Technical Report

UCAM-CL-TR-769
ISSN 1476-2986

Number 769

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Kilim: A server framework with lightweight actors, isolation types and zero-copy messaging

Sriram Srinivasan

February 2010

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Kilim: A Server Framework
with Lightweight Actors, Isolation Types & Zero-copy Messaging
Sriram Srinivasan

Internet services are implemented as hierarchical aggregates of communicating components: networks of data centers, networks of clusters in a data center, connected servers in a cluster, and multiple virtual machines on a server machine, each containing several operating systems processes.

This dissertation argues for extending this structure to the intra-process level, with networks of communicating *actors*. An actor is a single-threaded state machine with a private heap and a thread of its own. It communicates with other actors using well-defined and explicit messaging protocols. Actors must be light enough to comfortably match the inherent concurrency in the problem space, and to exploit all available parallelism. Our aims are two-fold: (a) treat SMP systems as they really are: distributed systems with eventual consistency, and (b) recognize from the outset that a server is always part of a larger collection of communicating components, thus eliminating the mindset mismatch between *concurrent programming* and *distributed programming*.

Although the actor paradigm is by no means new, our design points are informed by drawing parallels between the macro and micro levels. As with components in a distributed system, we expect that actors must be isolatable in a number of ways: memory isolation, fault isolation, upgrade isolation, and execution isolation. The application should be able to have a say in actor placement and scheduling, and actors must be easily monitorable.

Our primary contribution is in showing that these requirements can be satisfied in a language and environment such as Java, without changes to the source language or to the virtual machine, and without leaving much of the idiomatic ambit of Java, with its mindset of pointers and mutable state. In other words, one does not *have* to move to a concurrency-oriented language or to an entirely immutable object paradigm.

We demonstrate an open-source toolkit called Kilim that provides (a) ultra-lightweight actors (faster and lighter than extant environments such as Erlang), (b) a type system that guarantees memory isolation between threads by separating internal objects from exportable messages and by enforcing ownership and structural constraints on the latter (linearity and tree-structure, respectively) and, (c) a library with I/O support and customizable synchronization constructs and schedulers.

We show that this solution is simpler to program than extant solutions, yet statically guaranteed to be free of low-level data races. It is also faster, more scalable and more steady (in increasing scale) in two industrial strength evaluations, interactive web services (comparing Kilim Web Server to Jetty) and databases (comparing Berkeley DB to a Kilim variant of it).

Acknowledgments

To Alka, my dearest. Life's a wonderful journey, and I have the best travel companion.

To Amma and Appa, for the easy laughter, love and encouragement, and the early mornings and late nights they have lavished on me. This dissertation is dedicated to Amma. She would have exclaimed "*hai*"!

To my parents-in-law, for being an additional set of parents. One should be so lucky.

To my advisors. Prof. Jean Bacon, for making this excursion possible, for allowing me to gradually discover my interests and strengths without pressure, and for her gentle and selfless guidance to collaborate with other groups. To Prof. Alan Mycroft, for setting distant Bézier control points that so significantly influenced the arc of my learning, for his exhortations and for his unwavering belief that I would get there. To Dr. Ken Moody, for his critical reading of my papers and this document, for his cheerful and enthusiastic cross-pollination of ideas, and tours of beers and wines.

To Reto Kramer, for his friendship, guidance, technical knowledge, and a brain the size of a planet. There are a few dissertations lurking in there. To Lauri Pesonen and David Pearce, for the critical reviews, for the long coffee sessions packed with techie talk, and for sharing my enthusiasm for programming languages. To Eiko Yoneki, Pedro Brandão, David Eyers, Dave Evans, Minor Gordon and Niko Matsakis for sharing their impressive knowledge, opinions and good humor, for challenging my biases and assumptions, and for the gentle prodding to get on with it.

To Boris Feigin and Kathryn Gray for introducing me to the world of types and semantics, and holding me accountable to rigor.

To seminal influences in my professional life: Ed Felt, Anno Langen, Uday Rangaswamy, Gopalan Sankaran, Jolly Chen, Paul Ambrose and Terence Parr. I am a professional learner because of them. To Tony Hoare, Simon Peyton Jones, Mooly Sagiv and Uday Khedker for giving me their valuable time and for just the right mix of hard questions and encouragement. To Tim Harris, Martin Richards, Cliff Click, Chris Newcombe and the Berkeley DB team for lending their ear, time and patience.

To Samuel Kounev, Anil Madhavapeddy, Sandy & Elisabeth Fraser for their viral enthusiasm and constructive encouragement. I hope to pass that on.

To various cafes in Berkeley and Cambridge: the Beanery, Roma, Nero and King's, where I was both pupa and pupil!

And finally, in a dissertation that speaks of networks, load-sharing and fault-tolerance, it seems only fitting to acknowledge my own wonderful network, one that shares my load and tolerates my foibles: my sister Vidya and little Malavika for being a rock of support; Praveen & Shaila and the cuties, Anjali and Sanjana; Lizette, for the food, coffee, the Wire and the good cheer, and to all our dear friends: Ajit, Alka (Raghuram), Amita, Anil, Arjuni, Ashwini, Aslam, Athula, Banny, Caroline, Conor, Deepak, Flavia, Geetanjali, Hanna, Haresh, Hiroo, Jennifer, Kavita, Kinkini, Kirtana, Krishna, Larry, Lata, Lise, Maithili, Mark, Mayank, Namrata, Naren, Nasreen, Nishanth, Niteen, Nupur, Peggy, Prasad, Raghu, Raman, Rao, Sanjay, Santosh Jain, Sarita, Satish, Sarah, Savita, Scarlet, Shaheen, Shobana, Simona, Tara, Tom, Trish, Usha Jain, Uma, Vandana, Viji, Vinod, Vipul.

Thank you all.

Contents

1 Introduction **11**

1.1 Overview . . . 11
1.2 Thesis Contributions . . . 13
1.3 Kilim: Communicating Actors for Java . . . 15
1.4 Example . . . 18
1.5 Dissertation Outline . . . 18

2 Background **21**

2.1 Present-day server architecture issues . . . 21
2.1.1 Data plane: State transformation . . . 21
2.1.2 Data plane: Immutable objects . . . 27
2.1.3 Control plane . . . 27
2.2 Hardware trends and server-side design opportunities . . . 31
2.3 Communicating Actors . . . 32
2.4 Related Work . . . 34
2.4.1 Control Plane . . . 34
2.4.2 Data Plane . . . 35
2.4.3 Isolation: not an isolated quest . . . 37

3 Kilim Abstract Machine **39**

3.1 Syntax . . . 39
3.2 Operational Semantics . . . 41

4 Type System **47**

4.1 Design Choices . . . 48
4.1.1 Putting them all together . . . 51
4.2 Phase-1: Base Type Inference . . . 52
4.2.1 Phase-1 Termination and Soundness: Proof sketch . . . 53
4.3 Typestates and Heap Abstraction . . . 54
4.3.1 Object typestate . . . 54

4.3.2 Shape graphs . . . 55
4.3.3 States + Shape graph = Shape-state graph . . . 56
4.3.4 Properties of Shape-State graphs . . . 59
4.4 Phase-2: Message verification . . . 62
4.4.1 Normalization . . . 62
4.4.2 Semantic Rules . . . 62
4.5 Soundness of message verification . . . 67
4.6 Mapping to Java . . . 72
4.6.1 Sharable types . . . 73
4.6.2 Arrays . . . 74
4.6.3 Collections and Iterators . . . 74
4.7 Related work . . . 74

5 Kilim architecture and implementation 81
5.1 Actors and Fibers . . . 81
5.1.1 Continuations: An overview . . . 82
5.1.2 Automatic Stack Management . . . 83
5.1.3 JVM constraints . . . 85
5.1.4 Optimizations . . . 86
5.1.5 Cooperative Preemption . . . 87
5.1.6 Lightweight Threads: Related Work . . . 87
5.2 Communication and Coordination . . . 89
5.3 Scheduling . . . 91
5.3.1 Scheduler Hopping . . . 91
5.3.2 Scheduling: Related Work . . . 92
5.4 Network Support . . . 93
5.4.1 The Kilim I/O architecture . . . 95
5.4.2 Network Support: Related Work . . . 96

6 Evaluation and Performance 97
6.1 In-core benchmarks . . . 97
6.2 Network benchmark: Kilim I/O vs. Thread-per-request . . . 98
6.2.1 Results . . . 99

6.3 CPU + Network benchmark: Kilim Web Server 100
6.3.1 KWS architecture 101
6.3.2 KWS application: auto-completion 101
6.3.3 Results 102
6.4 CPU + Disk: Berkeley DB 104
6.4.1 Berkeley DB: original architecture 104
6.4.2 Berkeley DB: Modifications 104
6.4.3 Berkeley DB: Benchmark and Results 105
6.5 Qualitative Evaluation: Kilim in practice 106

7 Conclusions and Future Work **109**
7.1 Future Work 110

8 Notation and Nomenclature **113**

Bibliography **115**

1 Introduction

1.1 Overview

This dissertation is about the structure of highly concurrent internet servers such as web, email, game, chat, name and app-servers, as well as storage managers and load balancers.

An industrial-strength server is expected to deal with a large number of concurrent requests, the number being much greater than the number of available processors. Each request results in a split-phase workflow combining computation, network and disk activity. The server must cater to varying traffic patterns and multiple request classes each with its own service-level guarantee (often specified in terms of 99^{th} percentile latency and latency jitter). It must support dynamic component upgrades (plugins, servlets, spam filters) and must be engineered with fault- and performance-isolation aspects to help remote administration, online diagnosis and rapid recovery from component failures.

A single server is rarely deployed all by itself, for load-sharing and fault-tolerance reasons. Larger scales promote increasing levels of redundancy and backup, to the point that the failure of an entire data center or network can be tolerated. In other words, zooming outwards on a deployment setup, we find that the demands on a single server, listed above, are merely a miniature version of the demands placed on a cluster, which in turn resemble in miniature form those placed on the data center.

If the requirements are similar, we ask whether the architecture of a data center or cluster has any relevance to the way an individual server is structured. As Fig. 1.1 shows, a data center is connected to other data centers and contains clusters offering specialized services (data storage, application modules, traffic routing and so on). Clusters are networked with other clusters and are built with individual server machines (typically SMP), which in turn contain independent but connected components (operating system kernel and application processes).

The following phrases are key to this line of enquiry: active components (not always reactive), asynchronous messaging, isolation, scheduling (timing of deployment) and location affinity (assignment of application layers to clusters and of servers to machines). Isolation takes myriad forms: execution-isolation, memory-isolation, fault-isolation and upgrade-isolation. These architectural aspects apply at all levels.

Our thesis is that this architecture should continue on to the intra-process level as well:

> *Servers are best built using "actors", components with private state* and *a private thread of execution, that communicate with each other using well-defined asynchronous messages and protocols. Actors must be isolated, monitorable, application-schedulable, and lightweight enough to exceed the number of concurrent user-level activities.*

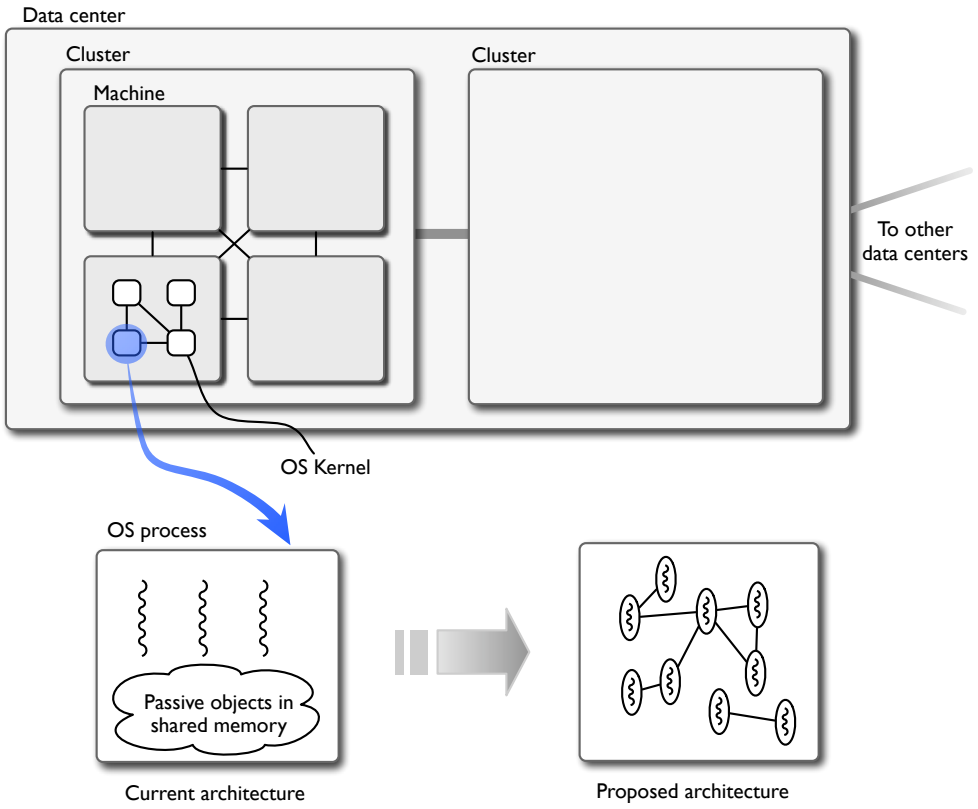


Figure 1.1: Hierarchical, nested networks in a data center. We propose extending this structure to the intra-process level.

Each of these characteristics sets the communicating actors proposal apart from current mainstream approaches. All servers are currently built upon manipulating passive objects in shared memory, regardless of the control plane (kernel threads, user-level threads, event-loops, coroutines[51]). None of these approaches (object-orientation included) provides isolation in the myriad forms listed earlier. Edward Lee[117] puts it succinctly: "things happen to objects, actors make things happen". Further, none of the extant control plane schemes combine simplicity of development with adequate application-level control over failure-handling, placement (processor affinity), and scheduling of activities. We will elaborate on these issues in the next chapter.

Déjà Vu?

The communicating state machine model is by no means new or rare. It has been widely adopted at every scale: (a) modeling approaches (Tony Hoare's `CSP`,[91] Robin Milner's π-calculus[124] and Carl Hewitt's Actors[87]), (b) model checkers like `SPIN`,[94] (c) at the implementation level, seen in isolated operating system processes communicating with others via IPC, to the kernel via syscalls, and to the graphics processor using OpenGL pipelines etc.). And of course, replicated state machines are necessary at the distributed system level.[159] In many of these cases, the underlying hardware (CPU, memory, network) is shared, but isolation is maintained either physically (via processor-supported address space partitioning), or logically (for example, via linear type systems or copying data to avoid data races).

At the intra-process level, we have seen generations of concurrent languages that provide isolation

and fine-grained schedulable tasks. Notable examples are Per Brinch Hansen's `Concurrent Pascal`,[83] `MESA`,[111] `occam-pi`,[134] `Ada`,[38] `Concurrent C`,[71] `Limbo`,[188] `Erlang`[13] and `Sing#`,[60] just to name a few. None of these languages have been widely adopted (yet) in building highly concurrent servers*. Why?

One or more of the following (technical) reasons may be relevant. First is the hegemony of the Algol-family of languages; one cannot discount the tool support and extant training in imperative languages with pointers and mutable state. This model is deeply entrenched, as any book on basic data structures can attest.

Second, commodity processor architectures such as Intel and ARM have co-evolved with these languages, which means certain common patterns have found expression in silicon. For example, the call/return mechanism and stack allocation are cheap due to the presence of stack registers, register windows and stack segments. Shared memory constructs are fast due to built in CAS (compare and swap) and cache coherence machinery. In turn, C has become the official "systems" programming language. If Lisp or Lisp machines had had their way, we might possibly be able to count on hardware support for garbage collection, tail recursive calls and closures.

The third reason is performance. In server-side environments, high performance always holds sway over cleanliness of a model, even consistency. For example, most modern large-scale services do away with database joins and data normalization, in the pursuit of raw speed. Trading availability for consistency, ACID (atomicity, consistency, isolation, durability) guarantees have given way to BASE[54, 146] (Basically Available, Soft state, Eventually consistent).

Finally, not all approaches provide configurable control over scheduling and processor assignment.

Which brings us to the topic of how our work differentiates itself from existing work.

1.2 Thesis Contributions

Our main contribution is to bring the communicating actor model to the world of mainstream imperative languages, pointers and mutable state, subject to the requirements and rigors of server-side environments.

We address the problem of memory isolation by taking a novel approach: we treat messages as a distinct category of objects, the only kind that can be exchanged between actors. All other objects are confined to the actor in whose private heap they are created. Messages are mutable, but subject to export restrictions: (a) they are tree-structured and, (b) linearly owned — readable and writable, indeed visible, by only a single actor at a time.

We present a type system (Chapter 4) that statically differentiates between ordinary internal objects and exportable messages, and further enforces the message constraints above. By controlling ownership and reining in arbitrary pointer aliasing for messages, the type system *statically guarantees* absence of low-level data races. The only way to affect an actor's state is by sending it a message, and that too only after the actor explicitly imports the message into its context and chooses to change its state. This relieves the programmer from having to worry about unexpected changes to an actor's state or the effect of memory models. At the same time, it permits messages to be unencapsulated (public, non-final

*`Erlang` is seeing an encouraging trend at the time of this writing.

fields), and to be passed by reference (zero-copy messaging). The next section describes these aspects in some detail.

The second contribution of our approach is to show that it is possible to retrofit a high-level language such as Java with actors light enough to be in the hundreds of thousands, faster to context-switch*, preemptible (to a degree) and more scalable than concurrent languages such as Erlang that have low-level support for stack switching. This, in spite of the lack of access to the stack in the JVM†, leave alone modify or switch stack contexts.

We present a framework called Kilim[103]‡ that combines a code generator, an actor framework, the type system and a small run-time library. We aim to show that with just a small number of internally consistent design principles and techniques it is possible to have speed, statically guaranteed safety, reliability and programming simplicity, and to continue to use existing tools (debuggers, profilers, IDEs) and libraries without too much change.

The final contribution — most crucial for a server-side programmer — is to show vastly improved performance metrics. We compare industrial-strength products, noted for their performance in their respective categories, to their Kilim equivalents: Jetty for dynamic web services, and Berkeley DB for transactional data stores. We demonstrate lower latency and jitter, much better scalability and fairness.

Nomenclature

We appropriate the term *actor* to represent a state machine with a control stack of its own (a private thread) and an isolated heap. Our usage of this word differs from the original definition in Carl Hewitt's[87] and Gul Agha's[3] pioneering work on the actor model. There, an actor is an active object (with its own thread) with a singular identity to which messages can be addressed. We are agnostic as to whether an actor has one identity or has multiple facets. Our usage bears closer resemblance to processes used in process-calculi,[124] but we reserve the term *process* for operating system processes in this document.

The terms *thread* and *concurrency* are often used to *collectively* refer to a slew of facilities: automatic stack management provided by the operating system, preemptive context switching, and coordination using shared memory and locks§. We will use the term *multi-threaded programming* to refer to this context, but the term *thread* will be used to refer solely to the control plane (the first two aspects), separating the data plane aspects of synchronization and data exchange. In particular, we will use the term *kernel thread* when the thread's stack is managed and context-switched by the operating system.

Finally, the term *server* in this document refers to one or more kernel threads, possibly spread over multiple operating system processes, that together provide a single service. Many "event vs. thread" debates[107,181] implicitly assume a single process context (more on this in the next chapter), which need not always be the case (examples are database servers such as Postgres, and web servers such as Flash). The point is that the resolution of such debates in favor of one or the other does not say whether a single process can provide a particular service with the required guarantees (performance, fault-tolerance).

*With the caveat that all performance comparisons are snapshots in time

†A method cannot reflectively discover any details about the caller's activation frame.

‡A Kilim is a flexible, lightweight Turkish flat rug woven with fine threads.

§For example, Edward Lee's paper on *The Problem with Threads*[117] uses the term this way.

1.3 Kilim: Communicating Actors for Java

Our original aim was to build a communicating actors framework for low-level systems programming, with an emphasis on type-directed memory isolation, fast hardware-supported context switching, and explicit interface automata specifications in the style of Clarity[44] and Melange[120] . We were also interested in the idea of using the SMP cache-coherence machinery for passing messages selectively and lazily by reference, or sending short messages directly in the style of URPC.[24] Finally, control over actor placement influences locality of instruction and data caches and simplifies garbage collection considerably. Building such features into a language is still the long-term aim.

We investigated `LLVM` and the `D` language in some detail, but switched to Java, partly for its rapid adoption in industrial environments, its excellent and portable tool support and libraries, but mainly for its portable memory consistency model. Early prototypes also showed surprising performance numbers, a testament to the quality of the JVM. Most of the ideas listed above have been incorporated into Kilim, the current toolkit. Kilim is a combination of a bytecode post-processor called the *weaver*, a type-checker, and a small, extensible run-time library (67k). We present the highlights below, and devote the rest of the dissertation to explaining each of these points in some detail.

Ultra-lightweight threads

Java threads are expensive in terms of creation time, context-switching speeds and space. Kilim's *weaver* provides cooperative, ultra-lightweight threads by transforming Java bytecode using a variant of continuation passing style. It requires that the programmer specially identify methods that may need to suspend mid-way, only to resume later based on some condition. Such methods are called *pausable*, and marked by the annotation `@pausable` (see example in §1.4). The annotation is similar in spirit to checked exceptions in that all callers and overriding methods must be marked `@pausable` as well.

The weaver injects code in all pausable methods to provide automatic stack management.[2,67] The engineering details of this transformation are explained in Chapter 5. Kilim's actor threads are quick to context-switch and do not need pre-allocated private heaps.

Kilim permits multiple user-level schedulers (including programmer-supplied ones) to partition kernel threads and processors, and for actors to migrate to different schedulers; this gives flexible control over placement and scheduling.

Communication via mailboxes

Kilim contains a run-time library of type-parametrized *mailboxes* for asynchronous message-passing and a prioritized `select` operator to choose from a number of ready mailboxes (similar to CSP's `alt`[91]). Mailboxes have optional dynamically settable bounds to prevent producers from overrunning consumers. In the typical server scenario where a user request proceeds from one stage of processing to another, blocking an actor puts a back pressure on the earlier stages, all the way back to a specific socket connection. This is a fine-grained version of SEDA-style I/O conditioning.[186] Mailbox references are first-class values that can be incorporated into messages, as in π-calculus.[124]

These features permit a dynamically evolving topology of interconnected components. They separate the initiator of a request from the thread that reaps the result, useful for example in RPC chains.[162] They also allow deferred synchronous approaches where multiple requests may be initiated before

waiting for a response. Mailbox references are intended to be shareable across address spaces (§7.1), but the current implementation is an intra-process solution only.

Messages as a special category

Traditionally, most messaging solutions treat messages specially. Java, for example, requires any object that can be sent as a value between address spaces to be marked as `Serializable`. However, serialization is an expensive operation even in a network scenario, and prohibitively so in an intra-process setting. For this reason, we are interested in passing message objects by reference, serializing them only if they cross process boundaries. But passing objects by reference between actors requires us to confront the problems of pointer aliasing. We will defer to Chapter 4 to discuss these problems in detail, as well as the scores of attempts to bound the problem space. Below we provide an overview of the special treatment accorded to messages by Kilim.

We treat messages as philosophically distinct from, and much simpler than, other Java objects. Messages are:

- *Unencapsulated values* without identity (like their on-the-wire counterparts, XML, C++ structs, ML datatypes and Scala's case classes). The public structure (instance fields are public and mutable) permits pattern-matching,[123] structure transformation, delegation and flexible auditing at message exchange points; these are much harder to achieve in the presence of encapsulation.
  The public structure also results in far less syntactic noise compared to the setters and getters seen in object-oriented wrappers.[74] Note that these wrappers do not really provide any encapsulation because they return pointers to internal objects and retain pointers supplied to them; their setters and getters merely add to accidental complexity.
- *Not internally aliased.* A message object may be pointed to by at most one other message object (and then only by one field or array element of it). The resulting tree structure can be serialized and cloned efficiently, and effortlessly stored in relational, XML and JSON schemas). This structure is widely adopted by convention, almost without exception. Examples include events or messages in most server frameworks (in fact, all the ones that we are aware of), windowing systems, and the Singularity operating system.[60] Enforcing this convention simplifies the type system enormously. This style of simplification is philosophically similar to the `SPARK`[20] subset of `Ada`, in that it is sometimes worth limiting the full generality of the language in exchange for guaranteed reliability.
- *Linearly owned.* A message belongs to, and is visible to, a single actor at a time. After sending a message, the actor loses complete visibility of that message (and all reachable objects). This allows efficient zero-copy message transfer (by reference) where possible, without the danger of any mutations to that data being simultaneously visible to any other actor. The programmer has to explicitly make a copy if needed, and the imperative to avoid copies puts a noticeable "back pressure" on the programmer.
  Linearity is often informally specified in commercial code; figure 1.2 shows an example taken from the Apache MINA[125] framework (an event-driven asynchronous API over Java's NIO facility).

Statically-enforced isolation

We enforce the above properties at compile-time. Isolation is interpreted as *interference-freedom*, obtained by keeping the set of *mutable* objects reachable from an actor's instance fields and stack

```
class HttpSessionContext {
    ....
  /**
   * Writes the specified response back to the client.
   * The response <i>must not</i> be modified after it has been
   * submitted for comittment - irrespective of whether a commit
   * is successful or not.
   *
   * A request may have only one response committed to it. The return
   * value of this method can be used to determine whether the
   * supplied response will be used as the single response for this
   * request.
   * <p>
   * Application code must not modify a response in any way after it
   * has been committed to a request. The results of doing so are
   * undefined.
   *
   */
   boolean commitResponse( HttpResponse response );
}
```

Figure 1.2: Example of informally specified linearity, seen in Apache MINA documentation

totally disjoint from another actor's. Kilim's weaver performs a static intra-procedural heap analysis that takes hints from isolation qualifiers specified on method interfaces.

The uniqueness properties of the type system can be used for recycling resources such as buffers and security capabilities. If Kilim were to target a lower-level machine without garbage collection (as was the original plan), such tracking would be necessary for memory management even for immutable objects.

Portability

Kilim is designed to meet the requirements of production server-side systems. With a view towards immediate industrial adoption, we imposed the following additional constraints. The solution must be portable, in that it should not depend on a particular JVM or rely on facilities afforded by specific compilers (such as the pluggable type system in an upcoming version of `javac`[140]). For the same reason, it should not require any syntactic modifications to the Java source language, which unfortunately precludes many interesting constructs. It should remain within the idiomatic confines of Java (using exceptions, mutable state and so on), except, of course, for the concurrency constructs – those are replaced with message passing.

For the reasons stated above, Kilim's `weaver` is a bytecode post-processor (Fig. 1.3). It performs the type checking and CPS transformation; the transformed code runs in conjunction with the supplied run-time library.

The Kilim toolkit provides mechanism, not policy, and offers largely orthogonal feature-sets. For example, if the JVM were to natively provide lightweight threads, one should be able to use the messaging mechanisms and the isolation type checking aspect without any change. The mailboxes are a library construct; unlike built-in primitives such as channels in some concurrent languages, mailboxes are not given special treatment by the Kilim weaver. This permits variants in terms of number of producers and consumers (currently mailboxes have *n* producers and one consumer), customized versions

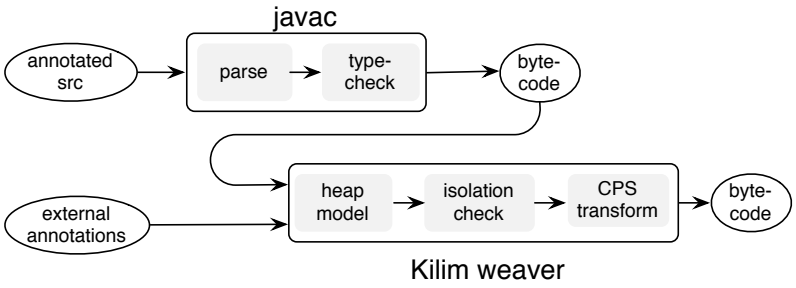


Figure 1.3: `javac` output post-processed by Kilim `weaver`

for primitive types, and buffering strategies (unbuffered, fixed size buffers). The scheme also permits other constructs such as *futures* and Haskell-style `MVars` to be provided at a later stage, and lets the type system guarantee safety.

1.4 Example

Figure 1.4 illustrates a simple Kilim program (names that have significance to Kilim are colored gray). It sets up a chain of actors, where each actor knows about its mailbox and that of the next in the chain (and notably, is not given the other actor's reference). The main thread pushes an empty `StringBuffer` into the first actor's mailbox, which writes "hello" into the buffer and passes the modified `StringBuffer` on to the next actor, and so on. The last actor appends "world", prints it out and exits.

The structure of an actor is analogous to that of a Java thread: it extends the `Actor` class instead of Java's `Thread`, and its entry point is `execute()` instead of `run()`. Note the use of the `@pausable` annotation on `execute`, because it calls the mailbox's `get()` and `put()` methods, which may pause if the mailbox does not have messages or (respectively, is full). Note that `main` does not have to be declared pausable because it only calls a non-pausing method called `putnb()`. Chapter 5 has more details on mailboxes, actors, and support for cancellation and scheduling.

The following steps illustrate the compilation, post-processing and running steps.

```
> javac -d ./classes Chain.java
# Post-process using weaver
> java kilim.tools.Weave -d ./classes examples.Chain
# Run example
> java -cp ./classes:$CLASSPATH  examples.Chain
```

1.5 Dissertation Outline

Chapter 2 ("Background") argues why the current paradigms of mainstream server-side design are flawed or inadequate not just for future trends, but for present day requirements and conditions as well.

```
1  package examples;
2  import kilim.*;
3
4  // A mailbox for exchanging StringBuffer objects.
5  class Smb extends Mailbox<StringBuffer>{}
6
7  public class Chain extends Actor { // instead of ''extends Thread''
8      Smb mymb, nextmb;
9      public Chain(Smb mb, Smb next) {mymb = mb; nextmb = next;}
10
11     @pausable
12     public void execute() {  // equivalent of Thread.run()
13         while(true) {
14             StringBuffer sb = mymb.get();
15             if (nextmb == null) { // last actor.
16                 System.out.print(sb + "world");
17                 System.exit(0);
18             } else {
19                 sb.append("hello ");
20                 nextmb.put(sb);  // send to the next actor
21             }
22         }
23     }
24
25     public static void main(String args[]) {
26         Smb mb = new Smb();
27         Smb nextmb = null;
28         // Set up the chain
29         for (int i = 0; i < 1000; i++) {
30            new Chain(mb, nextmb).start();
31            nextmb = mb;
32            mb = new Smb();
33         }
34         // Send a message to the first element in the chain.
35         nextmb.putnb(new StringBuffer());
36     }
37 }
```

Figure 1.4: Sample Kilim program: Each actor in a chain of actors appends "hello" to a mutable buffer and passes it on. The last actor prints the buffer.

Chapter 3 ("Kilim Abstract Machine") is a precursor to the chapter that follows. It presents the operational semantics for an abstract machine that is similar to the JVM, but with support for memory-isolated actors and message passing. Messages are physically moved from one actor's state to another.

Chapter 4 ("Type System") presents the isolation type system and a proof of soundness. It restricts the set of allowable KAM (Kilim Abstract Machine) programs to ones where a dangling reference is never used. This means that a Kilim program deemed type-safe by the weaver is guaranteed not to access any part of a message that has been sent, thereby ensuring an absence of data races at compile-time.

Chapter 5 ("Kilim architecture and implementation") describes the bytecode transformation to achieve lightweight actors, and a discussion of the design points employed in the run-time library. Many of these issues are pertinent to other mainstream imperative languages as well.

Chapter 6 ("Evaluation and Performance") demonstrates micro-benchmarks (comparisons to Erlang on tasking and message passing performance) and two macro-benchmarks: a CPU+network comparison pitting the Kilim Web Server against Jetty (a popular industrial-strength web server), both of which are based on the same underlying Java NIO libraries, and a CPU+disk comparison pitting Berkeley DB (a widely adopted key-value store) to its Kilim variant.

Background 2

We observed earlier that messaging between isolated state machines is widely adopted; we now present the case for the approach to be applied at the intra-process level as well.

The structure of this chapter is as follows. We first present arguments for why present-day server-side solutions create a fair amount of *accidental* complexity, in the words of Fred Brooks,[36] and why they are sometimes inadequate to handle the *essential* complexity of the problem as well. We then take a brief look at emerging trends in processor and network architectures (§2.2). We argue (§2.3) that the communicating actor model is better suited to handle present-day problems as well as likely future scenarios. Finally, we examine some notable research work related to building highly concurrent servers.

2.1 Present-day server architecture issues

In this section, we study mainstream server-design approaches, and in the section on related work (§2.4), we will examine other proposed solutions that are notable, although not necessarily mainstream.

At a *logical* level, all server-side frameworks and applications have concurrent threads of control that transform the state space in a collaborative fashion. We study how threads of control synchronize (the *control plane*; §2.1.3) separate from what it means to collaboratively update the system's state space (*data plane*; §2.1.1). Two logically equivalent implementations may differ considerably on other axes of practical importance: the cost model, composability, fragility in the face of software evolution, run-time diagnosis and so on.

2.1.1 Data plane: State transformation

In this section we focus on that part of the system state that two or more threads of control may be interested in modifying concurrently. We categorize different concurrency control schemes according to the way they segregate updates to the state space.

In *temporal partitioning* schemes, the memory in which the updatable state resides is shared, and different threads of control take turns directly modifying the state space in-place; only one is permitted to update it at any one time. In *spatial partitioning* schemes, the state space is spread over partitioned memory spaces, and multiple partitions may contain data pertaining to the same logical entities. Memory can be partitioned physically (memory banks, hardware-supported process isolation) or logically partitioned (with a type system); either way, each thread of control has its own space for in-place modification.

We explore these two categories in some detail, and argue that update isolation via *spatial partitioning is preferable to temporal partitioning*.

Temporal partitioning via locks

The primary temporal partitioning scheme uses locks to serialize updates to shared memory, and is by far the most popular of all partitioning approaches.

All mainstream languages possess fine-grained memory sharing semantics. This is true even in cases where there are no pointers, such as lexically shared variables amongst closures. Instructions such as `CAS` (compare and swap) and `LL/SC` (load-linked, store-conditional) help perform atomic updates on shared memory cells and enable a variety of synchronization constructs such as semaphores, monitors, locks, mutexes and condition variables. The provision of fine-grained data updates (different threads can perform updates of a single memory word) and fine-grained locking (each Java object has a private lock) can make for a highly concurrent system, but as experience has shown,[26,117,173] it is also extremely error-prone and hard to reason about. Here are some prominent issues with fine-grained locking and sharing:

Locks and multiple objects. An object does not get to decide its atomicity context, the application does. While monitors are useful for making consistent changes to a single object (a bank account, for example), there is the need to atomically effect changes to *multiple* objects; for example, a bank transfer requires that changes to two accounts is made atomically. The programmer must make arrangements to safely update a set of objects without the language's help, which is a hard problem when different threads of control are updating different but non-disjoint sets of objects. This is closely related to the next point.

On-demand lock acquisition. Since locks are acquired on demand as a thread proceeds, multiple threads may acquire different sets of locks in different orders depending on their control flow. This is likely to cause deadlocks (see §2.1.3). Frequently, a lock implicitly grants permission to the lock owner to update a given set of objects, but errors creep in when the same set of objects can be reached via another traversal that does not pick up the required locks along the way (more on this in the next section).

Separating locking and unlocking. If one procedure locks an object and another unlocks it, and if the procedures can be called in any order and any number of times, it is practically impossible to reason about that object's consistency. This is a problem even if the locks are reentrant. Monitors and Java's `synchronized` blocks couple locks and unlocks with lexical and dynamic scope, which is a good thing, but there are other problems with it, as we shall see in the control plane section.

Locking is expensive. Applications are sensitive both to the overhead of locking and to the time taken within critical sections. Lock contention can be extremely expensive for a waiting process, especially if it is forced to suspend until the lock is available. Much literature on concurrency is focussed on highly concurrent data structures. The problem does not lie here; after all, data structures offer bounded and predictable time windows of contention, and are available in the form of libraries written by expert programmers. The problem lies in using the lock paradigm in a more general setting (in the application area, and by less trained programmers); in one instance (from personal experience), a single log write inserted within a lock window produced bizarre timing behavior, depending on the information contained in the log data.

Locking is optional Mainstream languages do not record (or require the programmer to declare) whether an object can be accessed by multiple threads concurrently, or require that a sharable object be

always accessed under a lock. Errors due to lock omission are highly non-deterministic; they may not even show up as a problem until run on a different processor configuration. As Per Brinch Hansen[84] points out, the term *monitor* in the Java and .NET environments is used erroneously to refer to objects with built-in locks, since a monitor by definition must always hide state and protect it using a built-in mutex. Most concurrency errors can be traced to the fact that these are optional facilities.

Note that locking is a logical synchronization mechanism that acquires a permission before executing a corresponding action. As such, locks can be used in a message-passing system to regulate access to a shared actor, and one should expect to see the problems above in such a situation as well. However, the scale of such problems is considerably less than the usage of locks in mainstream languages due to the fact that the more fine the grain of sharing, the more are the chances that a given data structure has multiple access and update paths through its constituent objects. Coupled with the undecidability of precise static pointer alias detection,[148] it is hard to guarantee absence of low-level data races.

In the section on related work, we will briefly cover lock-free data structures, another way to synchronize access to shared data without indefinitely blocking the threads.

