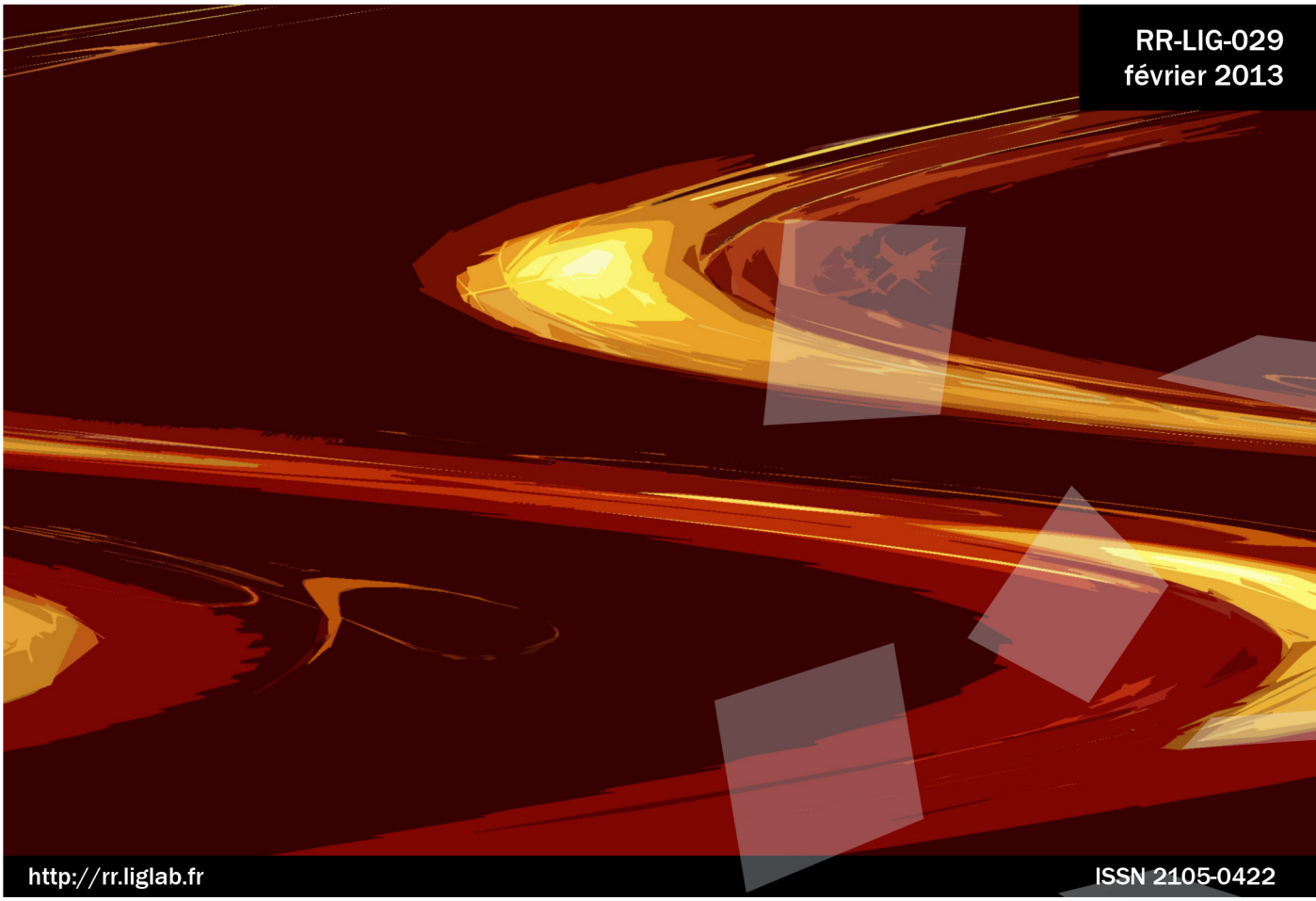
Message Passing: A Case for Mixing Deep-Copy and Migration

Benoit Claudel, Fabienne Boyer, Noel de Palma, Olivier Gruber

Les rapports de recherche du LIG

Message Passing
A Case for Mixing Deep-Copy and Migration

Benoit CLAUDEL, Post doc, LIG, INRIA, France
Fabienne BOYER, Associate Professor, LIG, University of Grenoble (UJF), France
Noel DE PALMA, Professor, LIG, University of Grenoble (UJF), France
Olivier GRUBER, Professor, LIG, University of Grenoble(UJF), France



RR-LIG-029
février 2013

http://rr.liglab.fr
ISSN 2105-0422

L I G

Message Passing
A Case for Mixing Deep-Copy and Migration

Benoit Claudel[1] and Fabienne Boyer[2] and Noel De Palma[2] and Olivier Gruber[2]

[1] INRIA, France
[2] Université de Grenoble

Abstract. This paper proposes an ownership model that permits to combine both a deep-copy and migration semantics within one consistent message-oriented programming model for Java. We argue that both semantics are necessary to cover the complete spectrum of application patterns. We also argue that one memory isolation mechanism is enough to provide strict memory isolation (isolate style) and message-oriented isolation suited for concurrent programming (actor style). Our proposal combines strict memory isolation, modeled on Java isolates (JSR 121), and a message passing that can send unconstrained object graphs, either migrating or deep-copying them. To our knowledge, our ownership model is the only model that does not specialize classes for their use in messages and still enables fast migration of unconstrained object graphs. Our performances show that the technology is suitable for interpreted virtual machines and are strong evidence that it is also suited for high-performance virtual machine based on JIT compilation.

1 Introduction

Memory isolation is traditionally provided by operating systems through the concept of processes, whereby real memory is virtualized into virtual address spaces. Recently, the concept of *Software Isolated Processes* (SIPs) has gained popularity amongst researchers. The idea is to rely on software mechanisms rather than hardware mechanisms to enforce memory isolation; the goal being to provide fine-grain memory isolation. To reach that goal, a large family of systems [12,17,5,15,6,18] rely on virtualization and more specifically on an object-oriented safe instruction set. By safe, we mean that memory references cannot be forged. The Java Virtual Machine (JVM) from Sun MicroSystems or the Common Language Runtime (CLR) from Microsoft are two examples of such virtual machines. Both provide an object-oriented instruction set, where object references are capabilities—an object is only reachable to those given its reference to.

