Kilim: Isolation-Typed Actors for Java
(A Million Actors, Safe Zero-Copy Communication)

Sriram Srinivasan and Alan Mycroft

University of Cambridge Computer Laboratory,
Cambridge CB3 0FD, UK
{Sriram.Srinivasan,Alan.Mycroft}@cl.cam.ac.uk

Abstract. This paper describes Kilim, a framework that employs a combination of techniques to help create robust, massively concurrent systems in mainstream languages such as Java: (*i*) ultra-lightweight, cooperatively-scheduled threads (*actors*), (*ii*) a message-passing framework (no shared memory, no locks) and (*iii*) isolation-aware messaging.

Isolation is achieved by controlling the shape and ownership of mutable messages – they must not have internal aliases and can only be owned by a single actor at a time. We demonstrate a static analysis built around isolation type qualifiers to enforce these constraints.

Kilim comfortably scales to handle hundreds of thousands of actors and messages on modest hardware. It is fast as well – task-switching is 1000x faster than Java threads and 60x faster than other lightweight tasking frameworks, and message-passing is 3x faster than Erlang (currently the gold standard for *concurrency-oriented* programming).

1 Imagine No Sharing

Computing architectures are getting increasingly distributed, from multiple cores in one processor and multiple NUMA processors in one box, to many boxes in a data centre and many data centres. The shared memory mindset – synonymous with the concurrent computation model – is at odds with this trend. Not only are its idioms substantially different from those of distributed programming, it is extremely difficult to obtain correctness, fairness and efficiency in the presence of fine-grained locks and access to shared objects.

The "Actor" model, espoused by Erlang, Singularity and the Unix process+pipe model, offers an alternative: independent communicating sequential entities that share nothing and communicate by passing messages. Address-space isolation engenders several desirable properties: component-oriented testing, elimination of data races, unification of local and distributed programming models and better optimisation opportunities for compilers and garbage collectors. Finally, data-independence promotes failure-independence [1]: an exception in one actor cannot fatally affect another.

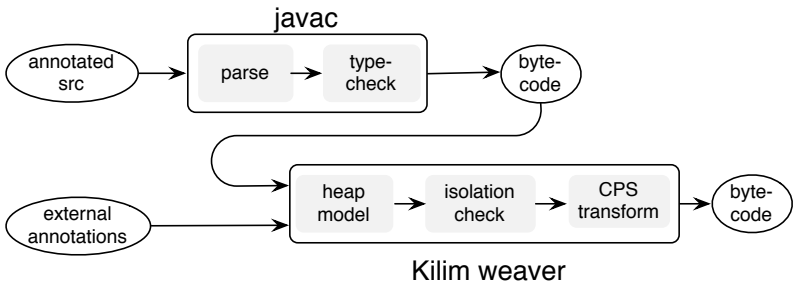


Fig. 1. `javac` output post-processed by Kilim `weaver`

1.1 Motivation

The actor and message-passing approach, with its coarse-grained concurrency and loosely-coupled components is a good fit for split-phase workloads (CPU, network and disk) [4] and service-oriented workflows. With a view to immediate industrial adoption, we impose the following additional requirements: (a) no changes to Java syntax or to the JVM, (b) lightweight actors[1] (c) fast messaging (d) no assumptions made about a message receiver's location and implementation language (e) widespread support for debugging, logging and persistence.

1.2 The Kilim Solution

This paper introduces Kilim[2], an actor framework for Java that contains a bytecode post-processor ("weaver", see Fig. 1) and a run-time library. We list below some important features as well as the design points:

Ultra-lightweight threads Kilim's *weaver* transforms methods identified by an `@pausable` annotation into continuation passing style (CPS) to provide cooperatively-scheduled lightweight threads with automatic stack management and trampolined call stack [3, 20]. These actor threads are quick to context-switch and do not need pre-allocated private heaps. The annotation is similar in spirit to checked exceptions in that all callers and overriding methods must be marked `@pausable` as well.

Messages as a special category For the reasons outlined above, we treat message types 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

[1] For example, threads are too heavyweight to assign per HTTP connection or per component in composable communication protocol state machines.

[2] Kilims are flexible, lightweight Turkish flat rugs woven with fine threads.

public structure permits pattern-matching, structure transformation, delegation and flexible auditing at message exchange points; these are much harder to achieve in the presence of encapsulation.
  - *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 and XML schemas. The lack of internal aliasing is less limiting in practice than would first appear, mostly because loosely-coupled components tend to have simple interfaces. Examples include events or messages in most server frameworks, windowing systems, the Singularity operating system [18] and CORBA valuetypes.
  - *Linearly owned.* A message can have at most one owner at any time. This allows efficient zero-copy message transfer where possible. The programmer has to explicitly make a copy if needed, and the imperative to avoid copies puts a noticeable "back pressure" on the programmer.

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 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.

Run-time support Kilim contains a run-time library of type-parametrised *mailboxes* for asynchronous message-passing with I/O throttling and prioritised *alting* [23]; SEDA-style I/O conditioning [36] is omnipresent. Mailboxes can be incorporated into messages, π-calculus [28] style. Space prevents us from presenting much of the run-time framework; this paper concentrates on the compile-time analysis and transformations.

The contribution of this work is the synthesis of ideas found in extant literature and in picking particular design points that allow portability and immediate applicability (no change to the language or the JVM).

1.3 Isolation Qualifiers and Capabilities: A Brief Overview

Drossopoulou *et al* [16] present in their brief survey the choices of syntactic representations for controlling aliasing. One issue they raise is the need to "develop lightweight and yet powerful [shape] systems". We have adopted "only trees may be transferred between actors" as our guiding principle.

The motivations given in Sec. 1.1 led us to choose a scheme with (*i*) a marker interface `Message` to identify tree-shaped message types which may contain primitive types, references to `Message`s and arrays of the above; and (*ii*) three qualifiers (`@free`, `@cuttable`, `@safe`) on method parameters, which we formalise within a calculus.

These qualifiers can be understood in terms of two orthogonal *capabilities* of an object in a tree: first, whether it is pointed to by another object or not (called

a *root* in the latter case) and second, whether or not it is structurally modifiable (whether its pointer-valued fields are assignable). The latter is a transitive property; an object is structurally modifiable if its parent is.

Given this, an object is *free*[3] if it is the root of a tree and is structurally modifiable. A *cuttable* object may or may not be the root, but is structurally modifiable. An object with a *safe* capability cannot be structurally modified (transitively so), and does not care whether or not it is the root. These capabilities represent in decreasing order the amount of freedom offered by an object (in our ability to modify it, send to another actor, to placel on either side of a field assignment). We use the term *send* (*sent*) to mean that the message is effectively transferred out of the sender's space after which the sender is not permitted access to the message.

Clearly, in all cases, a node in our `Message` tree can have at most one other object pointing to it[4]; in Boylands' terminology [9], all fields of our `Message`s are *unique*, which *provides a system-wide invariant that permits an easy intuitive grasp of our isolation qualifiers as deep qualifiers*. The *cut* operator (see below) can be read as an explicit version of the notion of *destructive reads* [9]. The *cuttable* and *safe* capabilities can be seen as variants of Boylands' *borrowed*.

The relationship between qualifiers and capabilities is this: the qualifiers are specified on method interfaces and imply a interface contract between a method and its caller and, in addition, bestow the corresponding capability on the object referred to by the method parameter. Sec. 3 gives the specifics.

The *cut* operator performs a specific structural modification: it cuts a branch of a tree, severing a subtree from its parent. In addition, it grants the root of the subtree a *free* capability. Only *new* and *cut* can create *free* objects.