Temporal partitioning via scheduling

The second way of ensuring that threads take their turns updating a portion of shared memory is by scheduling them appropriately. Scheduling can be implicit in the way the code is arranged to execute. The event-oriented style of code, essentially a single thread with a main loop, and seen in many (perhaps most) server and GUI frameworks, is a classic example of an implicit style of scheduling control. This pattern provides "run-to-completion" semantics, because the thread of control is not interrupted preemptively, which in turn ensures that state updates can be made consistently without using locks. These properties are often presented as the reason for the simplicity and performance of an event-driven style over a multi-threaded architecture.

While the event-driven style is the mainstay of most C-based non-threaded applications, it uses only a single processor. All event-vs-thread debates[107, 137, 181] implicitly assume a uniprocessor context for comparing single-threaded event loops to a multi-threaded programming style.

Another problem is that changes to the state machine tend to be brittle, because splitting up a relatively long-running computation into multiple intermediate states may invalidate the earlier assumption that the shared data structure is in a consistent state every time the flow of control returns to the main loop driver.

Temporal partitioning via *explicit* scheduling, while rare in typical server applications, is used widely in real-time and parallel programming environments. For example, the OpenMP library provides compiler pragmas to help the programmer annotate which sections and which loops can be evaluated in parallel. These schemes tend to be brittle or non-portable as they tend to hard-code assumptions about the hardware.

Temporal partitioning, shared objects and relaxed memory consistency

All temporal partitioning approaches rely on updating shared objects in place; we now examine what it takes to share memory between processors.

As processor speeds have exponentially outstripped memory speeds, a wide variety of tricks have been employed to blunt the impedance mismatch. Several levels of caches have been introduced, memory updates are deferred by write-behind caches (in the hope of batching writes), instructions are routinely eliminated by the compiler, or reordered due to compiler optimizations and instruction level

```
class Foo {
 private Bar bar = null;

 synchronized
 public Bar get() {
   if (bar == null) {
     bar = new Bar();
   }
   return bar;
 }
}
```

```
class Foo {
 private Bar bar = null;
 public Bar get() {
   if (bar == null) {
     synchronized(this) {
       if (bar == null)
         bar = new Bar();
     }
   }
   return bar;
}
```

Figure 2.1: Double-checked locking error. The code on the right is incorrect.

parallelism. On a uniprocessor, these tricks vastly improve performance, yet provide an illusion of a single memory, and behave as if the instructions and data were evaluated in program order (*as-if-serial* semantics).

However, given multiple processors, it is prohibitively expensive to provide sequentially consistent[110] cache coherence. Consequently, every multiprocessor system can at most provide some form of *relaxed consistency*, which means that at any given point in time, different threads may have different values of a particular memory address (in registers or in the cache), and thereby for a particular language-level variable. Java programmers discovered[147] the pitfalls of relaxed consistency the hard way when the commoditization of multi-core processors began to smoke out bugs not just in well-tested software, but also in in well-worn idioms.

Subsequently Java,[100,121] the .NET platform and lately C++ [27] have elucidated portable memory model semantics that recognize the reality of relaxed consistency, without sacrificing well-known optimizations and reordering mechanisms (both in the compiler and the processor). They do this by identifying special operations such as constructor calls, `synchronized` blocks, access of `volatile` variables etc. to serve as checkpoints, where writes to memory before a checkpoint is guaranteed to be visible to other processors after the checkpoint. The compiler is not permitted to reorder code across these checkpoints.

The key here is that the program must be "correctly synchronized" to prevent low-level data races, but there is no automatic way to detect if a program is incorrectly synchronized. The classic double-checked locking optimization idiom[158] is a textbook example of code that could be expected to work in a uniprocessor, but fails in a multiprocessor context (see Fig. 2.1).

The code on the left (of Fig. 2.1) lazily initializes an instance field, under the protection of the parent object's monitor. Since acquiring a monitor's lock can be expensive under contention, and because the field is going to be initialized just the first time, programmers balk at having to penalize every subsequent access just to check if the field is non-null. One common pattern was the code shown on the right, where we skip the synchronization in the common case where `bar` is non-null. This is erroneous. It is possible that due to compiler and ILP-induced reordering, the updates to the corresponding object's fields (the object pointed to by `bar`) have not yet made it into that thread's processor. Without a synchronization barrier, there is no incentive for the thread's cached values to be updated. Given the deeply entrenched mental model of imperative control flow, it is hard to reconcile with the fact that updates do not necessarily happen in program order, across threads.

```
1 class String {
2   private int hashcode;
3   public static int hashcode() {
4     int h = this.hashcode;
5     if (h == 0) {
6       h = ... compute hash ...
7       this.hashcode = h;
8     }
9     return h;
10  }
11 }
```

```
1 class String {
2   private int hashcode;
3   public static int hashcode() {
4
5     if (this.hashcode == 0) {
6       int h = ... compute hash ...
7       this.hashcode = h;
8     }
9     return this.hashcode;
10  }
11 }
```

Figure 2.2: Memoization and benign data race. The code on the right is incorrect

```
1 class String {
2   private int hashcode;
3   public static int hashcode() {
4     int h = this.hashcode;
5     if (this.hashcode == 0) {
6       h = ... compute hash ...
7       this.hashcode = h;
8     }
9     return h;
10  }
11 }
```

Figure 2.3: Legal (but unsafe) rewrite of Fig 2.2(right)

The next example (Fig. 2.2) is an even more subtle illustration* of the perils of relaxed consistency. It considers two ways of lazily computing the hashcode for an immutable `String` object. Unlike the previous example, this example updates just a single `int` field.

The code on the left (of Fig. 2.2, as seen in the JDK source (Java Development Kit)) exemplifies a benign data race; in the worst case, one expects that two threads will find the hash code to be `0`, race to compute it and to overwrite the hashcode field to the same value. The field being a 32-bit integer, this update is guaranteed by Java to be atomic. What could go wrong? It appears that the example on the right is equivalent (and has the same benign data race), but even expert programmers have trouble discerning a possible (though highly improbable) data race in the code on the right. In the absence of any synchronization checkpoints, the compiler is free to rewrite the code on the right to that shown in Fig.2.3.

This transformation, while correct in an uniprocessor setting, introduces two reads of `hashcode` (Fig.2.3, lines 4 and 5). If `this.hashcode` was zero initially, but updated by another thread between those two lines, the method will skip the *if*-block and return the cached value of h (zero). Even with a defined memory model, this is not the level of reasoning or attention one can expect from even seasoned programmers.

We end this section on memory consistency models with the disquieting note that processor memory models are specified informally,[131] and that this is likely to be a rich source of bugs in the near

*Thanks to Jeremy Manson for this example.

future. One needs constructs (such as messaging) that remove all memory consistency concerns out of the hands of the larger programming community, by concentrating them in a few maintainable concurrency constructs.

False sharing

Cache-coherent shared memory can also result in *false sharing*. Caches invalidate an entire line (typically 32 to 512 bytes) even if just one word was updated. If the code were to have an array of data items, and say thread `0` modifies item `0`, thread 1 modifies item 1 and so on, we would experience cache thrash although the updates are logically isolated. Such patterns are seen in arrays of locks, arrays of shared counters (for statistical purposes), and lock-free versions of concurrent array-based queues. Performance problems due to cache line ping pongs are exceedingly hard to detect by a visual examination of the code, and programmers have to pad structures to cache line widths to ensure that isolated units of data are mapped to separate lines.

We can safely conclude that programming with a combination of fine-grained access to shared memory, in-place updates and locks is not simple in a multi-processor environment.

Data plane: Spatial Partitioning of updatable state

We now turn our attention to spatial partitioning schemes that physically or logically partition, or spread, the state space among multiple threads of control. Either way, the threads of control do not share memory directly, but work on their own copies of the state. At some point, the threads of control resolve their differences and come to a shared understanding — *consensus* — on what constitutes the latest consistent overall state.

The important characteristics of this approach are: memory is not shared, state is replicated, and consistency is lazily achieved (as opposed to the locking approach, where the permission is acquired before the update is made). It is also known as an *optimistic* approach, in that a thread makes a change to a private copy assuming that no other thread is likely to touch that logical bit of data, but is prepared to redo or abort the computation if there is consensus that that bit of data has indeed been updated.

Spatial partitioning is of course necessary at the distributed system level, but has proven to be a sound engineering approach even on multiprocessor machines, although it is possible to address all of system memory and share objects directly. Many prominent database and web servers use the multi-process approach to take advantage of hardware-supported memory isolation. Our preference for spatial partitioning stems from the observation that the underlying hardware abstraction necessarily exposes itself as a distributed system (relaxed memory consistency) and that the expectation of lazy conflict resolution is in tune with that of the underlying hardware.

The biggest advantage of spatial partitioning of memory is that every bit of code has the guarantee that all variables in scope, and all objects reachable from those variables, belong to a single updater. One does not have to worry about objects getting modified unexpectedly, nor about memory consistency (as long as the scheduler issues a memory fence before switching the thread of control over to another processor). This permits compilers and processors to preserve all the optimizations made for the uniprocessor era, and for us to continue with the familiar program-order semantics. Objects can be traversed and mutated at a fine-grained level, and reachability defines the logical extent of a thread's heap. The problem is that pointers and global variables make it impossible to statically verify confinement. Chapter 4 discusses the Kilim type system and other related work on this issue.

The disadvantage of the spatial partitioning approach is that multiple threads working away on their own copies (until reconciliation) might involve duplication of data and effort. This turns out to be not much of an issue in a thread-per-user or thread-per-connection architecture in servers; except for games, most applications do not involve interactions between users.

2.1.2 Data plane: Immutable objects

Another common approach is to build systems with aggregates of immutable values. Reducing the amount of concurrently mutable data in the system is always sound from the concurrency point of view. Threads can share immutable objects by reference. A transition to a new state is achieved by atomically switching to another collection, mutating the aggregate in place, or using a persistent data structure. There are a few issues, however.

If the aggregates are mutable in-place (at some level, one has to reflect the fact that the logical state space is mutable), we still have the problem of making consistent changes to these aggregates, which reduces to one of the two partitioning schemes above. Still, the fact that there are fewer shared mutable objects to worry about is a good thing.

Next, the immutability property must be deep. An object is immutable only if all reachable objects are immutable too; however, this definition indicates that one cannot have an immutable collection of mutable objects. In an essentially-mutable language such as Java where there is no way of identifying or tagging a group of objects, there is no way to distinguish one reachable object from another (a cons cell versus a contained element). In §4.7, we will briefly examine ownership types that provide a mechanism to tag objects in a parametrized fashion.

The third pitfall is that for items that are big and likely to change often (protocol buffers, for example), it does not always pay to make them immutable, because any change results in deep copies (which is why we categorize this pattern of programming as a spatial partitioning scheme).

Finally, many immutable objects feature memoization. As we saw in the discussion of Fig. 2.2, *observational immutability* has its weak spots. Boyland[33] catalogues other issues related to adding a `readonly` modifier to Java.

2.1.3 Control plane

Most modern APIs — language-level interfaces as well as service-level interfaces provided by web services, file systems, databases etc. — are implemented as function calls. This permits a given task to be handled in a structured, hierarchically decomposable fashion, and the code to be written in a sequential and imperative style. The thread, which encapsulates the program counter, the call stack and local variables is an integral part of the state machine, because it implicitly records the work that has already been performed. One convenience of using threads to encapsulate the callstack — and arguably the source of many problems as well —- is that a thread-local context is available to store objects such as transaction context, security context and profiling information which are of interest to all methods on the callstack, without having to thread them through method parameters.

There are some critical problems with this service-provided API approach, however.

Control plane: Client-side issues

The fundamental problem with functions as the means of requesting service is that functions are expected to run to completion, which means that the caller is *blocked*, prevented from handling any further events or issuing any other requests until the callee finishes. This is an issue in most modern GUI frameworks and up-calls in network protocols today. GUIs have been known to freeze when a remote file system connection is broken, for example. It is also impossible to add cancellation semantics to long-running computations that may block, because there is no opportunity for a blocked client to notice or act upon a cancellation notification. The underlying issue is that the problem context is inherently concurrent, but the code is forced into a single serial execution. Such a serialization also prevents use of other processors even when inherently parallel solutions exist.

The following example illustrates all these issues. The Java Transactions API (a clone of the X/Open XA API) provides a transactional *coordinator* interface to be used by clients, and a separate *resource* interface implemented by databases and queue managers. Multiple resources are coordinated by the two-phase transaction coordinator. Now, consider a client that begins a transaction and involves two databases in that transaction before committing it.

```
tx = Transaction.begin(); // associate transaction tx with the current threads
db1.dostuff();            // db1 and db2 implicitly register themselves ...
db2.dostuff();            // ...  with tx from the thread-local context
tx.commit();              // launch two-phase commit of db1 and db2
```

The transaction coordinator's `commit` method can be (naively) implemented in a linear fashion in the following way. Note that the `prepare`, `commit` and `rollback` methods are all part of a standard *resource* interface.

```
class Coordinator {
  void commit() {
    // Prepare-phase
    for (Resource r: getRegisteredResources()) {
      if (!r.prepare()) {
         this.rollback()
         return;
      }

    logPrepare(); // record commit decision to disk-based transaction log
    // Commit phase
    for (Resource r: getRegisteredResources())
      r.commit()
  }
}
```

The synchronous request/response nature of the standard interfaces permits the code to faithfully follow the two-phase protocol. The problem, one among many, is that the client is blocked until `commit` is done. Neither the client nor the transaction coordinator can react to a timeout or cancellation once the commit process is on its way. Also, once the first prepare phase is over and the transaction is committed to disk (`logPrepare`), the rest can be done offline without stalling the client; this is not the case here. The resources ought to be preparable in parallel, but are forced to do so serially due to the blocking nature of `prepare`. Such problems are omnipresent, from system-call interfaces to RPC in enterprise-frameworks such as Java EE (Enterprise Edition), CORBA, .NET and so on.

What if the service were to offer completely asynchronous interfaces? For example Some network and file-systems, for examples, offer async or non-blocking interfaces (such as POSIX AIO), where the issuing of the request is separated from handling of the response. We believe that such a separation merely shifts the problem to the latter aspect. If the response is delivered as a callback, then the caller cannot easily deliver a response to *its* caller as a result value; it too is forced to defer its response in a separate callback. This is known as an inversion of control; instead of the code driving the control flow in a sequential, imperative fashion, one programs in a reactive style*. Facilities such as anonymous functions, block closures (in Smalltalk, Ruby and Scala[156]), or even Java's ersatz version in the form of inner classes avoid the problems faced by C, in that they permit response-handling code to be placed lexically close to the request-initiating code.[81] All these closure and callback-based solutions however have the reverse complication: they make it more complicated for the client to block. Further, it is easy to lose sight of the fact that the callback could be invoked in a different kernel thread; care must be taken to protect shared lexically scoped data, and barrier synchronization constructs should be used to keep in lock-step.

The third alternative to completely synchronous or completely asynchronous interfaces is a deferred synchronous approach; instead of passing in a callback function or a closure as a parameter to a request, the invocation immediately returns a *promise* or *future* object, allowing the client to proceed asynchronously while the request is being processed. The client can pass this object around as a first-class primitive, in some cases even to other processes (incidentally, a potentially non-scalable solution because it requires distributed garbage collection). The client can either poll the *future* object or wait on it until a response has been generated; either way, it blocks its own caller. To avoid this, it needs to return a *future* of its own which depends on the earlier future, and so on up the call stack. Such composition of futures need first-class support for higher-order functions to be achieved elegantly. The advantage of the deferred synchronous approach is that most implementations permit waiting on multiple *future* objects simultaneously, which allows the caller to initiate many requests and wait at one rendezvous spot for all responses.

The other problem with blocking the client (other than being unable to respond to any other event) is that in all mainstream approaches, the callstacks are mapped directly to kernel threads, which are expensive to create and to context-switch. What if threads were so lightweight that blocking them is a non-issue? All concurrency-oriented languages have constructs such as *spawn*, *cobegin* or *par* to spin off additional user-level threads. Our framework, Kilim provides such a facility as well: a Java program can quickly spawn hundreds of thousands of ultra-lightweight user-level threads (Chapter 6 quantifies the terms *lightweight* and *heavyweight*). Kilim's mailboxes, described later, provide deferred synchronous behavior and multiplexed waiting on multiple mailboxes. Note that it is still possible for the code to block a kernel thread by making a blocking system call. There are three options: (i) Use a facility such as *scheduler activations*[9] where the kernel scheduler coordinates with a user-level thread scheduler to dispatch I/O notifications (ii) tolerate the call, in the hope that the blockage will not last for too long (iii) provide an adapter that blocks the user-level thread, but internally issues asynchronous calls to the kernel. Since the first option is not widely available, our approach is to rely on the latter two; we will elaborate our design in Chapter 6.

In conclusion, the ideal control plane on the client side involves a combination of asynchronous messages and blocking calls in conjunction with lightweight threads and deferred synchronous constructs. For software evolution, one needs to be able to change the client from a synchronous to

*Sometimes called the Hollywood principle: "Don't call us, we'll call you".

deferred synchronous interface without modifying the interface. Note that asynchronous messages can be point-to-point (oneway) or broadcast (publish-subscribe mechanisms). In this thesis and in the Kilim framework, we concentrate on oneway messaging only.

Control plane: Service-side issues

We now survey the problems introduced by the function-oriented API approach from the service provider's point of view. Services and objects react to events and do not have the luxury of driving the control flow imperatively.

The most common pattern is to use top-level function calls as entry points into the service. However, the service cannot dictate the order in which to receive events, and must be multi-thread safe to coordinate multiple client threads. The trouble is compounded[117] in patterns such as subject-observer which involve *reentrant* callbacks or conversations with the client. This is fertile ground for deadlocks, as two threads can obtain locks in reverse orders, because locks are obtained on demand, decided by the flow of control.

This problem is closely aligned to another that crops up often in practice. Monitors and Java's *synchronized* construct protect all service calls with first-come first-served locks*. Since the lock queue is hidden as part of the monitor implementation, a service cannot know which clients are blocked, leave alone being able to handle clients out of order. Consider the case of a list of transactions waiting on an object's monitor. One might like the transaction with the earliest deadline to proceed first, or the one that started earliest (assuming heuristically that it has performed more work and is more expensive to rollback); such heuristics cannot be implemented in a system that does not expose the queue.

A more imperative or active way is for the service to explicitly state the events or messages it is willing to handle at any point in the code. Concurrent languages have constructs for message reception (*receive* in Erlang, *accept* in Ada), multiplexing (CSP's *alt*, for example), and pattern matching; these permit the service to use its own stack for processing the message. By pulling the message into its own context, a service can decide the order of event processing. Note that in the database example above, real databases are able to reorder transactions precisely because they handle the message in a different operating system process or machine (hence a different thread by construction) and also because they use asynchronous messages, not RPC, between client-side drivers and the servers.

The point is that function-call based interface *pushes* events to the service in an indeterminate order, while a messaging interface allows the service to *pull* messages in a relatively deterministic order.

A related problem with the function-call approach is that for any higher-level protocol that requires multiple function calls, the service has to squirrel away the intermediate state somewhere, to pick up from where it left off in the last call. This problem is routinely encountered by web service developers. In a case where the service has its own thread and stack, the stack automatically tracks intermediate state. In chapter 5 we will examine many continuation-based frameworks (Kilim included), some developed exclusively for web interaction.

The next problem with a functional API for the service side is that there can be at most one response to a given request, which cannot handle single-request/multiple-response patterns of streaming or result-batching; examples are, retrieving rows from a database, lines from a file and bytes from a

*Most monitor *implementations* are FIFO for fairness reasons, even if the specification does not require it

socket. In all these cases, the caller does not have to wait for the entire data to come in before processing can begin. Ideally, a server ought to be able to be push multiple responses, and correspondingly, the client ought to be able to pull the data at its own convenience. Traditionally (in C, Java), a callback-based interface is used here, but that solution inverts the flow of control.

2.2 Hardware trends and server-side design opportunities

Let us examine hardware and systems trends at both the micro and macro scales (within a processor, and within a data center respectively). These scales are not entirely unrelated, as cloud and utility computing services are serving an increasingly large number of personal and corporate computing and storage needs.

Chip-level processing. Computing speed has depended on our ability to pack more and more transistors on a single die, and on increasing clock frequencies. The heat generated by these two architectural features, and the resulting power and cooling requirements have garnered as much attention from data center administrators as the increases in speed. At the time of this writing, it is estimated that the cost of the energy expended by a server over its lifetime (four years) exceeds the cost of the server itself. The scale of operations can be inferred from the fact that one of the largest data centers at this time is a 192 MW facility. Benchmark metrics such as Sun Microsystems' SWaP (Space, Watts and Performance) seek to normalize standard performance metrics (such as TPC) by rack height and power consumption.

Another consequence of increasing processor speeds is the exponentially widening disparity between CPU and memory speeds. A simple cache miss to memory can cost about 200-300 clock cycles and dominates performance metrics. Many levels of cache hierarchy and complicated instruction level parallelism techniques have been introduced to fill this gap, but it is untenable to stretch it further.

These problems have forced a move towards chip-level multi-processing (CMP, also known as multi-core processors), where more processors are packed into a single die, and where each processor is simpler and operates at lower frequencies. This has the desirable effects of making the system more compact (by sharing on-chip resources such as L2 cache and memory controller), reducing the mismatch between processor and memory speeds, and running cooler. Since power consumption has a cubic relationship with frequency, one can either consume far less power or enable more cores within a fixed power budget.

Many-core chips are already a reality for niche applications such as graphics, multimedia and signal processing. Let us presume for a moment that chips with hundreds of cores will be commoditized[29] in general-purpose computing as well. In such a context, applications written with a single-threaded, event-oriented mindset will necessarily underperform. There are many more such applications than meet the eye. For example, the up- and down-calls in network protocol stacks,[78] and push-pull strategies of routers such as Click[106] are products of a single-processor mindset, and will need to be reengineered. Protocol layers will need to be independently schedulable to take advantage of a number of cores.

Chip-level multi-threading (CMT) is a still finer grain of division, where a single core's execution resources are mapped to several (but fixed) number of threads. When a thread's execution stalls due to an off-chip request, the processor moves on to another thread. Even as memory latency is the

single biggest factor in a server's performance, the memory bandwidth is increasing[163] at a rapid clip (60GB/s/chip). CMT allows many more threads to stand by waiting for memory.

The point is that hardware parallelism is getting cheaper. Clearly, single-threaded servers do not map well to this trend. As it turns out, present day *multi-threaded* applications are not always *transparently* portable to the many-core era either, as many of them rely implicitly on low processor counts. For example, using a shared mutable B-tree subjects the root to exponentially rising latch contention (with the number of processors). The solution might be to partition data (*sharding*) or replicate data amongst processors. In other words, we should treat a single multi-core or multi-processor box as the distributed computing environment that it is,[23] and pay attention to locality and interconnect traffic; such non-shared, messaging solutions resemble the ones adopted at the level of the data center. Section 2.4.3 discusses a few notable projects that are taking this route.

Fast networking. Changes in processor design apart, the drop in memory prices coupled with Gigabit networking represent an inflection point in the way online storage may be structured in the future. The RAMClouds project[138] proposes to aggregate the memories of thousands of servers to provide a 1000x increase in throughput coupled with very low latency (5-10μs). The new slogan is "RAM is the new disk, disk is the new tape".

Mixed language. Finally, there is increased usage of multiple languages on the server-side, often presented as a mixture of "systems programming languages" and "scripting" languages (e.g. Google's AppEngine, YouTube). These environments too present isolated subsystems that communicate using well-defined messages. We will argue later for an asynchronous messaging scheme.

2.3 Communicating Actors

Our thesis is that a system of actors with lightweight, memory-isolated threads and communicating with explicit messages is well equipped to handle the problems of present-day systems and the opportunities of the future. As Edward Lee says, "things happen to objects, actors make things happen".[117]

Advantages

Threads and blocking calls permit code to be written in a linear, imperative style. When threads are light enough to comfortably match or exceed the concurrency requirements of the *problem*, blockinga non-issue.

Memory isolation avoids the problems and mistakes such as low-level data races and the subtleties of memory models associated with the current shared-memory paradigm. We advocate an actor model that is not internally concurrent (*atomic*, in the taxonomy of Briot[35] et al. , so that all messages sent to it are handled in sequence. This architecture is equivalent to obtaining a monitor-lock per actor. The only way an external entity can affect an actor's state is by sending it a message, and the actor can check its invariants at message sending and reception points.

Messaging gives equal syntactic treatment to both requests and responses. By dissociating the number of responses for a given request, the actor can implement push-style responses. In fact, an actor can even autonomously send out a message; it does not always need to be reactive. Messages can be acted upon in a batch, which improves spatial and temporal cache locality.

Messaging avoids lost notifications because it unifies the control and data planes: a message contains the changed data as well as the signal. Note that messaging by itself does not prevent deadlocks, but

buffered messaging and the coarse-grained concurrency nature of actor systems in practice make the problem considerably more tractable. Channel-based messaging combines the ownership information of locks and the count of notifications inherent in semaphores.

Messaging pipelines permit easily visualizable dataflow models. These are particularly useful cases where the connection topology is static and the rates of message arrival and the cost of message processing are known in advance. For such networks, a scheduling plan can be created ahead of time.[143]

The actor architecture permits debugging, testing and performance profiling in isolation. Just as one debugs hardware by putting probes on the wire, message sending and receiving primitives can be instrumented to monitor the data that enters and exits an actor.

Static isolation guarantees can help garbage collectors and compilers to aggressively optimize use of the current hardware infrastructure, more than is possible today. Since all objects are accessed by a single actor at a time, and since in our approach, all objects reachable from a given object are also guaranteed to be visible by the same actor (and only that actor) at a given point in time, the garbage collector can perform heap traversals on a per-actor basis without fear of interference. A compiler (and the programmer) need not account for the possibility that reordering memory accesses and hoisting computations out of a loop may be visible outside that actor.

Issues

This section examines a few problems seen with actor-oriented systems *in practice*.

The term *actor* as defined originally by Hewitt[87] refers to an active object, whose identity is used for sending messages. In practice, a single mailbox used to receive messages could suffer performance problems, because messages retained in the mailbox tend to get pattern matched over and over until they are drained, an issue in languages such as Erlang. In addition, since an object can only have a single type (or a single instantiated type in the case of parametrized types), actor systems tend to be untyped to accommodate unrelated message types in the same mailbox.

Without low-level hardware support, pipelines and active objects (actor) are much more expensive than virtual function calls on passive objects, which necessarily puts a back pressure on the programmer to reduce the frequency of messaging. Although process calculi and actor systems can model shared memory approaches, in practice sharing passive objects is much more efficient at a fine-grained level. While a judicious use of message batching irons out performance differences, the problem with the actor model is that the coarseness of an actor (the grain of schedulability) is fixed at compile-time; as the actor becomes finer, the fraction of switching overhead increases and the opportunity to run a computation to completion (until invariants are re-established) is reduced.

In a language with pointers and mutable state, pointer aliasing is a burdensome issue. An actor can embed a pointer to some data structure in a message, and then subsequently modify that data structure. It is difficult to track such breaches of isolation. We will devote considerable attention to this issue in the next few chapters. One can avoid aliasing by making messages immutable (§2.1.2), or by copying data items, but these can both lead to considerable extra work for the garbage collector. Immutability must be a deep property in order to be usable, but given that there is no clear-cut difference between a resource such as a socket and an ordinary data item, one has to ensure that all objects in a message are immutable or pure (have no side-effects).

2.4 Related Work

Concurrent systems[16] have been the richest veins of research since the early days of computing; entire books can not cover all the research work. In this section, we discuss principles and languages that have been used to implement systems that have seen a measure of end-user usage, or those (such as software transactional memory) that are gaining currency at the time of this writing.

Briot[35] et al. supplies a useful taxonomy for object-oriented concurrent and distributed systems. Peter Van Roy[153] presents informal operational semantics of all concurrent models in a high-level language, Oz; in particular, the treatment of declarative concurrency and logic variables is recommended. While the determinism of declarative concurrency is valuable for reasoning and correctness, we have not yet pursued that avenue because our focus has been on those server-side systems that tend to be non-deterministic due to changing traffic patterns and failures. That said, the approach of wiring together components in a static dataflow configuration is intuitive and appealing; tiny sensor systems have been written in the communicating state machine model using the `nesc` programming language[69] (including the TinyOS[68] operating system), and the BBC's Kamaelia[102] framework is a python library with a vast number of media components.

2.4.1 Control Plane

Most concurrent languages provide a similar slew of facilities for spawning threads of control, and synchronizing transfer of control between them. We discuss thread creation and scheduling, and service invocation approaches on the client and service end of things following the discussion of §2.1.3.

One basic primitive common to all concurrent languages is a lightweight thread (the ability to create tens of thousands of them would classify it as lightweight for our purposes). The Kent Retargetable `occam-pi` project,[134] which combines CSP and the mobility features of the π-calculus, is the standard bearer for lightness and speed; it demonstrates switching and process-creation speeds in the range of tens of nanoseconds on 800Mhz processors.

Most concurrency languages, Kilim included, separate forking threads from joining. Some, like `Mesa`[111] and Erlang[13] take a functional approach where any function can be used as a starting point for a thread, and some, like Ada and Kilim take an object-oriented view (with defined entry points). Some such as `occam` and `Cilk` provide systemic and syntactic support for fork-join parallelism (such as a *par* construct), where the lifetime of a child thread never exceeds its parent's lifetime. This pattern is useful for parallel programming, but less useful for the workflow or staging style often seen in server systems.

Next, take service invocation. On the client end of things, languages (such as `Mesa`, `ABCL` and `SR`) allow the client to invoke a service synchronously, asynchronously or in a deferred synchronous fashion. John Reppy's Concurrent ML generalizes the deferred synchronous approach; instead of calling `receive` on a channel (which waits until a channel has data), one can call `recvEvt` on a channel. This returns a form of a future object called an `event`; the program can wait on this object using `sync`. Multiple event objects can be waited upon using a non-deterministic primitive, `choose`. The power of the system comes from the ability to compose combinators of event objects, and to `sync` on them. Kilim's mailboxes can be invoked in both synchronous or asynchronous fashion (§5.2), and additionally sup-

port callbacks for such higher-order event-composition. The limitations of the Java language's syntax render the Kilim approach considerably less elegant than using Concurrent ML.

On the service end of things, the ability to selectively receive messages is important to imperatively drive the flow of control. Kilim's facility for blocking on multiple mailboxes permits filtration on both the content and the source of events. `Erlang` and `Ada` provide syntactic support for selective message reception and guarded expressions (that need to be true in order to receive an event), but provide no control over source-level filtration.*

It is often necessary for an application to filter or reorder events based on the event's source. For example, a database may give a higher priority to a request from an older transaction. In most monitor-based approaches (including `Mesa`, `Ada` and Java) where event-delivery is via method calls, and in actor-based approaches with a single mailbox (`Erlang`), the event source is not provided by the framework. The ability to receive messages on multiple ports relieves the programmer from having to separate events by source. Such a pattern is used widely in practice; for example, the Internal Revenue Service in the US uses different P.O. box numbers for different criteria, thereby getting the tax payer (the source of the document) and mail service to implicitly sort the tax returns for them.

Finally, a word about scheduling control. Typically, concurrent languages and environments do not permit the application programmer to dictate thread placement, processor affinity and order of scheduling. We believe that spatial and temporal cache localities are important considerations for server-side programmers intent on squeezing performance, especially in many-core configurations where inter-core communication bandwidth may be a limiting factor. We will discuss this further in chapter 5.

2.4.2 Data Plane

The `Concurrent Pascal`[83] and `Mesa` languages were seminal influences on most concurrent languages, in protecting shared objects with monitors. As it turns out, we are still prone to the same issues that Lampson and Redell[111] cataloged in an `Mesa` experience report in 1980. As Per Brinch Hansen[84] argues, many of these problems can be traced to the fact that mainstream languages and environments have weakened the original definition of monitors, by permitting internal state to be visible publicly, and without an automatic lock. Cantrill and Bonwick,[39] on the other hand, presents a strong endorsement of shared memory and fine-grained locking; our only caveat is that most programmers do not have the expertise of the authors.

In locking approaches, the thread holding the lock can perform actions in a critical section, but if it is context-switched out, no other thread waiting on that lock makes progress. A *non-blocking* algorithm[65] does not employ locks or critical sections. The key difference is that unlike the locking approach where a thread only does its own thing, in a non-blocking approach, a thread performs some other thread's work if it is unable to do its own, as long as the overall system moves forward. Here is an analogy. Consider a laundry room with a washer and a drier. Assume that the washer has finished washing its load, but the person who started that cycle has gone out for a long walk. Your options are: (a) wait (block) until that person returns and moves the clothes to the drier or, (b) you transfer his clothes yourself before loading your own. The latter is non-blocking. It is particularly useful

*This is not a huge problem; the source task or process can embed its id in a message.

in preventing priority inversion (where a lower-level thread owns a lock and gets swapped out by a higher-level thread that happens to want that lock).

Non-blocking algorithms are decidedly non-trivial; there are a handful of experts in the world able to produce portable and correct lock-free code. Instead of general algorithmic patterns, the focus now is on packaging this expertise[72] into lock-free container classes (lists, queues and so on). The problems described in §2.1.1 remain, however; concurrency guarantees are only for the supplied container classes, not for the elements they contain. That said, we expect to use these for Kilim's internal data structures, such as mailboxes and schedulers (at the current stage of evolution, non-blocking structures in Java are more expensive).

A far more general, and more tractable, approach to lock-free updates of shared memory is Software Transactional Memory (STM). As with database transactions, all reads and writes within an `atomic` block belong to a transaction, and tracked accordingly in a separate log. At commit time, the writes are *atomically* committed to shared memory if and only if the read and write sets have not been updated by another transaction. Otherwise, the transaction either aborts or retries automatically.

Software transactions are composable[85,86] when implemented as language-supported lexical scopes and are deadlock-free regardless of the pattern of access. The flip-side is an indefinite number of aborts or retries — livelocks — in the presence of contention, as seen in optimistic database transactions. This is a non-trivial performance issue. Livelocks, unlike deadlocks, are hard to detect. As functions compose and more and more functionality gets pulled within the scope of a transaction, the window of contention increases dramatically. The other problem related to STM performance is that the more fine-grained the data to be mutated, the more the transactional state that needs to be maintained and compared at commit time.

STM is meant for providing serializability, and does not cater to looser consistency; for example, if an unrelated value (such as time) were to be read while processing a transaction, an update to that value would roll the transaction back. But worse is omitting to include the read of a relevant value inside a transaction; this has the same problem as lock omission. Cascaval[42] et al. express considerable skepticism about STM's viability as a replacement for fine-grained locking. The other counter to STM is that traditional locks are not heavyweight any more[14,189] (5-9 instructions for acquiring and releasing locks), and that whole program analyses exist[28] to remove unnecessary synchronization.

STM is however perfect for a language built upon principled (and rare) mutability. Languages such as Erlang, Clojure and Haskell are built on a foundation of immutable values and use functional data structures and multiple versions of data instead of in-place updates. Haskell provides the best approach to STM, wherein the type system statically verifies that within the scope of a transaction, only specially identified transactional types can be modified; no others can be modified and no I/O can be performed. The language can automatically retry the transaction because the old state and inputs are immutable and not consumed.

For most languages, mutability and pointers are the norm, and almost without exception, pointers embedded in messages allow contents to be simultaneously visible and mutable by both the sender and receiver. We seek *guaranteed* isolation.

The `E` and `Limbo` languages embrace in-place updates and pointers, yet guarantee isolation. Actors do not share memory and limit pointer aliasing, mutability and reachability to the confines of an actor. In order to communicate with others, an actor either exchanges immutable values or clones (deep-copies) object graphs. The cost of cloning is a concern in a low-level systems framework (protocol layers, for

example). Erlang too copies values between actors' heaps for ease of garbage collection,[99] but an analysis directed storage allocation scheme exists[41] to statically discover which allocations are purely local and which may be exported. Mesa's hardware-supported copy-on-write approach is another possibility; however, such an approach needs to distinguish between cloneable data values and (uncloneable) references to finite accountable resources (e.g. database connection).

Kilim is similar to E and Limbo, but the mechanism used to guarantee isolation is a significant point of difference. Our approach is to avoid cloning by sending messages by reference, but as we mentioned in Chapter 1, the type system enforces linear ownership of a message. This permits the receiver to assume ownership of the object and to mutate it in-place without the worry of the sender or any other actor being able to observe that object. The programmer has to explicitly clone a message if he needs to retain a copy; this renders the cost of cloning explicit in the code.

2.4.3 Isolation: not an isolated quest

We briefly describe a few recent research projects that serve as reference markers for the Kilim project. All these approaches have one architectural aspect in common: memory-isolated communicating actors.

The Multitasking Virtual Machine[53](MVM) provides multiple logical JVMs (*isolates*) within a single process. Communication is via serialized messages. The Incommunicado project[139] provides a lightweight mechanism for helping isolates communicate, by deep copying objects instead of serialization, because the machine and the class files used by the isolates are the same.

In "The End of an Architectural Era (It's Time for a Complete Rewrite)", Stonebraker[169] et al. propose rewriting databases as a collection of interacting database actors, called *sites*. Each site runs on a single core, and is a single-thread state machine that implements a query interpreter and has its own copy (or partition) of data and accompanying indices. Fault-tolerant storage is achieved not by archiving on disk, but by replicating copies in RAM among different sites.

The Barrelfish[22] project is a multi-kernel operating system based on similar principles. A kernel runs on a single core, and communicates using explicit asynchronous protocols with kernels on other cores. State is replicated, not shared. It is built on the premise that operating systems currently make optimizations and assumptions — related to the consistency model, the cache hierarchy, and the costs of local and remote cache access — that are unscalable and sometimes not even applicable to newer processors of the same family. They demonstrate the limitations imposed by the interconnect bandwidth, and show that message passing can be much more efficient than shared memory. At its current stage of evolution, it is unclear whether the message passing interface is available to the application level, whether one can have multiple actors per core, and whether actors in a process can run on separate cores.