The safe instruction set must be completed by a safe Inter Process Communication. A safe IPC, based on message passing between two software-isolated processes, maintains memory isolation. Current approaches [5,15,6,18] maintains memory isolation through deep copy, thereby preventing the creation of cross-boundary references, but for the controlled references on message queues. In

the Java world, the reference is the isolation API (JSR 121) first introduced in the Multitasking Virtual Machine (MVM) [5]. The MVM introduces the *isolate* and *link* concepts to Java. An isolate, constructed as an instance of the Isolate class, ensures a strict isolation between applications or software components. Furthermore, isolates guarantee a clean and robust termination, leaving the Java Runtime Environment in a consistent state. A Link provides a low-level communication layer designed for synchronous message passing. In [15], the message passing between isolates has been extended with a Cross-Isolate Method Invocation (XIMI), modeled on regular RMI.

An essential design point regarding isolation is the message passing model and if it preserves the object-oriented programming model or not. The isolate model and many others rely on a costly deep copy because it is the only existing existing mechanism that fully respects the object-oriented programming model. Some other solutions [12,17] argue for message passing based on a more efficient migration mechanism, but they are limited to messages that are tree structures. Furthermore, such approaches rely on static analysis that requires the introduction of new keywords, annotations, or other changes to the type system. Hence, classes are specialized for their use within messages, not only hindering their reuse but often introducing *hidden copies* because of the impedance mistmach introduced between regular data structures and messages. This means that whenever an existing object-oriented structure needs sending, it has to be translated and therefore copied into a message format. Since object-orientation encourages encapsulation and reuse, most developers will be unaware of the implementation nature of the data structures they use. Not knowing if internally they are graphs or trees, developers will be forced into translating most data structure they send.

In this paper, we propose a message passing model that combines the best of both worlds. The first contribution of this work is an ownership model based on first reachibility that provides the necessary knowledge of which objects are part of a message and which are not. The second contribution is a message framework based on our ownership model that allows to either migrate or deep-copy messages containing unconstrained object graphs. We argue that both semantics must be available to support the full range of application patterns. The third contribution of this paper is a simple and efficient design for message passing in the context of strict memory isolation.

Our proposal combines the isolate concept that provides lightweight memory isolation and the actor concept that provides featherweight concurrency based on message passing. In our proposal, each actor has its own isolate. Communications between isolates solely happen through message passing. Each actor has one or more message queues but one message queue belongs to only one actor. Actors are the unit of concurrency with one logical thread per actor, reacting to delivered messages, Erlang style. Each reaction runs to completion, following a traditional event-driven programming style. Logical threads carrying out reactions are multiplexed onto kernel threads.

Messages may contain unconstrained object graphs, captured by our ownership model. Our ownership model is based on first reachibility from roots of

ownership. Newly created objects are created free, that is, without an owner. Roots of ownership are actor objects and message objects, that is, an actor or a message is created as owning itself. The first time a free object becomes reachable from a owned object, the ownership is propagated. Notice that ownership propagation is therefore transitive. Hence, by the time a message has been constructed and is about to be sent, all the objects that must be sent along with the message are owned by that message.

Messages are always sent asynchronously, with either a deep-copy or a migration semantics. Both semantics are natively implemented in the Jam virtual machine (JamVM [14]), an open-source Java Virtual Machine with a state of the art threaded interpreted but no Just-In-Time (JIT) compilation. However, our technology is not interpreter specific in any way, adatping it to a JIT-based JVM requires changing the JIT to include read and write barriers for the few bytecodes manipulating object references. This barrier requirement is the downside of our proposal since the associated overhead is paid by all applications, that they use message passing or not. It is therefore paramount that the barrier overhead be low. We evaluated our barrier overhead on the SpecJVM98 benchmark, a benchmark that does not use message passing at all. Our overhead varies between 1.2% and 3.9%, with an average at 2.9%. We consider this overhead quite acceptable considering the robustness benefits of actor-based programming and our preserved ability to migrate unconstrained graphs. Furthermore, the presence of a JIT helps lowering even further the overhead of both read and write barriers. As reported in [2], the overhead of barriers is less than 6% and typically around 2% in high-performance virtual machines.

Our deep copy has comparable performance to other optimized deep-copy approaches such as MVM [15] and JX operating system [6]. It achieves an eight-fold improvement over standard RMI in the local case, that is, between actors within the same address space. We further optimized our message passing between actors spread across address spaces when micro-kernel technology is available. Tailoring our message passing to use the low-level IPC of microkernels, we are able to maintain an four-fold improvement across address spaces compared to RMI in the local case.

Our migration consistently beats our deep copy, whenever a migration semantics is appropriate. Our migration overhead on message-intensive applications remains between 2.1% and 7.6% of regular Java, that is, the same Java program without isolation. This is to compare to the overhead of a deep-copy approach that already suffers an overhead over 40% for medium-size messages such as a linked list of one hundred small objects.

This paper is structured as follows. In section 2, we introduce our programming model based on actors and isolates. In section 3, we discuss the details of our ownership mechanism. In section 4, we present our object migration. In section 5, we presents experimental results. In section 6, we discuss related work. We finally conclude in section 7.