As an aside, we provide an additional (unchecked) escape interface `Sharable` that allows the programmer to identify classes that do not follow our message restrictions, yet can be safely transferred across to another thread. These may include immutable classes and those with internal aliasing.

2 Example

Fig. 2 shows a simple Actor class `TxtSrvr` that blocks on a mailbox awaiting a message, transforms the message and responds to a reply-to mailbox specified in the message itself.

`TxtMsg` is a message class identified as such with the marker interface `Message`. The programming model for actors (`TxtSrvr` here) is similar to that for Java threads – replace `Thread` with `Actor` and `run()` with `execute()`. Similarly, an actor is spawned thus: `new TxtSrvr().start();`

The entry point of a Kilim task is `execute()`, the only method of the actor required to be public. Its other non-private methods may only have message-

[3] Note: parameters have qualifiers, objects have capabilities; we write `@free` for the programmer-supplied qualifier and *free* for the corresponding object's capability.

[4] At most one *heap alias*. Multiple local variables may also have the same pointer value.

```
import kilim.*;
class Mbx extends Mailbox<TxtMsg> {}

class TxtSrvr extends Actor {
  Mbx mb;
  TxtSrvr(Mbx mb) {this.mb = mb;}

  @pausable
  public void execute() {
    while(true) {
      TxtMsg m = mb.get();
      transform(m);
      reply(m);
  }}

  @pausable
  void reply(@free TxtMsg m) {
     m.replymb.put(m);
  }

  // @safe is default, so optional
  void transform(@safe TxtMsg m) {···}
}
```

```
class TxtMsg
  implements Message
{
   Mbx replymb;
   byte [ ] data;
}
```

```
// Sample driver code

// spawn actor
Mbx outmb = new Mbx();
new TxtSrvr(outmb).start();

// Send and recv message
Mbx replymb = new Mbx();
byte [ ] data = ...
outmb.put(new TxtMsg(replymb, data));
... = replymb.get();
```

Fig. 2. Example Kilim code showing annotations for message and stack management. Kilim's semantic extensions are in bold.

compatible parameters and results. The `@pausable` annotation on a method informs Kilim's `weaver` that the method may (directly or transitively) call other pausable methods such as `Actor.sleep()` and `Mailbox.get()`.

The blocking call (to `Mailbox.get()`) in an infinite loop illustrates automatic stack management. A typical state machine framework would have the programmer rewrite this in a callback-oriented style and arrange to return to a main loop; this style is prevalent even in multi-threaded settings because threads are expensive and slow resources.

Kilim's mailboxes are type-specific and thread-safe message queues, and being sharable objects (see Sec. 5.2), they can be passed around in messages. They support blocking, timed-blocking and non-blocking variants of `get` and `put`. An actor may simultaneously wait for a message from one of many mailboxes using `select` (like CSP's *alt* [23]). Rudimentary I/O throttling is provided in the form of bounded queue sizes (default is unbounded), and the caller of `Mailbox.put()` is suspended if the queue is full (which is why `reply()`) must be marked as pausable in the example.

The isolation qualifier `@free` on the `reply()` method's parameter is a contract between the caller (`execute()`) and the callee. The weaver checks that the

$$
\begin{array}{l}
\mathit{FuncDcl} ::= \mathit{free}_{\text{opt}}\ m(\vec{p} : \vec{\alpha})\ \{\ (lb : \mathit{Stmt})^{*;}\ \} \\
\quad \mathit{Stmt} ::= x := \texttt{new} \quad | \ x := y \\
\qquad\qquad | \ x := y.f \quad | \ x.f := y \quad | \ x := \texttt{cut}(y.f) \\
\qquad\qquad | \ x := y[\cdot] \quad | \ x[\cdot] := y \quad | \ x := \texttt{cut}(y[\cdot]) \\
\qquad\qquad | \ x := m(\vec{y}) \mid \texttt{if/goto}\ \vec{lb} \mid \texttt{return}\ x
\end{array}
$$

$x, y, p \in$ variable names	$f \in$ field names
$lb \in$ label names	$m \in$ function names
$sel \in$ field names $\cup\ \{[\cdot]\}$	$[\cdot]$ pseudo field name for array access
$\alpha, \beta \in$ isolation qualifier $\{\mathit{free},\ \mathit{cuttable},\ \mathit{safe}\}$	
null is treated as a special readonly variable	

Fig. 3. Core syntax. All expressions are in A-normal form. Variables not appearing in the parameter list are assumed to be local variables.

caller supplies an object with a *free* capability to the callee and subsequently does not use any local variables pointing to or into the message. In turn, `reply` cedes all rights to the message after calling the mailbox's `put()` method (because the latter too has a `@free` annotation on its formal parameters).

The `transform()` method does not require its supplied arguments to be *free*. This means that `execute()` is permitted to use the message object after `transform()` returns. Note also that `transform()` is not marked with `@pausable`, which guarantees us that it does not call any other pausable methods.

3 Core Language

Fig. 3 shows our core syntax, a Java-like intra-procedural language. The language is meant for the isolation checking phase only; it focuses solely on message types and its statements have a bearing on variable and heap aliasing only. We confine ourselves to purely intra-procedural reasoning for speed, precision and localising the effect of changes to code (whole program analyses sometimes show errors in seemingly unrelated pieces of code).

Primitive fields and normal Java objects, while tracked for the CPS transformation phase, are not germane to the issue of isolation checking. A program in this language is already in A-normal form (all intermediate expressions named).

Isolation Qualifiers and Capabilities We mentioned earlier that isolation qualifiers (α, β) are specified in the form of annotations on method parameters and return values. Like normal types, they represent the capabilities of the arguments expected (an object must be at least as capable). Internally to the method, the qualifiers represent the *initial* capability for each parameter object; the object's capability may subsequently change (unlike its Java type). Other objects' capabilities are inferred by a data-flow analysis (Sec. 5). In all cases, we enforce the invariant that there can be at most one heap pointer to any message object.

The list below informally describes object capabilities (Fig. 8 has the precise semantics). It bears repeating that they reflect a lattice composed of two boolean properties – root node or not and, whether or not its pointer-valued fields are assignable (structurally modifiable).

***free*:** The *free* capability is granted to the root of a tree by *new* and by *cut*, and to a method parameter marked as `@free`. A *free* object is guaranteed to be a root, but not vice-versa. It is field-assignable to another non-*safe* object and can be used as an argument to a method with any qualifier.

***cuttable*:** This capability is granted to an object obtained via a field lookup of another non-*safe* object, from downgrading a *free* object by assigning it to a field of another (it is no longer a root) and to a method parameter marked `@cuttable`. This capability permits the object to be cut, but not to be assigned to another object (because it is not necessarily a root). This capability is transitive: an object is *cuttable* if its parent is.

***safe*:** The *safe* capability is granted to a method parameter marked `@safe` or (transitively) to any object reachable from it. A *safe* object may not be structurally modified or further heap-aliased or *sent* to another actor.

The qualifiers on method parameters impose the following interface contracts on callers and callees:

`@free`**:** This allows the method to treat the parameter (transitively the entire tree rooted there) as it sees fit, including *sending* it to another actor. The type system ensures that the caller of the method supplies a *free* argument, and subsequently forbids the use of all local variables that may point to any part of the tree (reachable from the argument).

`@cuttable`**:** The caller must assume that the corresponding object may be cut anywhere, and must therefore forget about all local variables that are reachable from the argument (because the objects they refer to could be cut off and possibly *sent* to another actor).