The Erlang project was one of the original inspirations for Kilim. We wish to combine the extant experience and familiarity of the Algol family of languages with some of the systemic features of the Erlang environment (that is, we borrow ideas from the run-time, not the language), especially its failure recovery model.[12] The essential idea is to avoid exception handling altogether. Instead, *fail-fast* semantics let an actor crash, which in turn might lead to the crash of other dependent actors. At the topmost level of this cascade are a hierarchy of supervisor objects which re-spawn actors from known consistent state. Another key lesson from Erlang is the fallibility of micro-benchmarks. A C programmer would

not be faulted for assuming that an environment such as Erlang would be slow in a systems context, given the following features: interpreted code, isolated actors and data copied between heaps. Yet, end-user systems written in Erlang are more than acceptably fast in practice,[57] and win on many other counts such as fault-isolation and simplicity. Latest versions of Erlang have just-in-time compilers and true SMP support.

The project with an approach very similar to ours is the Singularity operating system, written in a language called Sing#. Singularity is a microkernel that relies on a type system (discussed in Chapter 4) for creating so called software isolated processes (SIPs). It supports pointers and garbage collection within a SIP, and separates message objects from internal objects. Messages are hierarchically structured, as in Kilim. Such features (along with security policies) permit device drivers, the kernel and user-code to be all part of the same address space and hardware protection ring. Messages are allocated from (and live in) a special *exchange heap* . Communication and synchronization is performed through channels, which can specify a (bidirectional) contract in the form of a finite state machine specification. This approach is more expressive and powerful than guard expressions in Erlang and ADA; it separates the interface automaton from internal code. We find this approach much more intuitive compared to session types.[70]

The current version of Kilim is but a starting point on the path that Singularity has blazed. The essential difference is that ours is an evolutionary approach, which allows a programmer to mix and match the vast set of extant Java libraries along with the messaging mindset. In the rest of this dissertation, we will show that it takes just a few modifications to retrofit the communicating actor approach into a mainstream language, and that it is possible to improve speed while guaranteeing safety and preserving the simplicity of sequential code.

3 Kilim Abstract Machine

This chapter presents an imperative and concurrent abstract machine called the Kilim Abstract Machine (KAM) to discuss the ideas presented in the next chapter. The KAM is an intermediate language similar in spirit to portable virtual machines such as the JVM*, the .NET CLR, Parrot and LLVM. The reason for an abstraction such as the KAM is to highlight a different approach to two aspects: memory model and concurrency-awareness.

The JVM[119] and others feature a shared memory architecture where threads are part of a library, and the language and execution engine are not aware of scheduling and interleaving semantics. All objects are potentially sharable; each object in the JVM is provided with a private lock that tracks (a) which thread currently owns it and, (b) the threads waiting to obtain the lock. The JVM provides modifiers such as `volatile` and `synchronized`, and primitives such as `wait` and `notify` to control thread execution and to specify consistency points where data updated by one thread is visible to other threads. Importantly, a method or class definition does not provide any hints about whether an object is concurrently accessible by multiple threads, which renders concurrent reasoning hard and forces compilers to be conservative in their (sequential) optimizations.

The KAM, in contrast, features a communicating sequential actors model where each actor is a thread with a private heap. It distinguishes, via a type system, the objects that are never going to leave the confines of an actor (*structs*), from objects that may be exchanged between actors, *messages*. A third category of objects, *mailboxes*, provides a globally accessible id-space and synchronization primitives. The type system constrains the shape and pointer aliasing of message objects.

This chapter describes the operational semantics of the machine and the next chapter presents the type system. Our choice of primitives and semantics is primarily driven by a pragmatic view; we wish to remain within the idiomatic ambit of environments such as the JVM, .NET and C, and we eschew features that may not translate to efficient code in these environments (tail call optimization is not always available, for example). At the same time, we chose isolation as a key feature of the underlying model to firstly simplify the mapping (and hence the proof of soundness) between the static and dynamic semantics and secondly, to be able to track that piece of memory in a garbage collector (similar to Singularity's Exchange Heap).

3.1 Syntax

Fig. 3.1 shows the syntax for the Kilim Abstract Machine visually separated between features commonly found in typical virtual machines such as the JVM, and Kilim-specific primitives for tasking and communications. A Kilim program consists of a list of named structures (`struct` for internal objects and `msg` for exportable messages). Mailboxes are typed; a mailbox typed as $M mbx$ may only be

*We will henceforth use the JVM as a representative model for the others.

$x, y, p \in$ variable names $\quad m \in$ method names $\quad b \in$ variable names (for mailboxes)
$S \in$ struct type names $\quad M \in$ message type names $\quad f \in$ field names
$lb \in$ label names $\quad n \in \mathbb{Z}$

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \overline{\mathit{Struct}}\ \ \overline{\mathit{Func}}\ \ \overline{\mathit{Msg}} \\
\mathit{Struct} & ::= & \texttt{struct}\ S\ \{\ \overline{\tau\ f}\} \\
\mathit{Msg} & ::= & \texttt{msg}\ M\ \{\ \overline{\tau\ f}\ \} \\
\mathit{Func} & ::= & \tau\ m(\overline{q\ \tau\ p})\ \{\ \overline{lb_{opt} : \mathit{Stmt}}\ \} \\
\mathit{Stmt} & ::= & x\ =\ \mathit{Rhs} \mid\ x.f\ =\ y \mid x.f\ =\ \texttt{null} \mid \texttt{ifeq}\ x\ y\ lb \mid \texttt{return}\ x\ \mid \\
 & & \texttt{spawn}\ m(\overline{x})\ \mid\ \texttt{put}\ b\ x \\
\mathit{Rhs} & ::= & \texttt{null} \mid n \mid x \mid x.f \mid \texttt{new}\ S \mid m(\overline{y}) \mid \\
 & & \texttt{new}\ M\ \mid \texttt{newmbx}\ M \mid \texttt{get}\ b\ \mid \texttt{select}\ \overline{b} \\
\tau \in \mathit{Type} & ::= & \texttt{int} \mid S \mid\ M\ \mid\ M\ \texttt{mbx} \\
q \in \mathit{Qual} & ::= & \texttt{whole}\ \mid\ \texttt{part}\ \mid\ \texttt{safe}\ \mid\ \texttt{none}
\end{array}
$$

Figure 3.1: KAM syntax. Shaded parts indicate Kilim support for communicating actors.

used to transfer message type M. Struct names, message names, mailbox types together with `int` form the set of *base types*.

A function contains a list of statements, which are simple three-address constructs. In addition to the usual statements for transfer of control (`if`, `return` and function calls) and assignment, one can spawn a new actor (`spawn`), create a new sharable mailbox (`newmbx`), put and retrieve messages from mailboxes (`put` and `get`), and use `select` to select a non-empty mailbox from a set of mailboxes (similar to CSP's `alt` construct but with no guard expressions). The `new` statement creates a structure or message object in the heap and returns an object reference.

Local variables contain integer values, object references and a distinguished `null` value. Variables are treated like untyped registers and spring into existence on assignment. Global variables and static class members are not present.

We shall distinguish between the terms *message object* and a *message*. The former refers to a single object, whereas the latter refers to the entire cluster of objects transitively reachable from a given (root) object. That is, when we refer to a program variable `x` as a message, we implicitly include all the objects reachable from that variable (via its various fields), and transitively so. To *export* or *send* a message is to use it as an argument to the `spawn` and `put` primitives, which remove the message (the entire cluster of linked message objects) from one heap and transfer it to another actor or a mailbox.

A message not reachable from any other object in an actor's heap is termed a *whole message*, otherwise it is a *part message*. The type system ensures that only whole messages are sent.

A function declaration contains a list of typed parameter declarations and a typed result. Each parameter has a name (unique to the function signature), a base type and a type qualifier. Type qualifiers are required for message types to dictate ownership and consumption policies; the next chapter delves into the details of qualifiers and their effects. Struct types must always be qualified with `none` (recall

that is an abstract syntax, not a higher-level language syntax exposed to the programmer; it is simpler to be explicit for the sake of uniformity). Each function parameter is a local variable and a function signature is the only means to associate a variable with a type; a type inference mechanism infers the rest (§4.2).

Well-formedness rules. A program is well-formed according to the following rules. Structure names and message type names are unique within a program, and while mutual and cyclic type references are permitted, message type names cannot be used to type fields in a `struct` declaration and vice-versa; this is the first step towards separating between internal and exportable objects. Mailboxes are the only entities sharable between actors, and both object and message declarations may freely refer to the same mailboxes Field names are unique within a structure and parameter names are unique within a function signature. Function names are globally unique (no overloading) and a distinguished function called `main` (no arguments) is required to serve as the program's entry point. Statements in a function may optionally have labels (unique within the function) and the target label in an `if` statement must exist within the function. The function supplied to `spawn` may only have mailbox and integer-typed parameters. Functions returning a message type can only return `whole` messages (the return qualifier in the method signature must be `whole`), and `put` and `get` are built-in primitives that take and return `whole` messages only.

We assume automatic garbage collection but omit features such as subtyping, virtual method dispatch, access privileges, exceptions and inline structs; these are largely orthogonal to our expository goals. §4.6 presents further details on how these issues are mapped to the Java setting.

3.2 Operational Semantics

The dynamic semantics of KAM is defined as a small-step transition relation on a *world configuration*. A world W consists of a set of actors, and a shared post office P which keeps track of a set of sharable mailboxes. The grammar for the runtime configuration is in Fig. 3.2. (Please note that the notation and nomenclature is summarized on page 113)

Each actor has a private heap that maps object ids to values, object ids and mailbox ids among them. Mailbox ids are globally unique names that can be shared by embedding them in messages. This is equivalent to *mobility* of channel names in the π-calculus model.[124]

The operational semantics is shown in two parts: actor-local transitions (Fig.3.4) where the world evolves by transforming a single actor A; the rest of the world (the other actors As and the post-office P) are unchanged and unaware of the changes to A. Fig. 3.5 shows transitions that have a global effect (the change may be visible to another actor); this includes modifications to the set of actors, the set of mailboxes in the post-office, or changes to a particular mailbox.

The initial world configuration $(A, \emptyset)$ is a single actor and an empty post office, where the actor A is initialized with an empty heap and an activation frame loaded with the body of the distinguished entry function `main`: $A = ((\texttt{body(main)}, \emptyset), \emptyset)$.

An actor ceases to exist after popping the last activation frame off the callstack, when it executes `return`. Note that the semantics for `return` are present in both figures 3.4 and 3.5 to account for the non-empty and empty callstacks respectively. The first one merely changes the local actor, the second changes the list of actors.

$W \in World$	$=$	$\overline{Actor} \times PostOffice$	Global configuration: bag of actors and a central post-office. One actor is non-deterministically chosen for reduction.
$A \in Actor$	$=$	$Callstack \times Heap$	Actor configuration: call stack with a private heap
$S \in Callstack$	$=$	$\overline{Frame}$	Call stack: list of activation frames
$F \in Frame$	$=$	$\overline{Stmt} \times Locals$	Activation frame: A list of instructions (Fig. 3.1) and a local variable store. The sole candidate for reduction (for this actor) is the head of the instruction list ($\overline{Stmt}$) of the frame at the top of the actor's call stack.
$\sigma \in Locals$	$=$	$Var \mapsto Val$	Local variable store (or register file): maps variable names to values.
Val	$=$	$Id \cup n \cup \{\texttt{null}\}$	Values. Id is a domain of unforgeable object identities (distinct from integers) that refers to both objects and mailboxes.
$H \in Heap$	$=$	$Id \mapsto Object$	Partial map of ids to objects. Actors and mailboxes each have a heap of their own.
$Object$	$=$	$FieldName \mapsto Val$	Objects are not nested; fields contain references to objects or mailboxes or are integer-valued or `null`.
$P \in PostOffice$	$=$	$Id \mapsto Mbx$	A collection of mailboxes indexed by mailbox id
$B \in Mbx$	$=$	$\overline{Id} \times Heap$	Mailbox. A collection of distinguished root object references, pointing to objects in the mailbox's own heap

Figure 3.2: KAM runtime configuration

A mailbox is a passive repository of messages. It stores whole messages in a private heap and keeps them separate using an additional list of *root* references. There is no lookup mechanism for mailboxes; an actor can use a mailbox only if it has obtained its id from another actor or created one itself. The fact that object and mailbox identities are unforgeable permits one to create private channels of communication. An actor can never directly affect the state of another actor (except for spawning one). Clearly, this means that for actors to communicate, they must use one or more mailboxes.

Put, Get. The `put` statement makes use of a *split* operation to partition the actor's heap into two disjoint parts (as in separation logic), one containing those objects reachable from the supplied object, and the rest. The former is separated from the actor's heap and merged into the mailbox. Note that this may leave dangling references from the remaining heap, and the actor's local variables, into the heap partition that is now merged into the mailbox.

In addition to transferring objects from an actor's heap to a mailbox's heap, `put` adds the root reference to a distinguished set of root references. A subsequent `get` does the reverse: it selects one of these roots from the given mailbox, and transfers all reachable objects into the actor's heap.

Select. The `select` statement takes a list of mailbox ids and returns the index of a mailbox ready for interaction. As implemented currently, `select` is only used to signal that a mailbox is `get`-ready (is non-empty), but does not signal a `put`-ready mailbox (signifying that the mailbox has one or more slots available). While easily rectified, this issue is orthogonal to the main thrust of our work.

$fields(T)$	$=$	$\langle\overline{\tau\ f}\rangle$, given a structure (correspondingly message) declaration of the form `struct` $T\{\overline{\tau\ f}\}$
$args(m)$	$=$	$\overline{p}$, the list of parameter names, given a method declaration of the form $\tau\, m(\overline{q\ \tau\ p})\{\overline{lb_{opt} : stmt}\}$
$body(m)$	$=$	$\overline{stmt}$, given a method declaration of the form $\tau\, m(\overline{q\ \tau\ p})\{\overline{lb_{opt} : stmt}\}$
$block(m, lb)$	$=$	$(s_k,\ \cdots\ s_n)$, the list of statements beginning at label lb in function m, which is of the form $\tau\ m(\overline{q\ \tau\ p})\{lb_0 : s_0,\ \cdots,\ lb_k : s_k,\ \cdots,\ lb_n : s_n\}$ and $lb = lb_k$ for some k.
$split\ (\rho, \mathrm{H})$	$=$	(H_{rem}, H_ρ) where $\rho \in Id$ and $H, H_{rem}, H_\rho \in Heap$ Splits heap H into H_ρ, the cluster of ρ, and the remaining heap H_{rem} $H_\rho = H(\rho')$, for all $\rho' \in dom(H)$ and $\rho' \in cluster(\rho, H)$ $H_{rem} = H(\rho')$, for all $\rho' \in dom(H)$ and $\rho' \notin cluster(\rho, H)$
$cluster(\rho, H)$	$=$	Smallest set X such that $\rho \in X\ \wedge\ (\rho' \in X \Rightarrow \rho \rightsquigarrow_{\mathsf{H}} \rho')$. (see defn. 3.1)
$merge(H_1, H_2)$	$=$	$\lambda\rho.\ \lambda f.\ \begin{cases} H_1(\rho)(f) & \text{if } \rho \in dom(H_1) \\ H_2(\rho)(f) & \text{if } \rho \in dom(H_2) \\ undef & \text{otherwise} \end{cases}$

Figure 3.3: Auxiliary function definitions used in Figs. 3.4 and 3.5

Remark. We assume blocking semantics for `get` and `select` and non-blocking semantics for `put`; the reason for not explicitly modeling an actor's scheduling state is that it has no bearing on actor isolation. That is, it wouldn't make an actor any more or less memory-isolated if `get` were to return `null` instead of blocking until the mailbox is non-empty.

The configuration is stuck when any actor is stuck, and halts when the list of actors is empty. An actor is stuck if it accesses a null or dangling reference, but importantly, is not considered stuck when blocked on a `get` or `select`.

Definition 3.1 (Reachability)**.** Given a heap H, we define n-step reachability as follows (f ranges over field names):

Let $\rho \rightsquigarrow^1 \rho'$ if $\rho, \rho' \in dom(H) \wedge \exists f.H(\rho)(f) = \rho'$

Let $\rho \rightsquigarrow^n \rho'$ if $\rho, \rho' \in dom(H) \wedge \exists f.H(\rho)(f) \rightsquigarrow^{n-1} \rho'$

We use the following notation to indicate a path between two objects in the heap.

$\rho \rightsquigarrow_{\mathsf{H}} \rho' \triangleq \exists n.\rho \rightsquigarrow^n \rho'$.

$$((\; x = \texttt{null} \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma[x \mapsto \texttt{null}]) :: S,\ H)$$

$$((\; x = n \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma[x \mapsto n]) :: S,\ H)$$

$$((\; x = y \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma[x \mapsto \sigma(y)]) :: S,\ H)$$

$$((\; x = y.f \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma[x \mapsto H(\sigma(y))(f)]) :: S,\ H)$$

$$((\; x = \texttt{new}\ T \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma[x \mapsto \rho]) :: S, H), \text{ where } \rho \text{ is fresh}$$

$$((\; x = m(\overline{y}) \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s_m, \sigma') :: (\; x = m(\overline{y}) \; :: s,\ \sigma) :: S,\ H)$$
where $s_m = body(m)$ and,
$\sigma'[p_i \mapsto \sigma(y_i)]$ and $p_i \in args(m)$

$$((\; x.f = y \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma) :: S,\ H[\sigma(x)(f) \mapsto \sigma(y)])$$

$$((\; x.f = \texttt{null} \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s,\ \sigma) :: S,\ H[(\sigma(x),\ f) \mapsto \texttt{null}])$$

$$((\; \texttt{ifeq}\ x\ y\ lb \; :: s,\ \sigma) :: S,\ H) \quad \rightsquigarrow \quad ((s',\ \sigma) :: S,\ H)$$
where $s' = \begin{cases} block(m_c, lb) & \text{if } \sigma(x) = \sigma(y) \\ s & \text{otherwise} \end{cases}$

$$((\; \texttt{return}\ x \; :: s,\ \sigma) :: (\; x' = m(\overline{y}) \; :: s', \sigma') :: S,\ H) \quad \rightsquigarrow \quad ((s',\ \sigma'[x' \mapsto \sigma(x)]) :: S,\ H)$$
See also `return` in Fig. 3.5

Figure 3.4: KAM Operational Semantics (part 1). Actor-local transitions of the form $(A \bullet As, P) \rightsquigarrow (A' \bullet As, P)$, where $A = ((\; stmt \; :: s, \sigma) :: S, H)$. S is the callstack, H is the actor-local heap (Fig. 3.3). The current function at the top of the call stack is denoted by m_c. (Note: m_c is not explicitly tracked in the rules, for simplicity of exposition)

$(((\texttt{spawn}\ m(\overline{y})::s,\ \sigma)::S,\ H)\bullet As,P) \rightsquigarrow (((s,\ \sigma)::S,\ H)\bullet\ ((s'::\sigma')::[\,],H')\bullet As,P)$
where $s' = body(m)$, $H' = \emptyset$
$\sigma'[p_i \mapsto \sigma(y_i)]$, $p_i \in args(m)$

$(((\texttt{return}\ x::s,\sigma)::[\,],\ H)\bullet As,P) \rightsquigarrow (As,P)$
See also `return` in Fig. 3.4

$(((\ b = \texttt{newmbx}\ T::s,\sigma)::S,\ H)\bullet As,P) \rightsquigarrow (((s,\sigma[b\mapsto\rho])::S,\ H)\bullet As,P)$
ρ is fresh.

$(((\ \texttt{put}\ b\ x::s,\sigma)::S,\ H)\bullet As,P) \rightsquigarrow (((s,\sigma)::S,\ H')\bullet As,P')$, where
$\rho_x = \sigma(x)$, $(H',H'') = split(\rho_x,H)$
$\rho_b = \sigma(b)$, $P(\rho_b) = (roots_b,H_b)$
$P' = P[\rho_b \mapsto B']$
$B' = (roots_b \cup \{\rho_x\}, merge(H_b,H''))$

$(((\ x = \texttt{get}\ b::s,\sigma)::S,\ H)\bullet As,P) \rightsquigarrow (((s,\sigma[x\mapsto\rho])::S,\ merge(H'',H))\bullet As,P')$
$(H',H'') = split(\rho,H_b)$
for some $\rho \in roots_b$ and where
$\rho_b = \sigma(b)$, $B = P(\rho_b) = (roots_b,H_b)$
$P' = P[\rho_b \mapsto (roots_b \setminus \{\rho\},H')]$

$(((\ v = \texttt{select}\ \overline{b}::s,\sigma)::S,\ H)\bullet As,P) \rightsquigarrow (((s,\sigma[v\mapsto\rho_b])::S,\ H)\bullet As,P)$
for some $\rho_b \in \{\sigma(b_0),\ldots,\sigma(b_n)\}$, and
$|roots(P(\rho_b))| > 0$

Figure 3.5: KAM operational semantics (part 2). World transitions of the form $(A\bullet As,P) \rightsquigarrow (A'\bullet As',P')$

Type System 4

The Kilim Abstract Machine, modeled on the JVM, stores untyped values (object references and integers) in the local variables and the heap. This chapter describes a type system that introduces aliasing and ownership constraints to eliminate two broad classes of errors.

One set of errors arises from using a value in the wrong context: using an object reference as an integer for example, or using an undeclared field name, or using an object of one type where another is expected, etc. Such simple type errors are easily prevented with a conventional Java-style type system that reasons about the types of values in the local variable store and in the heap. This is done in the first phase of type verification (§4.2).

However, simple type verification is not enough. One of the key aspects of languages with references is aliasing, where there are multiple ways of obtaining the reference to a given object. Aliasing takes the form of *variable* aliasing when more than one variable refers to an object, and *heap* aliasing, when there are multiple field references into an object (multiple incoming edges, if you visualize the heap as an object graph).

For all its efficiency benefits, aliasing is a constant thorn in the side of modular reasoning. Consider the following simple example:

```
1 class C {Socket sock}
2
3 void foo (C x, C y) {
4   close(x.sock);
5   process(y.sock);
6 }
```

This example closes a socket that belongs to `x`, and uses one that belongs to `y`. Clearly, it is a problem if `x` and `y` refer to the same object. But even if they were different objects, we have no information about whether they refer to the same underlying socket. The issue is that in pointer-based languages, one cannot tell the difference between *refers* and *belongs*. The second issue is that `foo` modifies objects reachable through its arguments, and these modifications persist after `foo` returns. Specification languages that describe a method's side effects tend not to be used in practice, either because they are incomplete or verbose; further, the deeper the nesting of method invocations, the more the possible side effects. In addition, a function's side effects may depend on the aliasing of its input arguments, which is why we cannot reason about a function on its own.

The paper "The Geneva Convention on the treatment of object aliasing" by Hogg[93] et al. is an excellent overview of the perils. They categorize all approaches to deal with the aliasing issue in terms of:

- *Alias Detection*. These approaches perform post hoc alias detection. However, given that precise

alias detection is undecidable[148] (they must necessarily be conservative), and that whole program analyses are NP-hard, we do not use them.

- *Alias Advertisement*. Programmer-supplied annotations advertise whether (and how) they create new aliases to parameters.
- *Alias Prevention*. Aliases can be implicitly and automatically prevented by many techniques, an important one being treating assignment as a deep-clone operator. We do not use these methods for efficiency reasons. As a side note, we prefer potentially expensive operations such as messaging and context switching to be explicitly spelled out, as a constant and visible reminder to the programmer.*
- *Alias Control*. These schemes enforce explicit control over aliasing, in that it is a compile-time error if an undesirable alias is created. We take this route, as described in the next section.

4.1 Design Choices

We now outline the key principles and design decisions for a second phase of the type verification process.

- **Ownership transfer**. As we mentioned in Chapter 1, we retain the mutable mindset of Java and other current systems programming languages, but make transfer of ownership a central feature in both the KAM and the type system, to ensure that an object is exclusively owned by a single actor at any time. The KAM does not offer immutable objects because we do not (yet) distinguish between objects that represent resources vs. normal data values; just because an object is immutable (say a file handle) does not mean that it is not a resource with ownership constraints. The type system outlined in this chapter enables a compiler to ensure that there are no dangling references.

- **Alias control only for messages**.

  Empirical evidence from industry suggests that although aliasing is a nuisance, it tends to be easily manageable in practice for purely *sequential* code, especially when garbage collection is provided. This is because sequential patterns of data access are well understood and bugs in such scenarios are usually detected and fixed easily. On the other hand, aliasing and concurrency are a toxic cocktail. Since messages are the only *kind* of objects to be exchanged among concurrent KAM actors, we enforce aliasing and ownership handoff constraints only on messages. Such constraints may be considered too onerous for objects in general. We discuss more elaborate methods of alias control in the section on related work (§4.7).

- **Tree Structure**. Kilim does not allow a message object to be heap aliased; it may be reachable from only one other object at most, and that too via a single field reference. That is, the following code is unacceptable: `x.f = y; x.g = y`. Multiple local variable aliases are permitted, however, as they are easy to track with a straightforward intraprocedural analysis.

  As we saw in §1.3, tree-structured messages are the norm in practice. In our experience, only compilers tend to communicate DAGs between stages, while typical server software use tree-structured events and messages by convention. For this reason, it is not a stretch to harden this

*This is the reason for the `@pausable` annotation, instead of the weaver discovering it behind the scenes.

pattern into a requirement. Tree-structured messages simplify alias analysis tremendously, and are fast to serialize onto a network.

We say an object is *owned* if there exists another object in the heap that refers to it via some field: $owned(\rho, H) \triangleq \exists \rho' . \exists f . H(\rho')(f) = \rho$

Definition 4.1 (Unique Ownership)**.** An object is *uniquely owned* if there is only a single "incoming arrow".

$unique(\rho, H) \triangleq (H(\rho_0)(f) = \rho) \wedge (H(\rho_1)(g) = \rho) \implies \rho_0 = \rho_1 \wedge f = g$ for all ρ_0, ρ_1, f and g

This definition permits loops (including self-loops) and does not care whether or not an object is garbage collectable. However, the type rule for field assignment (S-PUTFIELD on p. 63) prevents cycle creation.

- **Deep safety**. The type system adds the notion of a *safe* message to the KAM. A message marked safe can never be exported, updated, or *grafted* (assigned to another object's field). Safety is deep: if an object is *safe*, all members of its cluster are guaranteed safe as well. Contrast this to keywords such as Java's *final* or C++'s *const*, where marking a reference-valued variable or field as *final* does not prevent the object it points to from being mutated, rendering the concept fairly useless in practice. We will later use a predicate $safe : Id \mapsto \{\texttt{true}, \texttt{false}\}$, which is initialized from a type qualifier, discussed next.

- **Local, modular reasoning**. We use annotations or type qualifiers on method parameters as the sole mechanism to convey extra semantic information and programmer intent. This permits each function to evolve independently (as long as it holds up its end of the contract) and to be analyzable independently and quickly. We use a type system to infer and reason intra-procedurally about types akin to the JVM verifier. In addition, the type system tracks the state (*typestate*) of each message object, described next.

- **Type qualifiers for consumption semantics**.

  Our annotations ($\{\texttt{whole}, \texttt{part}, \texttt{safe}\}$) are used to specify the initial typestate of the supplied argument, and secondly, serve as an ownership or consumption contract between caller and callee.

  - The `safe` annotation is the mechanism to render the corresponding formal argument and its cluster *safe* (as defined above) for the duration of the function's activation, including nested calls. The callee guarantees that objects in the cluster remain unmodified and unexported. Objects marked `safe` can be passed as arguments to another method provided the corresponding parameters are annotated `safe`.

```
1  void foo (@safe M X) { // for some message class M
2   print (x);     // ok: print is safe.
3   put (b, x);    // error: put defined as void put(Mbx, @whole T)
4   y = x.f;       // y is transitively safe, ..
5   put (b, y);    //  ... hence error
6
7   z = new M;
8   z.f = x;       // error: safe object cannot be assigned ..
9   x.f = z;       //  .. nor assigned to
10 }
```

– Recall the definition of *whole* from p. 40; it refers to an object that is not reachable from any other object in the heap. The `whole` parameter annotation states two things: that it requires a *whole* object to be supplied, and second, it is the function's intention to assume ownership of the whole message (the object and its cluster) and to do with it as it pleases, including exporting to another actor, or grafting it on to another message object (assigning to another object's field), or letting the garbage collector collect it. We will refer to all these actions as *consuming* a message.
After the call is over, the caller must not use any variable references to any part of the cluster; it is expected to sever its ties to the object completely. To preserve the tree structure, only whole messages can be on the right hand side of an assignment statement (this restriction is only for message types). Of course, once assigned, they cease to be *whole*.

```
1 void foo (@whole M x, @whole M y) { // for some message class M
2  print (x);  // Ok. print is safe, and does not consume x.
3  x.f = y;    // Ok.
4              // y is no longer whole (it is part of x's cluster)
5  z = y.g;    // z reachable from x and y.
6  put (b, y); // Error. y is not whole.
7  put (b, x); // ok.
8              // x's cluster (including y and z) has been sent
9              // x, y, z are not valid any more (consumed)
10 print (z);  // Error.
11 }
```

– The `part` annotation is similar in that the callee can consume all or some part of the cluster, *except* for the argument object itself; that is, the callee is not allowed to assume ownership of the argument object (although it can mutate it). The other objects in the cluster can be exported (as long as they are dissociated from their parent objects). This annotation is typically used for the receiver in an OO setting; calling a method doesn't consume the object itself, but its cluster may look quite different after the call. Going with a rough fishing analogy, it is as if the argument object is the hook; whether it comes back with a fish or without the worm is not known in advance, but the hook itself is not lost.

```
1 void bar (@part M z) {..,] // for some message class M
2
3 void foo (@part M x, @whole M y) {
4  y.f = x;    // Error. x is not whole.
5  x.f = y;    // Ok.
6              // y is no longer whole (it is reachable from x)
7
8  z = x.g;    //
9  x.g = null; // z is now whole (severed from x, assuming x was a tree to begin with)
10
11 bar(x);     // Ok.
12             // x not consumed, but any variables dependent on x
13             // will not be trusted. Hence ...
14 print (y);  // Error. y dependent on x (from line 5)
15 put (b, z); // Ok. z is whole.
16 z = x.g;    // Ok. x has not been consumed.
17 }
```

We have thus far spoken of a method's parameters taken one at a time, in that a parameter annotation dictates the pre- and post-states of the corresponding argument. There is a second part of the contract between caller and callee, which applies to *pairs* of arguments: each non-safe argument must refer to clusters that are disjoint from those of *every* other argument. The following example illustrates the problem if such a restriction were not enforced:

```
1 void foo (@whole x, @safe y) {
2  send(x);
3  print(y);
4 }
5
6 // elsewhere
7 ...
8 b = a.f;   // b belongs to a's cluster
9 foo(a, b); // error: non-disjoint arguments
```

Here, `foo` seems perfectly in order by itself; it is allowed to send a message x that it owns, and to use the object y safely. However, if x and y were part of the same message, as happens in a caller-side aliasing relationship (line 9), the actor inadvertently uses the object after sending it (line 3). We thus enforce the rule that an argument supplied to a non-`safe` parameter must be disjoint from all other arguments to the called method.

Such a disjointness constraint is common to all linear type systems; the idea is that since the analysis is intra-procedural, each function can be independently analyzed with the knowledge that its mutable parameters have no aliasing relationship at function entry.

Kilim annotations are designed to provide isolation *guarantees*, and are not just passive pieces of information. The reduction in verbosity comes from choosing convenient defaults, and by putting constraints on messages (and messages only). This helps distinguish our approach from other annotation approaches, including more general ones such as ownership types. Only widespread usage can tell whether the Kilim design decisions are deemed acceptable or whether the level of verbosity is small enough.

4.1.1 Putting them all together

The design points outlined above signal an important shift in strategy from the earlier chapter. Instead of speaking of isolation between actors at message transfer points, we now enforce isolation and message structure at *all* method calls (even within an actor). In the KAM, sending a message may leave dangling references behind. The type system of this chapter is far stricter, in that it forces us to forget about any *potentially* dangling references after *every* method call.

This strategy allows us to reason about each method in isolation, which makes it robust (or less brittle) in the face of software evolution. It also allows the type system to treat a mailbox as a library, not as a built-in primitive, which permits us to add other concurrency mechanisms in the future:

```
1 class Mailbox<T> {
2   @whole T get() {..}
3   void put(@whole T msg) {...}
4 }
```

Recall that this strategy is only for message types; other types are not constrained any further. The downside of course is the necessity of annotating every message parameter. From experience, making `safe` the default annotation reduces the syntactic burden substantially.

4.2 Phase-1: Base Type Inference

This phase distinguishes between base types: messages, mailboxes, structs and integers. This phase is similar to the JVM type inference and verification process, but is considerably simpler than the JVM due to the lack of the inheritance. Fig. 4.1 shows the type lattice. The type $\top$ refers to an unknown type; there is no top level type such as Java's `Object`. The special type `null` is held only by a distinguished reference also called `null`.

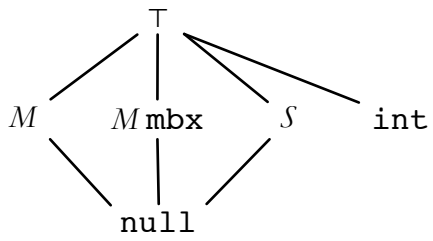


Figure 4.1: Kilim's type lattice for base types. $\top$ represents an unknown type.

The phase-1 type rules are shown in Figure 4.2. The rules are of the form $\Gamma \vdash stmt \Rightarrow \Gamma'$, where Γ is the type environment, defined as a total function:

$\Gamma : Var \rightarrow Type \cup \{\texttt{null}, \top\}$.

The algorithm is standard to the JVM type verification process.[118] We construct a control flow graph (CFG) based on implicit transfer of control between adjacent statements in a method and on explicit transfer of control instructions like `ifeq` and `return`. A forward dataflow analysis is performed on the join semi-lattice $\langle \Gamma, \sqsubseteq \rangle$ whose ordering and join operators are defined component-wise in terms of the type lattice (*lub* is the least upper bound):

$$\begin{aligned} \Gamma_1 \sqsubseteq \Gamma_2 &\triangleq \forall v.(\Gamma_1(v) \leq \Gamma_2(v)) \\ \Gamma_1 \sqcup \Gamma_2 &\triangleq \lambda v.lub(\Gamma_1(v), \Gamma_2(v)) \\ &\quad \text{where } v \in dom(\Gamma_1) \cup dom(\Gamma_2) \end{aligned}$$

The starting point for this analysis is the CFG node for method entry, seeded with an initial environment Γ_{init} for the current method m:

$$\Gamma_{init} \triangleq \lambda v. \begin{cases} \tau_i & \text{if } v = p_i, \text{where sig(m)} = \overline{q_i \tau_i} \rightarrow \tau_r \text{ and args(m)} = \overline{p} \\ \top & \text{otherwise} \end{cases}$$

Remark. We have described our approach in terms of intraprocedural data flow analysis and abstract interpretation,[132] as opposed to modern type systemic approaches that are typically specified in a functional style.[145] It allows us to test our ideas as additions to existing toolchains, with no changes

$$\text{T-Null}\ \frac{}{\Gamma \vdash \boxed{\texttt{x = null}} \Rightarrow \Gamma[x \mapsto \texttt{null}]} \qquad \text{T-Const}\ \frac{}{\Gamma \vdash \boxed{\texttt{x = n}} \Rightarrow \Gamma[x \mapsto \texttt{int}]}$$

$$\text{T-Assign}\ \frac{y : \tau \in \Gamma}{\Gamma \vdash \boxed{x = y} \Rightarrow \Gamma[x \mapsto \tau]} \qquad \text{T-New}\ \frac{\Gamma' = \Gamma[x \mapsto T]}{\Gamma \vdash \boxed{x = \texttt{new}\ T} \Rightarrow \Gamma'}$$

$$\text{T-PutField}\ \frac{x : T, y : \tau \in \Gamma \qquad f : \tau \in \mathit{fields}(T)}{\Gamma \vdash \boxed{x.f = y} \Rightarrow \Gamma} \qquad \text{T-PutNull}\ \frac{x : T, y : \tau \in \Gamma \qquad f : \tau \in \mathit{fields}(T)}{\Gamma \vdash \boxed{x.f = \texttt{null}} \Rightarrow \Gamma}$$

$$\text{T-If}\ \frac{x : \tau_0, y : \tau_1 \in \Gamma \qquad \tau_0, \tau_1 \in \{\tau, \texttt{null}\} \text{ for some } \tau}{\Gamma \vdash \boxed{\texttt{ifeq } x\ y\ lb} \Rightarrow \Gamma} \qquad \text{T-Return}\ \frac{\mathit{sig}(m) = * \rightarrow \tau_r \qquad x : \tau_r \in \Gamma'}{\Gamma \vdash \boxed{\texttt{return } x} \Rightarrow \Gamma'}$$

$$\text{T-Put}\ \frac{x : M \in \Gamma \qquad b : M\texttt{mbx} \in \Gamma}{\Gamma \vdash \boxed{\texttt{put } b\ x} \Rightarrow \Gamma} \qquad \text{T-Get}\ \frac{b : M\texttt{mbx} \in \Gamma}{\Gamma \vdash \boxed{x = \texttt{get } b} \Rightarrow \Gamma[x \mapsto M]}$$

$$\text{T-Select}\ \frac{b : M\texttt{mbx} \in \Gamma, \text{ for all } b \in \overline{b}.}{\Gamma \vdash \boxed{x = \texttt{select } \overline{b}} \Rightarrow \Gamma[x \mapsto \texttt{int}]} \qquad \text{T-NewMbx}\ \frac{\Gamma' = \Gamma[b \mapsto M\texttt{mbx}]}{\Gamma \vdash \boxed{b = \texttt{newmbx } M} \Rightarrow \Gamma'}$$

$$\text{T-GetField}\ \frac{y : T \in \Gamma \qquad f : \tau \in \mathit{fields}(T) \qquad \Gamma' = \Gamma[x \mapsto \tau]}{\Gamma \vdash \boxed{x = y.f} \Rightarrow \Gamma'}$$

$$\text{T-Invoke}\ \frac{\mathit{sig}(m) = \overline{q_i T_i} \rightarrow \tau_r \qquad \overline{\tau_i} = \Gamma(\overline{y}) \qquad \tau_i \leq T_i}{\Gamma \vdash \boxed{x = m(\overline{y})} \Rightarrow \Gamma[x \mapsto \tau_r]}$$

$$\text{T-Spawn}\ \frac{\mathit{sig}(m) = \overline{q_i T_i} \rightarrow \tau_r \qquad \overline{\tau_i} = \Gamma(\overline{y}) \qquad \tau_i \leq T_i}{\Gamma \vdash \boxed{\texttt{spawn } m(\overline{y})} \Rightarrow \Gamma}$$

Figure 4.2: Type Inference: Phase 1. (m_c denotes the function currently being analyzed)

required of the syntax or the existing type system. Second, annotations at the bytecode level allow us to experiment with other JVM languages such as Scala and Clojure. Finally, we also pursued this approach due to a parallel research interest in information flow and security,[59] which too requires one to track aliasing and provide flow-sensitive inference.