2 Strictly-isolated Actors

Actors fulfill a dual purpose. One is to guarantee strict memory isolation, within a single Java Virtual Machine (JVM). To achieve this, an actor is created within its own isolate. Our isolates are semantically equivalent to MVM isolates [5]. Therefore, actors can be dynamically created, started, and stopped. When an actor is stopped, its isolate is terminated. Since isolate termination is guaranteed to leave the JVM in a consistent state, actor termination enjoys the same guarantee.

```
public class Actor {
 public Actor(String args[]);
 public void start(Queue queues[]);
 public void stop();
 public void react(Message msg);
}
```

To protect actors against malignous interference [15], actors have disjoint object graphs, sharing being completely prohibited but for controlled message queues. In particular, actors do not share class objects, meaning that each actor has its own version of static variables for each class it uses. Since we do not rely on any static analysis at compile or load time, we support dynamic class loading through class loaders, like any regular JVM or the MVM. For efficiency, the JVM implements as much sharing as possible across actors and the classes they uses. In particular, the low-level code of methods is shared. Hence, actors, like the MVM isolates, are lightweight isolation domains that share dynamically loaded code.

```
final public class Queue {
  public Queue(Actor actor);
  public void migrate(Message msg, long delay);
  public void copy(Message msg, long delay);
}

public class Message {
  public Message();
  public Queue getQueue();
  public int addObject(Object object);
  public Object getObjectAt(int index);
  public Queue getQueueAt(int index);
}
```

Message queues provide one-way communication channels between actors, as illustrated by the definition of the corresponding classes above. Notice that a message queue belongs to one and only actor but an actor may contain one or more message queues. A message is an ordered list of objects that can be either deep-copied or migrate to a message queue. When a message is received on a queue, it is delivered to the corresponding actor through its *react()* method.

Listing 1 provides an example of a simple time server. Notice that we have adopted a callback-oriented style, typical of Finite State Machines. Per actor, we cooperatively schedule message reactions on one logical thread. When scheduled, an invocation of the *react()* method on an actor runs to completion. For featherweight actors, we felt it was important to avoid the overhead and complexities associated with Java monitors and thread race conditions. Identically to

```
public class Client extends Isolate {
 String clientName;
 Date date;
 Queue wakeupQueue = new Queue();
 Queue replyQueue = new Queue();
 Queue serverQueue;

 public void start(Queue args[]) {
   serverQueue = args[0];
 }
 public void react(Message msg) {
   Queue queue = msg.getQueue();
   if (queue==wakeupQueue) {
     Message msg = new Message();
     msg.addObject(clientName.clone());
     msg.addObject(replyQueue);
     serverQueue.migrate(msg);
     msg = new Message();
     // check time again in one minute
     wakeupQueue.migrate(msg,60000);
   } else if (queue==replyQueue) {
     date = (Date)msg.getObjectAt(1);
   }
 }
}
```

```
public class TimeServer extends Isolate {
 String serverName;
 Queue serverQueue = new Queue();

 public void start(Queue args[]) {
   ...
 }
 public void react(Message msg) {
   long time = System.currentTimeMillis();
   String clientName = msg.getObjectAt(0);
   // log client request
   log(clientName,time);
   Queue replyQueue;
   // extract reply queue
   replyQueue = msg.getQueueAt(1);
   // send back the current time
   msg = new Message();
   msg.addObject(serverName.clone());
   msg.addObject(new Date(time));
   replyQueue.migrate(msg);
 }

 ...
}
```

Fig. 1. Time Server Example

Kilim [17], we argue that better multi-core scalability is obtained by multiplexing logical threads on kernel threads.

3 Ownership

Before a message can be migrated, the very first challenge is to know which objects need to migrate. Rather than splitting the type system and/or requiring source annotation like [12,17], we advocate a new approach based on dynamic ownership, which we claim is rather natural for object-oriented developers. Our ownership is a runtime technology, totally dynamic and entirely automated. When applied to messages, it unambiguously tells which objects belong to which message.

Our ownership is defined as follows. Within each actor, new objects are created free and remain free until they are owned. Ownership propagates by reachability through references. A free object becomes owned as soon as it becomes reachable from a owned object. In other words, when an object L, owned by an object O, first refers to a free object R, the ownership propagates from L to R. Once owned, an object remains owned by the same object until it becomes garbage.

It is important to point out that propagating ownership does not entail any object copy. It is similar to coloring but instead of propagating a color, we propagate the identity of the owner object. The simplest implementation is to have an extra hidden reference per object, called the *owner reference*. When the

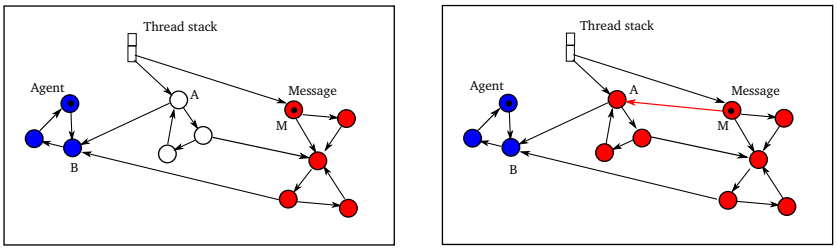


Fig. 2. Ownership Propagation

object is free, that owner reference is null. When the object is owned, that owner reference refers to the owner object.

We define only two roots of ownership, the Actor and Message classes. When an actor object is created, it is created as its own owner. The same is true for message objects. Therefore, any newly created object will remain free as long as it is used through local variables. However, as soon as it becomes transitively reachable from an actor object, it becomes owned by that actor. If it becomes transitively reachable from a message object first, it becomes owned by that message instead. Using first reachibility is quite natural in object-oriented programming where objects are created and first referenced from the data structure managing them and then shared with other data structures.

To implement our ownership, we introduce a write-barrier in the JVM on the instructions supporting the assignment of references, such as PUTFIELD and AASTORE bytecodes. PUTFIELD stores an object reference in an object field where AASTORE stores an object reference in an array object. Assuming a field f of an object L is assigned with the reference of an object R:

$$L.f = R; \tag{1}$$

The write barrier works as follows:

1. **The object L is free**, in which case there is no ownership to propagate. If the object R was free, it remains free. If it was owned, its owner remains unchanged.
2. **The objects L and R are owned**, nothing happens. If L and R are owned by the same owner, nothing needs to happen. If L and R are owned by two different owners, nothing happens since the owner of an object never changes through the lifetime of that object.
3. **Object L is owned and R is free**. When a free object is first referenced by an owned object, we propagate the ownership.