`@safe`**:** The caller can continue to use a message object (and all aliases into it) if it is passed to a `@safe` parameter. The callee cannot modify the structure.

The `cut` operator severs a subtree from its (*cuttable*) parent thus:

$$y = cut(x.sel) \qquad \stackrel{def}{=} \qquad y = x.sel; x.sel = null;$$

Crucially, and in addition, it marks y as *free*; ordinarily, performing the two operations on the right hand side would only mark y as *cuttable*. The `cut` operator works identically on fields and arrays. Because it is a single operation and because messages (and their array-valued components) are tree-structured by construction, the subtree can be marked *free*.

Remark 1. The most notable aspect of this calculus is that we amplify the requirement that at most one actor owns a given message into the stronger one that at most one dynamically active method stack frame may refer to a *free*

message. This is justified by the requirements that (*i*) a *free* object is a root object and (*ii*) the rules on passing it to a method expecting a `@free` parameter cause all local variables pointing to it to be marked inaccessible. Therefore inter-actor communication primitives of the form *send* and *receive* are treated as simple method calls; in other words, all that is required of an inter-actor messaging facility like the mailbox is that they annotate their parameters and return values (for send and receive operations respectively) with *free*, thereby trivially isolating the intricacies of inter-actor and inter-thread interaction, Java memory model, serialization, batched I/O, scheduling etc.

Remark 2. One could readily add an intermediate qualifier between `@cuttable` and `@safe`, say `@cutsafe`, which permits all modifications except cutting. That is, it could allow additions to the tree and nullification, but not extraction via *cut* for possible transfer of ownership.

In addition to matching object capabilities with isolation qualifiers on method parameters, Kilim enforces a rule to eliminate parameter-induced aliasing: arguments to a method must be pairwise disjoint (trees may not overlap) if any one of them is non-*safe*, and the return value, if any, must be *free* and disjoint from the input parameters.

3.1 Why Qualifiers on Variables Are Not Enough

One might hope that a simple type system à la PacLang [17] can be created by associating variables of `Message` type with isolation type qualifiers, which change with the program point. However, such type systems do not take relationships between variables into account. For example, if we know that x and y are aliases, or y points within the structure rooted at x, then passing x to a method accepting a *free* message (e.g. `Mailbox.put()`) must result in not only x but also y being removed from the objects accessible from the scope of the actor.

In other words, while it is convenient to think of *variables* as having a qualifier such as `@free`, it is really the *objects* that have such a qualifier. We need to analyse methods to infer variable dependencies; the next two sections expand on this subject.

We split isolation checking into two phases for exposition, although the implementation performs them pointwise on the control flow graph. These two phases are covered in Sec. 4 and Sec. 5.

4 Heap Graph Construction

A program may create an unbounded set of message objects at run-time. A compile-time analysis of such a program requires that we first create an abstract model of the heap, called a *heap graph*. Each *node* of this (necessarily finite) graph represents a potentially infinite set of run-time objects that have something in common with each other at a given program point, and different heap analyses differ on the common theme that binds the objects represented by the node.

$G: \langle L,\ E\rangle$	Heap graph is a pair of local var info L and edges E
$L \in \mathcal{P}(\langle Var, LNode\rangle)$	L = relation between local variable names and *nodes* (*LNode* is logically the nodes of the graph)
$E \in \mathcal{P}(\langle Node, sel, Node\rangle)$	E = a set of Node-Node edges labelled with field names
$LNode \in \mathcal{P}(Var)$	Heap Graph node; in this formalism the name of the node consists of the set of local variable names that may point to it. Well-formedness: $\langle x, N\rangle \in L \Leftrightarrow x \in N$
$Node \in \mathcal{P}(Var) \cup \{\emptyset\}$	Labelled nodes plus summary node.
Convenience: $L(x) \stackrel{def}{=} \{N \mid \langle x, N\rangle \in L\}$	set of *LNodes* to which a local variable might point.

Fig. 4. Heap Graph formalism following [37].

We base our heap graph abstraction on a simple variant of shape analysis [37]; we claim no novelty. Our contribution is the set of design choices (isolation qualifiers, tree-structure, local analysis, the `cut` operator) that make the problem simpler and faster to reason about; it is a shape-enforcement rather than a general analysis problem.

A heap graph G (see Fig. 4) is a pair $\langle L, E\rangle$; L is the set of associations between variable names and nodes, and E represents the set of labelled edges between nodes. A node may be pointed to by more than one variable and is identified by a label that is merely the set of variable names pointing to it (a reverse index).

Fig. 5 shows example heap graphs at two program points. The sample heap graph l_1 is represented algebraically as follows[5]:

$L = \{\ \langle a, \{a\}\rangle,\ \langle b, \{b, d\}\rangle,\ \langle d, \{b, d\}\rangle,\ \langle c, \{c, d\}\rangle,\ \langle d, \{c, d\}\rangle,\ \langle e, \{e\}\rangle\ \}$

$E = \{\ \langle \{a\}, f, \{b, d\}\rangle,\ \langle \{a\}, f, \{c, d\}\rangle,\ \langle \{b, d\}, g, \{e\}\rangle,\ \langle \{c, d\}, g, \{e\}\rangle\ \}$

The common theme among run-time objects represented by a shape analysis node is that they are all referred to by the set of variables in the node's label, at that program point, for any given run of the program – a node is an *aliasing configuration*.

In addition to the labelled nodes mentioned thus far, there is one generic *summary* node with the special label $\emptyset$ that represents all heap objects not directly referred to by a local variable. When a node ceases to be pointed to by any variable, its label set becomes empty and it is merged with the summary node (hence '$\emptyset$'—by analogy with the empty set symbol).

Note that edges originate or end in labelled nodes only; the heap graph does not know anything about the connectivity of anonymous objects (inside the $\emptyset$ node)

[5] Parallels to shape analysis [37]: G is their *static shape graph*, L is E_v with a layer of subscripting is removed; we write $\langle y, \{x, y, z\}\rangle$ for their $\langle y, n_{x,y,z}\rangle$.

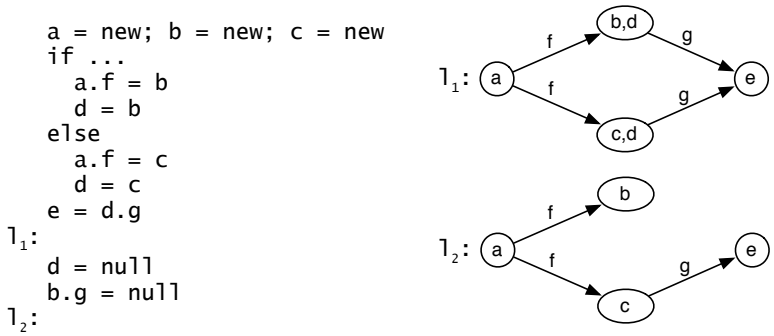


Fig. 5. Sample heap graphs at l_1 and l_2. Only edges E are shown; L is implicit.

The most important invariant in heap graph construction is that there cannot be an edge between two nodes whose labels are not disjoint. Without the invariant, an edge such as $\langle \{x, y\},\ f,\ \{x, u\} \rangle$ would represent the following impossible situation. x and y point to the same set of run-time objects (at that program point, on any run of the program). These objects in turn are connected to another bunch of objects, referred to by x and u. This is clearly not possible, because x's objects have both an outgoing and an incoming edge while its aliasing partners (y and u) only have one or the other edge. Non-disjoint alias sets can coexist in the graph, as long as they do not violate this invariant.