4.2.1 Phase-1 Termination and Soundness: Proof sketch

Each transfer function $[\![stmt]\!]$ of Fig. 4.2 is monotonic. Formally,

Proposition 4.1 (Monotonocity of $[\![stmt]\!]$). *For all statements stmt and for all pairs of type environments Γ_1 and Γ_2 such that $\Gamma_1 \sqsubseteq \Gamma_2$, we have $[\![stmt]\!](\Gamma_1) \sqsubseteq [\![stmt]\!](\Gamma_2)$.*

Termination of the phase-1 flow analysis algorithm follows automatically[5] from the monotonicity of the transfer functions together with the lattice $\langle \Gamma, \sqsubseteq \rangle$.

We claim soundness of this phase of type inference and verification by drawing parallels to the soundness of the JVM verifier type inference algorithm. Klein and Nipkow[104] demonstrate a mechanized operational and static semantics for a subset of the JVM, which is adequate for mapping our static and dynamic semantics. The phase-1 analysis does not deal with subtypes, arrays, casts and local

subroutines, some of which render the JVM type system unsound,[118] in the absence of a run-time check. The Java equivalent of a KAM message type is a super class `Message` inherited by all other message classes. The instructions related to spawning threads and mailboxes can easily be translated to equivalent Java library methods (as they are in the Kilim implementation as well).

4.3 Typestates and Heap Abstraction

The second phase of static verification enforces structural, aliasing and ownership handoff constraints on message objects. This section describes an abstraction called the *shape state graph*, the domain of analysis for the message verification rules described in §4.4. The next two subsections describe the two components of the shape state graph, *typestates* and the *shape graph* abstraction.

4.3.1 Object typestate

We touched on two important aspects in the introduction. The first is adding safety via the `safe` annotation, the guarantee being that a safe object is protected from modification, from grafting, and from export. The second is to eliminate heap aliasing.

Let us imagine that the KAM is instrumented with a run-time type system that checks the predicates $safe(\rho)$ and $owned(\rho)$ before any heap manipulation and function call. Recall that $safe(\rho) = \texttt{true}$ for all objects ρ reachable from a `safe` parameter (all others are implicitly mutable); a safe object remains so during the lifetime of this function's activation. However, it is possible to protect an unsafe object temporarily in a nested function call activation, if the argument qualifier is `safe`.

For convenience of analysis, we combine the two orthogonal predicates *safe* and *owned* into a single state map $\varsigma : Id \mapsto \{\mathbf{r}, \mathbf{w}, \mathbf{f}\}$, as shown below:

	$safe(\rho)$	$\neg safe(\rho)$
$owned(\rho)$	$\varsigma(\rho) = \mathbf{r}$ (readonly)	$\varsigma(\rho) = \mathbf{w}$ (writable)
$\neg owned(\rho)$	$\varsigma(\rho) = \mathbf{r}$ (readonly)	$\varsigma(\rho) = \mathbf{f}$ (free)

If an object is *safe*, it doesn't matter whether it is owned or not. A *writable* object can be written to, and a *free* object is one that is both writable and unowned. To this we add two more states: `null`, to track nullified variables and $\top$, to indicate an unknown or invalid state. These states can be totally ordered as $\texttt{null} < \mathbf{f} < \mathbf{w} < \mathbf{r} < \top$ to reflect a subtyping relationship.

The following table shows the preconditions enforced by the run-time type system (we use the notation ρ_v to refer to the object $\sigma(v)$). They formalize the statements made earlier in §4.1. (We show only the instructions that pertain to object references)

Instruction	Precondition
x = y.f	$\varsigma(\rho_y) \leq \mathbf{r}$
x.f = null	$\varsigma(\rho_x) \leq \mathbf{w}$
x.f = y	$\varsigma(\rho_x) \leq \mathbf{w}, \quad \varsigma(\rho_y) \leq \mathbf{f}$
return x	$\varsigma(\rho_x) \leq \mathbf{f}$, where $sig(m) = * \rightarrow \tau_r$ and τ_r is a message type
$m(..,x_i,...)$	1. $\varsigma(\rho_{xi}) \leq \begin{cases} \mathbf{r} & \text{if } q_i = \texttt{safe} \\ \mathbf{w} & \text{if } q_i = \texttt{part} \\ \mathbf{f} & \text{if } q_i = \texttt{whole} \end{cases}$, where $sig(m) = (..., q_i \tau_i, ...) \rightarrow \tau_r$
	2. $(cluster(\rho_i) \cap cluster(\rho_j) = \emptyset) \vee q_i = q_j = \texttt{safe}$
putb,x	$\varsigma(\rho_x) \leq \mathbf{f}$

The precondition for method invocation checks that the clusters of two arguments are disjoint, if either or both of them is non-safe. The precondition for $x.f = y$ requires that the object on the right hand side is *free* (unowned, writable), and is subsequently not permitted to be used on the right hand side of a field assignment as long it remains owned. This restricts the object graphs to be constrained to self-loops, rings and trees; however, an object may not be part of more than one cycle. Such non-overlapping cycles are benign in that there is nothing useful that can be done with an object in a cycle; it is unexportable and unassignable because it is not *free*. In any case, we will show later that the static rules presented in §4.4 rule out the possibility of cycles entirely.

4.3.2 Shape graphs

This section describes a scheme to statically track how objects connect to each other, whether an object is owned or not, and which objects belong to its cluster. There are any number of heap and pointer analysis schemes, all necessarily conservative (because precise alias detection is undecidable[148]) and the tradeoff is between speed of analysis, amount of programmer-supplied information, and precision. One property that we are particularly interested in is *strong nullification*, when two objects become no longer connected. Most analyses are unable to guarantee this property.

```
1 x = y.f      // y points to x via f
2 y.f = null   // y and x become no longer connected.
3 put ... y    // consume y and anything reachable
4 print x      // ok.
```

At run time, an actor's heap is a graph containing one or more independent messages at a given program point. An infinite number of runs of a program will create an infinite number of such concrete heaps (at the same program point). It is possible to statically approximate *all* such heaps with a single abstract heap graph containing a finite set of nodes and edges. Each node of this graph is an equivalence class representing an infinite number of run-time objects. If in any given run, two run-time objects are directly connected, we model the connectivity as an edge between the corresponding abstract nodes. Therefore, each edge of the graph represents the fact that there exists *some run* in which two runtime objects represented by the corresponding nodes are connected.

There are many ways of partitioning nodes into equivalence classes. We adopt the heap abstraction from the shape analysis work of Sagiv, Reps and Wilhelm[154] for our purposes, because it satisfies

the strong nullification requirement outlined earlier. Our approach is less general, and consequently simpler, because we want to *enforce* a tree shape, not accommodate all shapes.

The key idea in shape analysis is this: imagine that at a given program point, each run-time object is labelled by the set of variable names referring to it (the variable alias set). Since the number of program variables ($|V|$) is finitely determined by the source code, there is a finite number of these labels ($2^{|V|}$), or equivalence classes into which all objects can be slotted, *at any given program point*. The emphasis is important because the set of variables referring to an object changes with variable assignment; in other words, an object may be represented by different alias sets at different points in the code. Figure 4.3 shows an example with objects and their ids in square boxes, labelled with the set of variables referring to them. Two snapshots of the heap are represented by a single shape graph, where node n_x represents objects 10 and 43 because they are referred to by x, and node n_{yz} represents 12 and 51.

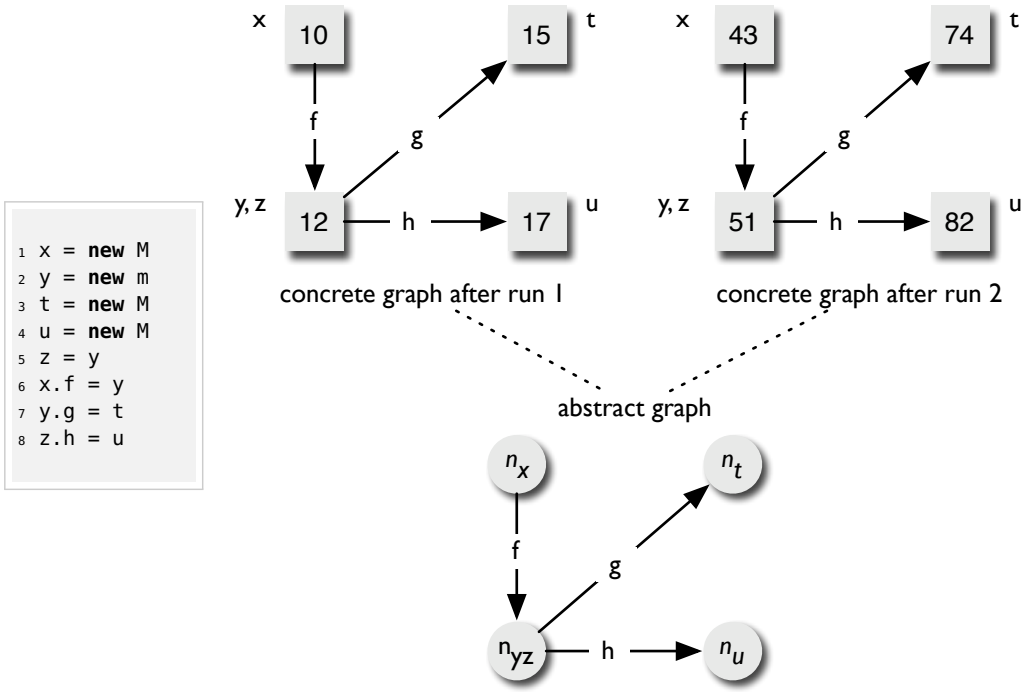


Figure 4.3: Concrete heap graphs after line 8, and their abstract version

However, in the presence of an *if/else* block, different runs of the program may take one or the other distinct path, resulting in different concrete graphs depending on the path taken. We make the same choice as Sagiv et al. of *merging* all abstract graphs at control flow merge points. For example, Fig. 4.4 shows the merged shape graphs after the *if/else* block. Note that in the final configuration (after line 11), none of the abstract nodes' labels contain d, as that variable is `null`. Note also the elimination of the $b.g$ edge.

4.3.3 States + Shape graph = Shape-state graph

In this section we assign a typestate to each node of a shape graph, forming a shape-state graph. We present an abstraction function β that translates a concrete runtime state δ to an abstract shape-state

```
1   a = new M; b = new M; c = new M
2   if ...
3     a.f = b
4     d = b
5   else
6     a.f = c
7     d = c
8
9   e = d.g
10  d = null
11  b.g = null
```

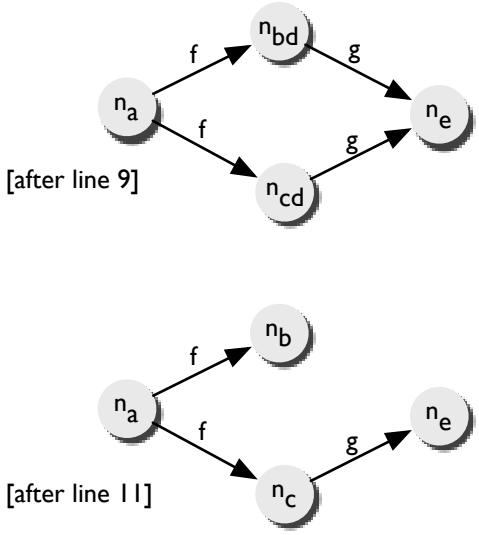


Figure 4.4: Shape graphs at two program points. Note that the shape graph after line 9 is a merger of the graphs from each branch of the if/else block.

graph Δ. We stress our focus on modular and local reasoning, which means the analysis is limited to the scope of a given function (intra-procedural analysis). We will also stress that the analysis is flow-sensitive, and the phrase *at a given program point* will be often expressed. This is similar to the flow-sensitive type qualifiers of Foster ,[64] except that they use constraint solvers for alias analysis.

Definition 4.2 (Concrete state)**.** The concrete state δ is an activation frame augmented with object states: $\delta = \langle \sigma, H, \varsigma \rangle$.

For example, the concrete state after run 1 in Fig. 4.3 is defined by:

$$\begin{aligned} \sigma &= \{x \mapsto 10, y \mapsto 12, z \mapsto 12, t \mapsto 15, u \mapsto 17\} \\ H &= \{\langle 10, f, 12\rangle, \langle 12, g, 15\rangle, \langle 12, h, 17\rangle\} \\ \varsigma &= \{10 \mapsto \mathbf{f}, 12 \mapsto \mathbf{w}, 15 \mapsto \mathbf{w}, 17 \mapsto \mathbf{w}\} \end{aligned}$$

Definition 4.3 (Shape-state graph)**.** Define the shape state graph Δ at a given program point as follows:

$\Delta \in SG$	$=$	$\overline{Node} \times \overline{Edge} \times State$	Set of abstract nodes and edges and node state map.
$n_T \in Node$	$=$		A graph node with the id n_T represents all run-time objects referred to by the variable alias set, or *label*, T at a given program point. The node $n_\emptyset$ denotes all objects in the heap that are not directly referred to by any variable.
$e \in Edge$	$=$	$Node \times Field \times Node$	An abstract edge $\langle n_S, f, n_T\rangle \in Edge$ exists if a run-time object represented by n_S is connected via field f to another object represented by n_T.
$\Sigma \in State$	$=$	$Node \mapsto \{\mathbf{r}, \mathbf{w}, \mathbf{f}, \mathtt{null}, \top\}$	The state of an abstract node.

For example, the shape-state graph of Fig. 4.3 is defined by $\Delta = \langle Ns, Es, \Sigma \rangle$, where

$$\begin{aligned} Ns &= \{n_x, n_{yz}, n_t, n_u\} \\ Es &= \{\langle n_x, f, n_{yz}\rangle, \langle n_{yz}, g, n_t\rangle, \langle n_{yz}, h, n_u\rangle\} \\ \Sigma &= \{n_x \mapsto \mathbf{f}, n_{yz} \mapsto \mathbf{w}, n_t \mapsto \mathbf{w}, n_u \mapsto \mathbf{w}\} \end{aligned}$$

Definition 4.4 (Label)**.** Define a labeling function $\mathscr{L}$ as the set of variable aliases referring to a run-time object ρ as follows: $\mathscr{L}(\rho, \delta) = \{ v \in Var \mid \sigma(v) = \rho, \text{ where } \delta = \langle \sigma, *, * \rangle \}$. For brevity and ease of presentation we will use the forms $\mathscr{L}(\rho)$ to implicitly denote $\mathscr{L}(\rho, \delta)$ and $\mathscr{L}'(\rho)$ for $\mathscr{L}(\rho, \delta')$.

Definition 4.5 (Abstraction Function β)**.** Define a family of functions β that transforms component-wise the concrete state $\delta = \langle \sigma, H, \varsigma \rangle$ to a shape state graph $\Delta = \langle \overline{n}, \overline{e}, \Sigma \rangle$. Please note that for ease of presentation, we use the symbols σ, ρ etc. in the implicit context of δ.

$\Delta = \beta(\langle \sigma, H, \varsigma \rangle)$	$\triangleq$	$\langle \beta(\sigma), \beta(H), \beta(\varsigma) \rangle$	Abstraction of concrete state.
$\overline{n} = \beta(\sigma)$	$\triangleq$	$\{ \beta(\rho) \mid \sigma(v) = \rho, v \in dom(\sigma) \}$	Set of abstract nodes.
$\overline{e} = \beta(H)$	$\triangleq$	$\{ \beta(\,(\rho, f, \rho_1)\,) \mid H(\rho)(f) = \rho_1 \}$	Set of abstract edges.
$n = \beta(\rho)$	$\triangleq$	$n_{\mathscr{L}(\rho)}$	Abstract node.
$e = \beta(\langle \rho, f, \rho_1 \rangle)$	$\triangleq$	$\langle \beta(\rho), f, \beta(\rho_1) \rangle$	Abstract edge if $\langle \rho, f, \rho_1 \rangle \in H$.
$\Sigma = \beta(\varsigma)$	$\triangleq$	$\lambda n_T . \bigsqcup_{T = \mathscr{L}(\rho)} \varsigma(\rho)$	An abstract node's state represents the least upper bound of the concrete states of its constituent objects.

The partial order for shape-state graphs $\Delta_1 = \langle Ns_1, Es_1, \Sigma_1 \rangle$ and $\Delta_2 = \langle Ns_2, Es_2, \Sigma_2 \rangle$ is defined as follows:

$$\begin{aligned} \Delta_1 \sqsubseteq \Delta_2 &\triangleq (Ns_1 \subseteq Ns_2) \wedge (Es_1 \subseteq Es_2) \wedge (\Sigma_1 \sqsubseteq \Sigma_2) \quad \text{where} \\ (\Sigma_1 \sqsubseteq \Sigma_2) &\triangleq \Sigma_1(n) \sqsubseteq \Sigma_2(n) \quad \text{for all } n \in Ns_1 \end{aligned}$$

We say Δ_2 is more conservative than Δ_1 if $\Delta_1 \sqsubseteq \Delta_2$.

The graph merge operation of Fig. 4.4 is defined as follows:

$$\Delta_1 \sqcup \Delta_2 \triangleq (Ns_1 \cup Ns_2,\ Es_1 \cup Es_2,\ \Sigma')$$

$$\text{where } \Sigma' = \lambda n . \begin{cases} \Sigma_1(n) & \text{if } n \notin dom(\Sigma_2) \\ \Sigma_2(n) & \text{if } n \notin dom(\Sigma_1) \\ \Sigma_1(n) \sqcup \Sigma_2(n) & \text{otherwise} \end{cases}$$

The abstract interpretation process introduces two levels of imprecision: (*i*) a node's state is the least upper bound of the states of the objects it represents (Def. 4.5), and (*ii*) the merging of shape graphs at CFG merge points as defined above. In spite of this imprecision, the shape-state graph contains a wealth of information, which we examine next.

4.3.4 Properties of Shape-State graphs

In this section we study the inverse of abstraction: *concretization*. Given a shape-state graph, we examine what can be definitely asserted about *all* the concrete states it represents. This idea is important because during static analysis, we have necessarily to work only with abstract graphs and draw conclusions from them.

Compatibility

Observe that in any given run of the program, and at a particular program point, if x and y are aliases (refer to the same object), and x and v are aliases, it necessarily means that all three are aliases; that is $\{x, y\}$ and $\{x, v\}$ are not distinct alias sets. It follows that the corresponding abstract graph cannot have two nodes with non-disjoint labels (n_{xy} and n_{xv}). Formally,

$$\beta(\rho) \neq \beta(\rho') \implies \rho \neq \rho'$$

However, since the shape analysis algorithm merges multiple shape graphs (Fig. 4.4), the resulting graph may have nodes whose labels are not disjoint. This is not a problem; it merely represents a disjunction of possibilities ("either x is aliased with y or x is aliased with u, but never at the same time"). This property will be used to tease apart some of these disjunctions using the notion of node compatibility.

Definition 4.6 (Compatibility)**.** Two nodes are defined as compatible only if they can possibly represent objects from a single concrete state

$$\mathit{compatible}(n_A, n_B) \triangleq A = \emptyset \vee B = \emptyset \vee (A \neq B \wedge A \cap B = \emptyset)$$

Clearly, there can never be an edge between two incompatible nodes, even with the merging process. This is called the *edge compatibility constraint*.[154] The rules described in the next section maintain this invariant.

Reachability

We define $n_A \rightsquigarrow_\Delta n_B$ in the same manner as the concrete reachability relation of Defn. 3.1 (p. 43), to mean a path between two nodes in a shape graph.

A concrete edge has a corresponding abstract edge in the shape state graph. That is,

$$\begin{array}{rcll} & H(\rho_1)(f) = \rho_2 & \implies & \langle \beta(\rho_1), f, \beta(\rho 2)\rangle \in \Delta \text{ (by construction)} \\ \implies & \rho_1 \rightsquigarrow_{\mathsf{H}} \rho_2 & \implies & \beta(\rho_1) \rightsquigarrow_\Delta \beta(\rho_2) \end{array}$$

Since merging of abstract graphs does not lose information (edge sets are simply merged), a path between concrete objects definitely has a counterpart between the corresponding abstract nodes (transitive argument).

Conversely:

$$\langle \beta(\rho_1), f, \beta(\rho 2)\rangle \notin \Delta \implies \neg(H(\rho_1)(f) = \rho_2))$$

This means that, if there is no path between two shape nodes, we can assert that there exists no path between two corresponding run-time objects. In contrast, if there *is* a path between two shape nodes, it may or may not correspond to an equivalent run-time path. At the very least, all the nodes on the path must be pairwise compatible for there to be an equivalent concrete path (if at all). We use this property to define a shape cluster.

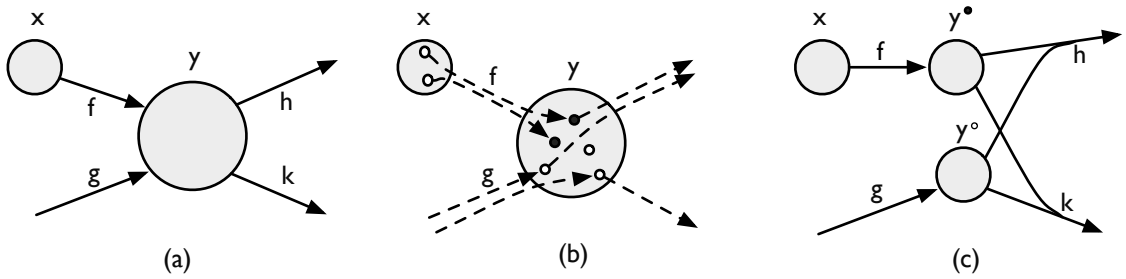


Figure 4.5: Materialization: Extracting the target objects of expression $x.f$

Definition 4.7 (Shape Cluster)**.** We define a shape cluster as all the compatible shape nodes reachable from a given node.

$$\mathit{scluster}(n : \mathit{Node}, \Delta) \triangleq \text{Smallest set } X \text{ such that } n \in X \ \wedge (n' \in X \implies n \rightsquigarrow_{\Delta} n' \wedge \mathit{compatible}(n, n'))$$

Note that unlike *cluster* (Fig.3.3), the shape cluster is not a reflexive, transitive closure of the reachability relation; it is a subset. What is important is that $\mathit{cluster}(\rho, H)$ is represented completely by $\mathit{scluster}(\beta(\rho), \Delta)$. The message verification rules described in §4.4.2 use this property. When an object is supplied as an argument to a `whole` parameter (in a method invocation), we approximate all possible reachable concrete objects by the corresponding reachable cluster in the shape graph. All such nodes are considered suspect, in that they may refer to exported object references after the call.

Materialization and sharing

Figure 4.5a shows an abstract graph, and subfigure 4.5b gives an idea of the concrete possibilities it represents (and what it doesn't), across many runs of the program. A few of the concrete nodes have incoming and outgoing edges, some have only one or the other, and others have no edges. The only factor common to parallel concrete edges (the $x.f$ edge, for example) is that their source and target endpoints have the same label. This leads us to two important observations.

First, the presence of multiple abstract edges incident on an abstract node (f and g in Fig. 4.5a) do not necessarily mean that a constituent concrete node is actually heap aliased; conventional shape analyses maintain a separate *is-shared* predicate to know whether there are two or more concrete incoming edges. In our case, the static semantics consult the state map Σ to prevent aliasing (see S-PUTFIELD in §4.4.2). This allows us to make the definite claim that multiple edges incident on an abstract node merely represent the overlaying of well-formed concrete trees.

Second, during the abstract interpretation process, if we encounter a statement of the form $y = x.f$, the figure should clarify that although the abstract edge $x.f$ is incident on the node y, it targets only a fraction of y's concrete nodes (shown as $y^{\bullet}$). Because we ensure that a concrete object can never be shared, the fraction of nodes $y^{\bullet}$ targeted by $x.f$ is disjoint from the rest, y°. Later, we will find it necessary to "scoop out" this fraction, or *materialize* as it is known in the shape analysis literature.[154]

Fig. 4.5c shows the effect after materializing the target of $x.f$. Note that there are no incoming g-labelled edges incident on $y^{\bullet}$. However, we can never know how the outgoing concrete edges are split up. For this reason, we have to be conservative and add abstract outgoing edges (h and k) from both $y^{\bullet}$ and y°.

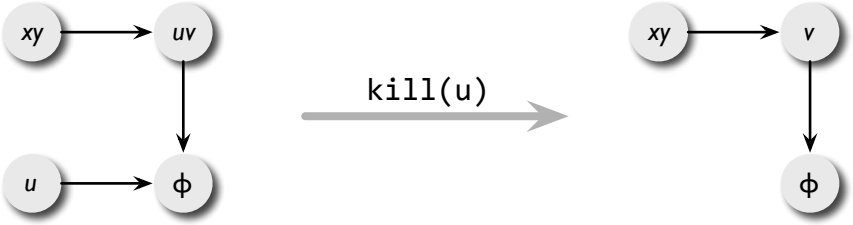


Figure 4.6: Kill

State of a variable

A node defines a set of variable aliases. Suppose the states of two nodes are given as follows: $\Sigma(n_{uv}) = \mathbf{f}$ and $\Sigma(n_{vz}) = \mathbf{r}$. This indicates that using the variable u will always yield a *free* object (unowned, writable), and using z will always be a *safe* object. However, the variable v will in some cases refer to *free* objects, and sometimes to *safe* objects. The only conservative assumption about the state of the object obtained via v is to take the least upper bound of the states. We overload the function Σ as follows:

$$\Sigma(v : Var) \triangleq \bigsqcup_{\{\, n \in \Delta \mid v \in vars(n) \,\}} \varsigma(n)$$

where $vars(n_T : Node) \triangleq T$

Kill

We can scavenge a dead variable from an activation frame without changing the behavior of the program. We define a kill pseudo-instruction for both the concrete and shape states. The concrete state merely removes the variable from its list of local variables; the corresponding change to the shape state graph may be larger because some nodes merge with $n_\emptyset$ (Fig.4.6)

Definition 4.8 (Kill)**.** Given $\delta = \langle \sigma, H, \varsigma \rangle$ and $\Delta = \langle Ns, Es, \Sigma \rangle$, define the overloaded functions *kill* as follows. We use the notation $n_{X-v} \triangleq n_{X \setminus \{v\}}$, to strip a variable off a node's label X.

$$\begin{aligned}
kill(\delta, v : Var) &\triangleq \langle \sigma', H, \varsigma \rangle \text{ where} \\
\sigma' &= \lambda v'. \begin{cases} undef & \text{if } v = v' \\ \sigma(v) & \text{otherwise} \end{cases} \\
\\
kill(\Delta, v : Var) &\triangleq \langle Ns', Es', \Sigma' \rangle \text{ where} \\
Ns' &= \{\, n_{X-v} \mid n_X \in Ns \,\} \\
Es' &= \{\, \langle n_{S-v}, f, n_{T-v} \rangle \mid \langle n_S, f, n_T \rangle \in Es \,\} \\
\Sigma' &= \lambda n_X.\, \Sigma(n_X) \sqcup \Sigma(n_{X \cup \{v\}})
\end{aligned}$$

We use *kill* in our typestate rules not just to scavenge, but also to invalidate certain variables after a method call. If a program survives such invalidation, we know that the variables are dead by fiat, and can thus be scavenged. In the limit case, one could invalidate all variables and be left only with $n_\emptyset$; the post-call abstract state would be accurate (no objects are labelled), but possibly completely useless as there are no valid variables left. The rules in the next section strive to keep the heap model accurate without losing precision unduly.

4.4 Phase-2: Message verification

We now have all the mechanisms required to discuss the message verification process. The semantics rules, enumerated next, are of the form $\Gamma;\Delta \vdash stmt \Rightarrow \Gamma;\Delta'$.

Each rule checks preconditions that conservatively cover the preconditions outlined in §4.3.1, and acts as a transfer function to produce the new shape state Δ'. The type environment from phase-1 remains constant in this phase. As in phase-1, we perform an abstract interpretation on the control flow graph of each function, using the lattice $\langle \Delta, \sqsubseteq \rangle$ and the merging operator defined in §4.3.3.

4.4.1 Normalization

For ease of presentation we simplify the rules by factoring out some aspects common to all instructions. This is done using a *normalization* pass on the code. We rewrite each statement that uses the same variable on both the left and right-hand sides to ones using fresh temporary variables. For example $x = x.f$ becomes $t = x.f; x = t$. Next, each non-null assignment instruction of the form $x = \ldots$ is preceded with $kill(x)$ to separate the "kill" step from the "gen" step. By the same token, all statements of the form $x.f = y$ are assumed to be preceded with $x.f = \texttt{null}$.

4.4.2 Semantic Rules

Notation. We define $\Delta = \langle Ns, Es, \Sigma \rangle$ and $\Delta' = \langle Ns', Es', \Sigma' \rangle$ in all the rules below, and use $*$ as a wildcard while selecting edges. We use the following auxiliary function in the rules.

$$nodes(v : Var) \triangleq \{ n \in Ns \mid v \in vars(n) \}$$

New. The S-NEW rule adds a fresh node n_x to the graph. No new edges are added.

$$\text{S-NEW}\ \frac{\begin{array}{lcl} Ns' & = & Ns \cup \{n_x\}, \quad n_x \text{ is fresh} \\ Es' & = & Es \\ \Sigma' & = & \Sigma[n_x \mapsto \mathbf{f}] \end{array}}{\Gamma;\Delta \vdash \boxed{x = \mathsf{new}\ M} \Rightarrow \Gamma;\Delta'}$$

Assign. This rule merely relabels the nodes; the auxiliary function a_x adds x to a node's label if it already has y. (F ranges over field names). The new graph is isomorphic to the old one, modulo x.

$$\text{S-ASSIGN}\ \frac{\begin{array}{l} \vdash \Sigma(y) < \top \\ \text{let } a_x \triangleq \lambda n_Z. \begin{cases} n_{Z \cup \{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases} \\ \begin{array}{lcl} Ns' & = & \{ a_x(n_Z) \mid n_Z \in Ns \} \\ Es' & = & \{ \langle a_x(n_S), F, a_x(n_T) \rangle \mid \langle n_S, F, n_T \rangle \in Es \} \\ \Sigma' & = & \lambda n_T.\ \Sigma(n_{T \setminus \{x\}}) \end{array} \end{array}}{\Gamma;\Delta \vdash \boxed{x = y} \Rightarrow \Gamma;\Delta'}$$

NullVar. Recall that x has already been killed by normalization (*kill*). We introduce a dummy node n_x so that the shape state records the fact that x has been initialized to `null`. n_x does not correspond

to any object in the heap. Note that the shape graph can be widened to introduce new nodes that do not correspond to any objects in the heap, without affecting the fact that every concrete object has a counterpart in the shape state graph.

$$\textsc{S-NullVar}\ \frac{\begin{array}{lcl} Ns' & = & Ns \cup \{n_x\} \\ Es' & = & Es \\ \Sigma' & = & \Sigma[n_x \mapsto \texttt{null}] \end{array}}{\Gamma;\Delta \vdash \boxed{x = \texttt{null}} \Rightarrow \Gamma;\Delta'}$$

NullField. As long as x is writable and non-null, we can remove all edges of the form $x.f$. This *strong nullification* property is one of the hallmarks of the shape analysis framework. The original target nodes of $x.f$ become *free* if (and only if) they are no longer reachable in the new configuration.

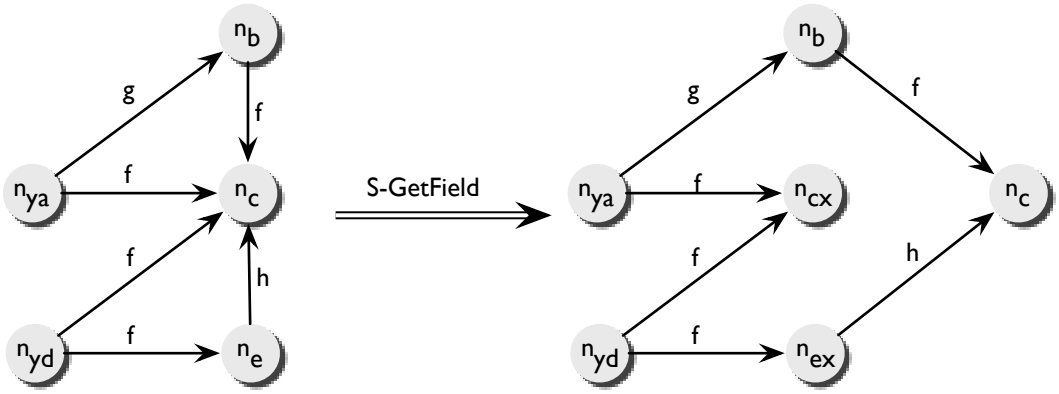
$$\textsc{S-NullField}\ \frac{\begin{array}{lcl} \vdash \mathbf{f} \leq \Sigma(x) \leq \mathbf{w} & & \\ Ns' & = & Ns \\ Es' & = & Es \setminus \{\langle n_X, f, * \rangle \in Es \mid x \in X)\} \\ \Sigma' & = & \lambda n_Z. \begin{cases} \mathbf{f} & \text{if } \Sigma(n_Z) = \mathbf{w} \wedge \langle n_X, f, n_Z \rangle \in Es \wedge \langle *, *, n_Z \rangle \notin Es' \text{ where } x \in X \\ \Sigma(n_Z) & \text{otherwise} \end{cases} \end{array}}{\Gamma;\Delta \vdash \boxed{x.f = \texttt{null}} \Rightarrow \Gamma;\Delta'}$$

PutField. The precondition is that all x-nodes (X) are *writable* and all y-nodes (Y) are *free*. The precondition that the nodes of x and y be disjoint ensures that a cycle is not created. For example, $x.f = y$ would create a cycle if x and y were aliases. The rule connects all X to all Y with an f-edge (*strong update*), then downgrades the y-nodes to *writable*. This operation has no effect if Y is empty (meaning $x.f$ is `null` along all CFG paths); recall that normalization has already removed all $x.f$ edges prior to the evaluation of this rule.

$$\textsc{S-PutField}\ \frac{\begin{array}{lcl} \vdash \Sigma(x) \leq \mathbf{w}, \Sigma(y) \leq \mathbf{f}, x \neq y & & \\ \vdash \mathit{nodes}(x) \cap \mathit{nodes}(y) = \emptyset & & \\ Ns' & = & Ns \\ Es' & = & Es \cup \{\langle n_X, f, n_Y \rangle \mid \Sigma(y) = \mathbf{f}\} \quad \text{where } x \in X, y \in Y \\ \Sigma' & = & \lambda n_Z. \begin{cases} \mathbf{w} & \text{if } \langle n_X, f, n_Z \rangle \in Es' \quad \text{where } x \in X \\ \Sigma(n_Z) & \text{otherwise} \end{cases} \end{array}}{\Gamma;\Delta \vdash \boxed{x.f = y} \Rightarrow \Gamma;\Delta'}$$

GetField. Informally, for a statement of the form $x = y.f$, we identify all the $y.f$ edges in the graph and add x to the label of the target nodes of these edges. But due to materialization (Fig. 4.5), we can be more precise, albeit with a concomitant increase in the complexity of the rule.

In the example figure below, the S-GETFIELD rule transforms the shape state graph on the left to the one on the right.