It is important to point that the JVM propagates ownership transitively, recursively applying the same write barrier on all encountered references. Indeed,

in the last case above, despite the fact that R is free, the graph of objects reachable from R might be composed of either owned or free objects. All reachable free objects will become owned; already owned objects will keep their owner.

Figure 2 illustrates an example of ownership propagation. On this left hand side, we have an actor with its logical thread and a message M, already created. There is also a free object A, reachable from the thread stack. Notice that the object A is an object graph, holding a reference to an object already owned by the actor. Going to the right hand side, we are adding the object A to the message M, which propagates the ownership of the message M onto the object A and those reachable from A. Notice that this propagation stops on objects already owned, such as the object B.

4 Migration

With the ownership in place, we know what to migrate when a message is sent. This section discusses how we achieve migration, strictly preserving memory isolation although all actors hosted in a single virtual machine share a single object heap, managed by a single garbage collector. In other words, objects owned by different actors are intermixed within that single object heap, along with objects owned by messages. This means that actors do not have private memory pages or segments; the structure of the object heap is solely dictated by allocation and garbage collection considerations, not by isolation considerations. Therefore, preserving strict memory isolation in the presence of message passing requires to maintain the following two invariants:

- The sending actor does not retain references upon migrated objects.
- Migrated objects do not retain references in the sending actor.

A naive solution is to search for such references and nullify them. We would need to scan the thread stack of the sending actor as well as all free objects reachable from that stack—cutting references on objects owned by the message being sent. We would also need to scan all objects owned by the sending actor and apply the same cutting rule. Additionally, we need to scan the message for outgoing references. The most likely case is the finding of references of message queues within messages. Since we do not restrict the programming model, other references may be found as well and must also be cut. This is equivalent to a garbage collection at the granularity of the sending actor, which we call a *scan-and-cut process*. This scan-and-cut process needs to happen for each message send, which is obviously not a practical solution in the general case.

Existing appoaches for message passing avoid this scan-and-cut process as follows. With a deep-copy approach, the scan-and-cut process is unnecessary since objects are recursively copied across isolate boundaries, which cannot create cross-isolate references. The tradeoff, copy versus scan, works well for small messages for which it is cheaper to deep copy messages than scan-and-cut the entire actor sending that message. Unfortunately, the copy overhead grows rapidly as the size of messages grows. Furthermore, for message intensive applications,

the copy process stresses the garbage collector, adding to the global overhead of a deep-copy approach. In contrast, migration solutions achieve a zero-copy overhead and guarantee that the scan-and-cut process is unnecessary through static analysis. The price to pay here is that messages must be tree structures (no aliasing) and that hidden copies are likely.

Given our goals, migrate unconstrained graphs and high level concurrency through featherweight actors, we have to craft a new and efficient runtime solution. The key idea of our solution ressembles the *tail-recursion* optimization applied to message passing. Tail recursion can optimize out deep recursive calls on a stack if the recursive call happens to be the last instruction of each call. For message passing, if the send instruction happens to be the last instruction of a reaction, the stack is empty and there is no free object still alive within the sending actor. Therefore, with a *tail-send pattern*, the scan-and-cut process is unnecessary for the stack and free objects.

It is interesting to notice that a *tail-send pattern* does not have to impose that the send instruction be syntactically the last instruction. With asynchronous message passing, the send instruction may appear anywhere in the code and the actual sending may actually be delayed until the end of the reaction. This point is important to preserve a natural programming model, without compromising performance. This leaves us with optimizing the scan-and-cut process on objects owned by the sending actor, which can be done through the introduction of remembered and revocable indirections.

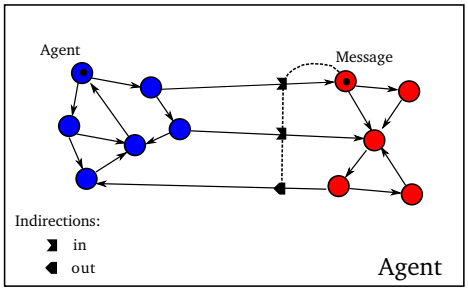


Fig. 3. Introducing Indirections

As depicted in Figure 3, we introduce a revokable indirection whenever a reference crosses an ownership boundary. Such indirections are introduced by a write barrier. In fact, we only need to slightly extend the write barrier already in place that propagates ownership. Each owner remembers both incoming and outgoing indirections, which enables a fast revocation process upon message passing. It is worth mentioning that special care is necessary for outgoing references on message queues since it is legal to embed a message queue reference within a message.

5 Evaluation

We implemented our actor technology by modifying the JamVM, an open-source Java Virtual Machine. We compiled the JamVM with GCC version 4.1.2, with O3 optimization turned on. All experiments were conducted on an Intel Core 2, dual core, at 2.2Ghz and 2GB of memory.

The JamVM has no Just-In-Time (JIT) compiler but rather relies on a threaded interpreter that rewrites Java bytecodes into a more effective internal representation, mostly saving branch instructions and most of the overhead of decoding operands. It is important to point out that there is nothing in our design and implementation that is specific to an interpreted approach. Using a virtual machine with a JIT, one can expect an improvement of the overall performance, but we argue that we would reach the same conclusions regarding message passing; a thesis that we will defend throughout this section.

5.1 Barrier overhead

The first facet of our design to evaluate is the intrinsic cost of our barriers (read and write). This is important since applications pay these barrier overheads even though they are not using message passing. The write barrier propagates ownership and create indirections across ownership boundaries; the read barrier eagerly skips indirections.

In the JamVM, we changed the rewriting of the four bytecodes concerned with reading or writing references to and from objects (including arrays). The instruction GETFIELD (resp. PUTFIELD) allows to read (resp. write) an instance variable. The instruction AALOAD (resp. AASTORE) allows to read (resp. write) an element of an array. The instruction GETSTATIC (resp. PUTSTATIC) allows to read (resp. write) class variable. The read and write barriers only need to be added when such instructions apply to a reference field.