Given the control flow graph *CFG* mentioned earlier, we use the following equations to construct the heap graph G after every program point. The analysis is specified in terms of an iterative forward flow performed on the lattice $\langle G, \subseteq \rangle$. We merge the heap graphs at control-flow join points to avoid the exponential growth in the set of graphs (like [37], unlike [29]). This means all transfer functions operate on a single heap graph (rather than a set of graphs).

$$
\begin{aligned}
G^{init}_{out} &= \langle\ \{\ \},\ \{\ \}\ \rangle \\
G^{l}_{in} &= \bigcup \{G^{l'}_{out} \mid (l', l) \in CFG\} \\
G^{l}_{out} &= [\![\ \cdot\ ]\!](G^{l}_{in})
\end{aligned}
$$

The second equation merges the graphs from the CFG node's incoming edges (simple set union of node and edge sets). $[\![\ \cdot\ ]\!]$ represents the transfer functions for each CFG node (Fig. 6). Note that *if_goto* and *return* do not have transfer functions; they are turned into edges of the CFG.

The transfer functions are simpler than the ones in shape analysis because they do not deal with sharing (attempts to share are faulted in the isolation checking phase). Note that the heap graph may have nodes with multiple incoming edges, but it reflects a may-alias edge, not an edge that induces sharing. The node labelled e in Fig. 5 represents two disjoint sets of run-time objects,

Notation: V (any Node), S (source Node), T (target Node)		
S_x	$\stackrel{def}{=}$	$S \cup \{x\}$
S_x^y	$\stackrel{def}{=}$	$\begin{cases} S \cup \{x\} & \text{if } y \in S \\ S & \text{otherwise} \end{cases}$
$kill(G, x)$	$\stackrel{def}{=}$	$L' = \{ \langle v, V' \rangle \in L \mid v \neq x \wedge V' = V \setminus \{x\} \wedge \langle v, V \rangle \in L\}$ $E' = \{ \langle S \setminus \{x\},\ sel,\ T \setminus \{x\} \rangle \mid \langle S,\ sel,\ T \rangle \in E\}$
$[\![entry(mthd)]\!]\ G$		$\mathbf{L}'' = \bigcup_i \{ \langle p_i,\ \{p_i\} \rangle \}$ where p_i is the i^{th} parameter of $mthd$ $\mathbf{E}'' = \{\}$
$[\![x := \texttt{new}]\!]\ G$		$G' : \langle L', E' \rangle = kill(G, x)$ $\mathbf{L}'' = L' \cup \langle x,\ \{x\} \rangle, \quad \mathbf{E}'' = E'$
$[\![x := y]\!]\ G$		$G' : \langle L', E' \rangle = kill(G, x)$ $\mathbf{L}'' = \{ \langle v,\ V_x^y \rangle \mid \{v,\ V\} \in L' \}$ $\mathbf{E}'' = \{ \langle S_x^y,\ sel,\ T_x^y \rangle \mid \langle S,\ sel,\ T \rangle \in E' \}$
$[\![x.f := y]\!]\ G$		$E' = E \setminus \{ \langle S,\ f,\ * \rangle \in E \mid x \in S \}$ $\mathbf{E}'' = \begin{cases} E' & \text{if } y \equiv null \\ E' \cup \{ \langle S,\ f\ , T \rangle \mid x \in S \wedge y \in T \} & \text{otherwise} \end{cases}$ $\mathbf{L}'' = L$
$[\![x[\cdot] := y]\!]\ G$		$\mathbf{E}'' = \begin{cases} E & \text{if } y \equiv null \\ E \cup \{ \langle S,\ `[\cdot]'\ ,\ T \rangle \mid x \in S \wedge y \in T \} & \text{otherwise} \end{cases}$ $\mathbf{L}'' = L$
$[\![x := y.sel]\!]\ G$		$G' : \langle L', E' \rangle = kill(G, x)$ $\mathbf{L}'' = L'$ $\cup \{ \langle t,\ T_x \rangle \mid \langle t, T \rangle \in L' \wedge \langle y, S \rangle \in L' \wedge \langle S, sel, T \rangle \in E' \}$ $\cup \{ \langle x,\ T_x \rangle \mid \langle y, S \rangle \in L' \wedge \langle S, sel, T \rangle \in E' \}$ $\mathbf{E}'' = (\ E' \setminus \bigcup \{ \langle y,\ sel,\ * \rangle \in E' \})$ $\cup \{ \langle y, sel, T_x \rangle \mid \langle y, sel, T \rangle \in E' \}$ $\cup \{ \langle T_x,\ sel,\ U \rangle \mid \langle T, sel, U \rangle \in E' \}$
$[\![x := cut(y.sel)]\!]$		$[\![y.sel := null]\!] \quad \circ \quad [\![x := y.sel]\!]$
$[\![x := m(\vec{v})]\!]\ G$		$G' : \langle L', E' \rangle = kill(G, x)$ $\mathbf{L}'' = L' \cup \{ \langle x,\ \{x\} \rangle \}$ $\mathbf{E}'' = E'$

Fig. 6. Transfer functions $[\![\cdot]\!]$ for heap graph construction. They transform $G : \langle L, E \rangle$ to $G'' : \langle L'', E'' \rangle$. '$*$' represents wildcards and *sel* represents field and array access.

one of which has incoming edges from the $\{b, d\}$ set of objects and the other from $\{c, f\}$.

The transfer function for $x := y.f$ deserves some attention. It associates x with all nodes T pointed to by $y.f$, which may or may not have been created as yet by the analysis procedure. Fig. 7 covers both possibilities. In the case where a node does not exist, it is treated as if it belongs as a discrete blob inside the summary node, implicitly referred to by $y.f$ (the grey region in Fig. 7). In this case, the node is *materialized* [37] out of the summary node and all edges outgoing from that node are replicated and attached to the newly materialized node. This replication is necessary because we do not have precise information about which portion of the anonymous heap (represented by the summary node) is responsible for the outgoing edges (the grey blob, or the non-grey portion). Note that we do not have to replicate the incoming edges because we know that nodes are not shared and that the newly materialized node is already pointed to by the $y.f$ edge.

Shape analysis provides *strong nullification* and *disjointness* [37], as illustrated in Fig. 5 by the transition from heap graph at l_1 to that of l_2. Unfortunately, shape analysis cannot do the same for arrays: setting "$x[i] = y$" tells us nothing at all about $x[j]$. However, `cut` performs strong nullification even on arrays, because our type system ensures that the array's components are disjoint both mutually and from the variable on the right hand side.

Remark 3. There is an important software engineering reason for having *cut*, instead of relying on shape analysis to inform us about disjointness: we want to make explicit in the code the act of cutting a branch from the tree and giving the subtree a *free* capability. Most methods do not need to cut; they can have the default `@safe` qualifier, which allows them to (transitively) modify the arguments, but not *cut* or *send* the object.

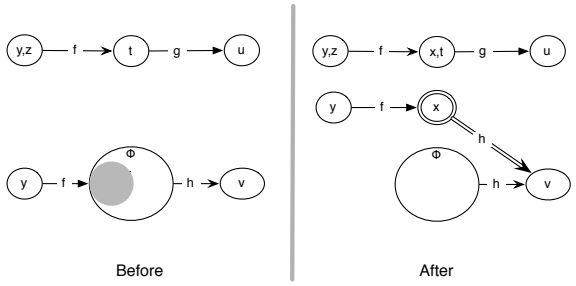


Fig. 7. Example heap graph before and after transformation by $[\![x := y.f]\!]$. Double lines show the newly materialized node and edge. The grey blob is the portion of the anonymous heap that is the implicit target of $y.f$

5 Isolation Capability Checking

Having built heap graphs at every program point, we now associate each labelled node n in each heap graph with a *capability* $\kappa(n)$, as mentioned earlier. All runtime objects represented by n implicitly have the same capability.