Denote the source and target nodes of these edges by Ys and Ts respectively. In the example, $Ys = \{n_{ya}, n_{yd}\}$, and $Ts = \{n_c, n_e\}$. We identify a subset of the target nodes YFs as those that only have incoming $y.f$ edges and no other; in the example, $YFs = \{n_e\}$. These can be simply relabeled, as shown by the n_{ex} node on the right, because the runtime objects represented by these shape state nodes are all exclusively targets of $y.f$ nodes. Each node in the rest of the target nodes (that is, in $Ts \setminus YFs = \{n_c\}$) represents (i) runtime objects that are targets of $y.f$, and (ii) unrelated runtime objects that are targets of other unrelated edges. These two sets represent disjoint sets of runtime objects, given the invariant that no message object may be pointed to by more than one object at run time. After the transformation, the first part is materialized out as a separate node n_{cx}, while the latter continues to be n_c.

$$
\begin{array}{l}
\vdash \mathbf{f} \le \Sigma(y) \le \mathbf{r} \\
\text{let } Ys = \{n_Y \mid \langle n_Y, f, * \rangle \in Es, n_Y \in Y\} \\
\text{let } Ts = \{n_T \mid \langle n_Y, f, n_T \rangle \in Es, n_Y \in Y\} \\
\text{let } YFs = \{n_T \mid \nexists n_Z.\ (\langle n_Z, F, n_T \rangle \in Es \wedge n_T \in Ts \wedge (y \notin Z \vee F \neq f))\} \\
Ns' = Ns \setminus YFs \\
\qquad \cup \{n_{T \cup \{x\}} \mid n_T \in Ts\} \\
\\
Es' = Es \\
\qquad \setminus \{\langle n_Y, f, n_T \rangle \mid n_Y \in Ys, n_T \in Ts\} \ \boxed{1} \\
\qquad \setminus \{\langle n_T, *, * \rangle \mid n_T \in YFs\} \ \boxed{2} \\
\qquad \cup \{\langle n_Y, f, n_{T \cup \{x\}} \rangle \mid n_Y \in Ys, n_T \in Ts\} \ \boxed{3} \\
\qquad \cup \{\langle n_{T \cup \{x\}}, F, n_U \rangle \mid \langle n_T, F, n_U \rangle \in Es\} \ \boxed{4} \\
\\
\Sigma' = \lambda n_Z. \begin{cases} w \sqcup \bigsqcup_{n_y} \Sigma(n_Y) & \text{if } \langle n_Y, f, n_Z \rangle \in Es' \ \boxed{5} \\ \texttt{null} & \text{if } Z = \{x\} \wedge Ys = \emptyset \ \boxed{6} \\ \Sigma(n_{Z \setminus \{x\}}) & \text{if } n_{Z \setminus \{x\}} \notin YFs \ \boxed{7} \end{cases} \\
\qquad \text{for all } n_Y, n_T \in Ns, y \in Y \\
\hline
\Gamma; \Delta \vdash \boxed{x = y.f} \Rightarrow \Gamma; \Delta' = (Ns', Es', \Sigma')
\end{array}
\quad \text{S-GETFIELD}
$$

For edges, we remove all $y.f$ edges incoming to Ts (1) and retarget them to the updated targets (3). We remove all edges outgoing from YFs (2) because these will not have a counterpart in the new shape

state. Finally, we add edges from the updated Ts (4). Note that n_{ya} and n_{yd} point to the new node n_{cx}, while the n_b to n_c edge is preserved.

As for the typestate, nodes in the updated Ts get their state from their parent Ys in the new configuration (5), or `null` if there were no $y.f$ edges at all (6). All others remain the same (7).

Method Call. The method call ensures that non-`safe` arguments are disjoint from every other argument, then invalidates (kills) all variables that could potentially be pointing to an object that may have been exported by the method call. All linear type systems have variants of this style of consumption.

$$
\begin{array}{l}
\text{let } sig(m) = \overline{q\tau} \to \tau_r \\
\text{let } vcluster(v : Var) = \bigcup_{n \in nodes(v)} scluster(n) \\
\text{let } disjoint(v, v') = vcluster(v) \cap nodes(v') = \emptyset \text{ where } v, v' \in Var \\
\text{let } N_k = \bigcup_i \begin{cases} vcluster(y_i) & \text{if } q_i = \texttt{whole} \\ vcluster(y_i) \setminus nodes(y_i) & \text{if } q_i = \texttt{part} \end{cases} \\
\text{let } V_k = \bigcup_{n \in N_k} vars(n)
\end{array}
$$

$$
\text{S-CALL} \quad \frac{
\begin{array}{l}
\vdash \Sigma(y_i) \le q_i \\
\vdash (q_i = q_j = \texttt{safe}) \vee disjoint(y_i, y_j), \text{ for all } i \neq j \\
\Delta' = fold(kill, V_k, \Delta) \\
Ns'' = Ns' \cup \{ n_{\{v\}} \mid v \in V_k \} \\
\qquad \cup \{ n_x \}, \text{ provided } \tau_r \text{ is a message type} \\
Es'' = Es' \cup \{ \langle n_{\{x\}}, f, n_\emptyset \rangle \mid f \in fields(\tau_r) \} \text{ if } \tau_r \text{ is a message type} \\
\Sigma'' = \lambda n_T. \begin{cases} \top & \text{if } T \cap V_k \neq \emptyset \\ \mathbf{f} & \text{if } T = \{ x \} \text{ and } \tau_r \text{ is a message type} \\ \Sigma'(n_T) & \text{otherwise} \end{cases}
\end{array}
}{
\Gamma; \Delta \vdash \boxed{x = m(\overline{y})} \Rightarrow \Gamma; \Delta''
}
$$

The function *vcluster* is a union of all clusters accessible from each of $nodes(v)$ and when adjusted for the qualifier (N_k), corresponds to concrete objects that are reachable from v that are permitted to be exported or grafted onto other objects. We have no way of knowing how the callee will change these objects. The variables in the set V_k belong to the labels of nodes in N_k, and which *may* be left with dangling references after the call. For this reason, we conservatively invalidate them all using the functional *fold* function to kill each variable successively. Consider the example below.

```
1  int h(safe  a) {
2    if ...
3      y = a                 Σ(n_ay) = r
4    else
5      x = new M             Σ(n_x) = f
6      y = new M
7      x.f = y               n_x ⇝ n_y
8  }
9                            //Before call:  nodes(y) = {n_ay, n_y}, scluster(n_x) = {n_x, n_y}
10 send(x)                   //Consume vcluster(x) = scluster(n_x)
11                           //After call; Σ(n_x) = Σ(n_y) = ⊤ and Σ(n_a) = r
12 print(a)                  // ok.
```

Disjointness of a non-safe argument from all other arguments is verified by ensuring that its *vcluster* does not contain the nodes of the other arguments. The callee can thus be assured that if any of its input arguments is writable, it is independent of any other argument.

Using *kill* serves another important purpose, besides invalidation. It eliminates uncertainty about the post-call state by retaining only those variables and nodes and edges that are guaranteed to remain untouched; these include `safe` arguments and those inaccessible to the called method. This way, we ensure that both the concrete and abstract models have been brought into sync. Note that we introduce dummy nodes corresponding to all the invalidated variables to record the fact that they were explicitly invalidated ($\top$ indicates the status is unknown). This is a technical device which ensures that, after a graph merge (at CFG merge points), we can tell the difference between a variable that was invalidated on one path and not on the other, as shown below.

```
1  if ... {
2    a = y
3    send(y)  // Σ(n_y) = Σ(n_a) = ⊤
4  } else {
5    b = y = new M // Σ(n_by) = f
6  }
7  // Merge point
8  // Graph contains Σ(n_y) = Σ(n_a) = ⊤, Σ(n_by) = f
```

In this example, it is safe to use b (at line 6) because it only refers to n_{by}, whose typestate is valid. But it is not safe to use y because its state is $\Sigma(n_y) \sqcup \Sigma(n_{by}) = \top$.

Finally, we create edges between the return variable x and the summary node $n_\emptyset$ to record the fact that we have a *free* object whose subtree is unlabelled.

Method Entry If the current method is called m, we introduce a shape node for each message argument p_i, a dummy variable and node $\hat{p}_i$ to represent the parent node, if the argument is `part` (to keep track of ownership), and edges from these dummy nodes and to the summary node $n_\emptyset$, to represent potential concrete edges.

$$\text{S-ENTRY}\ \frac{\begin{array}{l}\text{let } sig(m) = \overline{q\ \tau} \to *\\ \text{let } \overline{p} = args(m)\\ \text{size Note: } i \text{ ranges from 1 to } |args(i)|\\ Ns = \bigcup_i n_{\{p_i\}}, \text{ where } q_i \neq \texttt{none}\\ \quad \cup \bigcup_i n_{\{\hat{p}_i\}}, \text{ where } q_i = \texttt{part} \text{ and } \hat{p}_i \text{ is fresh}\\ Es = \bigcup_i \langle n_{\{p_i\}}, f, n_\emptyset\rangle \text{ for all } f \in fields(\tau_i)\\ \quad \cup \bigcup_i \langle n_{\{\hat{p}_i\}}, \hat{f}, n_{p_i}\rangle \text{ where } q_i = \texttt{part}, \text{ and } \hat{f} \text{ is a fresh field}\\ \Sigma = \left[n_{\{p_i\}} \mapsto \begin{cases} \mathbf{r} & \text{if } q_i = \texttt{safe}\\ \mathbf{w} & \text{if } q_i = \texttt{part}\\ \mathbf{f} & \text{if } q_i = \texttt{whole}\end{cases}\right]\end{array}}{\Delta = (Ns, Es, \Sigma)}$$

Put, Get The rules presented so far permit us to treat `put` and `get` as ordinary function calls that transfer `whole` messages:

$sig(\texttt{put}) = (M\ \texttt{mbx},\ \texttt{whole}\ M) \rightarrow *$

$sig(\texttt{get}) = (M\ \texttt{mbx}) \rightarrow \langle \texttt{whole}\ M \rangle$

In practice, this permits us to implement communication primitives as library methods and not give them special treatment in the type checker. This way, we can implement other communication primitives such as futures and Haskell style MVars. We have also investigated using this type system for unique ownership and linear transfer of ownership of security capabilities.[59]

Note that `spawn` and `select` do not have message type parameters, and so do not need to be dealt with in this phase.

4.5 Soundness of message verification

This section shows that the message verification rules (§4.4) conservatively simulate the operational semantics of §3.2, or more precisely, the operational semantics augmented with a run-time type system (§4.3.1). Any program that passes the static test will never encounter dangling references and will always maintain the tree structure invariant of messages. We will use the notation $[\![\ldots]\!]_{\delta}$ and $[\![\ldots]\!]_{\Delta}$ to represent the operational and static rules respectively.

Proposition 4.2 (Monotonocity). *For all semantic rules of the form* $[\![\ldots]\!]_{\Delta}$ *and for all pairs of shape states* Δ_1 *and* Δ_2 *such that* $\Delta_1 \sqsubseteq \Delta_2$*, we have* $[\![\cdot]\!]_{\Delta}(\Delta_1) \sqsubseteq [\![\ldots]\!]_{\Delta}(\Delta_2)$

The proof structure is aligned closely to the excellent paper by Sagiv et al.[154] Our proof is necessarily simpler because objects are not heap-aliasable; on the other hand, we add function invocation and consumption policies and treat `null` specially (which permits us to detect null-dereferences in some cases).

Informally, the proof structure is as follows. Any run-time heap graph has a corresponding shape state graph. We prove that for a verified program, the transformation of the run-time graph due to any instruction (given the rules in Fig. 3.4 and 3.5, p. 44) is paralleled in a conservative fashion by the transformation of the corresponding shape state graph (the rules in §4.4.2).

Figure 4.7 shows a concrete pre-state $\delta = \langle \sigma, H, \varsigma \rangle$ transformed to a concrete post-state $\delta' = \langle \sigma', H', \varsigma' \rangle$. Likewise, the static verification rules transform the abstraction of the concrete-pre-state, the shape state graph Δ, to Δ'. To assert that Δ' soundly models δ', we compare it to the shape-state graph of the post-state, $\hat{\Delta}$ $(= \beta(\delta'))$, and verify that for each rule, $\hat{\Delta} \subseteq \Delta'$.

Theorem 4.1 (Local Safety Theorem). *For all statements stmt and for every concrete evaluation*

$[\![stmt]\!]_{\delta}(\langle \sigma, H, \varsigma \rangle) \rightsquigarrow \langle \sigma', H', \varsigma' \rangle$

$\beta(\langle \sigma', H', \varsigma' \rangle) \sqsubseteq [\![stmt]\!]_{\Delta}(\beta(\langle \sigma, H, \varsigma \rangle))$

PROOF. To prove $\hat{\Delta} \sqsubseteq \Delta'$, we need to show that each component is a subset of the other's.

$$n \in \hat{Ns} \implies n \in Ns'$$
$$e \in \hat{Es} \implies e \in Es'$$
$$\hat{\Sigma}(n) \sqsubseteq \Sigma'(n), \text{ for all } n \in \hat{Ns}$$

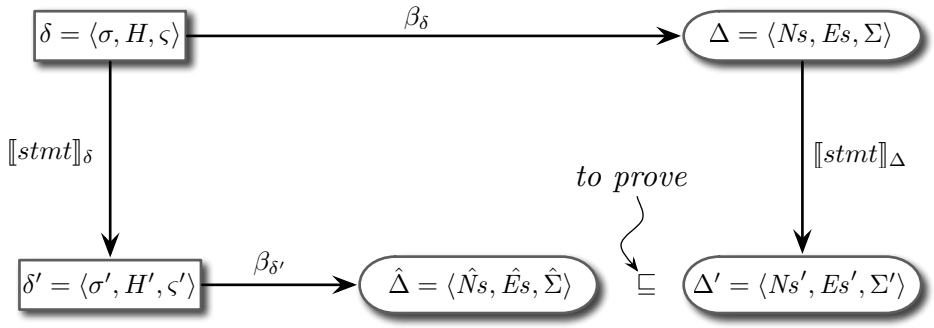


Figure 4.7: Parallel execution of concrete and abstract semantics

We will use phrases such as "going from $\delta \to \delta' \to \hat{\Delta}$" to indicate the direction of the proof, with reference to Fig. 4.7.

Case 1 $\beta([\![kill]\!]_\delta(\delta)) \sqsubseteq [\![kill]\!]_\Delta(\beta(\delta))$, where $\delta = \langle \sigma, H, \varsigma \rangle$

Let the variable to be killed be x. Let $\sigma(x) = \rho$. The object ρ is the only one to have its label modified in the concrete semantics;the heap remains the same. Define $X = \mathscr{L}(\rho)$ and $X' = X \setminus \{x\}$

I. (Proving $\hat{Ns} \subseteq Ns'$).
Going from $\delta \to \delta' \to \hat{\Delta}$
$\sigma(x) = \rho$
$\implies \mathscr{L}'(\rho) = X \setminus \{x\}$, from Def. 4.8
$\implies \hat{Ns} = \{n_{X'}\} \cup \bigcup_{\rho' \neq \rho} n_{\mathscr{L}(\rho')}$, from $\delta' \to \hat{\Delta}$
Similarly, going from $\delta \to \Delta \to \Delta'$
$Ns = \{n_X\} \cup \bigcup_{\rho \neq \rho} n_{\mathscr{L}(\rho)}$, from $\delta \to \Delta$
$\implies Ns' = \{n'_X\} \cup \bigcup_{\rho \neq \rho} n_{\mathscr{L}(\rho)}$. x removed, but $\mathscr{L}(\rho)$ unchanged.
Hence, $\hat{Ns} \subseteq Ns'$

II. (Proving $\hat{Es} \subseteq Es'$). Similar to above; the end point labels have changed, but the edges are unchanged

III. (Proving $\hat{\Sigma} \subseteq \Sigma'$).
Let $\varsigma_x = \varsigma(\rho) = \varsigma'(\rho)$, since the typestate is unchanged. There are three subcases:

(a) For all objects $\rho' \neq \rho$, the label is unchanged; if $Z = \mathscr{L}(\rho') = \mathscr{L}'(\rho')$, it is easy to see that $\hat{\Sigma}(n_Z) = \Sigma(n_Z) = \Sigma'(n_X)$

(b) Label changed for ρ, but $X' \neq \emptyset$. Since a variable can only point to a single object, and since X' is non-empty, $n_{X'}$ must represent only ρ.
$\hat{\Sigma}(n_{X'}) = \varsigma'(\rho) = \varsigma(\rho) = \Sigma(n_X) = \Sigma(n_{X'})$

(c) Label changed for ρ, and $X' = \emptyset$. Let $\varsigma_\emptyset$ be the least upper bound of all typestates of all objects without a label.
Since $X' = \emptyset$, ρ is represented by $n_\emptyset$, and hence influences $n_\emptyset$ typestate.
Going from $\delta \to \delta' \to \hat{\Sigma}$,

$\hat{\Sigma}(n_\emptyset) = \varsigma'(\rho) \sqcup \varsigma'_\emptyset$ from $\delta' \to \hat{\Delta}$
$\hat{\Sigma}(n_\emptyset) = \varsigma(\rho) \sqcup \varsigma_\emptyset$ because neither typestate has changed
Going from $\delta \to \Delta \to \Delta'$,
$\Sigma(n_X) = \varsigma(\rho)$, and $\Sigma(n_\emptyset) = \varsigma_\emptyset$
$\implies \Sigma'(n_\emptyset) = \Sigma(n_{\emptyset \cup \{x\}}) \sqcup \Sigma(n_\emptyset)$ from $[\![kill]\!]_\Delta$
$\implies \Sigma'(n_\emptyset) = \Sigma(n_X) \sqcup \Sigma(n_\emptyset)$
$\implies \Sigma'(n_\emptyset) = \varsigma(\rho) \sqcup \varsigma_\emptyset$

Hence, in all three cases $\hat{\Sigma}(n_\emptyset) \leq \Sigma'(n_\emptyset)$

Case 2 $\beta([\![x.f = \texttt{null}]\!]_\delta(\delta)) \sqsubseteq [\![x.f = \texttt{null}]\!]_\Delta(\beta(\delta))$, where $\delta = \langle \sigma, H, \varsigma \rangle$

I. (Proving $\hat{Ns} \subseteq Ns'$). Follows directly from the fact that σ is unmodified in $[\![x.f = \texttt{null}]\!]_\delta$

II. (Proving $\hat{Es} \subseteq Es'$).
Going from $\delta' \to \delta \to \Delta \to \Delta'$
Let edge $\langle \rho_1, g, \rho_2 \rangle \in H'$
$\implies$ $\langle \rho_1, g, \rho_2 \rangle \in H \wedge ((\sigma(x) \neq \rho_1) \vee (f \neq g))$ from $[\![x.f = \texttt{null}]\!]_\delta$
$\implies$ $\langle n_{\mathscr{L}(\rho_1)}, g, n_{\mathscr{L}(\rho_2)} \rangle \in Es \wedge ((x \notin \mathscr{L}(\rho_1)) \vee (f \neq g))$
$\implies$ $\langle n_{\mathscr{L}(\rho_1)}, g, n_{\mathscr{L}(\rho_2)} \rangle \in Es'$ from S-PUTNULL
$\implies$ $\langle n_{\mathscr{L}'(\rho_1)}, g, n_{\mathscr{L}'(\rho_2)} \rangle \in Es'$ because $\sigma' = \sigma$.
Going from $\delta' \to \hat{\Delta}$
$\langle \rho_1, g, \rho_2 \rangle \in H' \implies \langle n_{\mathscr{L}'(\rho_1)}, g, n_{\mathscr{L}'(\rho_2)} \rangle \in \hat{Es}$
Hence, $e \in \hat{\Delta} \implies e \in \Delta'$

III. (Proving $\hat{\Sigma} \subseteq \Sigma'$). Let $\rho_1 = \sigma(x)$ and $\langle \rho_1, f, \rho_2 \rangle \in H$. We will examine ρ_2 separate from all the others (because only its typestate is modified). This is the only rule where a node's typestate is upgraded to *free*.
$\varsigma(\rho_2) = \mathbf{w}$, (because the preconditions ensure that its parent ρ_1 is not marked `safe`).
$\varsigma'(\rho_2) = \mathbf{f}$, (from $[\![x.f = \texttt{null}]\!]_\delta$, ρ_2 is no longer owned, and was never marked `safe`)
let $\Sigma(n_Z) = \varsigma(\rho_2) \sqcup \varsigma_{rest}$, where n_Z node represents ρ_2, and other objects whose collective typestate (least upper bound) is denoted by ς_{rest} (which remains unchanged).

(a) If ς_{rest} is $\mathbf{r}$, at least one of the other objects is marked safe:
$\implies \Sigma'(n_Z) = \Sigma(n_Z) = \mathbf{r}$ from $[\![x.f = \texttt{null}]\!]_\Delta$ (upgrade to $\mathbf{f}$ not possible)
Going from $\delta \to \delta' \to \hat{\Delta}$,
$\hat{\Sigma}(n_Z) = \varsigma'(\rho_2) \sqcup \varsigma_{rest}$
$\hat{\Sigma}(n_Z) = \mathbf{r}$

(b) If ς_{rest} is $\mathbf{w}$, there must be at least one non $x.f$ edge in H, which means there must be an equivalent edge in Es:
$\implies \Sigma'(n_Z) = \Sigma(n_Z) = \mathbf{w}$ from $[\![x.f = \texttt{null}]\!]_\Delta$. (upgrade to $\mathbf{f}$ not possible)
Going from $\delta \to \delta' \to \hat{\Delta}$,
$\hat{\Sigma}(n_Z) = \varsigma'(\rho_2) \sqcup \varsigma_{rest}$
$\hat{\Sigma}(n_Z) = \mathbf{w}$

(c) If ς_{rest} is $\mathbf{f}$, there are no incoming edges (ρ_2 was holding the others back, so to speak). If n_Z has incoming edges (being conservative), we cannot upgrade to $\mathbf{f}$. If not, it satisifies all the conditions of $[\![x.f = \texttt{null}]\!]_\Delta$:
$\Sigma'(n_Z) = \mathbf{f}$ from $[\![x.f = \texttt{null}]\!]_\Delta$
Going from $\delta \to \delta' \to \hat{\Delta}$,
$\hat{\Sigma}(n_Z) = \varsigma'(\rho_2) \sqcup \varsigma_{rest}$
$\hat{\Sigma}(n_Z) = \mathbf{f}$

(d) For all other objects ρ, there is no change in $\varsigma(\rho)$.
If ρ belongs to n_Z as defined above (it belongs to *rest*), the sub-cases above show that $\hat{\Sigma}(n_Z) \leq \Sigma'(n_Z)$
If not, $\varsigma(\rho) \leq (\hat{\Sigma}(n_{\mathscr{L}(\rho)}) = \Sigma(n_{\mathscr{L}(\rho)}) = \Sigma'(n_{\mathscr{L}'(\rho)}))$
In all cases we have shown that $\hat{\Sigma}(n) \leq \Sigma'(n)$

Case 3 $\boxed{\beta([\![x.f = y]\!]_\delta(\delta)) \sqsubseteq [\![x.f = y]\!]_\Delta(\beta(\delta)), \text{ where } \delta = \langle \sigma, H, \varsigma \rangle}$

I. (Proving $\hat{Ns} \subseteq Ns'$). Follows directly from $\sigma' = \sigma$, in $[\![x.f = y]\!]_\delta$

II. (Proving $\hat{Es} \subseteq Es'$). The only concrete change is the introduction of a new $x.f$ edge

(a) Concrete edge was newly introduced
Going from $\delta' \to \delta \to \Delta \to \Delta'$
Let $\langle \rho_1, f, \rho_2 \rangle \in H'$ where $\sigma'(x) = \rho_1 \wedge \sigma'(y) = \rho_2$
$\implies$ $\sigma(x) = \rho_1 \wedge \sigma(y) = \rho_2 \wedge \varsigma(\rho_2) = \mathbf{f}$ from $[\![x.f = y]\!]_\delta$
$\implies$ $n_{\mathscr{L}(\rho_1)}, n_{\mathscr{L}(\rho_2)} \in Ns$ $\delta \to \Delta$
$\implies$ $\langle n_{\mathscr{L}'(\rho_1)}, f, n_{\mathscr{L}'(\rho_2)} \rangle \in Es'$ from $[\![x.f = y]\!]_\Delta$
Going from $\delta' \to \hat{\Delta}$
$\langle \rho_1, f, \rho_2 \rangle \in H' \implies \langle n_{\mathscr{L}'(\rho_1)}, f, n_{\mathscr{L}'(\rho_2)} \rangle \in \hat{Es}$

(b) All other edges. By inspection, all unrelated edges are preserved and no new ones are created.
Therefore, $e \in \hat{Es} \implies e \in Es'$

III. (Proving $\hat{\Sigma} \subseteq \Sigma'$).

(a) Newly introduced edge $\langle \rho_1, f, \rho_2 \rangle \in H'$ where $\sigma'(x) = \rho_1 \wedge \sigma'(y) = \rho_2$
Going from $\delta' \to \delta \to \Delta \to \Delta'$
$\varsigma'(\rho_2) = \mathbf{w}$, owned by ρ_1 by typestate definition
$\implies \varsigma(\rho_2) = \mathbf{f}$, precondition of $[\![x.f = y]\!]_\delta$
$\implies \Sigma(n_{\mathscr{L}(\rho_2)}) = \mathbf{f}$ $\delta \to \Delta$
$\implies \Sigma'(n_{\mathscr{L}'(\rho_2)}) = \mathbf{w}$ from $[\![x.f = y]\!]_\Delta$
Going from $\delta' \to \hat{\Delta}$
$\varsigma'(\rho_2) = \mathbf{w} \implies \Sigma'(\langle \mathscr{L}'(\rho_2) \rangle)$

(b) All other edges. By inspection, unrelated edges are preserved and no new ones are created, and no other objects' typestates are updated.
Hence, $\hat{\Sigma}(n) \subseteq \Sigma'(n)$ for all $n \in Ns'$ (Note: $Ns' = Ns$)

Case 4 $\boxed{\beta([\![x = y.f]\!]_\delta(\delta)) \sqsubseteq [\![x = y.f]\!]_\Delta(\beta(\delta)), \text{ where } \delta = \langle \sigma, H, \varsigma \rangle}$

I. (Proving $\hat{Ns} \subseteq Ns'$).

Assume that the $y.f$ edge refers to the concrete edge $\langle \rho_1, f, \rho_2 \rangle \in H$. We'll refer to ρ_2 as the target object. The only change in the concrete state is the update to ρ_2's label. $\mathcal{L}'(\rho_2) = \mathcal{L}(\rho_2) \cup \{x\}$

(a) Consider object ρ_2.
Going from $\delta \to \Delta \to \Delta'$
$x \notin \mathcal{L}(\rho_2)$ due to normalization
$\implies n_{\mathcal{L}(\rho_2)} \in Ns$ $\delta \to \Delta$
$\implies n_{\mathcal{L}(\rho_2) \cup \{x\}} \in Ns'$ from [3] in $[\![x = y.f]\!]_\Delta$
$\implies n_{\mathcal{L}'(\rho_2)} \in Ns'$
Going from $\delta \to \delta' \to \hat{\Delta}$
$n_{\mathcal{L}'(\rho_2)} \in \hat{Ns}$

(b) All objects other than ρ_2. Since their labels don't change, $n_{\mathcal{L}(\rho)}$ is present in Ns, Ns' and $\hat{Ns}$. Therefore, for all objects, $n_{\mathcal{L}'(\rho)} \in \hat{Ns} \implies n_{\mathcal{L}'(\rho)} \in Ns'$

II. (Proving $\hat{Es} \subseteq Es'$).

As before, let $y.f$ edge refer to the concrete edge $\langle \rho_1, f, \rho_2 \rangle \in H$. In the concrete semantics of $[\![x = y.f]\!]_\delta$, there are no changes to edges ($H' = H$).

We will consider three subcases from ρ_2's point of view; incoming concrete edges (there can be only one, due to well-formedness assumptions), outgoing edges, and all other edges that have nothing to do with ρ_2.

(a) Incoming edge; this must be the $y.f$ edge.
Going from $\delta \to \delta' \to \hat{\Delta}$
$\langle \rho_1, f, \rho_2 \rangle \in H \implies \langle \rho_1, f, \rho_2 \rangle \in H'$
$\implies \langle n_{\mathcal{L}'(\rho_1)}, f, n_{\mathcal{L}'(\rho_2)} \rangle \in \hat{Es}$
Going from $\delta \to \Delta \to \Delta'$
$\langle \rho_1, f, \rho_2 \rangle \in H \implies \langle n_{\mathcal{L}(\rho_1)}, f, n_{\mathcal{L}(\rho_2)} \rangle \in Es$, $\delta \to \Delta$, where $y \in \mathcal{L}(\rho_1)$
$\implies \langle n_{\mathcal{L}(\rho_1)}, f, n_{\mathcal{L}(\rho_2) \cup \{x\}} \rangle \in Es'$ from $[\![x = y.f]\!]_\Delta$
Hence $e = \langle n_{\mathcal{L}'(\rho_1)}, f, n'_{\mathcal{L}}(\rho_2) \rangle \in \hat{Es}' \implies e \in Es'$

(b) Edges outgoing from ρ_2; consider $\langle \rho_2, g, \rho_3 \rangle$ for some field g and some other message object ρ_3.
Going from $\delta \to \delta' \to \hat{\Delta}$, as above
$\langle n_{\mathcal{L}'(\rho_1), f, \mathcal{L}'(\rho_2)} \rangle \in \hat{Es}$
Going from $\delta \to \Delta \to \Delta'$
$\langle n_{\mathcal{L}(\rho_2)}, g, n_{\mathcal{L}(\rho_3)} \rangle \in Es$ $\delta \to \Delta$
$\implies \langle n_{\mathcal{L}(\rho_2) \cup \{x\}}, g, n_{\mathcal{L}(\rho_3)} \rangle \in Es'$ from $[\![x = y.f]\!]_\Delta$
$\implies \langle n_{\mathcal{L}'(\rho_2)}, g, n_{\mathcal{L}'(\rho_3)} \rangle \in Es'$
Hence, $e = \langle n_{\mathcal{L}'(\rho_2)}, g, n_{\mathcal{L}'(\rho_3)} \rangle \in \hat{Es} \implies e \in Es'$

(c) All other concrete edges that have nothing to do with ρ_2. Consider $\langle \rho_3, g, \rho_4 \rangle \in H$, where $\rho_2 \notin \{\rho_3, \rho_4\}$. This includes the case of non f-labelled edges out of y ($\rho_3 = \rho_1 \wedge f \neq g$).

As above, $\langle n_{\mathscr{L}'(\rho_3)}, g, n_{\mathscr{L}'(\rho_4)}\rangle \in \hat{Es}$
Going from $\delta \to \Delta \to \Delta'$
$\implies \langle n_{\mathscr{L}(\rho_3)}, g, n_{\mathscr{L}(\rho_4)}\rangle \in Es$
$\implies \langle n_{\mathscr{L}(\rho_3)}, g, n_{\mathscr{L}(\rho_4)}\rangle \in Es'$ no change to edge in $[\![x = y.f]\!]_\Delta$
$\implies \langle n_{\mathscr{L}'(\rho_3)}, g, n_{\mathscr{L}'(\rho_4)}\rangle \in Es'$ no change in end-point labels for $\rho \neq \rho_2$
Hence, $e = \langle n_{\mathscr{L}'(\rho_3)}, g, n_{\mathscr{L}'(\rho_4)}\rangle \in \hat{Es}' \implies e \in Es'$

III. (Proving $\hat{\Sigma} \subseteq \Sigma'$). Let $\rho_1 = \sigma(y)$ and $\langle \rho_1, f, \rho_2\rangle \in H$. No object typestates are modified in a getfield operation. That is, $\varsigma(\rho_2) = \varsigma'(\rho_2)$.

(a) Consider ρ_2. There are three sub-subcases depending on $\varsigma(\rho_2)$

i. $\varsigma(\rho_2) = \mathbf{r}$, it follows that $\varsigma(\rho_1)$, due to the deep properties of `safe`.
Going from $\delta \to \Delta \to \Delta'$
$\implies \Sigma(n_{\mathscr{L}(\rho_2)}) = \mathbf{r}$ $\delta \to \Delta$
$\implies \Sigma'(n_{\mathscr{L}(\rho_2)\cup\{x\}}) = \mathbf{w} \sqcup \Sigma'(n_{\mathscr{L}(\rho_1)})$ from [5] in $[\![x = y.f]\!]_\Delta$
$\implies \Sigma'(n_{\mathscr{L}'(\rho_2)}) = \mathbf{w} \sqcup \mathbf{r}$ both ρ_1 and ρ_2 are `safe`
$\implies \Sigma'(n_{\mathscr{L}'(\rho_2)}) = \mathbf{r}$
Going from $\delta \to \delta' \to \hat{\Delta}$,
$\varsigma'(\rho_2) = \mathbf{r}$, because typestate unchanged
$\implies \hat{\Sigma}(n_{\mathscr{L}'(\rho_2)}) = \mathbf{r}$

ii. If $\varsigma(\rho_2) = \mathbf{w}$, the reasoning is similar to the one above.

iii. If $\rho_2 =$ `null`, then there is no corresponding edge in Es. It follows that $\Sigma'(n_x) =$ `null`

(b) For all objects other than ρ_2, the typestate is the same as before.

In all cases, we see that $\varsigma(\rho) \leq \hat{\Sigma}(n_{\mathscr{L}(\rho)}) \leq \Sigma'(n_{\mathscr{L}(\rho)})$

Case 5 $\beta([\![x = m(\bar{y})]\!]_\delta(\delta)) \sqsubseteq [\![x = m(\bar{y})]\!]_\Delta(\beta(\delta))$, where $\delta = \langle \sigma, H, \varsigma\rangle$

The safety properties of the S-CALL rule follow from those of *kill* and *scluster*; as long as the set of variables to be invalidated is conservative, *kill* takes care of both safety and accuracy of the abstract heap model; we have sacrificed precision (detailed information) of the heap model in favor of certainty, without losing accuracy. As we pointed out earlier, in the limit case, one could invalidate all variables and have the post-call state to be accurate (only $n_\emptyset$ is left), but completely useless. We have already seen that *scluster* offers us the necessary coverage of all the nodes that could refer to objects whose state cannot be predicted (Defn. 4.7, pg. 60).

□

4.6 Mapping to Java

Java classes identified by the marker interface `Message` are treated as message types. That helps it keep internal objects separate from message objects, as long as the type checker is careful to maintain the original class even under casting.

All the restrictions imposed upon message types have to do with enforcing ownership protocols and containing aliasing. There are a few categories of objects that do not fall into our definition of a message.

4.6.1 Sharable types

Immutable *value* types such as `String` and `Date` ought to be freely exchangeable. However, simply being immutable is not enough, if the object is treated as a resource. For example, a security capability object may need to be strictly accounted for and passed carefully from one subsystem to another;[59] such a scheme would need to deal with aliasing and ownership, even if it is implemented as an immutable Java class. As a side note, it is surprisingly difficult to glean from an examination of the bytecode whether or not a class is immutable. Most classes such as `String` and `Date` are known as immutable, but in reality are *observationally immutable*: not all fields are marked final or computed in the constructor. Some, such as `hashcode` are lazily computed and cached. Some classes such as Java's `java.net.URL` permit inner classes to memoize its state.

Secondly, fitting into an existing thriving ecosystem such as Java means that there will be a number of classes that we would like to exchange (`Socket`), but which are not Kilim message types.

For these two reasons, we require that the programmer make explicit to the type checker that a type is sharable, via a marker interface called `Sharable`. Any class inheriting from `Sharable` is given a free pass by the type checker; a sharable object can be treated as a message, but is not subject to any constraints of aliasing or ownership, or the types of its fields. Objects obtained from fields of `Sharable` objects are treated as regular Java objects, unless they too are instances of a `Sharable` class. For this reason, it is best if the fields of such an object are encapsulated and made accessibly indirectly via thread-safe methods. Finally, the weaver limits static class fields to constant primitive or final `Sharable` objects and prevents exceptions from being message types. Clearly, the `Sharable` annotation is a safety loophole, but a necessary compromise.

For existing classes whose sources cannot be annotated, but whose behavior is known, the programmer can supply "external annotations" to the weaver as a text file:

```
1 class java.lang.String implements Sharable
2
3 interface java.io.PrintStream {
4   void println(safe  Object o);
5 }
6
7 class StringBuffer implements Message
```

This scheme works as if the annotations were present in the original code. Clearly, it only works for non-executable annotations (type-qualifiers used for validation); `Pausable` cannot be used as it results in code transformations. Further, externally annotated libraries are not intended to be passed through Kilim's weaver; the annotations serve to declare the library methods' effects on their parameters.

If a method parameter is not a message type, but is type compatible (upcast to `Object`, for example), then the absence of an annotation is treated as an escape into unknown territory; the weaver treats it as a compile-time error.

4.6.2 Arrays

Arrays are another kind of object that we would like to be able to exchange. The Kilim weaver treats an array of primitive types, `Message` and `Sharable` types as a potential message. If an array is obtained from a message object, it is subject to message verification rules. Expressions of the form $x[i]$ are simply treated as getfield or putfield expressions of a special field type $[\cdot]$. To simplify the type checker, we provide a special *destructive read*[32] function called *cut* that reads a field or an array slot and nullifies it in one atomic operation, similar in sprit to Java's post-increment operator. That is,

$x = \texttt{cut}(y.f) \quad \equiv \quad x = y.f;\ y.f = \texttt{null}$, and

$x = \texttt{cut}(y[i]) \quad \equiv \quad x = y[i];\ y[i] = \texttt{null}$

In both cases, the function yields a *free* object (assuming the array is an array of messages)

4.6.3 Collections and Iterators

Collections and iterators have traditionally been a problem for any type system that enforces ownership and encapsulation protocols, including linear types, ownership types and universe types. One issue is that an iterator object results in multiple aliases, and is distinct from the collection itself.

The other issue has to do with return values. Once a value is returned, the function exercises no control over it. This poses a problem for strict ownership transfer protocols, including ours. Having transferred an object to a collection, a `get` or `lookup` method cannot simply return a contained object without losing control. The obvious option of removing the object from the collection and returning it untethered is not practical.