To evaluate the barrier overhead when message passing is not used, we ran the SpecJVM98 with and without our barrier runtime checks. Figure 4 gives the overall results, showing that our barrier is inexpensive. The smallest overhead is as low as 1.2% and stay below 3.9%, with an average at 2.9%. Although counter intuitive, the fact that the JamVM is interpreted makes this overhead actually higher than it ought to be. First, Java compilers produce notoriously unoptimized code, expecting the JIT to optimize out redundant object accesses. Hence, an interpreter pays an extra barrier for every redundant object access. Also, JIT compilers typically reduce the overall number of barriers compared to object accesses. This is consistent with barrier overheads well below 6% and as low as 2% in high performance virtual machines as reported in [2].

5.2 Message-passing overhead

To measure the overhead of message passing, we use a straightforward client-server interaction where a client actor sends a message to a server actor that responds by recreating the very same message and sends it back. Throughout

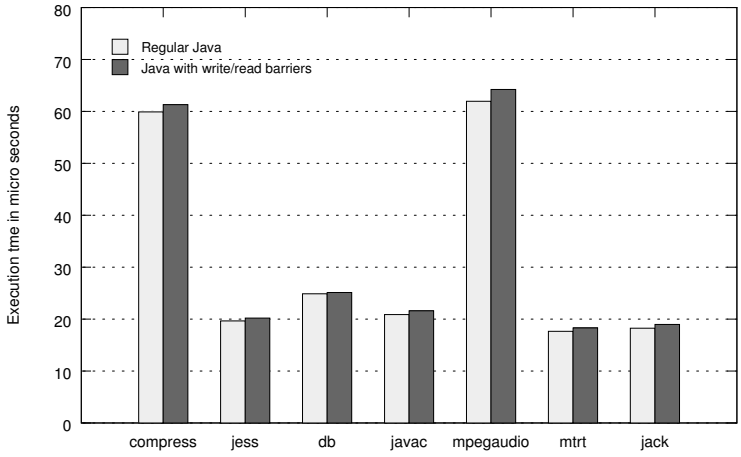


Fig. 4. Barrier Overhead on the SpecJVM98

the experiments, we vary the nature and size of messages, capturing the overhead of different facets of our design. For each shape of message, we ran eleven times the experiment with ten thousand client-server interactions per run. First, we averaged over the ten thousand interactions to get the time of sending a message, from which we computed the median over the values for the eleven runs. We considered the following shapes of messages:

- **Empty message**. This message is an empty message, that is, a single object, instance of the Message class.
- **Singleton message**. This message carries a single data object, so we are in fact sending two objects (the data one and the Message instance).
- **int[100] message**. This message carries an array of one hundred integers.
- **Object[50] and Object[100]**. This message carries an array of objects, instances of the class Object. We vary the length of the array, either fifty or one hundred.
- **List[50] and List[100]**. This message carries a graph, instance of the Java collection LinkedList (java.util package). We vary the size of the list, either fifty or one hundred elements, instances of the class Object. In the standard implementation of the LinkedList, the overall number of objects is about twice the size of the list.

Our first experiment compares both our migration and deep-copy technology against regular Java without memory isolation. By regular Java, we mean that we only removed the memory isolation property, everything else remaining the same. We still have two actors creating the same messages. Message passing still happens through the message queues but messages are simply passed by reference, with no extra check. In other words, the overhead for message creation as well as thread scheduling are the same.

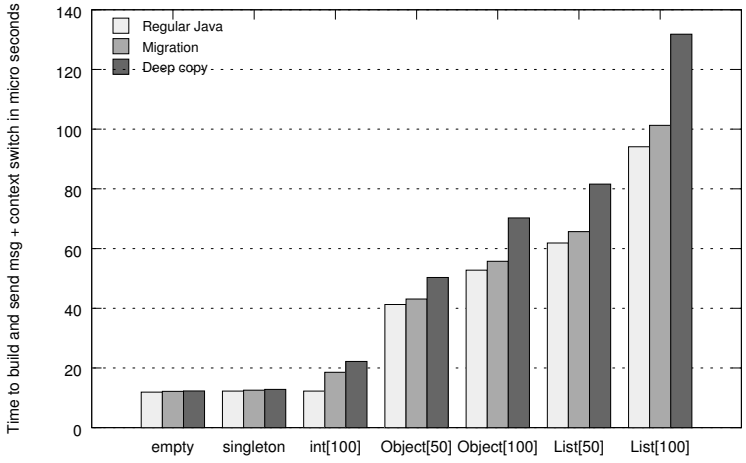


Fig. 5. Positioning our Isolate Technology

The results are given in Figure 5. All our experiments were ran with garbage collection disabled, thereby ignoring the extra pressure put on garbage collection by a deep copy approach. Our deep-copy mechanism is similar to the one advocated by JX kernel [7,6]. Similarly to both XIMI [15] and JX kernel [7,6], we report at least a ten-fold improvement over standard RMI in the local case, that is, messages exchanged across actors within the same virtual machine. We therefore consider that our deep-copy mechanism is state of the art.

Nevertheless, our migration mechanism consistently beats our deep-copy mechanism in this experiment. In fact, migration incurs only a small overhead compared to regular Java, between 2.1% and 7.6%. This performance delta is therefore the intrinsic measure of the overhead of our migration technology that enforces strict memory isolation. Figure 6 shows that overhead, isolated from the overhead of creating messages (but still including thread scheduling). The strength of migration is apparent with an almost constant overhead irrespective of the size of the message.