Fig. 8 shows the monotone transfer functions operating over the capability lattice in a simple forward-flow pass. At CFG join points, the merged heap graph's nodes are set to the minimum of the capabilities of the corresponding nodes in the predecessor heap graphs (in the CFG). For example,

```
a = new        // κ(a) := free
if ...
   b.f = a     // κ(a) := cuttable
               // join point. κ(a) := min(free, cuttable)
send(a)        // ERROR: κ(a) is not free
```

Note that the function κ has been overloaded to work for both variables and nodes; a variable's capability is the *minimum* capability of all the nodes to which it points.

The transfer function for method calls may be better understood from Fig. 9. The matrix matches capabilities of the argument with the corresponding parameter's isolation qualifier and each cell of the matrix reflects the effect on the capabilities of objects reachable from the argument.

5.1 Soundness

A formal proof of correctness is left to future work. Below, we outline the intuitions that justify our belief that the system is correct, and which we expect could form the basis of a proof.

Firstly, we require all built-in functions that can transfer a message (or a tree of such messages) from one actor to another do so via a `free` parameter or result. Therefore it is necessary to ensure that when an object is so transferred, no part of it remains accessible by the caller after the call – i.e. all local variables that can reference it can no longer be used. Of course, using conventional stack frames, there may remain pointers into the transferred structure, but the critical requirement is that all variables that may refer to these are marked with capability $\bot$. This effect is achieved by a combination of heap graph analysis followed by the capability dataflow propagation.

Secondly, we need to ensure that all operations in the language preserve the invariant that messages are tree-structured and that only the root of a message is ever marked as *free*. This requires a careful examination of each language primitive. Critical cases are:

- x := `cut`$(y.f)$. If y is a well-formed tree, the modified y and x are also well-formed.

Assumption 1: the current method's signature is $\textit{free } mthd(\vec{p} : \vec{\alpha})$.
Assumption 2: E and L used (e.g.) in *dependants* result from Heap Graph analysis for the current instruction.

$[\![entry(mthd)]\!]\ \kappa$	$=$	$[\vec{p} \overset{\kappa}{\mapsto} \vec{\alpha}]$
$[\![x := \texttt{new}\ T]\!]\ \kappa$	$=$	$\kappa[x \overset{\kappa}{\mapsto} \textit{free}]$
$[\![x := y]\!]\ \kappa$	$=$	$\kappa[x \overset{\kappa}{\mapsto} \kappa(y)]$
$[\![x.f := y]\!]\ \kappa$	$=$	$\textit{precondition} : \kappa(y) = \textit{free}$ $\kappa[y \overset{\kappa}{\mapsto} \textit{cuttable}]$
$[\![x := y.f]\!]\ \kappa$	$=$	$\kappa[x \overset{\kappa}{\mapsto} s]$ $s = \begin{cases} \textit{safe} & \text{if } \kappa(y) = \textit{safe} \\ \textit{cuttable} & \text{if } \kappa(y) \in \{\textit{free}, \textit{cuttable}\} \end{cases}$
$[\![x := m(\vec{y})]\!]\ \kappa$	$=$	$\textit{precondition} : \beta_i \sqsubseteq \kappa(y_i)\ \wedge\ (\forall i \neq j)(\textit{disjoint}(y_i, y_j) \vee \beta_i = \beta_j = \textit{safe})$ $\kappa \begin{bmatrix} \textit{dependants}(y_i) \cup \{y_i\} \overset{\kappa}{\mapsto} \bot, & \text{if } (\beta_i = \textit{free}) \\ \textit{dependants}(y_i) \overset{\kappa}{\mapsto} \bot, & \text{if } (\beta_i = \textit{cuttable}) \end{bmatrix} \left[x \overset{\kappa}{\mapsto} \textit{free}\right]$ (assumption: m's signature is $\textit{free } m(\vec{\beta})$. Return value is always *free*)
$[\![x := \textit{cut}(y.f)]\!]\ \kappa$	$=$	$\textit{precondition} : \kappa(y) \in \{\textit{free}, \textit{cuttable}\}$ $\kappa[x \overset{\kappa}{\mapsto} \textit{free}]$
$[\![\texttt{return}\ x]\!]\ \kappa$	$=$	$\textit{precondition} : \kappa(x) = \textit{free} \wedge \forall i(\alpha_i = \textit{cuttable} \Longrightarrow \textit{disjoint}(x, p_i))$ κ (no change)

where:

$\kappa(n) : LNode \mapsto Capability$		gives the Capability associated with a node $n \in LNode$
$(Capability, \sqsubset)$	$=$	$\bot \sqsubset \textit{safe} \sqsubset \textit{cuttable} \sqsubset \textit{free}$
$\kappa(v)$	$\overset{def}{=}$	$min(\kappa(n)),\ n \in L(v)$
$\kappa[v \overset{\kappa}{\mapsto} c : Capability]$	$\overset{def}{=}$	$\kappa[n \mapsto c], n \in L(v)$
$\textit{dependants}(v)$	$\overset{def}{=}$	$\{v' \mid n \in L(v) \wedge n' \in L(v') \wedge n' \in \textit{reachset}(n)\}$ where $\textit{reachset}(n) = \bigcup_{\{n,\ \_,\ n'\} \in E} \{n'\} \cup \textit{reachset}(n')$
$\textit{disjoint}(x, y)$	$\overset{def}{=}$	$x \neq y\ \wedge\ (x\ \notin\ \textit{dependants}(y)\ \wedge y\ \notin\ \textit{dependants}(x)$

Fig. 8. Transfer functions for capability inference. Standard precondition: variables used as rvalues must be valid (i.e. $\sqsupset\bot$)

	Isolation qualifier β		
$\kappa(y)$	**free**	**cuttable**	**safe**
free	$\kappa' = \kappa[y \overset{\kappa}{\mapsto} \bot, \vec{z} \overset{\kappa}{\mapsto} \bot]$	$\kappa' = \kappa[y \overset{\kappa}{\mapsto} \bot]$	$\kappa' = \kappa$
cuttable		$\kappa' = \kappa[y \overset{\kappa}{\mapsto} \bot]$	$\kappa' = \kappa$
safe			$\kappa' = \kappa$

Fig. 9. Effect of the call $m(y)$ – where m's signature is $m(\beta\ p)$) – on the capabilities of y and on the dependants $\vec{z}$ of y. A blank indicates the call is illegal.

– $x.f := y$. This is the only form that can create heap aliases and its preconditions ensure that no more than one heap alias is created for any object. Further, y simultaneously loses the property of being a root and being *free*.

The correctness of the heap graph analysis rests on it being a special case of shape analysis (we can omit the "heap-sharing" flag).

Together, the argument is that each single step of evaluation preserves the property that only the root of a message can ever be *free*.

As a consequence each heap node is only accessible from at most one method in one actor as *free*, and therefore accessible from at most one actor.

5.2 Interoperation With Java

Java classes identified by the marker interface `Message` are treated as message types. We have treated Java objects and `Message`s as immiscible so far. This section describes the manner in which they can be mixed and the effect of such mixing on correctness.