Our approach is to avoid iterators and return values altogether. Instead, we provide a few special collection classes for messages (hashmaps, lists and sets) that can be supplied *actions*; the introduction of closures to the Java source language should clean up the syntax considerably.

```
1    m = new MsgHashMap();
2    m.put(key, value);
3    ...
4    m.doWith(key, new Action(){
5        void action(@part  Value v) {
6
7        }
8    });
9    m.iterate(new Action() {
10       void action(@part  Entry e) {
11           Key k = e.key; Value v = e.value;
12       }
13   });
```

4.7 Related work

Pointer aliasing is widely used for efficiency. Hackett[80] describes nine ways in which aliasing is used in systems software. The problems raised by aliasing[93] are significant enough to have spawned a rich

vein of research, and in this section we briefly categorize the approaches and discuss the highlights in roughly chronological order.

What differentiates our work from extant literature is not so much the mechanics of the analysis as in our particular design points: to recap briefly, making messages data-oriented (unencapsulated C-like structures), requiring a tree-structure for ease of analysis and for run-time speed, and providing deep implicit capabilities for safety and ownership transfer. Much of our analysis framework can be used in other areas that treat objects as resources, areas such as secure information flow and resource accounting of security capabilities and database connections, for example. Our desire to require minimal changes to the Java toolchain resulted in the Kilim *weaver*, a post hoc analysis tool. Matthew Papi's work on pluggable types for Java[140] uses annotations as type qualifiers, and allows the Java compiler to be enhanced with user-defined types. This facility is expected to be a standard part of the compiler in future versions of Java.[101] Unfortunately, in the absence of type inference machinery in the Java compiler, we feel this approach only serves to add to the syntactic burden.

Immutability, Readonly

When objects are immutable, low-level data races are non-existent (and higher-level data coordination is easily handled using a transaction or workflow coordinator). However, unless a language and environment makes it the default from the ground up (as in Clojure and Haskell), one has to address the mutable aspects of the language. Boyland[33] highlights the complexities of different schemes to add immutability to object-oriented languages. Many attempts[179] to fix the shallow nature of Java's *final* and C++ *const* by providing transitive readonly semantics are flawed because they cannot easily mix immutables and mutables: an immutable container of mutable items, for example, or vice-versa. The other problem is that of *observational exposure*. When a variable (and not the underlying object) is typed readonly, and if it is possible to modify the object via some other variable, the original readonly variable can see the changes as they are happening. Worse, partial updates may be viewable in concurrent systems. This is the reason why we have `safe` as a temporary readonly qualifier, and why variables deemed to be referring to possibly inaccessible objects are rendered invalid.

Pointer Analysis

We originally started with static alias detection and verification in order to not require any programmer annotation. Precise alias analysis is undecidable,[148] and every analysis must necessarily trade speed for precision and coverage. There is a vast amount of literature available on pointer analysis of C programs,[88] that ranges from fast and imprecise analyses such as Steensgard[168] and Andersen,[8] to slow, but precise shape analyses.[154] The solutions differ on whether they are flow-sensitive and context-sensitive, favor local or global whole-program analysis,[187] offline vs. online[89] and choice of implementation representation (constraint sets, BDDs[187]), etc. Many design choices are also based on the application for which they are intended, such as purity and escape analysis.[45,46,155]

The key problem with most alias detection schemes, even given enough time to do whole program analysis, is that they are necessarily limited in their precision in the presence of unrestricted aliasing, multi-threading and shared memory. For this reason, most post hoc analyses (whether online or offline) are used for optimization purposes, not for building correct, modular systems from the ground up. In our experience, even with annotations for isolation and consumption semantics, we found that many analyses based on *pointsTo* sets were too imprecise. As Sagiv[154] et al. point out, these analyses create abstract heap nodes based on their allocation point — all objects allocated at line l are associated with an abstract node n_l. This is equivalent to fixing the set of run-time objects that a variable

may point to, which renders them unable to provide strong nullification and disjointness preservation guarantees.

Shape analysis,[154] on the other hand, is much more precise but can be extremely slow (of the order of tens of seconds for a single file). For this reason, we limit the problem to shape enforcement, thereby making it into a post hoc type system and verification scheme. We add consumption semantics, null-reference tracking and function calls. A less cumbersome alternative (from the programmer's point of view) might be to extend the inference to package boundaries using a limited inter-procedural shape analysis. In future work, we intend to study the unified memory analysis scheme of Marron[122] et al. , as it claims to be significantly faster than intra-procedural shape analysis.

Separation logic

Separation logic[151] extends Hoare logic[90] with logical separations of the heap (this is a direct influence on the heap separation model in the KAM). It provides spatial connectives to reason about parts of the heap in isolation, and rules to reason about merged heaps. Much of separation logic work is devoted to low level C-like languages with address arithmetic and no automatic memory management; Matthew Parkinson's PhD thesis[142] applies separation logic to provide local, modular reasoning for abstract data types and objects. As with any logic based approach (and unlike type and annotation-based approaches), separation logic provides a rich alternate language to make assertions about several variables together and about data structures at any point of the program ("x is a tree, and $x \rightsquigarrow y$"); the flip side is that at its current level of evolution, it appears that the speed of analysis and the level of programmer involvement required are still impediments to widespread industrial adoption (esp. when the structures become more complicated than trees, and when concurrency is involved).

As the limitations of post hoc alias detection and verification schemes became clearer, and as we discovered how brittle many analyses are in the face of evolving software, we shifted our approach to static and modular *enforcement* of aliasing and ownership behavior.

Linearity and Uniqueness

If an object is transferred *linearly* from one variable to another, aliasing is impossible, and at any point, the holder of the linear object has complete rights over it, including deallocation. Memory management — a prime mover behind much research on linearity — belongs to the general category of resource management. Approaches to linearity depend on whether they consider the object to be *used once*, or whether there is only a *single extant reference* to it at run time.

Gerards's linear logic[73] and Lafont's use of it for resource consumption and accounting[109] has been used by type theorists to provide linear types.[183] Linear types prevent aliasing altogether by ensuring that a reference is consumed (like C++'s `auto_ptr`) when it is passed as a parameter or when it occurs on the right hand side of an assignment. Because no aliases exist, a local analysis is enough to show whether the object can be garbage collected, and objects can be updated in-place .[17,18] Clearly, it would be too onerous to give this treatment to every single type, which means a distinction between linear and non-linear types. This leads to the question of whether a linear type can contain or refer to a non-linear type, and vice versa. Wadler[183] provides a `let!` construct that permits a non-linear type to be treated as a linear-type in exchange for temporarily surrendering all aliases to the non-linear type.

Paclang[58] is an industrial example of using uniqueness and transfer of ownership. It is a simple imperative, concurrent language for high speed packet processing and handling, and is a simplification of Kobayashi's quasi-linear types[105] (no higher order functions). It defines a special datatype called `packet`

(an array of bytes) that can be transferred between memory banks and processors. At any one time, a packet may have multiple references, but only within a single thread. The notion of treating a message as a distinct entity was a seminal influence on Kilim. Kilim's message type is an inductive structure with named fields and separately identifiable components (with corresponding aliasing issues).

Guava[15] is a dialect of Java that prevents data races by separating objects into three categories: unsharable objects (cannot `synchronize` on them), `Monitor` objects that can be shared by reference, and immutable `Value` objects which are deep-copied on assignment. A `move` operator allows values to be transferred without copying.

Adding uniqueness to Java-like type systems (those without inference) requires one to explicitly declare uniqueness everywhere it is used, for example, `unique Msg x = y.f`, In addition, older approaches required unique variables to be *destructively read*, where the right hand side is nullified to atomically prevent aliasing. Both of these points add to the syntactic burden; Boyland's alias-burying[32] scheme adds inference and delayed destructive reads via a post hoc static analysis that uses annotations equivalent to `whole` and `safe` on method signatures. We followed their lead in this respect.

Regions

From eliminating aliasing altogether, we next examine proposals to permit aliasing in a controlled fashion. The key idea is to have named regions that can group objects, and to track aliasing at the level of regions.[17] In Tofte and Talpin's type and effect system,[177] each object belongs to a region. New regions can be allocated and deallocated dynamically, and region lifetimes are associated with lexical scope; for this reason, memory management is handled at the region level. Region names can be passed polymorphically to functions and since regions can be deallocated, dangling references are prevented by requiring functions to state which regions they use, and how. One of the best known examples of a region based memory management system is Cyclone,[77] a dialect of C.

This notion of automatic or compiler-generated names as equivalence classes of objects occurs repeatedly in various guises, such as owners in ownership types and guards in capability systems, etc. We examine a number of such proposals next.

Objects and Encapsulation

Encapsulation of state is one of the cornerstones of object orientation and seen as an important tool towards reining in aliasing. The notion of private and protected access in object oriented languages merely protect a single object, but not networks of objects. Encapsulation approaches differ on how they draw boundaries around networks of objects — effectively marking a region — and the restrictions on aliasing that they place across the boundary.

Hogg[92] proposed a scheme called Islands. An island is a cluster of objects, and connected to other islands via *bridge* objects (similar to Kilim mailboxes). An ordinary object is invisible from outside and may only point to a bridge or to other objects in its island. The method parameters of bridge objects must be qualified as *free* or *read*, the equivalents to our `whole` and `safe`, thereby guaranteeing that what goes in and comes out of an island is unaliased. This makes it possible to have Hoare-style specifications on the bridge's methods. The *read* mode has the problem of observational exposure discussed earlier. Almeida's Balloon types[7] has similar support for strong encapsulation, but relies on a static analysis instead of a type system to reduce the syntactic burden. The encapsulation requirements of Islands and Balloons have been considered too strict by researchers.

Confinement types[180, 191] treat the package as a natural boundary, allowing package-scoped classes access only to objects within the same package or in nested packages. The encapsulation achieved here is necessarily weaker, and the problem of concurrent modification is orthogonal to it.

The Universe type system[56] by Dietel and Müller has the notions of *rep* and *peer*. A *rep* object is part of the representation of its owning object and is fully contained within. Ownership transfer was addressed in a later extension.[129] Two objects are *peers* if they have the same owner (cons cells in a linked list, for example). An object can only modify the fields of its own rep objects or that of a peer, commonly called as the *owner-as-modifier* approach. All other references are marked as readonly, in that a non-rep or non-peer field cannot be used to effect a modification, so although aliasing is permitted, the proposal limits the places where a given object can be modified. This scheme has the problem of observational exposure.

Flexible Alias Protection[133] addresses the read-reference concept differently. Objects that cross an abstraction boundary — called *argument* objects — can be passed in a special kind of read mode called *arg*-mode; a reference in this mode can only view immutable parts of an object, thus accommodating aliasing without compromising safety. The proposal combines this with other modes we have seen before: *free* and *rep*. Aliasing is either limited to related internal objects, or must be in the benign *arg* form to be made public. We decided not to add this facility to Kilim, because we wanted to explore the ramifications of isolation and ownership transfer.

Treating all reachable objects as nested or owned objects in need of encapsulation is untenable. After all, a collection and its elements are independent. Ownership type schemes pioneered by Clarke[47, 49] draw an encapsulation boundary using a dependency relationship. When an object directly modifies a field of another object or calls a method that modifies the object's state, the former is said to depend upon the latter. By this definition, a list does not depend upon, or own its elements.

In ownership types, every object has an owner, and its type is parametrized by an owner name (similar to regions). Owner names are supplied as polymorphic identifiers on class and method declarations. For example, the type `Foo<this>` refers to all objects of class `Foo` and owned by the current object; in effect, it declares a new type `Foo` for each owning object, which fixes the owner of an object for its lifetime and prevents transfer of ownership altogether. The type `Foo<bar>` refers to an ownership parameter `bar` obtained from a class or method declaration such as this: `class C<owner, bar>{...}`. This permits an object to refer to objects owned by other objects, and not mix one owning class with another.

Alias containment is achieved by ensuring that an owned object can never be public or reachable from a return value; this forces access to an object to always go through its owner. This is known as the owner-as-dominator approach, in contrast to the owner-as-modifier property of Universes.

Ownership type schemes do not handle iterators, because they are conceptually part of the collection but also publicly visible at the same time. Boyapati's SafeJava[31] adds *where*-clauses to further specify and constrain ownership relations. This, combined with a principled exception for inner classes addresses the iterator problem.

Ownership types as described so far have no idea about threads and unsafe access. Clark and wrigstad,[48] and subsequently SafeJava, combined ownership types with unique references to effect ownership transfer. The systems prevent variable aliases to unique references by providing explicit destructive reads: $x = y$ is equivalent to $x = y; y =$ `null`. To temporarily relax the uniqueness restriction, the systems provide a *borrow* construct that introduces a lexical scope where a unique variable may

be aliased. The aliases are destroyed and uniqueness is regained at the end of the scope. Unique references and ownership types together yield an *external uniqueness* property, where it is possible to ensure that a message object (for example) is uniquely transferred from one subsystem to another, while the ownership type scheme preserves OO encapsulation.

The problem with most ownership type schemes in practice is the syntactic burden, even with SafeJava-style type inference and well-chosen default annotations. Since each object must have an owner, an abstract data type must accommodate a corresponding owner parameter name for each object seen in its interface. For example, consider a Pair class below:

```
class Pair<owner, oa, ob>
  where oa ≽ owner, ob ≽ owner
{
    Object<oa> a;   // a owned by oa
    Object<oa> b;   // a owned by ob
}
```

Streamflex[164] and later ScopeJ[190] provide an implicit ownership type system for the real-time Java setting. Because memory allocation is crucial and real-time garbage collection is frowned upon, the scheme relies on lexical scopes to combine ownership and regions, which in turn means that ownership is implicitly tied to the lexical scope.

Capabilities and Permissions

We next examine capability-based systems.[52] The key principle here is to separate *possession* of a pointer from the capability to *use* the pointer. The static analysis tracks the type environment separately from the capability environment, and a pointer can be used only if there is a capability to use it. Such systems may even permit dangling references, and guarantee safety by retracting the corresponding access capabilities. Further, distinction is drawn between a *linear* capability, which indicates exclusive ownership and permission to deallocate a region, and a *non-linear* capability for reading and writing.

The "Capabilities for Sharing" proposal by Boyland [34] et al. unifies uniqueness and immutability. It associates pointer variables and fields with a set of capabilities: basic rights to read and write fields, exclusive rights to deny other variables the right to access the same object, and finally, an ownership right, which effectively makes a variable the exclusive owner of all rights to an object, and also the meta-capability to deny rights to other aliases.

AliasJava[6] is a dialect of Java that combines alias annotations for uniqueness (`unique`, `shared`), and ownership (`lent`, `owned`). These annotations are applicable at all type declaration points, but the implementation infers annotations using an intraprocedural constraint analysis. AliasJava provides alias control mechanisms that are oblivious to concurrent usage, and data races are possible. For example, it is possible to use a unique reference even while the reference has been lent out.

Typed Assembly Language[127] and later, Alias types[161] target lower-level compiler intermediate languages where memory management is important. Here, single objects are tracked (not regions) and pointers refer to abstract location names. Crucially, the capability and type environments are intermixed: while a pointer variable mentions a location, a capability mentions both the location and type of its contents. For example, a pointer variable's type is denoted by `ptr` ρ and the associated capability is $\{\rho \mapsto \tau\}$, which denotes a linear ownership of an object ρ whose contents are of type τ. The linear

capability allows the owner to change the underlying type of the object, as would be necessary in a memory manager; with a single change, all pointers pointing to that object point to a different object. Non-linear capabilities are denoted by $\{\rho \mapsto \tau\}^+$.

Alias types were later enhanced with regions,[184] and existential types to anonymously name regions. For example, the type $\exists\rho.\texttt{ptr}\ \rho$ denotes a pointer to some region with some name ρ. If this were a pointer to a pair, one would say $\exists\rho.[\{\rho \mapsto \langle\tau_1, \tau_2\rangle\}]\texttt{ptr}\rho$, which packages exclusive capabilities to a tuple inside an anonymous region. Semantic rules to pack and unpack such hidden names were added using a notion of packing and unpacking the hidden capabilities. This scheme allows one to describe recursive data structures and embedding of linear types into non-linear ones. Alias types were a seminal influence on languages such as Vault[55] and Sing$^\#$.[60]

Using existential types has the problem that each anonymous region generates its own unique type. Fähndrich and Deline[61] introduced *adoption* and *focus* to fix this problem. Here, an object is born in its own region (and being a singleton, is linear as well). Adoption allows a linear object to be adopted by another region; it thereby loses both its region and its linear capability. The *focus* operator provides a lexical scope which allows one to focus on a particular object, while barring any aliases of that region from being usable within that scope. When the scope of the focus operation ends, the region is back to being non-linear.

In recent work, Haller and Odersky[82] build on the above concepts to provide concurrency-safe external uniqueness and also represents an excellent mix of the state of the art and programming convenience. An `@unique` annotation on method parameters forces the caller to forgo the reference. This is coupled with `expose`, a feature similar to Vault's `focus`, that provides a block inside which modifications can be done and arbitrary internal aliases can be created, but references from outside the block are prevented from referring to an exposed object's innards. By temporarily revoking uniqueness capabilities to a unique object, it prevents that object from being viewed simultaneously by two different threads.

Their system is less general (and consequently simpler) than Vault on one count: they do not provide the parametrizable region naming scheme seen in Vault; instead, they enforce the rule that any two exposed parameters supplied to a method must belong to the same region.

5 Kilim architecture and implementation

The Kilim runtime consists of components typical of user-level threading infrastructures: lightweight threads, user-level schedulers, communication and synchronization mechanisms. The differences from existing frameworks stem mainly from our focus on a server-side engineer's checklist. This chapter presents some of the important design choices and engineering techniques, in the context of the checklist. We start with a summary:

Performance & scalability	Threading optimizations (§5.1.4) and results (§6.1) Kilim I/O library (§5.4), and results (§6.2) Scheduler Hopping (§5.3.1) Messaging types to pass messages by reference Simple serialiazation of messaging types
Portability	JVM-independent lightweight threading
Decoupling	Mailboxes for control isolation Messaging types for heap isolation
Safety	Messaging types
Overload control	Bounded buffers in mailboxes (§5.2) Staging support with multiple schedulers (§5.3.1)
Multi-processor support	Thread-pool control with schedulers, thread affinity Blocking & pausing support in mailboxes (§5.2) Multiple schedulers, scheduler hopping (§5.3)
Fault-tolerance	Actor-linkage, watch-dog and mailbox timers
Monitoring	Interceptors in all communication constructs
Simplicity	Automatic stack management (linear control flow)
Preemption	Watch-dog timers (§5.1.5)

5.1 Actors and Fibers

A Kilim actor is essentially a combination of a lightweight thread and a private heap, but these are not notions that the user needs to care about.

```
package kilim;
class Actor {
  public void start();
  public @pausable void yield();
  public @pausable void sleep(long ms);
  public void informOnExit(Mailbox mb);
  public ExitMsg join();
  public ExitMsg joinb();
  public @pausable execute() throws Exception {}
}
```

Applications define a subclass of the Kilim Actor class and override the `execute()` method, which serves as the starting point (like a program's `main` or a Java thread's `run` method). The `start()` method is equivalent to the `spawn` instruction in KAM. The type system ensures that an Actor's constructor and public methods only message or value type paramters. We will explain the `join`, `informOnExit` and `joinb` methods later.

The rest of this section addresses three requirements: multiplexing hundreds of thousands of actors onto a handful of kernel threads, portability and preemption. The Kilim framework has been written in a way that it can easily be ported to a VM that provides very lightweight user-level threads; in such a case, most of this section (except preemption) would be unnecessary. However, there is much more to an actor than just user-space threading, as the rest of the chapter will demonstrate.

Most JVMs do not permit stack inspection (user code cannot examine the calling method's variables and operand stack, for example), leave alone permit stack switching. The only portable alternative is to transform bytecode to have each actor cooperatively unwind itself and leave the kernel thread free to rewind some other task in its place. The transformation is a simple variant of single-shot delimited continuations.[37] The Kilim actor implementation uses such a facility, provided by the `Fiber` class.

Here is a brief detour on the notion of continuations before we examine the Kilim threading infrastructure.

5.1.1 Continuations: An overview

A continuation is a label (program counter) coupled with some data; *evaluating* a continuation is the moral equivalent of a goto-with-data. Given this, a traditional procedure call is a special case of evaluating two continuations, the transfer of control to the procedure, followed by a jump back to the caller. In languages that provide first-class continuations, the compilers often transform all code into Continuation-Passing Style[10] (CPS). Here, each procedure signature is transformed to include an extra continuation parameter, which contains the return label to jump back to (along with the caller's activation frame); the last thing performed by any procedure is to evaluate the return continuation. If the procedure were to call another procedure (instead of returning), it would generate its own return continuation to pass to the new procedure, and link the continuation given to it to the new one it generated. This way, the traditional call stack with its stack of activation frames is merely a linked list of continuations going back all the way to the first procedure. By providing a general goto-with-data paradigm, continuations unify[112,170] all forms of control flow patterns including call/return, exceptions, signals, callbacks, loops, recursion, backtracking, coroutines, task switching, ICON-style generators[76] etc. They correspond directly to the denotational semantics of programming languages.[170]

Four decades have passed since continuations were first discussed in earnest;[150] they are only just beginning to be adopted by the mainstream. Given native hardware support for procedure call stacks (register windows for parameters, stack registers), the predominance of synchronous procedure calls (including crucially the system call interface), and the stack-oriented mentality (stack traces for debugging, stack inspection used by the Java security framework), there is even less incentive for most mainstream systems implementations to provide first-class support for continuations. Further, experience with Scheme's `call-with-current-continuation` (call/cc) has shown them to be too inefficient and error-prone in practice, especially so in the presence of mutable global state and hardware parallelism; evaluating a continuation may involve a scheduling decision as well.

Our original interest in continuations was to use them for asynchronous handoff of messages between protocol layers, without building up a stack or holding the caller's stack hostage. This is a problem central to all reactive, event-driven frameworks such as GUIs, timer services, signal handlers, network I/O handlers, and interrupt service routines. If the callback routine (the handler) in any of these cases fails to terminate (or just takes its own sweet time), it runs the danger of stalling the entire service of which it is a part. Our second interest was in providing ICON-style generators, an elegant generalization of iterators.

The Kilim framework uses a restricted form of continuations that always transfers control to its caller but maintain an independent stack. It satisfies the goals above with the following architectural choices:

Suspend-Resume. We preserve the standard call stack, but provide a way to *pause* (suspend) the current stack and to store it in a continuation object called `Fiber`). The Fiber is resumed at some future time. Calling `Fiber.pause()` pops (and squirrels away) activation frames* until it reaches the method that initiated `resume()`. This pair of calls is akin to `shift/reset` from the literature on delimited continuations; they delimit the section of the stack to be squirrelled away. Although Kilim fibers are not part of the public API, they have been used by other projects.[1,136] The entity responsible for resuming the Fiber differentiates the next two points.

Schedulable Continuations. Kilim actors are essentially thread-safe wrappers around Fibers. A scheduler chooses which Actor to resume on which kernel thread, thereby multiplexing hundreds of thousands of actors onto a handful of kernel threads. Kernel threads are treated as virtual processors (similar to the scheduler activations[9] approach) while actors are viewed as agents that can migrate between kernel threads.

Generators. The Kilim framework also provides generators†, essentially suspendable iterators. When resumed, they yield the next element and promptly pause again. Generators are intended to be used by a single actor at a time, and run on the thread-stack of that actor, which means that although the actor is technically running, it is prevented from executing any of its code until the generator yields the next element.

5.1.2 Automatic Stack Management

The Kilim toolkit provides a tool called a *weaver* that transforms the bytecode of each method along the call chain, as shown in Figure 5.1. This is a well-known technique,[66,96,152,175] and the differences are

*Also known as *stack-ripping*[2] or *trampolining*.
†equivalent to Python's `yield`.

Original

```
1  g(int n) {
2   for (int i = 0; i < n; i++) {
3    h();
4   }
5   ...
6  }
7
8  h() {
9   ...
10 }
```

Woven

```
1  g(int n, Fiber f) {
2    if (f.isResuming)              // prelude
3     switch(f.pc) {
4      case H1: i = 0; n = 0; goto H1;
5     ...
6    }
7    for (int i = 0; i < 10; i++) {
8      H1:                          // pre-call
9     f.down();
10    h(f);
11    f.up();                      // post-call
12    if (f.isPausing)
13      if (!f.hasState)
14        f.state = new State_I2(i,n);
15        f.pc = H1;
16      }
17      return;
18    } else if (f.hasState)
19      State_I2 st = (State_I2) f.state;
20      i = st.i1;  n = st.i2;
21    }
22   }
23   ...
24 }
25
26 h(Fiber f) {
27  ...
28 }
```

Figure 5.1: Kilim code *weaving*: Providing support for suspending and resuming the stack. Note that the transformation is on bytecode, hence the presence of `goto` statements.

primarily in some of the optimizations described later. We present below the gist of our transformation (unless otherwise noted, all line numbers refer to the right hand side of Fig. 5.1).

First, the method's signature gets an extra argument called a Fiber. This object is used as a peg to hang saved information for a subsequent restoration. The method's entry is patched with a *prelude* section, essentially a set of jumps to each call site. Finally, each call site (line 3 on the left) is sandwiched with pre- and post-call sections.

Let us look at stack suspension first (the post-call section in Fig. 5.1). Once `h` returns, `g` examines the fiber object to see whether the call returned normally or if it signalled that it wants to *pause*. If the latter, line 14 squirrels away to the fiber the information that it will need upon resumption (`i`, `n` and the code offset to return to `H1`). We use a canonical class tailored to the call site to store the information: in this example, `State_I2` has room for exactly two integers. Once `g` returns, its caller does the same style of post-call processing, and so on all the way up the call chain.

Stack resumption is split between the prelude and post-call sections. The prelude restores the program counter and jumps to the call site that earlier triggered the suspend. In the post-call section, if the fiber signals that `h()` returned normally, the state is restored (line 19) before moving on to line 23.

5.1.3 JVM constraints

The JVM's verifier and the threading-related constructs place a number of other constraints on portable stack switching. This section catalogues some of the major issues.

Verifier Unlike C or machine language, one cannot simply insert a jump instruction; the JVM verifier performs liveness and data type analyses to ensure that for each instruction in the method, the live local variables and operand stack have the same data type and that the operand stack has the same depth, regardless of the path taken to get to that point. The Kilim weaver performs the same analysis as the Java verifier and ensures that the frame's local variables are initialized with default values (corresponding to the variable's type) along newly introduced edges in the control flow graph (Line 4 in Fig. 5.1).

Allocation and Initialization The `new` expression (say, `new Foo(m())`) in Java is split into allocation, argument evaluation and the constructor call at the bytecode level:

```
t1 = new Foo // allocate instance
t2 = m()
t1.<init>(t2)  // constructor call
```

The verifier prevents jumps in and out of sections between the `new` and the corresponding constructor call(`<init>`). Observe that the transformation of Fig. 5.1 violates this constraint: the prelude jumps directly to a call site without regard to the target label's context. Our simple solution is to pull the allocation and initialization parts into a separate helper method (there is no run-time overhead since it gets inlined at run time):

```
t2 = m()
t1 = new_init_Foo(t2)

private static Foo new_init_Foo(y) {
    x = new Foo(); x.init(y);
    return x;
}
```

This solution performs local surgery on the original bytecode, yet easily handles complicated expressions with conditional constructs such as this:

```
new Foo(check? m(new Bar(n())): new Baz().n())
```

It is worth noting here that we explicitly disallow calling pausable methods *within* constructors (that is, a constructor cannot be marked `pausable`), because an object should not be receiving messages until it is fully initialized.

Java subroutines The Java VM has the notion of a local subroutine, originally intended for code in `finally` blocks to be callable from both the try and catch sections. There are two problems with this construct: (*i*) it prevents normal intra-procedural analysis techniques because, by definition, it is a separate subroutine and (*ii*) a subroutine's bounds are nebulous — the Java specification *informally* defines a subroutine as the set of instructions reachable from the `jsr` instruction (jump to subroutine); this could legally encompass the whole method. Most compilers no longer emit this instruction and

prefer to simply duplicate the `finally` block; the Kilim weaver follows suit, by inlining the transitive closure of instructions at the subroutine's call site.

Thread Affinity The `monitorenter` and `monitorexit` instructions are thread-affine and would require a fiber to be restored in the same kernel thread as the one in which it was reified. We take the simpler alternative of prohibiting pausable calls within synchronized blocks, with the intent to discourage sharing. However, we do provide a run-time way of pinning an actor to a particular thread, in order to coexist with libraries that rely on the kernel thread's identity. This is distinct from higher-level logical locks built atop messaging primitives to synchronize and sequence actor operation.

5.1.4 Optimizations

We have thus far discussed reasonably well-known techniques for switching stacks portably in Java. This section describes a number of performance-oriented optimizations that set Kilim apart from competing approaches.

Pausable methods. To avoid transforming all code, we introduce a Java method annotation called `@pausable` that helps the programmer mark those methods that wish to suspend and resume. This annotation is an interface contract similar to a checked exception; a method is pausable if it calls or overrides or implements a pausable method. The side benefit of this explicit annotation is that it alerts the programmer to the potential cost of stack switching (and hence puts a noticeable back-pressure on the design).

As Adya et al.[2] noted, such an explicit marker also helps with software evolution, because one cannot suddenly turn a non-pausing method into a pausable one without the caller being affected.

Return, not throw. Some researchers have used exceptions as a `longjmp` mechanism to unwind the stack; we use `return` because we found exceptions to be more expensive by almost an order of magnitude, especially because they have to be caught and re-thrown at each level of the stack chain to reify the call chain. Further, an exception clears the operand stack, which forces one to take a snapshot of the operand stack before making a call; in our experiments, lazy storage and restoration performs better.

Lazy storage and restoration. Figure 5.1 (line 14) shows that the frame is captured only when the callee signals a suspension (and then only if it was not captured in an earlier iteration). It is restored to its original values only when control needs to be passed to the original code following the invocation (line 23). Note that the control plane (the program counter) is restored eagerly in the prelude (with dummy values for the data to keep the verifier happy), but the data plane is restored only when absolutely required. Our experience with Kilim-based projects (Chap. 6) has been that intermediate frames in a call chain tend to get used far less than the topmost frame, and it is frequently unnecessary to eagerly initialize the local variables with the stored values.

In general, mainstream programming languages and compilers do not offer tail-call optimization (TCO), nor are programs written in a way to make use of TCO even when available. As a consequence, call chains tend to be large (fifty deep is quite common in Java-based servers). Storing and restoring such stacks would undoubtedly drag performance down, but as it turns out, the actor architecture never encounters this problem. This is because subsystems are chopped up into actors with their own private

stacks. For example, a typical call chain in a Java server contains layers of methods related to socket handling, parsing, message dispatching, and application code. The equivalent actor architecture splits these layers up into several actors that suspend and resume independently and at different frequencies.

Parameters, not thread-locals. Some approaches[96] do not modify the signature of each pausable method, preferring instead to pass the continuation object as a thread-local variable that associates information with the kernel thread. While the approach is syntactically cleaner, we found that the thread-local facility in the `pthreads` library and in the present implementations of the JVM is about 10x slower than passing the fiber object as a parameter (because they involve hash-table lookups). This has considerable impact when each method in the call chain is forced to use it. Of course, a faster thread-local facility (such as using the x86 segment registers) renders moot the performance advantage. However, by changing the signature, we use the JVM's verifier in ensuring that unwoven code is never loaded by accident. This too is a non-issue if a classloader incorporates the weaver and prevents the presence of unwoven code altogether.

Constant and Duplicate propagation. The Kilim weaver's static analysis not only tracks which registers and stack elements are live at any point in the code, it also tracks which of them have duplicated or constant values. These variables (if live) can be initialized and restored using embedded code, instead of relying on the `State` object.

5.1.5 Cooperative Preemption

The Kilim actor class provides two methods called `sigKill()` and `sigYield()`, which send internal notifications to the actor via a boolean flag (marked volatile) in the fiber. This flag is checked on entry to every pausable method. An actor can provide a custom implementation of this method, but as with Unix signal handling or any multi-threaded code, must ensure that the data is accessed in a thread-safe manner and that the overridden method does not itself block indefinitely. One could also take the Erlang approach of letting an actor die, and let a hierarchy of supervisor objects bring the system back to a consistent state.

This facility provides a measure of cooperative preemption; as long as the actor calls a pausable method, this flag is guaranteed to be checked. This of course does not prevent a runaway actor. It is trivial for the weaver to add a check at every back-edge of a loop, but in our limited experience, runaway actors are rare enough that it may not be worth burdening every loop of every method.

5.1.6 Lightweight Threads: Related Work

There are any number of solutions for lightweight cooperative task management, even within the Java world. To our knowledge, none of these solutions provide any support for preemption.

Adya et al.[2] observe that the notions of cooperative tasking and automatic stack management are often conflated. There are many Java solutions that belong solely to the former, such as Java's built in `Executor` service, Doug Lee's *FJTask*[115]) and Cilk.[25] Run-to-completion services are useful especially in fork-join parallelism (such as those found in MPI applications), but are not general enough for long-lived ad-hoc workflow tasks.

The Capriccio project[182] modified the user-level POSIX threads (`pthreads`) library to avoid overly conservative pre-allocation of heap and stack space, relying instead on a static analysis of code to size

the stack, and embedded code to dynamically grow the stack where necessary. They report scalability to 100,000 preemptively-scheduled threads. Meanwhile Linux sports a new threading infrastructure (NPTL: Native Posix Thread Library) with tremendously improved performance;[95] we have not determined how Capriccio fares in comparison.

Pettyjohn et al. [144] generalise previous approaches to implementing first-class continuations for environments that do not support stack inspections. However, their generated code is considerably less efficient than ours; it relies on exceptions for stack unwinding, and splits the code into top-level procedures, which results in loops being split into virtual function calls. The transformation also changes access modifiers: private methods become package-public for the continuation to resume a method. Kilim's local surgery combined with lazy suspension and resumption of the data plane gives excellent performance and its preservation of the original call stack chain preserves the developer's mental model of the control flow.

There are many frameworks that transform Java bytecode internally into a style similar to ours; we examined three for which the implementation details were available: Brakes,[178] RIFE,[152] and JavaFlow.[96]

The JavaFlow project uses thread-local variables to pass the continuation instead of modifying method signatures, as we do. While clean, this approach is error-prone because in the absence of any information from the interface (or some global knowledge of all the classes in the program) it is impossible to know whether the called interface or virtual method is pausable or not. A non-transformed piece of code would not know to look for yielding returns. In our case, the verifier would refuse to load the class because of the signature mismatch.

JavaFlow correctly transforms a method if it can reach a `suspend()` invocation. But it unnecessarily transforms all non-pausable methods reachable from there as well, leading to substantial code bloat. None of these projects do liveness or value analysis as of this writing. This means they must store all local variables regardless of whether they will be used after resumption. We analyzed some popular Java projects (Apache Tomcat, JBoss, Antlr and the JDK) and discovered only about 30-40% of the information on average is used across method invocations. Finally, none of these projects handle subroutines and constructor calls with pausable methods.

We refer the reader to a more detailed comparison presented in our paper;[165] importantly, the Kilim runtime performance was significantly better than any of the other portable frameworks we tested.

There are also a number of projects that have attempted to reify the state of the stack for thread persistence and migration reasons; Wei Tao's PhD thesis[174] is a good account of the problem. We have chosen not to provide for thread migration because a data center typically undergoes rolling upgrades, and it is possible for one machine's software version to differ from another. None of the schemes described in this chapter, including Tao's work, account for this reality. Another problem is the inability to transfer unserializable state that is tied to the machine, such as socket connections, database connections and file handles.

Xavier Leroy neatly sums up all the challenges of bytecode verification and provides formalizations and algorithms for doing type analysis.[118] The Kilim weaver does value analysis in addition to types, and tracks duplicate values and constants. We settled on the ASM toolkit[108] (in preference to SOOT and BCEL) for its speed and compactness, but used our own verification and analysis engine.

5.2 Communication and Coordination

Kilim consolidates all communication, notification and coordination activities via the `Mailbox` class, a type-safe multiple producer, single-consumer concurrent queue used as follows:

```
Mailbox<MyMsg> mbx = new Mailbox<MyMsg>();
mbx.put(msg);
mbx.put(10); // Type-error
MyMsg m = mbx.get();
```

Type-safe concurrent access is however only one of the aspects of communication and coordination. We now highlight a few design features that make `Mailbox` a one-stop shop for a variety of needs.

Blocking and Pausing. We use the term *blocking* to refer to any method that can block the current kernel thread and *pausing* to refer to the Kilim suspension mechanism. In the current environment, all existing thread-safe libraries and system calls are oriented towards blocking the thread and it would be impractical to provide a pausable equivalent for all blocking calls. Our measurements indicated that although threads are not lightweight, they are not so heavy that one has to be utterly parsimonious in their use. For this reason, we take a somewhat relaxed attitude towards kernel thread creation. Still, there is a limit to the number of kernel threads one can create, and to help Kilim programs straddle the blocking and pausing worlds, we provide variants of `get` and `put` in the mailbox, as shown below.

```
class Mailbox<T> {
    @pausable void put(@whole T msg); // pause actor until put succeeds
    void          putb(@whole T msg);  // block thread until put succeeds
    boolean       putnb(@whole T msg); // non-blocking put (return true on success)
    @whole T      get();
    ...
}
```

The mailbox thus becomes an adaptor that permits the producer and consumer to have independent semantics; one can block while the other pauses. This way an actor can supply or retrieve data from a chain of blocking Java I/O streams that are completely oblivious to Kilim.

Type support. The listing above shows that the API is not only type-safe with respect to the structural Java type, it also carries the *`whole`* annotation for linear ownership of mutable data (recall from Chapter 3 that this annotation forces the type to be a message type, but the linearity restriction does not apply to classes marked *`sharable`*).

Selection. Kilim provides simple CSP-style ALT'ing[91] in the form of a `select()` call that resumes the calling actor when any of the n mailboxes supplied to it is `get`-ready. We intend to provide the full generality of Concurrent ML[149] and JCSP[185] (guards, prioritized selection, waiting for both `get`- and `put`-ready events and so on).