The exact overhead of migration is however dependent on the number of indirections to scan and cut; an overhead illustrated in Figure 7. In this experiment, we again used the client-server interaction discussed above, but we only exchanged linked list. Both sides (client and server actors) iterate over the list included in the message, simulating a search filter. Depending on how we iterate over the list, we create indirections are not. If we iterate over the list using only local variables, no indirections are created. If we iterate over the list using an instance field of an object owned by the actor, there are as many indirections created as there are elements in the list. As expected, migration introduces minimal overhead when used properly. It still outperforms a deep copy when poorly used,

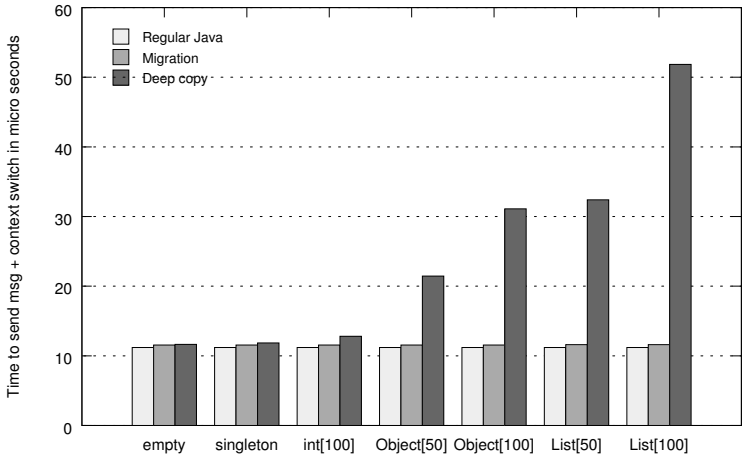


Fig. 6. Migration Microbenchmark

but the overhead can become rapidly significant if poor programming practices are used.

To pursue our comparison between migration and deep copy, we designed the next experiment to illustrate that migration and deep copy are in fact complementary semantics. The sweet spot of deep-copying messages is sending existing objects that are to be kept by the sending actor. The complementary sweet spot of migrating messages is when sending objects that allocated for the sole purpose of constructing messages. The experiment is designed as follows. We have a set of objects that we want to sort on some criteria and remove doublons of. The sorted objects are then to be sent using a linked list. One simple approach to achieve this has the following steps. A message is first created and a linked list added to it. A hash map is then created and filled with objects from the set, this step removing doublons. Then, the hash map is iterated over and its elements are sorted while being inserted in the linked list. Finally, the message is sent.

The important point with respect to comparing migration and deep copy is the origin of the object set. Figure 8 illustrates the results. No matter what the origin is, a deep-copy semantics can be used to send the message with a constant overhead of 9.5%. In contrast, the use of a migration semantics requires to know if the objects in the set are already owned by the sending actor or not. If they are free objects, migration can be used as usual with an overhead as small as 0.9%, clearly outperforming a deep-copy approach. However, if the objects are already owned, adding them to a message will not produce the expected result. Doing so would only create out-going indirections that would be nullified upon sending the message. The result would be sending a linked list with null references as elements. This is a typical case of hidden copies. To have the desired message passing, the already owned objects have to be cloned manually before they are

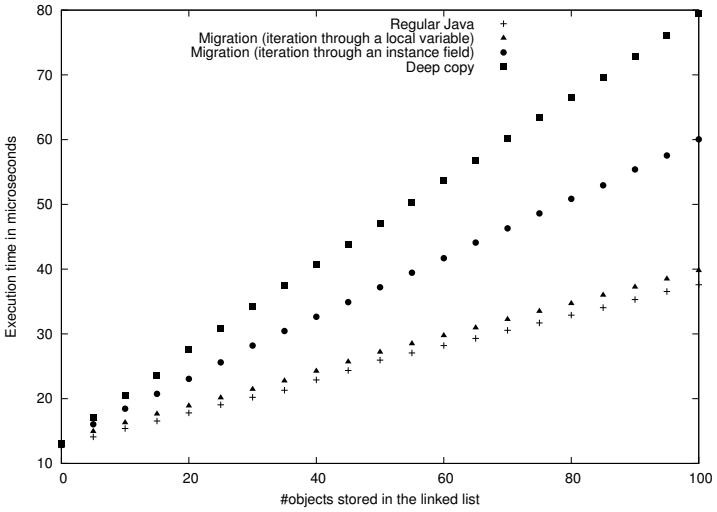


Fig. 7. Indirection Overhead

inserted in the message. This is not only cumbersome but results in poorer performance than a deep copy semantics.

One last point is important to discuss about this overall comparison of migration versus deep copy: would our conclusions remain valid in a high-performance JIT-enabled virtual machine? We argue that not only our conclusions remain valid but the importance of offering an efficient migration in order to limit the overhead of message passing increases. While the presence of a JIT improves the overall performance of Java code, it will have no effect on the effectiveness of message passing since it is implemented in C. Therefore, the overhead of message passing, as a percentage, will increase as the overall execution time of Java code decreases. This only makes the issue of adopting a migration semantics whenever meaningful more pressing.

5.3 Cross-address-space overhead

Additionally to evaluate messaging between actors within a single address space, we also evaluated messaging between actors across address spaces. In this case, the usual approach is to use standard RMI over standard sockets. To maintain better overall performance, in an homogeneous environment, we experimented with optimizing our deep-copy mechanism using the efficient low-level IPC provided by microkernels.

Figure 9 depicts our architecture on the XNU-1228.15.4 kernel that is part of the Mac OS X, introducing how we map actors and Java virtual machines onto tasks and threads. We map one JVM per kernel task, multiplexing actors per JVM and therefore per task. All actors hosted by one JVM share the same

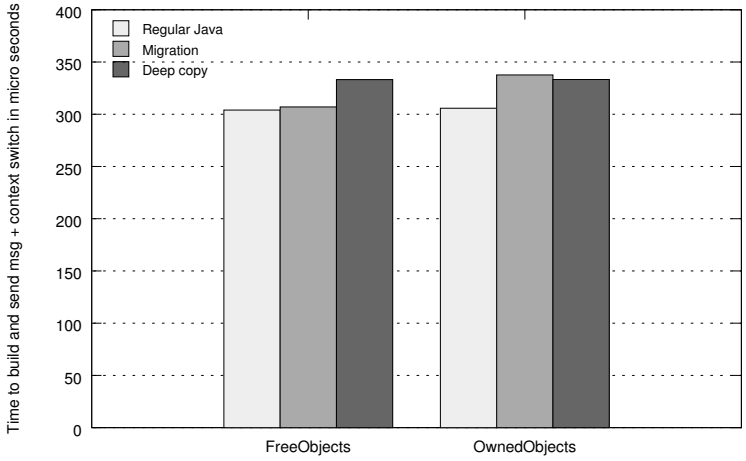


Fig. 8. Migration versus Deep Copy

object heap and the same garbage collector. We use one single kernel thread per task, using it to multiplex the reactions of actors.