Immutable classes such as `String` and `Date` already satisfy our interference-freedom version of isolation—even though, in the JVM they may be implemented as references shared by multiple actors, this sharing is benign. However, if the programmer wants to share a class between actors and is aware of the implications of sharing such a class (the mailbox is an example), he can have the class implement a marker interface `Sharable`. The weaver does not perform any checks on objects of such and therefore permits multiple heap-aliases. Clearly, this is a potential safety loophole. Objects derived from fields of `Sharable` objects are treated as regular Java objects, unless they too are instances of a `Sharable` class.

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. For existing classes whose sources cannot be annotated, but whose behaviour is known, the programmer can supply "external annotations" to the weaver as a text file (Fig. 1):

```
class java.lang.String implements Sharable
interface java.io.PrintStream {
  void println(@safe Object o);
}
```

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.

The `@safe` annotation implies that the method guarantees that the parameter does not escape to a global variable or introduce other aliases (such as a collection class might), guarantees that are ordinarily given by message-aware methods.

Kilim accommodates other object types (needed for creating closures for the CPS transformation phase), but does not track them as it does message types. We take the pragmatic route of allowing ordinary Java objects (and their arrays) to be referenced from message classes but give no guarantees of safety. We do not implement any run-time checks or annotations or restrictions (such as a classloader per actor) on such objects.

Finally, the weaver limits static class fields to constant primitive or final `Sharable` objects and prevents exceptions from being message types.

6 Creating Scalable, Efficient Actors

Traditional threading facilities (including those available in the JVM) are tied to kernel resources, which limits their scalability and the efficiency of context switching. We map large numbers of actors onto a few Java threads by the simple expedient of rewriting their bytecode and having them cooperatively unwind their call stack. Unwinding is triggered by calls to `Actor.pause` or `Actor.sleep` (the mailbox library calls these internally).

A scheduler then initiates the process of rewinding (restoring) another actor's call stack, which then continues from where it left off. Much of the mechanics of transformation has been covered in an earlier paper [33]; this section summarises and highlights some of the important engineering decisions.

Unwinding a call stack involves remembering, for each activation frame, the state of the Java operand stack and of the local variables and the code offset to which to return. A call stack can unwind and rewind only if the entire call chain is composed of methods annotated with `@pausable`. Each pausable method's signature is transformed to include an extra argument called a *fiber* (of type `Fiber`), a logical thread of control. The fiber is a mechanism for a method to signal to its caller that it wants to return prematurely. The fiber also acts as a store for the activation frame of each method in the call hierarchy as the stack unwinds. The activation frame of a method consists of the program counter (the code offset to jump back to), the operand stack and the local variables. When the callee pauses (calls `Actor.pause()` or `mailbox.get()`), the caller examines the fiber, learns that it is now in a pausing state, stores its activation frame on the fiber and returns. And so on all the way up past the call chain's starting point, the actor's `execute()` method. This way, the entire call stack with its control and data elements is reified onto the fiber. The process is easily reversed: each method consults the fiber upon entry, jumps directly to the resumption point and restores its state where necessary.

This is conceptually equivalent to a continuation passing style (CPS) transformation; it is however applied only to pausable methods and produces single-shot continuations. The transform inlines local subroutines (reachable from the `jsr` instruction and used in try-finally blocks). Finally, the A-normal form of the CFG helps deal with the restriction imposed by the JVM that one cannot branch to an offset between `new` and the corresponding constructor invocation.

Transforming Java bytecode has the advantage that its format has remained constant while the source language has undergone tremendous transformations (generics, inner classes and soon, lambda functions and closures). It also allows us to perform local surgery and to `goto` into a loop without modifying any of the original code. Finally, it is applicable to other JVM-based languages as well (e.g. Scala).

Fig. 10 shows a sample CFG of a pausable method that makes a call to another pausable method, before and after the transformation performed by Kilim's `weaver`. The CFG shows extended basic blocks (multiple out-edges that account for JVM branching instructions and exception handlers), with the `invoke` instruction to a pausable method separated out into its own block. We will henceforth refer to this basic block as a *call site*.

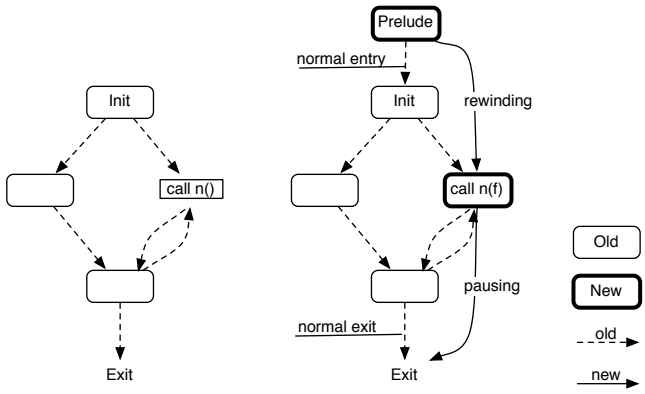


Fig. 10. CFG before and after transform

The `weaver` adds one *prelude* node at entry, modifies each call site and adds two edges, one from the prelude to the call site to help recreate the stack and another from the call site to the exit to pause and unwind the stack. It also adds a node at the entry to every catch handler. None of the original nodes or edges are touched, but the `weaver` maintains the JVM-verified invariant that the types and number of elements in the local variables and operand stack are identical regardless of the path taken to arrive at any instruction. This means that we cannot arbitrarily jump to any offset without balancing the stack first. For this reason, the stack and variables may need to be seeded with an appropriate number of dummy (constant) values of the expected type before doing the jump in the prelude.

6.1 Implementation Remarks

While the general approach is similar to many earlier approaches [5, 25], we feel the following engineering decisions contribute to the speed and scalability of our approach.

We store and restore the activation frame's state lazily in order to incur the least possible penalty when a pausable method does not pause. Unlike typical CPS transformations, we transform only those methods that contain invocations to methods marked `@pausable`. Our heap analysis phase also tracks live variables, duplicate values and constants. The latter two are never stored in the fiber; they are restored through explicit code. These steps ensure the minimum possible size for the closure. To the extent we are aware, these analyses are not performed by competing approaches.

In contrast to most CPS transformations on Java/C# bytecode, we chose to preserve the original call structure and to rewind and unwind the call stack. One reason is that CPS transformations also typically require the environment to support tail-call optimisation, a feature not present in the JVM. Second, the Java environment and mindset is quite dependent on the stack view of things: from security based on stack inspection to stack traces for debugging. In any case, the process of rewinding and unwinding the call stack turned out to be far less expensive than we had originally suspected, partly because we eagerly restore only the control plane, but lazily restore the data plane: only the topmost activation frame's local variables and operand stack are restored before resuming. If the actor pauses again, the intermediate activation frames' states are already in the fiber and do not need to be stored again.

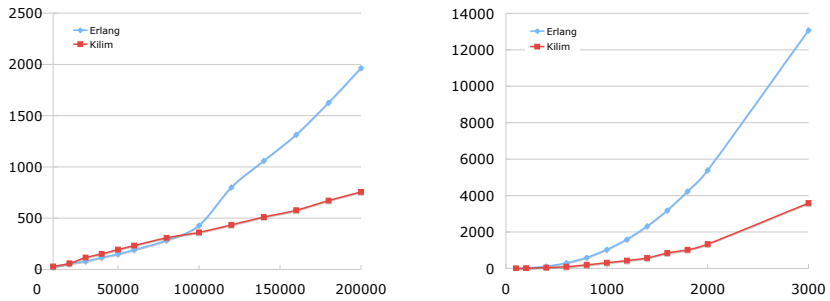
Some researchers have used exceptions as a `longjmp` mechanism to unwind the stack; we use `return` because we found exceptions to be more expensive by almost two orders of magnitude. Not only do they have to be caught and rethrown at each level of the stack chain, they clear the operand stack as well. This unnecessarily forces one to take a snapshot of the operand stack *before* making a call; in our experience, lazy storage and restoration works better.