Overload Control. Mailboxes have optional bounds for overload control — `put()` pauses when this bound is reached and resumes only when the mailbox is drained. This is useful for structuring SEDA-like stages.[186] The current design requires the programmer to specify the bound, and further research is needed to help determine such a number adaptively; it is never a good idea to embed magic numbers.

Support for Push/Pull. In addition to blocking or pausing, the mailbox supports the traditional subject-observer pattern to accommodate sources or sinks that are neither threads nor actors.

```
class Mailbox<T> {
 void addSpaceAvailableListener(EventSubscriber);
 void addMsgAvailableListener(EventSubscriber);
}
```

This facility can be used to set up a chain of mailboxes, for example, each of which is an "inline" data transformer (much like a traditional protocol stack). The value of push/pull communications is illustrated beautifully in the Click modular router project.[106] The disadvantage of course is that the entity doing the pushing and pulling gets its stack held up until the chain finishes processing. The Kilim project was initiated precisely due to the problems of such subsystem coupling.

Interceptors. `Mailbox` supports arbitrary `put` and `get` hooks that allow the message to be transformed, discarded (eliminating duplicates) or simply examined by a debugger. This style of interception permits debugging or profiling probes to be inserted at coordination points, a luxury not available to most other forms of coordination (shared memory or logic variables).

Coordination. Unlike other toolkits (such as JCSP), mailboxes are not central to the infrastructure; we expect developers to provide alternative synchronization schemes as well. For example, Kilim provides a `Cell`, a special Mailbox type with just one slot, equivalent to Haskell's `MVar`. Single assignment `Future` (or promises) for declarative concurrency can be added trivially as a subclass of `Cell`. Finally, classic concurrent signalling mechanisms such as semaphores are possible as well. In all these cases, the Kilim paradigm expects the library author to provide support for the other aspects (such as support for push/pull, monitoring, naming services, linear typing etc.) as well.

Exit Monitoring. Kilim allows actors to monitor the fate of other actors via a special exit message, as shown below (similar to Erlang process-linking). Any number of mailboxes can be registered with an actor or exit messages.

```
Mailbox<ExitMsg> exitmb = new Mailbox<ExitMsg>();

actor1.informOnExit(exitmb);
actor2.informOnExit(exitmb);

ExitMsg m = exitmb.get(); // wait for an ExitMsg
```

The common case of fork-join parallelism — the parent forking off a number of children and waiting for them to finish before proceeding — is handled with a `join()` convenience method (there are pausing and blocking variants of this method as well):

```
a = new MyActor().start();
a.join();
```

5.3 Scheduling

The attractive aspect of using a user-level scheduler is to offer the application the ability to dictate the order, timing of actor execution, and the execution context (including affinity to threads and processors). The Kilim scheduler interface binds together a collection of runnable actors, a pool of kernel-threads and a scheduling policy. We treat kernel threads as *virtual processors*, and the supplied schedulers only create as many threads as there are processors. However, a watchdog timer monitors when no thread has become idle in a while (either because they are busy or blocked in system calls) and spawns an extra kernel thread (up to a configured limit).

```
class MyActor extends Kilim.Actor{ ... }

new MyActor().start();           // schedule on default scheduler

Scheduler s = new Scheduler(10); // create a separate thread pool with 10 threads
new MyActor().start(s);          // explicitly assign a scheduler.
```

Two types of scheduler implementations are supplied, a simple round-robin implementation and another that manages a network *select*-loop (§5.4).

We are particularly interested in custom scheduler implementations with application-specific knowledge. For example, we have experimented with a scheduler that is savvy about database transactions and gives priority to those suspended actors that have been around the longest (earliest-deadline-first scheduling). Section 6.4 describes this setup and its performance profile.

Another example is a network simulator, where a custom scheduler installs mailbox interceptors and dictates the logical ordering of message delivery; this way a `put` or `get` can be artificially delayed or "lost". Given a level of indirection between logical time and real clock time, it should be possible to write applications that run largely unchanged in simulation and real settings.

5.3.1 Scheduler Hopping

An application can contain multiple schedulers at run-time; they are merely independent thread pools. Actors are allowed to switch between schedulers, similar to agent migration:

```
class MyActor extends Actor{
    @pausable void foo() {
        ...
        resumeOnScheduler(otherScheduler); // yield here,
        // resumed in one of otherScheduler's threads
    }
}
```

We use the concept of scheduler hopping in two ways. The first is to explore its use in data partitioning. The conventional approach in a shared-nothing scheme is to send messages to a database actor. The alternative is to partition the data and have each thread own its own logical slice of the data (based on key-hashing, for example). To access a piece of data, the actor uses a partition-savvy scheduler to hop to the thread that owns the data. With one call, the actor transforms itself into a database-side stored procedure with direct and exclusive access to the data.

Another use for scheduler-hopping is to easily emulate SEDA-style staging.[186] A network task can be written as a single read-process-write chain, but different parts could be handled by different schedulers. This is an easy mechanism for controlling overload (SEDA's objective) which achieves good cache locality if threads are pinned to processors.

It is possible to separate out staging as a separate, dynamically configured aspect of the application. Consider:

```
class MyActor extends Actor{
    @pausable void execute() {
        stage("packet_assembly");
        ... // frame packet
        stage("parse");
        ...
        partition = getPartition(msg.key);
        stage(partition);
        ... call database
        stage("output");
    }
}
```

The idea is for the application programmer to identify parts of the application that can be logically scheduled separately although the code is written in a linear style. This scheme permits late binding of stage names to scheduler instances, and even dynamic repartitioning based on processor configurations, load balancing and scheduler specialization; for example, the *packet_assembly* and *output* stages may be assigned to the network scheduler (§5.4). Even if all stages were mapped to a single thread (on a uniprocessor), the scheduler could still arrange to do *wavefront scheduling*, sequentially moving from stage to stage resuming only those actors that are runnable in the current stage. All new actors that get ready for a particular stage are processed in the next generation (even if they belong to the current stage.)

5.3.2 Scheduling: Related Work

We have found the following pieces of work instructive for what can be achieved with control over scheduling: the cohort scheduling work by Larus and Parkes,[113] and the work-stealing algorithms and hierarchical task structures of the Cilk project.[25] Minor Gordon's PhD thesis[75] on "Stage scheduling for CPU-intensive servers" provides a good overview of scheduling algorithms.

Although every threading framework must have a scheduler, there is surprisingly little literature on application-controllable scheduling for servers, leave alone having a combination of scheduling policies for different components or aspects. Note also that it is not just the ability to replace the scheduler instance (which many provide), but the rest of the framework must provide information that the scheduler can use (such as the use of mailbox interceptors and stage names in Kilim). Dynamic mapping of stage names to scheduler instances can help researchers study the effect of switching the mix of scheduling policies at run time.

We are particularly excited about hierarchical scheduling as exemplified by the BubbleSched framework.[176] This framework allows schedulers to be themselves schedulable; each scheduler maps its constituent schedulable entities to a finer range of virtual processors. At the leaf is a scheduler that

maps multiple tasks to a single core. This pattern has also been described earlier as CPU inheritance scheduling.[63]

Constraining data access via temporal scheduling is illustrated by the FairThreads projects ,[30] which has a mixture of cooperative and preemptive threads, the former managed by a fair scheduler. The implementation relies on Java threads, however. Another approach to coupling scheduling to data access is illustrated by CoBoxes,[157] which partitions data between multiple `CoBox` instances. A task runs under the aegis of a single `CoBox` with complete uninterrupted access to the `CoBox`'s data. Nested CoBoxes are supported for finer grained parallelism.

Finally, we must mention the elegant simulation framework JIST.[21] Their bytecode transformation is similar to that of Kilim. Its style of advancing the logical time is somewhat unintuitive, and we believe that it is better to shift the logical timing aspects into Kilim-style staging, message passing and scheduling support.

5.4 Network Support

No server framework would be complete without support for network I/O. We briefly examine some of the pros and cons of dealing with blocking and non-blocking I/O facilities, then describe the Kilim architecture: fast non-blocking I/O with automatic stack management. A socket is non-blocking if it merely returns the number of bytes read or written, instead of blocking the current thread until some I/O could take place.

A thread-per-connecton architecture is simple to write because each client session can block without affecting anyone else:

```
while (true) {
    // loop runs in a separate thread dedicated to a single socket
    readRequest();
    ... process event
    writeResponse()
}
```

While the linear style of coding is undoubtedly attractive, this approach has not curried favour with server-side engineers because of the performance problems enumerated in Chapter 2. Instead, engineers have found that a combination of non-blocking sockets and a kernel supported *selector* call such as `select`, `kqueue` or `epoll` is a lighter, faster, more scalable alternative to dedicating a kernel thread per connection.

The following style of event-loop based handling of non-blocking sockets prevails among all kinds of servers (regardless of language):

```
while (true) {
    wait for read write or connect events
    for each ready socket
        dispatch in this or another thread (socket, event)
}

// called by dispatcher
handle(socket, event)  {
    // warning: do not block the thread
}
```

The handlers are essentially finite state machines that resolve the I/O event with the socket's state. To avoid bringing the server to a sudden halt, one must take extreme care to ensure that loops and blocking system calls finish in reasonable time. This problem tends to get especially acute in generic frameworks, as seen in early operating systems, and unfortunately in many modern-day windowing and server toolkits, where a mistake by an application programmer freezes all operations. A kernel thread dedicated to each socket is clearly much simpler to write; each thread can independently block for however long it wishes.

Improvements in kernel thread architecture have turned the tide: latency is very good, scalability is good enough and the speed of context switching and synchronization are low enough for certain categories of servers such as SMTP*, where conversations tend to be short and bursty.

At the same time, event-loop based non-blocking I/O has run into rough weather because in order to use multiple processors, the event handlers are dispatched in threads other than the event loop. There are three issues with this architecture:

First, sockets must be carefully shared between the threads. The strategy employed by the Java NIO library of protecting sockets with locks has problems of its own. If the event loop thread is blocked (on `select`), then an attempt to use a socket from a handler thread (even an attempt to close it) would block the handler thread until the selector wakes up and releases the socket's lock. This forces the programmer to ensure that all such blocking activities are handled sequentially in a single thread (typically the event loop's).

Second, all selector events are level-triggered; if the socket is ready to read, the selector keeps reporting the event until the socket has been drained fully. Since the event will be handled at some future time in a separate thread, one is forced to unsubscribe that socket from the selector to avoid needless notifications. This requires a subsequent re-subscription for the next round, which involves waking up the selector thread; a very expensive proposition.

The third problem is that the handler still does not have the luxury of blocking its thread indefinitely (because threads are considered resource intensive and are consequently pooled), which means the original complexity and inversion of control remains.

Lately, the consensus is that while event-driven I/O scores well in terms of scalability and low memory consumption, it does not on program simplicity and the other performance measures.

*See Paul Tyma's presentation on this issue http://www.mailinator.com/tymaPaulMultithreaded.pdf.

5.4.1 The Kilim I/O architecture

One contribution of the Kilim I/O implementation is that it is possible to have it all: straight-forward sequential style (seen in thread-per-socket code), latency better than blocking I/O, and the scalability and parsimony of resource usage of event-driven I/O. It is made possible through judicious use of scheduler hopping and by amortizing the cost of selector wakeup.

We first assign each socket to a `Session` actor, which in turn divides its time between two schedulers: (i) a selector scheduler that manages a select loop thread and (ii) a session scheduler that coordinates a separate pool of session executor kernel threads. When an actor's socket action (read, write, connect, accept) fails, it hops over to the selector scheduler to register the socket for the appropriate ready event, then hops back to the selector scheduler when the event is ready. By itself, this registration and context-switching strategy would be terribly expensive, so we adopt the following alternative.

When a socket action fails, the session actor yields (without changing schedulers), letting another session run for a change. When the session is eventually rescheduled, there is a chance that the socket is now ready to perform the transfer; if so, we have avoided the cost of scheduler hopping and event subscription with the selector. If instead the I/O action fails again, the cycle repeats, but not indefinitely. After a few such failed polling attempts, the session concludes that the socket is inactive after all and reverts to the original plan of moving over to the selector scheduler and registering itself with the selector for appropriate socket-ready events. Further work is required to adaptively determine the number of polling attempts and to ensure fairness, to prevent a busy connection from ceding control to other connections.

This scheme has a number of advantages. It is highly adaptive to server load, both in terms of the traffic per connection as well as the number of connections. If the number of connections increases, the interval between successive polls of a given socket increases proportionally, which in turn increases the chances that the socket is ready to read or write (as the case may be) the next time it is polled. If the traffic on a particular connection drops, it may be moved from the active polling list to the selector scheduler.

Meanwhile, the selector's work is lighter because it only manages relatively inactive sockets. This scheme is considerably better than both blocking I/O and competing non-blocking event Java frameworks in terms of performance, scalability and memory consumption (see §6.2). A socket is not registered with the selector unless necessary, which permits a socket to be configured to be blocking, which allows the use of libraries that depend on blocking I/O.

Finally, we obtain the simplicity of sequential code because the session's logic looks identical to the multi-threaded thread-per-session version outlined earlier; the only difference is the pausable annotation. That is, the readRequest() internally looks as follows:

```
@pausable
Request readRequest()  {
  while not request fully assembled
    readMore()
  return parseRequest(bytes)
}
```

This part is identical to the thread-per-request architecture. The difference is in `readMore()`:

```
@pausable
readMore() {
  while (not socket ready)
    if yielded thrice
      hop to selector and wait for socket ready event
    else
      Actor.yield()

   // at this point, socket is definitely ready with data
  readSocket()
}
```

One particularly useful aspect of having a selector scheduler manage the event loop is that it can run actors just as any other scheduler would, whether or not that actor has anything to do with a socket. Those actors wait on mailboxes, timer events and so on. In fact, one does not even need a separate session scheduler, which makes it identical to the uniprocessor event-loop code of yore. This decision can be taken at deployment time or at run-time and for selected actors.

5.4.2 Network Support: Related Work

One measure of the complexity of a solution is the number of wrappers and tutorials needed to simplify it for the "the average programmer". Java's support for non-blocking I/O has spawned a cottage industry of such wrappers. In contrast, the thread-per-connection support hardly needs further elaboration, a testament to its simplicity; much of it arising due to its sequential nature.

Few event-loop frameworks support a combination of user-level threads and non-blocking I/O. StateThreads[167] is a fast C-based tasking facility with built-in support for I/O, but as with most other tasking frameworks, cannot directly utilize multiple threads and processors. The TaskJava project[62] provides similar support in Java.

The Kilim framework's approach is unique in that I/O events and other events are given equal status; one can wait on both I/O events , messaging events and external events (UI events, for example). Scheduling is exposed to the application programmer so that any task can run on any scheduler, making possible a wide variety of mappings of tasks to schedulers, selectors, and threads.

Evaluation and Performance 6

Kilim is designed for task parallel applications that divide their time between CPU, network and disks (*split-phase*). This chapter first demonstrates Kilim's performance in isolated settings: purely CPU activities such as the speed of actor creation and messaging, then purely network I/O to add a fresh perspective to the ongoing blocking vs. non-blocking I/O debates. Next, we examine split-phase operations with more real-life constraints. We study the CPU-network interaction using the Kilim Web Server, then CPU-disk interaction using a variant of the Berkeley DB library modified to use Kilim. In all cases we compare a Kilim based solution to an industrial strength one.

6.1 In-core benchmarks

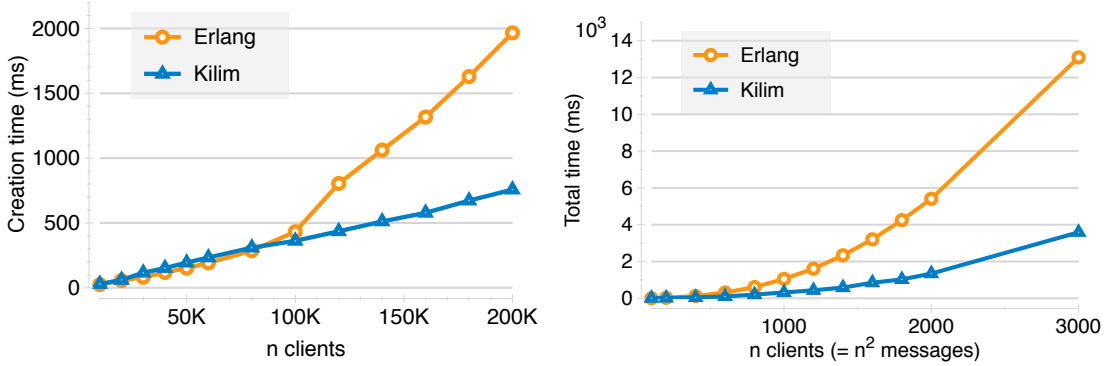
The Erlang language is often described as a concurrency-oriented language. Since it typifies the lightweight thread and messaging paradigm in server settings, we evaluated Kilim against Erlang on three of the latter's chief characteristics: ability to create many processes, speed of process creation and speed of message passing.

The in-core tests were run on a 3GHz Pentium D machine with 1GB memory, running Fedora Core 6 Linux, Erlang v. R11B-3 (running HIPE) and Java 1.6. All tests were conducted with no special command-line parameters to tweak performance. Ten samples were taken from each system, after allowing the just-in-time compilers (JITs) to *warm up*. The variance was small enough in all experiments to be effectively ignored.

Process creation The first test (Fig. 6.1(a)) measures the speed of (lightweight Erlang) process creation. The test creates n processes (actors) each of which sends a message to a central accounting process before exiting. We measure the time taken from the start to the last exit message arriving at the central process. Kilim's creation penalty is negligible (200,000 actors in 578ms, a rate of 350KHz), and scaling is linear. We were unable to determine the reason for the knee in the Erlang curve.

Messaging Performance The second test (Fig. 6.1(b)) has n actors exchanging n^2 messages with one another. This tests messaging performance and the ability to make use of multiple processing elements (cores or processors). Kilim's messaging is fast (9M+ messages at 0.54 μs per message, which includes context-switching time) and scales linearly.

The important lesson from this test is that the performance of the stack winding and unwinding logic is pretty much a non-issue for most server-side requirements.



(a) Creation and Destruction (b) Messaging

Figure 6.1: Erlang vs. Kilim times (lower is better). Kilim messaging is 3X faster.

Exploiting parallelism The dual-core Pentium platform offered no tangible improvement (a slight decrease if anything) by running more than one thread with different kinds of schedulers (all threads managed by one scheduler vs. independent schedulers). We ran the messaging performance experiment on a Sun Fire T2000 machine with 32G total memory, eight cores on one chip and four hardware threads per core. We compared the system running with one thread vs. ten. Fig. 6.2 demonstrates the improvement afforded by real parallelism. Note also that the overall performance in this case is limited by the slower CPUs running at 1.4 GHz.

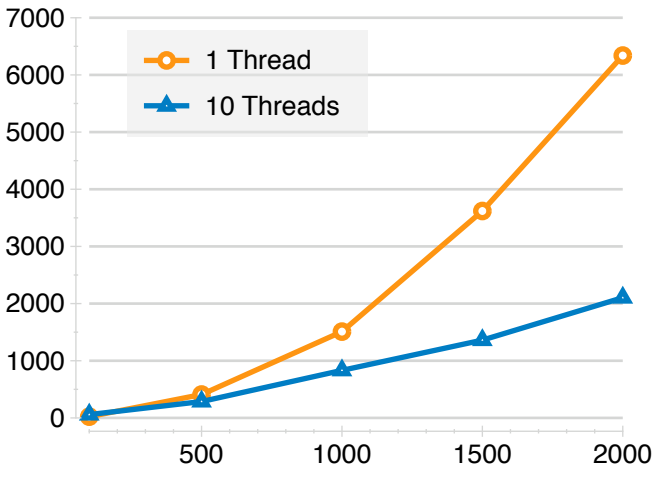


Figure 6.2: Kilim messaging performance and hardware parallelism. Lower is better.

6.2 Network benchmark: Kilim I/O vs. Thread-per-request

This section compares the performance of the Kilim I/O architecture with thread-per-request blocking I/O. We wrote a simple echo server, that echoes a 500-byte packet sent by the client. For reasons to be detailed in §6.3.2, we wrote our own load-generator, `Mob`, that can simulate thousands of TCP

and HTTP sessions. A client — one of 24,000 such concurrent sessions, running on four 16-core machines — makes 1000 requests in a closed loop fashion (waiting for the response before sending the next request). The thread-per-request model started giving connection problems beyond about 8000 simultaneous clients, hence Fig. 6.3 does not extend the comparison beyond that number; Kilim was stable until the measured limit.

The tests were run on the 585-node Darwin high performance computing cluster at the University of Cambridge*. The job coordination system on this cluster ensures exclusive control over the compute nodes and permits several instances of the Mob stress-testing client to be started concurrently.

The test environment is as follows.

Hardware	Dual socket Dell 1950 1U rack mount server
CPU	2 CPUs per node x 2 Intel Woodcrest cores per CPU = 4 cores in total per node @ 3.00 GHz per core
Primary Storage	8 GB per node (2 GB per core)
Network	Gigabit Ethernet
Operating System	ClusterVisionOS 2.1 Linux kernel 2.6.9-67.0.4.EL lustre.1.6.4.3smp x86 64
Server setup	JVM heap: 800M, TCP listen backlog: 1000. All other settings left unmodified.
Client setup	JVM heap: 800M. A maximum of 4 nodes (16 processors) simulate up to a total of 24,000 clients.

6.2.1 Results

Figure 6.3 shows two metrics. The bottom half shows completed calls per second with different numbers of concurrently connected clients. The top half represents a measure of fairness. It shows the number of in-flight client calls which, ideally, should be near the actual offered load (the number of configured clients) since this is an I/O intensive benchmark. Note that this number makes sense only when the average latency is comparable; otherwise a server could merely hold on to a pending request from every client and would be erroneously deemed fair. However, when all else is comparable, the number of active pending calls is a measure of whether the server (and the test setup) avoid convoy effects.

The two models have roughly equal throughput when the number of connected clients is low, but the thread-per-request model's fairness is poor, which means that some clients will see far worse turnaround time than others. Kilim I/O shows better and more uniform scalability while maintaining fairness. We are confident that there exist several design parameters that can improve the latency numbers for Kilim even in the lower ranges (such as adaptively tweaking the number of times we poll the socket before returning to the selector scheduler). This is left for future work.

*http://www.hpc.cam.ac.uk/darwin.html

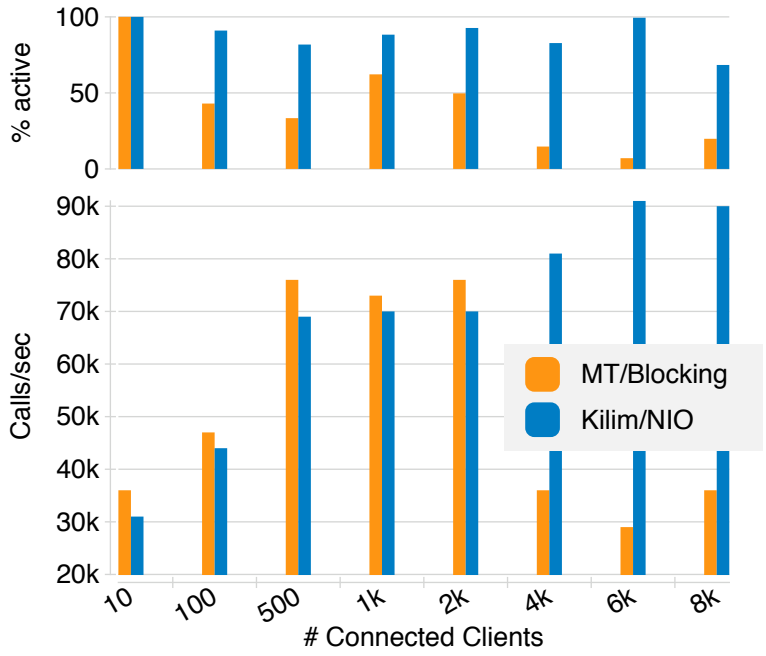


Figure 6.3: Kilim I/O vs. Thread-per-request blocking model. Higher is better.

In reality of course, a server is never this simple. The next section investigates a mixture of CPU and network activity with more realistic workloads.

6.3 CPU + Network benchmark: Kilim Web Server

As the terms "internet" and "web" become increasingly synonymous and the world prepares to consolidate services in the "cloud", servers on the internet are moving beyond serving fixed-size files using short-lived TCP connections to providing dynamically generated content (webmail and search), coordination and routing (games), streaming (live audio/video), event notification (server-push) and online interaction (AJAX applications). This immediacy of interaction between clients and servers requires us to re-evaluate traditional web infrastructure components (content-caches, proxies) and to redesign servers to simultaneously reduce latency while catering to increasing concurrent workload. In fact, as we shall soon see, even tools such as `httperf` used for stress-testing web servers need revisiting.

We modified an existing Java EE* compliant web server called TJWS, but the current API is too constraining in many ways: it is built on thread-blocking primitives, does not adequately support server-push models, a servlet is tailored to generate a response for a single request which spreads interactive conversations over several servlet instantiations, and so on.

We then wrote the Kilim Web Server (KWS), an embeddable HTTP 1.1 compliant framework, to explore the applicability of lightweight threads and message passing in the entire application and networking stack. Considering that some of the major subsystems are taken care of in the core Kilim

*Java Enterprise Edition

package itself (I/O support, kernel thread management, scheduling support), KWS is fairly lightweight (∼ 1300 lines)*, but demonstrates excellent performance and stability.

6.3.1 KWS architecture

Each client socket connection is handled by a `Session` actor. As currently written, the `Session` is responsible for reading the socket, parsing the request, assembling the body (if the transfer encoding is chunked) and decompressing the body if necessary. One can choose to either hand-over the assembled request to a separate actor, or if one knows a-priori the category of requests that are going to come over a particular port, one can have the Kilim I/O framework spawn a custom session that can handle request assembly, application specific functionality and the response as one linear chain. This pattern is common to high performance websites; they dedicate separate ports (in fact, different IP addresses) to file transfer, interactive services etc. Our test application takes this route.

The Kilim I/O support is especially suited to handling such linear flow of control. For example, it is easy to add SSL negotiation, admission control (for new socket connections) or overload control (to throttle client requests) either inline in the Session, or as a separate actor to handle just that aspect.

There are a few performance engineering aspects of the KWS architecture that deserve an aside. First, we use the Ragel state machine library † to generate fast parsers — the generated code fits compactly in a small instruction cache, and all operations are on byte arrays (no function call overhead). Second, object creation is delayed until absolutely needed. For example, an HTTP header is a UTF-8 encoded string with several key-value pairs, most of which are never examined. Instead of decoding the header and creating Java String objects for all the components of the header, KWS merely records the offset and length pairs (occupying the top and bottom halves of a 32-bit integer). Third, KWS uses the Kilim non-blocking I/O library mentioned earlier. This is not only considerably faster than a kernel-thread-per-request and blocking I/O, it is parsimonious with memory (at 25,000 connections, the resident memory used is 768M in the application described below). Fourth, all file transfers are done using Java support for `sendfile`, where available. When dealing with up to 90,000 requests/second (Fig. 6.3), these optimizations add up.

6.3.2 KWS application: auto-completion

Static web server performance has been well-studied and several specialized and extremely lightweight file servers have been developed.[141] The purpose of the following benchmark is to investigate the serving of short, dynamic content in an interactive context, as seen in server-side auto-completion on search bars and on forms, game and chat servers, etc. Messages are asynchronous (overlaid on the HTTP request/response paradigm), very short (a few hundred bytes, in contrast to dynamic content from news portals) and require low latency for online response, etc.

The sample benchmark application is a simple word completion web service that, given a prefix string, binary-searches a list of 400,000 words and returns up to ten words that complete the prefix string. Each of (up to 24,000) clients makes a connection, and repeatedly requests words with no intervening think time; it is an exercise in studying response rates and overload handling. Unfortunately,

*There is no support yet for SSL, cookie generation and session stickiness.

†http://www.complang.org/ragel/

none of the existing load generators such as `ab`, `httperf`[128] and `Siege`[160] addressed our requirements satisfactorily. For example, in spite of specifying 2000 sessions to be started, the maximum number of sockets opened with `httperf` would be of the order of 700. The second factor is that these tools report the typical statistical measures – average, standard deviation, and median – none of which are really useful in practice. Most high-performance commercial websites use the 99.9^{th} percentile of latency measures and other metrics, in a bid to address the worst case experience.[54] The third factor leading to writing our own stress-testing client, called `Mob`, was that we wanted access to the raw call-level data for subsequent analysis; for one, this makes it possible to aggregate statistics from separate nodes.

The benchmark compared the performance of KWS with the popular (and beautifully engineered) Jetty Web Server .[97] This choice was dictated by the fact that Jetty is written in Java and is modular, allowing one to use only the core module without the heavyweight Java EE superstructure getting in the way. The feature set of Kilim and Jetty used by this test is roughly the same.

Each instance of `Mob` starts up to 6000 sessions (TCP connections), each making 1000 HTTP GET requests at a specified rate (300-1000 req/s). Raw data for each call (begin and end times, bytes transferred, number of errors etc.) is gathered during the middle third of each session, to ensure that all clients and servers are in a steady state. The request-rate parameter was ratcheted down from 1000 requests/sec to 300, until the server could repeatedly sustain the given load and number of sessions without giving any connection errors.

6.3.3 Results

In our test, it took 25,000 connections to finally saturate the 4 cores of the server (with Kilim). It goes without saying that this kind of a load is never delivered to a single server in practice (perhaps only in a denial of service attack); for one, any business fortunate enough to get 25,000 concurrent clients would scarcely rest its fortunes on the vagaries of a single machine. However, it does mean that significant loads can be handled by considerably less powerful machines, which is important as we enter the era of counting watts per transaction.

As before, we combine server throughput (requests completed per second) and a fairness metric (number of clients currently active in a call). We could not get Jetty to reliably scale beyond about 5000 concurrent sessions without getting connection timeouts (there is no overload or admission control in the core module). KWS sustains the maximum offered request rate all the way up to 24,000 clients, when the server CPUs (all 4 cores) are saturated. In addition to delivering significantly better performance, KWS shows close to 100% level of fairness in keeping its client occupied, as the top half of figure 6.4 demonstrates.

Figure 6.5 shows fairness in a different way. In a particular run with 24,000 concurrent client sessions, the histogram classifies requests by round-trip time (a total of 12 million calls). Notice that more than 95% of the clients show a steady latency of 400-450ms at full load (with the clients taking no think time). It shows that the server is fair to all connected clients, and such steadiness allows one to make reliable UI decisions.

Remark. In practice, data centers focus on two somewhat conflicting requirements: minimizing latency and providing redundancy. When a primary machine fails and the backup takes over, the only way to ensure that the 99.9^{th} percentile of latency is within acceptable bounds is to make sure there is enough CPU and network headroom on all machines. Although utilization of a machine or the

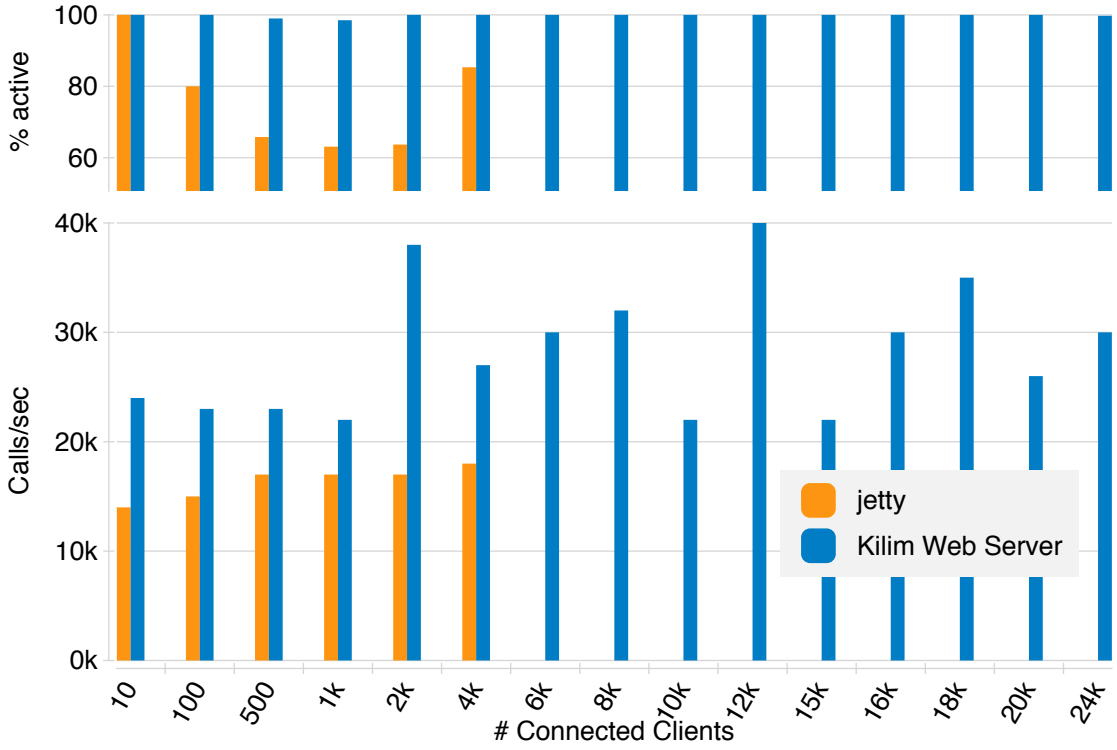


Figure 6.4: Auto-Completion service: Kilim Web Server vs. Jetty. Higher is better.

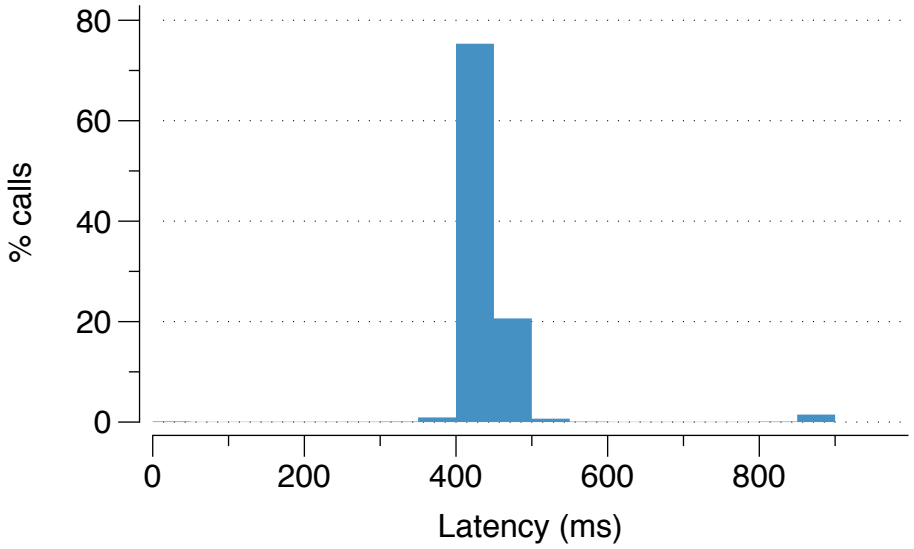


Figure 6.5: Kilim latency distribution at 24,000 clients. Almost all clients get a response between 400-500 ms.

network is not an interesting measure, our tests show that it is possible to handle a lot more clients with less (or less powerful) hardware.

6.4 CPU + Disk: Berkeley DB

Performance has become such an important differentiating factor that an increasing amount of data has been steadily migrated out of relational databases and into much simpler key-value schemes such as Google's `BigTable` . Underneath most of these lies the `Berkeley DB` (BDB) library, an embeddable, transactional key-value storage engine and one of the most popular storage managers in use today.

6.4.1 Berkeley DB: original architecture

Most such storage engines (including the version of BDB written in C) employ a B-tree to couple in-memory data structures to disk blocks. An update to the in-memory structure is mirrored by a corresponding in-place disk write. If the update is made under the aegis of a transaction, the database first writes the data to a write-ahead log and locks the corresponding records, and upon commit, the data is copied to the corresponding disk block in the data file.

The architecture of the Java version of BDB is unique among storage engines (it bears scant resemblance to even the C version). It eschews in-place update of disk blocks for a single append-only log-structured disk representation containing *all* data: application data, database meta-data and the transactional log. It is still a B-tree scheme, but without in-place disk updates. This design not only avoids separate disk writes of log and content data, the append-only strategy avoids random seeks, yielding excellent insert and write performance. As it turns out, the code has been so tightly written that for most typical workloads, it is extremely hard to force it to stress the I/O system much.

The downside is that the shared structure necessitates an elaborate latching and locking scheme. Latches are short-term reader-writer locks meant to protect shared data structures from concurrent updates, while the term "lock" is a read-write lock that protects application data until the completion of its owning transaction. To provide high concurrency, every B-tree node has its own latch. A root-to-leaf traversal involves latch-coupling: latch a node, then latch the appropriate child node, then let go of the parent latch. An insertion into a full leaf triggers node-splits from the leaf node all the way to the top, which means the latching order is bottom up, a situation primed for deadlock. There are still other areas where several updates have to happen concurrently (updates to lock manager, committing changes to B-tree nodes etc.), all of which require extensive use of Java monitors.

6.4.2 Berkeley DB: Modifications

Our original goal was to write a key-value store (using cache-adaptive log structured merge trees[135]) with an in-memory non-sharing model based on Kilim actors that vertically partition the key space amongst available processors. The difference from parallel database and distributed consistent-hashing approaches is that, while "locally distributed", the solution would still use shared memory to balance the assignment of partitions to processors (a simple non-blocking linearizable operation). Although much progress was made on this database and several fruitful discussions were held with the authors

of Berkeley DB and Facebook, the project has been temporarily set aside because a fair comparison to an existing product entails building a complete product in short order.

