential for the builders of tomorrow's software systems.

Bibliography

[1] The Responsive Manifesto. `http://www.reactivemanifesto.org/`, 2014.

[2] 10Gen Inc. MongoDB, 2013.

[3] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[4] Gul Abdulnabi Agha. Actors: A model of concurrent computation in distributed systems. 1985.

[5] Ricardo Alonso-Zaldivar. HealthCare.gov marred by technical problems on signup deadline day. *PBS News Hour*, 2014.

[6] Amazon, Inc. Simple Queue Service. `http://aws.amazon.com/sqs/`, 2014.

[7] Marc Andreesen. Digg's traffic is collapsing at home and abroad. *The Wall Street Journal*, 2011.

[8] Apache. Ant. `http://ant.apache.org`, 2013.

[9] Apple, Inc. Creating XPC Services, 2014.

[10] Greg Orzell Ariel Tseitlin. Netflix Operations: Part I, Going Distributed. `http://techblog.netflix.com/2012/06/netflix-operations-part-i-going.html`, 2012.

[11] Joe Armstrong, Robert Virding, Claes Wikstr, Mike Williams, et al. Concurrent programming in erlang. 1996.

[12] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the jalapeno jvm. In *ACM SIGPLAN Notices*, volume 35, pages 47–65. ACM, 2000.

[13] Michael Arrington. Twitter Said To Be Abandoning Ruby on Rails. *TechCrunch*, 2008.

[14] Phil Bagwell. Ideal Hash Trees. *Es Grands Champs*, 1195, 2001.

[15] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[16] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[17] Grady Booch. *Object Oriented Analysis & Design with Application*. Pearson Education India, 2006.

[18] Per Bothner. Kawa: Compiling Dynamic Languages to the Java VM. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, 1998.

[19] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[20] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.

[21] Chris Wanstrath. Mustache: Logic-less Templates. `http://mustache.github.io/`, 2014.

[22] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[23] Jeremy Cloud. Decomposing Twitter: Adventures in Service-Oriented Architecture. `http://www.infoq.com/presentations/twitter-soa`, 2013.

[24] Common JS Spec. Promises/A. `http://wiki.commonjs.org/wiki/Promises/A`, 2014.

[25] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[26] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[27] Ariel Tseitlin Cory Bennett. Chaos Monkey Released Into The Wild. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`, 2012.

[28] Douglas Crockford. The application/json media type for javascript object notation (json). `http://tools.ietf.org/html/rfc4627.txt`, 2006.

[29] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

[30] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.

[31] Dan Kegel. The C10K problem. `http://www.kegel.com/c10k.html`, 2014.

[32] David Wheeler. Sweet-expressions: A suite of readable formats for Lisp-like languages. `http://www.dwheeler.com/readable/sweet-expressions.html`, 2006.

[33] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121. ACM, 1986.

[34] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly, 2008.

[35] Matthew Flatt. Creating languages in racket. *Commun. ACM*, 55(1):48–56, January 2012.

[36] The Mozilla Foundation. The rust programming language. `http://www.rust-lang.org`.

[37] Steven E Ganz, Daniel P Friedman, and Mitchell Wand. Trampolined style. In *ACM SIGPLAN Notices*, volume 34, pages 18–27. ACM, 1999.

[38] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[39] Google, Inc. Dalvik. `https://code.google.com/p/dalvik`, 2014.

[40] Christopher Grant. SimCity launch crippled by server issues. *Polygon*, 2013.

[41] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. Ac: composable asynchronous io for native languages. *ACM SIGPLAN Notices*, 46(10):903–920, 2011.

[42] Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298. ACM, 1984.

[43] Hao He. What is service-oriented architecture. `http://www.nmis.isti.cnr.it/casarosa/SIA/readings/SOA_Introduction.pdf`.

[44] Heroku, Inc. Heroku Cloud Application Platform. https://www.heroku.com/, 2014.

[45] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[46] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 Symposium on Dynamic languages*. ACM, 2008.

[47] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[48] Michael N Huhns and Munindar P Singh. Service-oriented computing: Key concepts and principles. *Internet Computing, IEEE*, 9(1):75–81, 2005.

[49] Chris Hunt. Service Oriented Architecture at Square. http://www.confreaks.com/videos/1273-rubyconf2012-service-oriented-architecture-at-square, 2008.

[50] iMatix Corp. Zeromq. http://zeromq.org.

[51] Amazon Inc. Amazon Web Services. https://aws.amazon.com, 2014.

[52] Microsoft Inc. Windows Azure. http://azure.microsoft.com/en-us/, 2014.

[53] INRIA Inc. The Caml Language. http://caml.inria.fr.

[54] Jan Kneschke. lighttpd. http://www.lighttpd.net, 2014.

[55] JetBrains. Kotlin. http://kotlin.jetbrains.org.

[56] John Rose, Brian Goetz, and Guy Steele. State of the Values. http://cr.openjdk.java.net/~jrose/values/values-0.html, 2014.

[57] Joyent, Inc. Node.js. http://nodejs.org, 2013.

[58] Sepandar D Kamvar and Jonathan Harris. We feel fine and searching the emotional web. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 117–126. ACM, 2011.

[59] Samuel C Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. 1994.

[60] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *Formal techniques for Distributed Systems*, pages 1–25. Springer, 2009.

[61] Stephen G Kochan. *Programming in Objective-C*. Addison-Wesley Professional, 2011.

[62] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM.

[63] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.

[64] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[65] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in clu. *Communications of the ACM*, 20(8):564–576, 1977.

[66] Lori MacVittie. *XAML in a Nutshell*. O'reilly, 2006.

[67] Matthias Mann. Continuations Library. `http://www.matthiasmann.de/content/view/24/26/`, 2014.