Of course, we expected lower performance in the inter-task case than in the intra-task case; nevertheless we expected to be able to leverage the micro-kernel IPC and deliver better performance than standard RMI. Our results, given in Figure 10, confirm our intuition. While RMI is about ten times slower than a deep copy [15], our messaging across a task boundary is a little over twice the cost of a deep copy—so about five time faster than local RMI. The benchmark is exactly the same as the one underlying the results given in Figure 5, adding the inter-task performance figures.

The ratio between the deep-copy messaging and the inter-task messaging is easily explained. If we consider the last figures, we send by deep copy the linked list of one hundred elements in 131.80 microseconds, while we send the same message across tasks in 285.35 microseconds, with a ratio of 2.16. When sending across tasks, we actually incur two deep copies. The first deep copy happens on the sender side, when we deep copy the objects owned by the message into the buffer for the micro-kernel IPC. The second deep copy happens on the receiving side, when we deep copy out of the buffer into the object heap of the receiving task. The remaining time, above these two copies, is the time it takes for the micro-kernel to actually make the IPC happen.

Our implementation is a straight-forward use of the micro-kernel IPC. We use one single IPC per message we send across two tasks. Each IPC is composed of two out-of-band buffers. One buffer receives the deep-copied objects owned by the message we are sending. The other buffer receives the class names of these objects. Using two separate buffers allow us to send each class name only once,

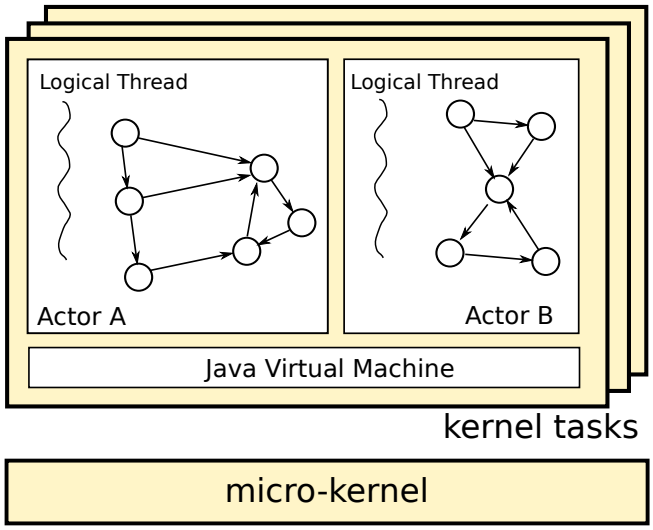


Fig. 9. SIPs over Tasks

irrespective of how many instances of actually deep-copied. Sending the class names is mandatory to know how to recreate the objects on the receiving side.

6 Related Work

This section describes related systems that provide memory isolation through software mechanisms. Three main categories may be considered. The first category leaves the Java type system untouched, either relying on Java serialization ([16,8,18]) or on deep copy mechanisms ([4,5,7,6]). Compared to these solutions, our work improves data exchanges through a zero-copy mechanism that preserves the isolation property. The second category ([9,1]) extends the type system, de facto limiting class reuse as well as changing developers' habits. In contrast, our solution does not requires any change to the type system. The third category ([12,13,11,17]) relies on static type checking to improve the efficiency of inter-domain communications. However, unlike our solution, these systems are limited to sending tree-based messages.

6.1 Copy-based solutions

The isolate API [16] allows to define Java programs that can run isolated in the same JVM. An isolate can be associated with one or more Java threads. Isolates communicate by message or remote method invokation (RMI). Objects are passed across isolate boundaries by serialization. The MVM[4,5] is an extension of the JVM that integrates the notion of isolate. The particularity of the MVM

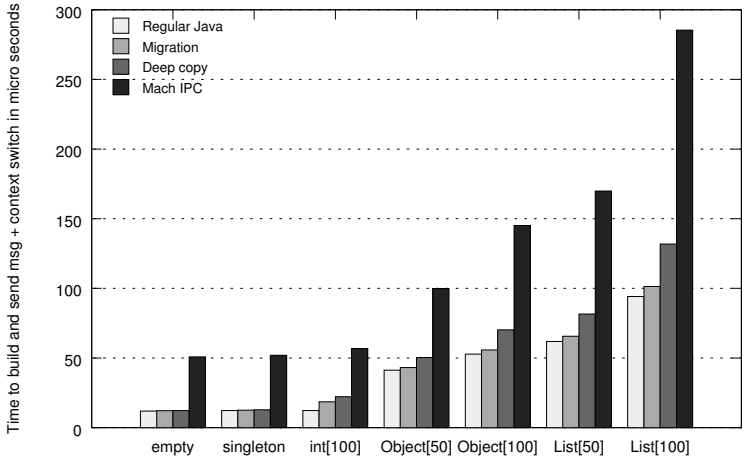


Fig. 10. Cross-Task Messaging

is to provide a cross-isolate method invokation (XIMI [15]) that passes objects by deep copy rather than by serialization.

The JX system [7,6] is a Single Address Space Operating System (SASOS [3,10]). The notion of domain represents the unit of concurency. Domains are written in Java and rely on the language safety to ensure memory isolation. Each domain has its own heap, garbage collector, and threads. Domains communicate through portals which are the only way to call methods on remote objects. Portals are the only objects passed by reference, all other data are passed by-value using a deep-copy mechanism similar to the one used by the MVM.