This section reports on a more modest experiment based on the following observations. The OS does not know about transactions or read-write locks or the fact that earlier transactions should be given priority to later ones (the idea being that a later transaction, having accomplished less, should be the one to block or rollback). Further, preemptive thread scheduling leads to split locks and priority inversion if a thread is de-scheduled while in possession of a logical lock. The odds of collision and priority inversion can only increase (exponentially, in the worst case) with an increase in the number of threads and hotspots.

We modified BDB (the Java version) to represent each application transaction as an actor; the original uses a thread-per-transaction model. The actors are coordinated using mailboxes instead of blocking the thread as traditional read-write locks are wont to do. Most of the existing structure of the code is left unmodified (latching, lock management, recovery, etc.) The idea behind the modification was the intuition that it should pay to dissociate hundreds of thousands of logical threads of control from actual threads, and to control their scheduling based on a user-level transaction-savvy scheduler. This way, most logical transactions can be taken through to completion instead of attempting to be fair. Also, because we create only as many kernel threads as processor cores (§5.3), thereby treating the kernel thread as a virtual processor, there are far fewer threads that are actually contending for a lock.

Before reporting on the performance results, here is a brief look at what the modifications to BDB entailed. About a fourth of the 5000 methods had to be annotated as `@pausable` (with a concomitant 18% increase in the size of the generated code). Several pre-existing classes (such as I/O streams and locks) that are wont to block the thread were replaced with pausable variants. All daemon threads (responsible for log cleanup, log writing etc.) were converted to actors. The biggest trouble came from an extensive use of synchronized blocks, which had to be replaced with an explicit acquire/release pair of calls; in addition, because Java locks are owned by the thread, we had to ensure that the actors were rescheduled in the same thread where they had acquired the lock. The scheduler used a priority queue (ordered by transaction start time) instead of a first-come first-served order. Given the usage pattern of key-value stores in many modern applications, this is a viable approach.

6.4.3 Berkeley DB: Benchmark and Results

The workload consists of n logical threads that attempt to get or update a random key (in an 80/20 ratio respectively). The BDB infrastructure's caching schemes are finely tuned and highly adaptive and most server-class machines are big enough to store the entire working set of accessed keys in memory. This means that most of the disk activity is due to the relatively infrequent updates. In other words, lock collisions and thread scheduling play a much higher part in the performance numbers than does disk I/O.

We measured the elapsed time for 1000 get/put calls made by each client on a database of size 100,000, averaged over five runs for each concurrent client count. The times were measured on an Apple MacBook Pro (dual-core 2.4GHz processor, 2G memory, 3MB L2 Cache, Mac OS X v.10.5). Figure 6.6 shows the inverse number: the number of requests handled (in thousands per second). In spite of the somewhat un-aesthetic modifications to the Berkeley DB core, the Kilim variant of BDB showed three improvements:

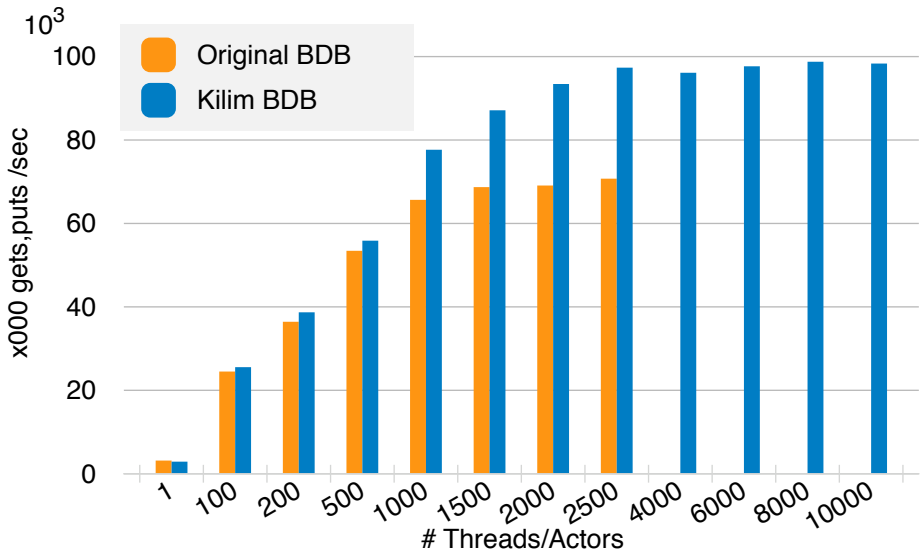


Figure 6.6: Original vs Kilim-based Berkeley DB. Higher is better.

Throughput. Kilim-BDB shows up to a 30% increase in throughput, then levels off due to CPU saturation. Note that the original threaded code starts the leveling off much earlier.

Scalability. Adding more actors is merely an exercise in adding objects to the heap and to the scheduler's run-queue and because of the run-to-completion property, it stands to reason that the Kilim variant can easily accommodate a few tens of thousands more threads of control; we expect this number to be limited only by application session and transaction timeouts.

Memory usage. The resident memory usage of the Kilim variant was routinely less than *half* of the thread version. What was surprising, however, is that Kilim-BDB had marginally more overall (4%) CPU usage than the original.

6.5 Qualitative Evaluation: Kilim in practice

The Kilim toolkit has been downloaded tens of thousands of times and has seen warm acceptance from Java server developers, particularly from the financial industry. At this early stage of adoption, there are, as yet, no published experience papers, so what follows is a brief, anecdotal mention of four different *types* of usage.

Financial Analytics

One London-based financial analysis firm is using Kilim to run thousands of *analytics* based on real-time data feeds; each analytic instance subscribes to particular stock market events and trends and feeds other analysis tasks. Many of these analytic applications are linear in that they wait for several events "inline", then proceed to the next stage of computation; the sequential nature of a Kilim actor is considerably less convoluted than a callback-driven piece of code.

Server infrastructure

A very fast web server, `asynchttp`*, based on the Kilim core framework, was written independently by Jan Kotek. Our adaptive I/O infrastructure was inspired by one key demonstration of the `asynchttpd` project, that there is a considerable difference in performance between polling all sockets in a round-robin fashion versus registering them with a selector for socket-ready events. The problem with this approach is that it sends the CPU into a looping frenzy if none of the sockets are active enough. We added support for detecting inactive sockets, by hopping over to the selector scheduler after a few failed polling attempts.

Actor Frameworks

The `Actor Foundry`† project at the University of Illinois at Urbana Champaign provides a purer actor-oriented programming model than Kilim: its objective is to treat everything as actors (the console, for example, is an actor to which you send a print message). The infrastructure uses the Kilim weaver for CPS transformation, but provides its own runtime infrastructure, including components for messaging, actors and schedulers. Unlike Kilim, they copy messages for safety. The project is currently exploring various issues related to fair scheduling.

Orchestration

Actor networks, workflow networks and dataflow networks have one idea in common: components have input and output ports and are wired together, often visually. The `Orc` project at the University of Texas at Austin provides a language to do this wiring. This project's focus is assembling data flows from distributed computations while taking into account fault-tolerance and wide-area communication latency. The language itself provides just three simple, but powerful operators: parallel computation, sequencing, and selective pruning, which are general enough to express all orchestration requirements.[126]

The `Orc` project uses Kilim for its threading transformation, to scale up to thousands of ongoing orchestrated activities. The Kilim programming model allows the programmer to wait for an event in the middle of a `for` loop.

*`http://kotek.net/asynchttpd`

†`http://osl.cs.uiuc.edu/af/`

7 Conclusions and Future Work

Server-side programmers face a daunting array of objectives and constraints, often at odds with each other: quality of service (low latency and latency jitter, reliability, overload control), effective use of resources (CMP/SMP boxes, multiple data centers, throughput, energy consumption), ease of development, security, online diagnosis, coexistence in a multi-language, multi-technology environment, and so on. These issues directly translate into a large number of choices in software architectures for the server-side developer.

This dissertation builds on the observation that a single server is typically deployed as *part* of a distributed setup, as well as *on* a distributed setup. In other words, a server is a part of a fault-tolerant, load-sharing cluster of servers, and separately, that the hardware platform on which a server is deployed, an SMP machine, makes no attempt to hide the fact that it is a distributed system. We propose that a distributed architecture at the intra-process level is one that seamlessly binds these two aspects. This makes the overall architecture somewhat self-similar, and solutions at the intra-process level can draw upon parallels from more macro levels.

Specifically, we propose a communicating actor architecture, where actors are heap-isolated state machines with private stacks, and communicate using explicit messaging protocols. In order for this architecture to be practical, we argue that the programming language and environment must support a clutch of architectural features: (i) Threads lightweight enough to comfortably exceed the number of concurrent users and tasks, (ii) memory isolation between actors, to promote lock-free programming and to eliminate having to think about memory consistency models, (iii) the ability to pull external events into the actor's space (the opposite of reactive programming) (iv) customizable, user-extensible synchronization and scheduling constructs.

Our contribution is in delivering this vision portably (without changes to syntax or to the underlying virtual machine) in a mainstream programming language such as Java, and without sacrificing the familiar idiomatic style of the language (sequential control flow coupled with pointers and mutable objects). This work is available as an open-source toolkit called Kilim.[103]

Kilim provides very lightweight and fast preemptible user-level threads, which promote linear, imperative control flow that is simple to understand and is composable, unlike a reactive inversion-of-control style fostered by callback-oriented code. The speed of context-switching and the memory footprint of Kilim actors compare very favorably to mainstream concurrent languages.

The Kilim type system statically guarantees that actors are memory-isolated by separating messages and internal objects, and by controlling pointer heap aliasing in messages. This is done by ensuring that each object in a message can have at most one other object — either an actor or another message object — point to it (and at most via a single field). This constraint promotes both linear ownership as well as a tree structure. The single ownership of messages and confinement of all other non-message objects fosters a programming model without locks and without fear of external modification. The

inductive tree structure of messages permits fast serialization and simple resource management (one can trivially recycle a buffer by separating it from its current parent, secure in the knowledge that no other object is pointing to it).

Finally, the Kilim toolkit allows messaging, synchronization and scheduling constructs to be user-customizable and extensible. The toolkit approach differs from existing concurrent languages with built-in fixed sets of primitives, and frameworks (such as Java Enterprise Edition and CORBA) that impose a fixed programming style. While the threading, type-system and run-time library aspects are largely orthogonal, we make the case that they reinforce each other's strengths when provided together. For example, one can build a Haskell-style `MVar` or a swap primitive while relying on the type system to control pointer aliasing.

7.1 Future Work

The Kilim framework is but a first step in demonstrating the viability of the communicating actor approach in a server environment. We will outline a few avenues of research and implementation, some that extend the Kilim framework as it exists, some that explore fresh implementations that preserve some or all of the original aims (portability, imperative control flow, pointers and mutable state).

Actor-Oriented JVM It is possible to improve Kilim's context-switching speeds by two orders of magnitude, by integrating actor support in the JVM. Unlike traditional coroutine support and green-thread implementations of earlier JVMs, the scheduling aspect must be user-customizable, and schedulers should be able to signal preemption. One challenge is in sizing the stack dynamically, should one opt for a stack-centric approach (in contrast to a CPS approach, where the call chain is in the heap).

A JVM's knowledge of actors and type-directed memory confinement simplifies garbage collection tremendously. It also permits better memory layout for both the internal state of an actor as well as for messages, and different garbage collection strategies for them. A VM's ability to make available actor-specific profiling and accounting metrics to the user-level would foster the creation of adaptive schedulers.

Distribution support The one-way "send-and-pray" semantics of asynchronous messaging presents a uniform veneer over collocated and remote targets. However, failure semantics are considerably different, and latency differences are considerable. That said, given the vastly improved speeds of networks and infrastructure reliability, it is worth exploring reliable facilities (such as lease-based garbage collection) to share resources such as mailboxes, for higher-order synchronization objects,[4] for well specified interface automata,[43,44,171] and for accounting of finite resources such as security capabilities. One useful optimization may be to pay particular attention to objects that are just shared between pairs of processes, because many workflows are linear or fork-join.

Orchestration As scripting approaches such as Kamaelia,[102] orchestration languages such as Orc[126] and dependency injection frameworks such as `guice`[79] have shown, there is much value to dynamic binding of components. When building a component, there should be no a priori knowledge of which other component it will connect to, or the rate of message arrivals. These parameters may be supplied at deployment time and may inform the creation of specialized, deterministic scheduling policies.

Syntactic improvements Languages such as Scala and F# have shown an elegant upgrade path for mainstream languages; in particular, the availability of higher-order functions, Smalltalk-style block closures, and pattern-matching is extremely attractive. There is much scope for providing other essential features we have mentioned earlier, such as statically guaranteed memory isolation and systemic knowledge of actor-scheduling and of actor boundaries in debuggers, profilers and so on.

Scheduling on a hierarchy of processors Many scheduling approaches recognize a hierarchy of schedulable objects, but not always a hierarchy of processing elements. It would be useful to develop a system for an entire cluster, and have a uniform way of integrating scheduling and placement aspects at all levels down to a particular core on a particular machine, especially in conjunction with orchestration approaches. It does not necessarily have to entail actor migration (due to scheduler hopping).

Interruption and Cancellation semantics A principled study of the value of different interruption approaches is required. The POSIX-style signal handling approach has the same problems of multi-threaded access to shared data, while passive notifications (sending a message to a specially marked "control" mailbox, for example) have the problem of leaving it to the programmer to explicitly poll for them. The second problem is how to react to such notifications, and the manner in which the signal handler can influence further processing; should it kill the actor or raise an exception? One promising avenue is to use the exception handling syntax to install lexically and dynamically scoped signal handlers, and further, to add resumption semantics for the flow of control to resume from where the exception was thrown.

Higher-level threads of control An application or workflow has several hierarchical threads of control and nested contexts, which is reflected in the way components and systems are connected and in the way in which data moves. For example, the user-level process of buying a widget may manifest itself as several TCP connections and short-term sessions. Much information can be gleaned from tracking such higher-level workflows, from profiling to denial of service attacks. One option may be for messages to carry around tokens representing hierarchies of contexts (on behalf of which work is being performed), and if actors can be made to attach those tokens (or generate dependent tokens) to outgoing messages, then a dynamic taint analysis can yield considerable information. These mechanisms could be further integrated with run-time instrumentation approaches such as dtrace[40] and profiling frameworks such as ARM (Application Response Measurement).[11]

Automated Reasoning JML[116] tools such as ESC/Java[50] attempt to find common run-timer errors using static analysis, with the help of theorem provers. The non-concurrent nature of an individual actor lends itself easily to such tools.

Security and Information flow We have only paid attention to data flow with the tacit understanding that the possession of a non-forgeable object identifier is equivalent to possessing a capability. We have, in related work,[59] explored the use of the type system to constrain the ownership and duplication of capabilities. A type-systemic approach needs a robust runtime component that prevents forgeries and permits revocation, especially in a distributed system. The E language and environment have much to teach us in this regard.

We currently treat data as just bits, not as *information* with differing levels of importance, secrecy and so on. Moving to an information-savvy system requires tracking control flow to prevent computations on low-valued information affecting higher-valued information.[98, 130]

Integrating operating systems and servers We believe that the structure of modern operating sys-

tems emphasizes a rigid and complicated division between user and kernel space, and that much of it is accidental complexity attributable to the use of C as a systems programming language. As the Singularity project has shown, given strongly typed languages and static verification schemes such as proof-carrying code, it is possible to treat the kernel as a collection of peer actors sharing the same physical address space and hardware protection level as any application object; mechanisms such as device-driver interrupts and scheduler activations are merely messages from one part of the system to another, and fault-isolation is a natural byproduct of an actor mindset. We intend to investigate type-directed, finer-grained isolation in the Barrelfish[22] operating system.

8 Notation and Nomenclature

$e \bullet S$	Add e to set S ($\{e\} \cup S$). Also used as a pattern match operator.	
$\triangleq$	Defined as	
$*$	Wildcard, dummy, place-holder	
$\overline{x}$	List of x	
$\overline{q\tau}$	List of pairs: $\langle q_0 \tau_0 \rangle, \langle q_1 \tau_1 \rangle, \ldots, \langle q_n \tau_n \rangle$	
$\langle \ldots \rangle$	An n-tuple	
$a \mapsto b$	Map $\triangleq \lambda x. \begin{cases} b & \text{if } x = a \\ undef & \text{otherwise} \end{cases}$	
$\mu[a \mapsto b]$	Extension of any map μ : $\lambda x. \begin{cases} b & \text{if } x = a \\ \mu(x) & \text{otherwise} \end{cases}$	
$\rightsquigarrow_\delta$	Concrete path (reachability)	pg. 59
$\rightsquigarrow_\Delta$	Abstract path	pg. 59
δ	Run-time function state (activation frame + typestate map)	Def. 4.2, pg. 57
Δ	Abstract function state corresponding to δ	Def. 4.3, pg. 57
$\Delta', \hat{\Delta}$	Abstract post-state	Fig. 4.7, pg. 68
β	Abstraction function. Converts $\delta = \langle \sigma, H, s \rangle$ to $\Delta = \langle N, E, \Sigma \rangle$	Def. 4.5, pg. 58
ρ	Object reference in the concrete state	
τ	Meta-variable ranges over types	Fig. 3.1, pg. 40
σ	Local register file. Maps var names to a value	Fig. 3.2, pg. 42
ς	Concrete object's typestate: $\rho \mapsto \{\texttt{null}, \mathbf{f}, \mathbf{w}, \mathbf{r}, \top\}$	§4.3.1, pg. 54
Σ	Abstract node's typestate: $n \mapsto \pi$, where $\pi \in \{\texttt{null}, \mathbf{f}, \mathbf{w}, \mathbf{r}, \top\}$	Def. 4.3, pg. 57
$\Sigma', \hat{\Sigma}$	Abstract post-state	Fig. 4.7, pg. 68
Γ	Phase-1 Type environment	§4.2, pg. 52
A	Actor	Fig. 3.2, pg. 42
b	Mailbox ids	Fig. 3.1, pg. 40
cluster	Cluster of objects reachable from object	Fig. 3.3, pg. 43
scluster	Shape cluster. Compatible nodes reachable from a given node	Def. 4.7, pg. 60
Es	Set of abstract edges	Def. 4.3, pg. 57
$Es', \hat{Es}$	Set of edges in post-state (corresponding to *Es*)	Fig. 4.7, pg. 68
$\mathbf{f}$	Free (unowned, writable). See ς	

f, g	Field names	Fig. 3.1, pg. 40
H	Heap of an actor	Fig. 3.2, pg. 42
lb	Label for jump instructions	Fig. 3.1, pg. 40
$\mathscr{L}$	Labeling function	Def. 4.4, pg. 58
M	Meta-variable for message structures names	Fig. 3.1, pg. 40
n_X	Abstract node with label X. A label represents a set of variable names	pg. 56
$n_\emptyset$	Abstract node with an empty label	pg. 56
Ns	Set of abstract nodes	Def. 4.3, pg. 57
$Ns', \hat{Ns}$	Set of abstract nodes in post-state	Fig. 4.7, pg. 68
P	Post Office	Fig. 3.2, pg. 42
q	Type qualifier	Fig. 3.1, pg. 40
r	Readonly (not writable; may or may not be owned). See ς	
s	Ranges over statements	Fig. 3.4, pg. 44
S	Call stack	Fig. 3.2, pg. 42
v, x, y	Variable names	Fig. 3.1, pg. 40
w	Writable and Owned. See ς	

9 Bibliography

[1] ActorFoundry. http://osl.cs.uiuc.edu/af/. [p. 83]

[2] 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*, pages 289–302, 2002. [pp. 15, 83, 86, and 87]

[3] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987. [p. 14]

[4] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007. [p. 110]

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. [p. 53]

[6] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 311–330, 2002. [p. 79]

[7] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 32–59. Springer-Verlag, 1997. [p. 77]

[8] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19). [p. 75]

[9] Anderson, Bershad, Lazowska, and Levy. Scheduler activations: Kernel support for effective user-level thread management. Technical Report 90-04-02, University of Washington, 1990. [pp. 29 and 83]

[10] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1991. [p. 82]

[11] Application response measurement (ARM). http://www.opengroup.org/management/arm. [p. 111]

[12] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology, Stockholm, 2003. [p. 37]

[13] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. [pp. 13 and 34]

[14] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for Java. In *Programming Language Design and Implementation (PLDI)*, pages 258–268, 1998. [p. 36]

[15] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of Java without data races. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 382–400, 2000. [p. 77]

[16] Jean Bacon. *Concurrent Systems*. Addison Wesley, 1998. [p. 34]

[17] Henry G. Baker. Unify and conquer. In *Conference on LISP and Functional Programming*, pages 218–226, 1990. [pp. 76 and 77]

[18] Henry G. Baker. 'Use-once' variables and linear objects - storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, 1995. [p. 76]

[19] Fred Barnes and Prof. Peter H. Welch. Communicating Mobile Processes. In Dr. Ian R. East, Prof David Duce, Dr Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 201–218, 2001.

[20] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003. [p. 16]

[21] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. JiST: an efficient approach to simulation using virtual machines. *Software—Practice and Experience*, 35(6):539–576, 2005. [p. 93]

[22] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Symposium on Operating Systems Principles (SOSP)*. ACM Press, October 2009. [pp. 37 and 112]

[23] Andrew Baumann, Simon Peter, Adrian Schuepbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Hot Topics in Operating Systems (HotOS)*, May 2009. [p. 32]

[24] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. Technical Report 90-05-07, University of Washington, 1990. [p. 15]

[25] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, July 1995. [pp. 87 and 92]

[26] Hans-Juergen Boehm. Threads cannot be implemented as a library. In *Programming Language Design and Implementation (PLDI)*, pages 261–268, 2005. [p. 22]

[27] Hans-Juergen Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Programming Language Design and Implementation (PLDI)*, pages 68–78. ACM, 2008. [p. 24]

[28] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 35–46, 1999. [p. 36]

[29] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007. [p. 31]

[30] Frederic Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, April 2006. [p. 93]

[31] Chandrasekhar Boyapati. *SafeJava: a unified type system for safe programming*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2003. [p. 78]

[32] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001. [pp. 74 and 77]

[33] John Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, 2006. [pp. 27 and 75]

[34] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 2–27, 2001. [p. 79]

[35] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998. [pp. 32 and 34]

[36] Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995. [p. 21]

[37] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Programming Language Design and Implementation (PLDI)*, pages 99–107, 1996. [p. 82]

[38] Alan Burns and Andy Wellings. *Concurrent and Real-Time Programming in Ada*. Cambridge University Press, 2007. [p. 13]

[39] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Communications of the ACM*, 51(11):34–39, 2008. [p. 35]

[40] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28. USENIX, 2004. [p. 111]

[41] Richard Carlsson, Konstantinos F. Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent languages. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 73–90, 2003. [p. 37]

[42] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6(5):46–58, 2008. [p. 36]

[43] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *Programming Language Design and Implementation (PLDI)*, pages 237–248, 1996. [p. 110]

[44] Prakash Chandrasekaran, Christopher L. Conway, Joseph M. Joy, and Sriram K. Rajamani. Programming asynchronous layers with CLARITY. In Ivica Crnkovic and Antonia Bertolino, editors, *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 65–74. ACM, 2007. [pp. 15 and 110]

[45] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Compiler Construction*, 2007. [p. 75]

[46] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 1–19, 1999. [p. 75]

[47] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2001. [p. 78]

[48] Dave Clarke and Tobias Wrigstad. External Uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 176–200, 2003. [p. 78]

[49] David G. Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 48–64, 1998. [p. 78]

[50] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413. [p. 111]

[51] Melvin Conway. Design of a separable transition-diagram compiler. In *Communications of the ACM*, 1963. [p. 12]

[52] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, pages 262–275. ACM, 1999. [p. 79]

[53] Grzegorz Czajkowski and Laurent Daynès. Multitasking without compromise: A virtual machine evolution. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 125–138, 2001. [p. 37]

[54] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating Systems Principles (SOSP)*, 2007. [pp. 13 and 102]

[55] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, pages 59–69, 2001. [p. 80]

[56] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005. [p. 78]

[57] ejabberd: Erlang Jabber/XMPP daemon. http://ejabberd.jabber.ru. [p. 38]

[58] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. Technical report UCAM-CL-TR-578, University of Cambridge, Computer Laboratory„ January 2004. [p. 76]

[59] David M. Eyers, Sriram Srinivasan, Ken Moody, and Jean Bacon. Compile-time enforcement of dynamic security policies. In *9th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 119–126. IEEE Computer Society, 2008. [pp. 53, 67, 73, and 111]

[60] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proc. of EuroSys*, 2006. [pp. 13, 16, and 80]

[61] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Programming Language Design and Implementation (PLDI)*, pages 13–24, 2002. [p. 80]

[62] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Partial Evaluation and Program Manipulation (PEPM)*, 2007. [p. 96]

[63] Bryan Ford and Sai Susarla. CPU inheritance scheduling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996. [p. 93]

[64] Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-sensitive type qualifiers. In *Programming Language Design and Implementation (PLDI)*, pages 1–12, 2002. [p. 57]

[65] Fraser and Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25, 2007. [p. 35]

[66] Stefan Fünfrocken. Transparent migration of Java-based mobile agents - capturing and reestablishing the state of Java programs. In *Mobile Agents*, pages 26–37. Springer-Verlag, 1998. [p. 83]

[67] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming (ICFP)*, pages 18–27. ACM, 1999. [p. 15]

[68] David Gay, Philip Levis, and David E. Culler. Software design patterns for TinyOS. *ACM Transactions of Embedded Computer Systems*, 6(4), 2007. [p. 34]

[69] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, pages 1–11, 2003. [p. 34]

[70] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *Programming Languages and Systems, European Symposium on Programming*, pages 74–90, 1999. [p. 38]

[71] Narain H. Gehani and William D. Roome. Concurrent C. *Software—Practice and Experience*, 16(9):821–844, September 1986. [p. 13]

[72] Anders Gidenstam and Marina Papatriantafilou. LFthreads: A lock-free thread library. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *Principles of Distributed Systems (OPODIS)*, volume 4878 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 2007. [p. 36]

[73] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. [p. 76]

[74] Google Protocol Buffers. http://code.google.com/apis/protocolbuffers/. [p. 16]

[75] Minor Gordon. *Stage scheduling for CPU-intensive servers*. PhD thesis, University of Cambridge, 2008. [p. 92]

[76] Ralph E. Griswold. Programming with generators. *The Computer Journal*, 31(3), 1988. [p. 82]

[77] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. *Programming Language Design and Implementation (PLDI)*, 37(5):282–293, May 2002. [p. 77]

[78] Rachid Guerraoui. *Introduction to Reliable Distributed Programming*. Springer, 2006. [p. 31]

[79] Guice. http://code.google.com/p/google-guice/. [p. 110]

[80] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *ACM SIGSOFT international symposium on Foundations of software engineering*, pages 69–80. ACM, 2006. [p. 74]

[81] Philipp Haller and Martin Odersky. Event-Based Programming without Inversion of Control. In *Proc. Joint Modular Languages Conference*, LNCS, 2006. [p. 29]

[82] Philipp Haller and Martin Odersky. Capabilities for External Uniqueness. Technical report, EPFL, 2009. [p. 80]

[83] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977. [pp. 13 and 35]

[84] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, 1999. [pp. 23 and 35]

[85] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 388–402, 2003. [p. 36]

[86] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Communications of the ACM*, 51(8):91–100, 2008. [p. 36]

[87] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977. [pp. 12, 14, and 33]

[88] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 54–61. ACM, 2001. [p. 75]

[89] Hirzel, Von Dincklage, Diwan, and Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29, 2007. [p. 75]

[90] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969. [p. 76]

[91] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. [pp. 12, 15, and 89]

[92] John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 271–285, 1991. [p. 77]

[93] John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard C. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992. [pp. 47 and 74]

[94] Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003. [p. 12]

[95] Blunt Jackson. NPTL: The New Implementation of Threads for Linux. *Dr. Dobb's Journal*, Aug 2005. [p. 88]

[96] JavaFlow: Apache Commons project for Java continuations. `http://jakarta.apache.org/commons/sandbox/javaflow`. [pp. 83, 87, and 88]

[97] Jetty webserver. `http://www.mortbay.org/jetty/`. [p. 102]

[98] Jif: Java + information flow. `http://www.cs.cornell.edu/jif/`. [p. 111]

[99] Erik Johansson, Konstantinos F. Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Proc. of The Workshop on Memory Systems Performance and The International Symposium on Memory Management (MSP/ISMM)*, pages 195–206, 2002. [p. 37]

[100] JSR 133. Java memory model and thread specification. `http://jcp.org/en/jsr/detail?id=133`. [p. 24]

[101] JSR-308 Annotations on Java Types. `http://jcp.org/en/jsr/detail?id=308`. [p. 75]

[102] Kamaelia: Concurrency made useful, fun. `http://www.kamaelia.org/Home`. [pp. 34 and 110]

[103] Kilim. `http://kilim.malhar.net`. [pp. 14 and 109]

[104] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006. [p. 53]

[105] Naoki Kobayashi. Quasi-linear types. In *Principles of Programming Languages (POPL)*, pages 29–42, 1999. [p. 76]

[106] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000. [pp. 31 and 90]

[107] Maxwell N. Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *USENIX Annual Technical Conference*, pages 87–100, 2007. [pp. 14 and 23]

[108] Eugene Kuleshov. Using ASM framework to implement common bytecode transformation patterns. In *Conference on Aspect-Oriented Software Development*. Springer-Verlag, 2007. [p. 88]

[109] Yves Lafont. Introduction to linear logic. In *Lecture notes for the Summer School on Constructive Logics and Category Theory*, August 1988. [p. 76]

[110] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979. [p. 24]

[111] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980. [pp. 13, 34, and 35]

[112] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. [p. 82]

[113] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In USENIX, editor, *USENIX Annual Technical Conference*. USENIX, 2002. [p. 92]

[114] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.

[115] Doug Lea. A Java fork/join framework. In *Java Grande*, pages 36–43, 2000. [p. 87]

[116] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005. [p. 111]

[117] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006. [pp. 12, 14, 22, 30, and 32]

[118] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 2003. [pp. 52, 54, and 88]

[119] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999. [p. 39]

[120] Anil Madhavapeddy. *Creating High-Performance, Statically Type-Safe Network Applications*. PhD thesis, University of Cambridge, 2007. [p. 15]

[121] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. *Principles of Programming Languages (POPL)*, 40(1), January 2005. [p. 24]

[122] Mark Marron, Deepak Kapur, Darko Stefanovic, and Manuel V. Hermenegildo. A static heap analysis for shape and connectivity: Unified memory analysis: The base framework. In George Almási, Calincaval, and Peng Wu, editors, *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 345–363. Springer, 2006. [p. 76]

[123] Bard Bloom Martin Hirzel, Nathaniel Nystrom and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Practical Aspects of Declarative Languages*, 2008. [p. 16]

[124] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999. [pp. 12, 14, 15, and 41]

[125] Apache MINA. http://mina.apache.org/. [p. 16]

[126] Jayadev Misra and William Cook. Computation orchestration: A basis for wide-area computing. In *Journal of Software and Systems Modeling*, 2007. [pp. 107 and 110]

[127] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System-F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. [p. 79]

[128] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998. [p. 102]

[129] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 461–478. ACM, 2007. [p. 78]

[130] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages (POPL)*, pages 228–241, 1999. [p. 111]

[131] Francesco Zappa Nardelli, Peter Sewell, Jaroslav Ševčík, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. Relaxed memory models must be rigorous. In *Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2)*, 2009. [p. 25]

[132] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. [p. 52]

[133] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998. [p. 78]

[134] Occam-pi programming language. http://www.cs.kent.ac.uk/projects/ofa/kroc. [pp. 13 and 34]

[135] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33, 1996. [p. 104]

[136] Orc: Orchestration language. http://orc.csres.utexas.edu/. [p. 83]

[137] John K. Ousterhout. Why threads are a bad idea (for most purposes), 1996. Presentation given at the 1996 Usenix Annual Technical Conference, January 1996. [p. 23]

[138] John K. Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. Technical report, Stanford University, 2009. [p. 32]

[139] Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynès. Incommunicado: efficient communication for isolates. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 262–274, 2002. [p. 37]

[140] Matthew M. Papi. Practical pluggable types for Java. Master's thesis, Massachusetts Institute of Technology, 2008. [pp. 17 and 75]

[141] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla. Comparing the performance of web server architectures. In *Proc. of EuroSys*, 2007. [p. 101]

[142] Matthew J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. [p. 76]

[143] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, Berkeley, 1995. [p. 33]

[144] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *International Conference on Functional Programming (ICFP)*, pages 216–227, 2005. [p. 88]

[145] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. [p. 52]

[146] Dan Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3):48–55, 2008. [p. 13]

[147] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, May 2000. [p. 24]

[148] Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994. [pp. 23, 48, 55, and 75]

[149] John H. Reppy. CML: A higher-order concurrent language. In *Programming Language Design and Implementation (PLDI)*, pages 293–305, 1991. [p. 89]

[150] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993. [p. 83]

[151] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002. [p. 76]

[152] RIFE web application framework. `http://rifers.org`. [pp. 83 and 88]

[153] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. [p. 34]

[154] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, January 1998. [pp. 55, 59, 60, 67, 75, and 76]

[155] Alexandru Salcianu and Martin C. Rinard. A Combined Pointer and Purity Analysis for Java Programs. In *MIT Technical Report MIT-CSAIL-TR-949*, 2004. [p. 75]

[156] Scala programming language. `http://www.scala-lang.org`. [p. 29]

[157] Jan Schäfer and Arnd Poetzsch-Heffter. Coboxes: Unifying active objects and structured heaps. In Gilles Barthe and Frank S. de Boer, editors, *IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 201–219. Springer, 2008. [p. 93]

[158] Douglas Schmidt and Tim Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Third Annual Pattern Languages of Program Design Conference*, 1996. [p. 24]

[159] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22, 1990. [p. 12]

[160] Siege: http load testing and benchmarking. `http://www.joedog.org/index/siege-home`. [p. 102]

[161] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In *Programming Languages and Systems, European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381, 2000. [p. 79]

[162] Yee Jiun Song, Marcos Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC Chains: Efficient client-server communication in geodistributed systems. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. [p. 15]

[163] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005. [p. 32]

[164] Jesper Honig Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in Java. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr, editors, *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 211–228. ACM, 2007. [p. 79]

[165] Sriram Srinivasan. A thread of one's own. In *Workshop on New Horizons in Compilers*, 2006. [p. 88]

[166] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java; a million actors, safe zero-copy communication). In *European Conference on Object-Oriented Programming (ECOOP)*, 2008.

[167] State Threads library for internet applications. `http://state-threads.sourceforge.net/`. [p. 96]

[168] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41, 1996. [p. 75]

[169] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160, 2007. [p. 37]

[170] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics which can deal with full jumps. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974. [p. 82]

[171] Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. Ph.D. thesis, Dept. of Computer Science, University of Illinois, Urbana, 1996. [p. 110]

[172] Herb Sutter. The free lunch is over: a fundamental turn toward toward concurrency. *Dr. Dobb's Journal*, March 2005.

[173] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005. [p. 22]

[174] Wei Tao. *A portable mechanism for thread persistence and migration*. PhD thesis, University of Utah, 2001. [p. 88]

[175] Hidehiko Masuhara Tatsurou Sekiguchi and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *Coordination Models and Languages*, pages 211–226. Springer-Verlag, 1999. [p. 83]

[176] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors: The BubbleSched framework. In *Euro-Par*, pages 42–51, 2007. [p. 92]

[177] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. [p. 77]

[178] Eddy Truyen, Bert Robben, Bart Vanhuate, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in Java. In *Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, pages 29–43. Springer-Verlag, 2000. [p. 88]

[179] Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. In *ACM International Conference on Object-Oriented Programming, Systems and Applications (OOPSLA)*, pages 211–230, 2005. [p. 75]

[180] Jan Vitek and Boris Bokowski. Confined types in Java. *Software—Practice and Experience*, 31(6):507–532, 2001. [p. 78]

[181] J. Robert von Behren, Jeremy Condit, and Eric A. Brewer. Why events are a bad idea (for high-concurrency servers). In *Hot Topics in Operating Systems (HotOS)*, pages 19–24, 2003. [pp. 14 and 23]

[182] Robert von Behren, J. Condit, F. Zhou, George Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Symposium on Operating Systems Principles (SOSP)*, 2003. [p. 87]

[183] Philip Wadler. Is there a use for linear logic? In *Partial Evaluation and Program Manipulation (PEPM)*, pages 255–273, 1991. [p. 76]

[184] David Walker and Greg Morrisett. Alias types for recursive data structures. In *ACM SIGPLAN Workshop on Types in Compilation (TIC) , Montral, Qubec* , September, 2000, volume 2071, pages 177–206. Springer-Verlag, 2001. [p. 80]

[185] Peter Welch. JCSP: Java Communicating Sequential Processes. `http://www.cs.kent.ac.uk/projects/ofa/jcsp`. [p. 89]

[186] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable Internet services. *Operating Systems Review*, 35(5):230–243, December 2001. [pp. 15, 89, and 92]

[187] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004. [p. 75]

[188] Phil Winterbottom, Sean Dorward, and Rob Pike. The Limbo Programming Language. In *Proceedings of Compcon 97*, 1997. [p. 13]

[189] Byung-Sun Yang, Soo-Mook Moon, and Kemal Ebcioğlu. Lightweight monitors for the Java virtual machine. *Software—Practice and Experience*, 35(3):281–299, 2005. [p. 36]

[190] Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213–241, 2008. [p. 79]

[191] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, 2006. [p. 78]