We chose to modify the method signatures to accommodate an extra fiber parameter in contrast to other approaches that use Java's `ThreadLocal` facility to carry the out-of-band information. Using `ThreadLocal`s is inefficient at best (about 10x slower), and incorrect at worst because there's no way to detect at run time that a non-pausable method is calling a pausable method (unless all methods are instrumented).

We have also noticed that the `@pausable` annotation makes explicit in the programmer's mind the cost of pausing, which in turn has a noticeable impact on the program structure.

7 Performance

Erlang is the current standard bearer for concurrency-oriented programming and sets the terms of the debate, from micro-benchmarks such as speed of pro-



(a) Creation and Destruction (b) Messaging

Fig. 11. Erlang vs. Kilim times. X-axis: n actors (n^2 messages), Y-axis: Time in ms (lower is better).

cess creation and messaging performance, to systems with an incredible 9-nines reliability [2]. Naturally, a comparison between Kilim and Erlang is warranted.

Unfortunately, no standard benchmark suites are yet available for the actor paradigm. We evaluated both platforms on the three most often quoted and much praised characteristics of the Erlang run-time: ability to create many processes, speed of process creation and that of message passing.

All 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.

Kilim's performance exceeded our expectations on all counts. We had assumed that having to unwind and rewind the stack would drag down performance that could only be compensated for by an application that could make use of the JIT compiler. But Kilim's transformation, along with the quality of Java's current run-time, was able to compete favourably with Erlang on tasking, messaging and scalability.

Process creation The first test (Fig. 11(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. The test measures the time taken from start to the last exit message arriving at the central object. 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. 11(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 in 0.54 μsec, which includes context-switching time) and scales linearly.

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 tried 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. 12 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.

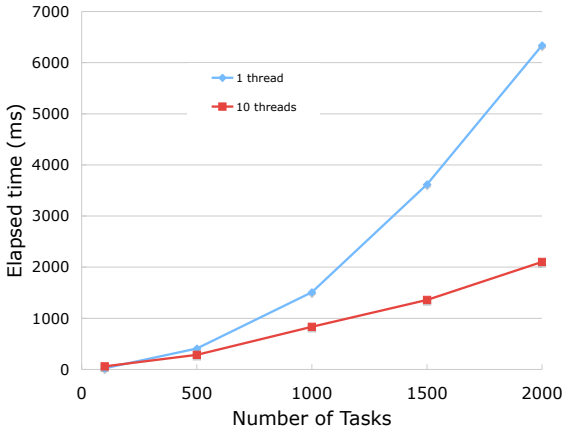


Fig. 12. Kilim messaging performance and hardware parallelism. (n actors, n^2 messages)

Miscellaneous numbers We benchmarked against standard Java threads, RMI objects and Scala (2.6.1-RC1) (within one JVM instance). We do not include these numbers because we found all of them to be considerably slower: a simple binary ping-pong test with two objects bouncing a message back and forth has Kilim 10x faster than Scala's Actor framework [22] (even with the lighter-weight `react` mechanism), 5x faster than threads with Java's `Pipe*Stream` and 100x faster than RMI between collocated objects (RMI always serialises its messages, even if the parameters are non-referential types). Larger scales only worsened the performance gap.

Interpreting the results One cannot set too much store by micro-benchmarks against a run-time as robust as that of Erlang. We are writing real-world applications to properly evaluate issues such as scheduling fairness, cache locality and memory usage. Still, these tests do demonstrate that Kilim is a promising step combining the best of both worlds: concurrency-oriented programming, Erlang style, and the extant experience and training in object-oriented programming.

8 Related Work

Our work combines two orthogonal streams: lightweight tasking frameworks and alias control, with the focus on portability and immediate applicability (no changes to Java or the JVM).

Concurrent Languages Most concurrency solutions can—on one axis—be broadly classified as a language versus library approach [11]. We are partial to Hans Boehm's persuasive arguments [6] that threads belong to the language, not a library. While most of the proposed concurrent languages notably support tasks and messages, few have found real industrial acceptance: Ada, Erlang and Occam. For the Java audience, Scala provides an elegant syntax and type system with support for actors provided as a library [22]; however, lack of isolation and aliasing are still issues. Scala has no lightweight threading mechanism, although the combination of higher-order functions and the `react` mechanism is a far superior alternative to callback functions in Java. JCSP [35], a Java implementation of CSP [23] has much the same issues.

The Singularity operating system [18] features similar to ours: lightweight isolated processes and special message types that do not allow internal aliasing. The system is written in a new concurrency-oriented language (Sing$^{\#}$), and a new run-time based on Microsoft's CLR model but with special heaps for exchanging messages. While ours is a more of an evolutionary approach, we look forward to their efforts becoming mainstream.

Tasks and Lightweight Threads None of the existing lightweight tasking frameworks that we are aware of address the problems of aliased references.

The word "task" is overloaded. We are interested only in tasking frameworks that provide automatic stack management (can pause and resume) and not run to completion, such as Java's `Executor`, FJTask [27]) and JCilk [14]. That said, we have much to learn from the Cilk project's work on hierarchical task structures and work-stealing scheduling algorithms.

The Capriccio project [4] 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 infer the appropriate size and locations to dynamically expand heap space. They report scalability to 100,000 preemptively-scheduled threads.

Pettyjohn *et al* [30] 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, it creates custom objects per invocation site, splits the code into top-level procedures which results in loops being split into virtual function calls.

Many frameworks such as RIFE [5], and the Apache project's JavaFlow [25] transform Java bytecode into a style similar to ours. RIFE does not handle nested pausable calls. Kilim handles all bytecode instructions (including `jsr`) and is significantly faster for reasons explained earlier (and in [33]).

Static Analysis Inferring, enforcing, and reasoning about properties of the heap is the subject of a sizable proportion of research literature on programming languages. We will not attempt to do this topic justice here and will instead provide a brief survey of the most relevant work. We heartily recommend [24], an "action plan" drawn up to address issues caused by unrestricted aliasing in the context of object-oriented programming languages.

Alias analysis concentrates on which variables may (or must) be aliases, but not on how sets of aliases relate to each other, an important requirement for us. We also require strong nullification and disjointness-preservation, something not available from most points-to analyses (*e.g.* [31]), because their method of associating abstract heap nodes with allocation points is equivalent to fixing the set of run-time objects that a variable may point to.

Shape analysis provides us the required properties because it accommodates dynamic repartitioning of the set of run-time objects represented by an alias configuration. However, the precision comes at the expense of speed. Our annotations provide more information to the analysis and pave the way for more modular inter-procedural analyses in the future.

Our approach is most closely related to Boyland's excellent paper on alias burying [10], which provides the notions of *unique* (identical to our *free*) and *borrowed*, which indicates that the object is not further aliasable (*cuttable* and *safe*, in our case). Boyland does not speak of safety from structural modifications, but this is a minor difference. The biggest difference in our approach is not the mechanics of the analysis, but in our design decision that messages be different classes and references to them and their components be unique. Making them different helps in dealing with versioning and selective upgrades (for example, one can have separate classloaders for actors and messages). Allowing free mixing of non-unique and unique references makes it very difficult to statically guarantee safety unless one extends the language, as with ownership types. This is an important software engineering decision; the knowledge that every message pointer is always unique and not subject to lock mistakes ensures that code for serialization, logging, filtering and persistence code does not need to deal with cycles, and permits arrays and embedded components to be exposed.