In the case of J-Kernel[8,18] the protection system is based on protection domains, software capabilities, and inter-domain communications. A protection domain constitutes a closed memory space with its own classloader and threads. Multiple protection domains run inside the same JVM process. Software capabilities are used to control object sharing between domains. A capability is the only way to access a resource and can be revoked at anytime by the system. The use of a revoked capability inside a domain throws an exception. Capability protection is also based on the Java language safety. When a domain terminates, all its capabilities are revoked and all its allocated memory is freed. Inter-domain communications are achieved through method calls on capabilities. Capabilities are passed by reference whereas other objects are passed by value using the Java serialization. J-Kernel does not require any modifications of the JVM.

6.2 Type-based Solutions

The Luna system[9] is similar to J-Kernel except that it extends the Java Type system with the concept of remote pointer instead of J-Kernel capabilities. This solution improves inter-domain comunication as it enables protection domain to

share directly well-identified objects in the system. To ensure domain isolation, a local pointer cannot be assigned in the field corresponding to a remote objects or used during a remote method invokation. Remote pointer can be revoked at any time by the system. However with this solution, shared objects must have been identified by the programmer (e.g., `String` for a local type and `String^` for a shared type). This specializes classes and thereby limits their reuse. It is interesting to note that shared types do not imply the exclusive ownership of shared objects.

KaffeOS[1] extends the JVM with the notion of processes and level of priviledges (supervisor and user). In particular each process owns its private heap and cannot reference objects inside other heaps. To ensure this property, KaffeOS uses the notion of *write barrier*[19]. Each time a new reference is written in a heap, the runtime check if it is authorized and throws an exception if not. As J-Kernel, each process has its own classloader and garbage collector. KaffeOS processes can communicate through shared heaps. This particular kind of heap contains only system classes and specific shared classes. A shared class must be part of the *shared.** package and its instances, called shared objects, cannot reference non-shared objects inside the private heaps of processes. The use of a special *shared package* is a serious limitation, unnecessarily fragmenting the type system—a simple String or Java collection may therefore not be shared.

6.3 Static checking for zero-copy messaging

Singularity[12,13,11] is composed of a microkernel and a set of SIPs that run in a single address space (SASOS design). A SIP may contain multiple threads. Software isolation is based on a safe programming language, *Sing#* that extends *C#*. A SIP is associated with a set of memory pages containing code and data. SIPs are closed object spaces, they cannot share writable memory with other SIPs and the code within a SIP is sealed at execution time. Communications occur using a message passing mechanism through *channels*. Messages and communication protocols exchanged through *channels* are specified using statically verifiable *contracts*. Data exchanges go through a special memory zone called the exchange heap that allows a safe-zero copy exchange of tree-like data structure. Unlike our solution, Singularity induces no runtime barriers since it is based on static checking. However only tree-shape objects can be exchanged between SIPs which is not the case with our solution.

Kilim[17] is a Java framework that provides the notion of Actors. Actors only communicate through message passing, with isolation being statically enforced. Messages are limited to tree-shaped data structures that can only contain primitive values, messages, and arrays of primitive values or messages. In Kilim, a message is reachable from only one field at any point in time and therefore belongs to a unique actor. Static analysis enforces that a sender of messages does not retain references to objects within sent messages. From static analysis, Kilim weaves at load-time the necessary code to ensure safe manipulation of messages. Therefore, Kilim does not require any modifications of the JVM.

7 Conclusion

This paper proposed an ownership model that permits to combine both a deep-copy and migration semantics within one consistent message-oriented programming model. We made the case that both migration and deep-copy semantics are necessary to cover application needs. We also argued that it is crucial to preserve the object-oriented programming model to avoid hindering class reuse and avoid hidden copies. To our knowledge, our ownership model is the only model that does not specialize classes for their use in messages and still enables fast migration of unconstrained object graphs. Our performances show that the technology is suitable for interpreted virtual machines and are strong evidence that it is also suited for high-performance virtual machine based on JIT compilation.

Acknowledgement: We wish to thank Laurent Daynès from Sun Research for his comments on this work and valuable insights on the MVM and XIMI.

References

1. Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, California, October 2000.
2. Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 143–151, New York, NY, USA, 2004. ACM.
3. Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *ACM Transactions on Computer Systems (TOCS)*, 12(4):271–307, 1994.
4. Grzegorz Czajkowski. Application isolation in the Java Virtual Machine. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 354–366, October 2000.
5. Grzegorz Czajkowski and Laurent Daynes. Multitasking without compromise: a Virtual Machine Evolution. *ACM SIGPLAN Notices*, 36(11):125–138, 2001.
6. Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, Monterey, California, June 2002.
7. Michael Golm, Jürgen Kleinöder, and Frank Bellosa. Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, pages 3–8, Elmau/Oberbayern, Germany, May 2001.
8. Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998.
9. Chris Hawblitzel and Thorsten von Eicken. Luna: a Flexible Java Protection System. *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 391 – 401, December 2002.

10. Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi Single-Address-Space Operating System. *Software: Practice and Experience*, 28(9):901–928, 1998.
11. Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the ACM EuroSys Conference*, pages 341–354, Lisbon, Portugal, March 2007.
12. Galen Hunt, James Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Redmond, Washington, October 2005.
13. Galen Hunt, James Larus, David Tarditi, and Ted Wobber. Broad New OS Research: Challenges and Opportunities. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 85–90, Santa Fe, New Mexico, June 2005.
14. JamVM: A Compact Java Virtual Machine. `http://jamvm.sourceforge.net/`.
15. Krzysztof Palacz, Grzegorz Czajkowski, Laurent Daynes, and Jan Vitek. Incommunicado: Efficient Communication for Isolates. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 262–274, November 2002.
16. Java Community Process. Application Isolation API Specification. `http://jcp.org/en/jsr/detail?id=121`.
17. Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java (A Million Actors, Safe Zero-Copy Communication). In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 104–128, Paphos, Cyprus, July 2008.
18. Thorsten von Eicken, Chi chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 369–393, 1999.
19. Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, Saint-Malo, France, September 1992.


L I G