[68] Griffin McElroy. 'Diablo 3' launch: The bugs, the errors and the fixes. *Polygon*, 2012.

[69] Daniel A Menasce. Web server software architectures. *IEEE Internet Computing*, 7(6):78–81, 2003.

[70] Microsoft Corporation. Asynchronous Programming with Async and Await (C# and Visual Basic). `http://msdn.microsoft.com/en-us/library/hh191443.aspx`, 2014.

[71] Microsoft Corporation. Internet Information Services. `http://www.iis.net`, 2014.

[72] Microsoft Corporation. Microsoft Axum. `http://social.technet.microsoft.com/wiki/contents/articles/6615.microsoft-axum-formerly-maestro.aspx`, 2014.

[73] Mikhail Vorontsov. Throwing an exception in Java is very slow. 2014.

[74] Shivakant Mishra and Rongguang Yang. Thread-based vs event-based implementation of a group communication service. In *Parallel Processing Symposium, International*, pages 0398–0398. IEEE Computer Society, 1998.

[75] Ken Moody and Martin Richards. A coroutine mechanism for bcpl. *Software: Practice and Experience*, 10(10):765–771, 1980.

[76] C. N. Mooers and L. P. Deutsch. Programming languages for non-numeric processing: Trac, a text handling language. In *Proceedings of the 1965 20th national conference*, ACM '65, pages 229–246, New York, NY, USA, 1965. ACM. Chairman-Floyd, R. W.

[77] Adam Nathan. *Windows presentation foundation unleashed*. Sams Publishing, 2006.

[78] Norman Maurer. Inline all the Things. `http://normanmaurer.me/blog_in_progress/2013/11/07/Inline-all-the-Things/`, 2013.

[79] Charles Nutter. The jruby project. `http://jruby.org`, 2013.

[80] Martin Odersky. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[81] OpenJDK. StackContinuations. `https://wikis.oracle.com/display/mlvm/StackContinuations`, 2014.

[82] Inc. Oracle. ForkJoinPool (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ForkJoinPool.html`, 2014.

[83] Oracle, Inc. Oracle Application Development Framework. `http://www.oracle.com/technetwork/developer-tools/adf/overview/index.html`, 2014.

[84] John Ousterhout. Why Threads Are A Bad Idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, 1996.

[85] John K Ousterhout and Ken Jones. *Tcl and the Tk toolkit*, volume 227. Addison-Wesley Reading, MA, 1994.

[86] Vivek S Paiz, Peter Druschely, and Willy Zwaenepoely. Flash: An efficient and portable web server. 1999.

[87] Mike P Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[88] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience*, pages 789–810, 1994.

[89] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[90] Rob Pike. The go programming language, 2009.

[91] Pivotal Software, Inc. RabbitMQ. `http://www.rabbitmq.com`.

[92] Plataformatec. Elixir. http://elixir-lang.org, 2013.

[93] PLT Design Inc. The Racket Language. http://racket-lang.org.

[94] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for java. In *Compiler Construction*, pages 325–341. Springer, 2002.

[95] Chet Ramey and Brian Fox. *Bash Reference Manual*. Network Theory Limited, 2003.

[96] Randall Munroe. xkcd: Lisp Cycles. http://xkcd.com/297/.

[97] Inc. Red Hat. Ceylon. http://ceylon-lang.org.

[98] Remi D. A little holiday present: 10,000 reqs/sec with Nginx! http://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/, 2008.

[99] Rich Hickey. Are We There Yet? http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey, 2009.

[100] Ronald Rivest. Network working group internet draft: S-expressions. http://people.csail.mit.edu/rivest/Sexp.txt, 1997.

[101] Sean Dawson Ruslan Belkin. LinkedIn Communication Architecture. http://www.slideshare.net/linkedin/linked-in-javaone-2008-tech-session-comm, 2008.

[102] Ryan Dhal. Long Stacktraces in Node.js. http://nodejs.org/illuminati0.pdf, 2010.

[103] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[104] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM, 2006.

[105] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5, 2007.

[106] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *European Conference on Object-Oriented Programming*, pages 104–128. Springer Berlin Heidelberg, 2008.

[107] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy Continuations for Java Virtual Machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 143–152, 2009.

[108] Stephen Toub. Async Performance: Understanding the Costs of Async and Await). http://msdn.microsoft.com/en-us/magazine/hh456402.aspx, 2014.

[109] James Strachan. Groovy. http://groovy.codehaus.org, 2013.

[110] The Apache Foundation. Commons Javaflow. http://commons.apache.org/sandbox/commons-javaflow/, 2014.

[111] The Apache Foundation. The Apache HTTP Server Project. http://httpd.apache.org, 2014.

[112] The Apache Software Foundation. ActiveMQ. http://activemq.apache.org.

[113] The Netty Project. Netty. http://netty.io, 2013.

[114] The nginx team. nginx. http://nginx.org, 2014.

[115] The RIFE Team. RIFE. http://rifers.org/, 2014.

[116] TIOBE. Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/.

[117] Christian Tismer. Continuations and stackless python. In *Proceedings of the 8th International Python Conference*, volume 1, 2000.

[118] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[119] Inc. Twitter. Finagle RPC. http://twitter.github.io/finagle/.

[120] Typesafe, Inc. Akka. http://akka.io.

[121] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, 2007.

[122] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[123] Vladimir Ivanov. JVM JIT-Compiler Overview, 2013.

[124] Andreas Richard Voellmy, Junchang Wang, Paul Hudak, and Kazuhiko Yamamoto. Mio: a high-performance multicore io manager for ghc. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 129–140. ACM, 2013.

[125] J Robert von Behren, Jeremy Condit, and Eric A Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS*, pages 19–24, 2003.

[126] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.