Type systems are generally monadic (do not relate one variable to another) and flow-insensitive (a variable's type does not change), although flow-sensitive type qualifiers [19] and quasi-linear types [26] are analogous to our efforts. Quasi-linear types have been successfully for network packet processing [17]; however packet structures in their language do not have nested pointers.

Ownership types [12, 7, 8] limit access to an object's internal representations through its owners. External uniqueness types [13] add to ownership types a linearly-typed pointer pointing to the root object. Each of these schemes offers powerful ways of containing aliasing, but are not a good fit for our current requirements: retrofitting into existing work, working with unencapsulated value objects and low annotation burden. An excellent summary of type systems for hierarchic shapes is presented in [16].

StreamFlex [32] relies on an implicit ownership type system that implements scoped allocation and ensures that there are no references to an object in a higher scope, but allows aliasing of objects in sibling and lower scopes. Their analysis relies on a partially closed world assumption. The type system is eminently suited for hooking together chained filters; it is less clear to us how it would work for long-lived communicating actors and changing connection topologies.

There are clearly domains where internal aliasing is useful to have, such as transmitting graphs across compiler stages. Although gcc's GIMPLE IR is tree-structured, one still has to convert it back to a graph, for example. A type system with scoped regions, such as StreamFlex's, permits internal aliasing without allowing non-unique pointers to escape from their embedded scope.

There are several works related to isolation. Reference immutability annotations [34, 21] can naturally complement our work. The Java community has recently proposed JSR-121, a specification for application level isolation; this ensures all global structures are global only to an isolate.

9 Conclusion and Future Work

We have demonstrated Kilim, a fast and scalable actor framework for Java. It features ultra-lightweight threads with logically disjoint heaps, a message-passing framework and a static analysis that semantically distinguishes messages from other objects purely internal to the actor. The type system ensures that messages are free of internal aliases and are owned by at most one actor at a time. This is in contrast to the current environment in all mainstream languages: heavyweight kernel threads, shared memory and explicit locking.

The techniques are applicable to any language with pointers and garbage collection, such as C#, Scala and OCaml.

Our target deployment platform is data-centre servers, where a user request results in a split-phase workflow involving CPU, disk and possibly dozens of remote services [15]; this application scenario helps distinguish our design choices from extant approaches to parallel and grid computing, which are oriented towards CPU-intensive problems such as protein folding.

Our message-passing framework lends itself naturally to a seamless view of local and distributed messaging. Integrating our platform with distributed naming and queueing systems is our current focus.

Another promising area of future work of interest to server-side frameworks is precise accounting of *resources* such as database connections, file handles and security credentials; these must be properly disposed of or returned even in the

presence of actor crashes. We expect to extend the linearity paradigm towards statically-checked accountability of resource usage.

Acknowledgements Thanks are due to Jean Bacon, Boris Feigin, Alexei Gotsman, Reto Kramer, Ken Moody, Matthew Parkinson, Mooly Sagiv, Viktor Vafeiadis, Tobias Wrigstad and the anonymous referees for their patient and detailed feedback. This work is supported by EPSRC grant GR/T28164.

References

[1] Armstrong, J.: Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, The Royal Institute of Technology, Stockholm (2003)
[2] Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall (1996)
[3] Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative Task Management Without Manual Stack Management. In: USENIX Annual Technical Conference, General Track. (2002) 289–302
[4] von Behren, R., Condit, J., Zhou, F., Necula, G., Brewer, E.: Capriccio: Scalable threads for Internet Services. In: 19th ACM Symposium on Operating Systems Principles. (2003)
[5] Bevin, G.: Rife http://rifers.org.
[6] Boehm, H.J.: Threads cannot be implemented as a library. In: ACM Conf. on PLDI. (2005) 261–268
[7] Boyapati, C., Lee, R., Rinard, M.C.: Ownership types for safe programming: preventing data races and deadlocks. In: Proc. of OOPSLA. (2002) 211–230
[8] Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: Proc. of ACM POPL. (2003) 213–223
[9] Boyland, J., Noble, J., Retert, W.: Capabilities for sharing: A generalisation of uniqueness and read-only. In: Proc. of ECOOP. Volume 2072 of LNCS. (2001) 2–27
[10] Boyland, J.: Alias burying: Unique Variables Without Destructive Reads. Softw. Pract. Exper. **31**(6) (2001) 533–553
[11] Briot, J.P., Guerraoui, R., Löhr, K.P.: Concurrency and distribution in object-oriented programming. ACM Comput. Surv. **30**(3) (1998) 291–329
[12] Clarke, D.G., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: Proc. of OOPSLA. (1998) 48–64
[13] Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Proc. of ECOOP. Volume 2743 of LNCS. (2003) 176–200
[14] Danaher, J.S., Lee, I.T.A., Leiserson, C.E.: Programming with exceptions in JCilk. Sci. Comput. Program. **63**(2) (2006) 147–171
[15] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: SOSP. (2007) 205–220
[16] Drossopoulou, S., Clarke, D., Noble, J.: Types for Hierarchic Shapes. In: ESOP. Volume 3924. (2006) 1–6
[17] Ennals, R., Sharp, R., Mycroft, A.: Linear types for Packet Processing. In: ESOP. Volume 2986. (2004) 204–218

[18] Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in Singularity OS. In: Proc. of EuroSys. (2006)
[19] Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: ACM Conf. on PLDI. (2002) 1–12
[20] Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: ICFP. (1999) 18–27
[21] Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable objects for a Java-like language. In: ESOP. Volume 4421. (2007) 347–362 http://www.cs.ru.nl/∼chaack/papers/papers/imm-obj.pdf.
[22] Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Proc. Joint Modular Languages Conference. LNCS (2006)
[23] Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (1978) 666–677
[24] Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R.C.: The Geneva Convention on the treatment of object aliasing. OOPS Messenger **3**(2) (1992) 11–16
[25] JavaFlow: The Apache Software Foundation: http://jakarta.apache.org/commons/sandbox/javaflow.
[26] Kobayashi, N.: Quasi-linear types. In: Proc. of ACM POPL. (1999) 29–42
[27] Lea, D.: A Java fork/join framework. In: Java Grande. (2000) 36–43
[28] Milner, R.: Communicating and Mobile Systems: the π-calculus. Cambridge University Press (1999)
[29] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)
[30] Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S., Felleisen, M.: Continuations from generalized stack inspection. In: ICFP. (2005) 216–227
[31] Salcianu, A., Rinard, M.C.: A combined pointer and purity analysis for Java programs. In: MIT Technical Report MIT-CSAIL-TR-949. (2004)
[32] Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: Streamflex: High-throughput stream programming in Java. (2007) 211–228
[33] Srinivasan, S.: A thread of one's own. In: Workshop on New Horizons in Compilers. (2006) http://www.cse.iitb.ac.in/∼uday/NHC-06/advprogram.html.
[34] Tschantz, M.S., Ernst, M.D.: Javari: adding reference immutability to Java. In: Proc. of OOPSLA. (2005) 211–230
[35] Welch, P.: JCSP http://www.cs.kent.ac.uk/projects/ofa/jcsp.
[36] Welsh, M., Culler, D.E., Brewer, E.A.: SEDA: An architecture for well-conditioned, scalable internet services. In: SOSP. (2001) 230–243
[37] Wilhelm, R., Sagiv, S., Reps, T.W.: Shape analysis. In: Proc. of CC. Volume 1781 of LNCS. (2000) 1–17