An Efficient Actor Name Management Architecture to Support Encapsulation, Distributed Execution and Strong Mobility

Rajesh K Karmani, Amin Shali, Gul Agha

University of Illinois at Urbana-Champaign

Abstract. In order to address the problem of programming multicore architectures, at least 15 implementations of the *Actor model* have been proposed in the past couple of years. Most of these implementations are in the form of libraries (written in Java, Ruby, Python, Scala, Squeak, C++ among others) which provide runtime support for actors. An issue is that these runtime libraries do not support *distributed execution* and *strong mobility*. While this reduces execution overhead on shared memory multicore computers, we argue that these actor properties are necessary for programming *scalable* multicore architectures. Many implementations have also compromised on *enforcing* one of the key requirements of *actor encapsulation*, namely, that an actor cannot directly access methods (or state) of another actor but may only send messages to it. We analyze the costs of supporting encapsulation, distributed execution and strong mobility in ActorFoundry, a Java library for actors, and show that the major overhead is the result of a naive implementation of location independent *actor names*. We then describe techniques to improve the efficiency of distributed actor name management. Results of our experiments demonstrate that, using these techniques, actor properties can be efficiently supported in libraries. We conclude by discussing the implications of name management for garbage collection and resource discovery.

1 Introduction

In the Actor model of programming [**?**], systems comprise of concurrent, autonomous entities called actors which do not share state. Hence, actors are inherently free from low-level data races. Moreover, access to local state only increases the locality of reference.

The synchronization primitive in the Actor model is asynchronous message-passing. While waiting for a lock, threads occupy system resources such as a native stack and possibly other locks. In contrast, actors waiting for a message do not hold any system resources. Hence asynchronous message-passing is less error-prone than locks. Moreover, the Actor model is an inherently concurrent model of programming. Hence it is widely considered to be a promising approach towards programming scalable multi-core architectures.

Some languages based on the Actor model that are currently being used include Erlang [**?**], SALSA [**?**], Ptolemy [**?**], E language [**?**]. Additionally there are various Actor libraries written on top of existing languages like Python (Parley [?], Stackless python), Ruby (RevActor, Dramatis, Stage [?]), C++ (Theron), Java (Kilim [?], ActorFoundry [?], Actor Architecture [?], JavAct [?], Jetlang, Jsasb, Actors Guild), Scala [?], and .NET (Microsoft's Asynchronous Agents, Retlang).

While the above-mentioned languages and libraries support concurrent entities and asynchronous message-passing, many of them overlook some key semantic properties of the Actor model of programming. These properties include encapsulation, distributed execution and strong mobility. Encapsulation is important for reasoning about safety properties. It also enables modular analyis and higher software maintainability. Modular reasoning allows composition of software components which in turn enables building large-scale open systems. Distributed execution and strong mobility facilitates load-balancing and fault-tolerance. Also, support for distribution allows a uniform programming model for multicores as well as the Internet. We would present arguments that support for distributed execution is important for programming scalable multicore architectures.

There is a debate going on whether the right model to program multicore architectures is a shared memory model or through message-passing. Due to the desirable properties of data race freedom and aynschrony, we believe that the right model is Actor model and hence, we belong to the message-passing camp. More than that, we believe that even the design and implementation of tools and run-time for multicores should also assume a distributed memory model. As the number of cores on a chip grow and we move towards *manycore* architectures, the logic and circuitry required for cache-coherence does not scale. For the software, assumption of a shared data structure by the parallel tasks can create bottlenecks (due to locks and other forms of synchronization) which may prevent exploiting the parallelism of an application. Conceptually, even the hardware cache is a shared data structure, only that it is managed by the hardware [?].

Such observations and other related problems have prompted the design of services based run-time and operating systems [?,?,?], where the services communicated by message-passing. We would term these services as coarse-grained actors. Therefore, we believe that support for distributed execution is important for achieving scalable performance on manycore, heterogeneous architectures, specially for dynamic, irregular apps [?]. We believe that distributed execution and mobility enable a separation of concerns: "what" from "where".

A survey of existing JVM-based Actor frameworks including Scala and Kilim shows the lack of these properties though. We describe a naïve implementation of encapsulation, distributed execution and strong mobility in the context of ActorFoundry, an Actor library for Java. We present a detailed analysis of the various costs. We identify the key sources of inefficiency in implementing these properties: mapping of actors to Java threads and location-independent naming. The first problem is addressed effectively in Kilim [?] through a Continuations Passing Style (CPS) transform on Java bytecode. The transform exploits properties of actors such as lack of shared state and message-driven scheduling, and hence allows light-weight actors that can be efficiently migrated.

In this paper, we focus on supporting location-independent naming in Actor systems. We discuss why efficiently providing location-independent naming can be a challenge. The major considerations are space requirement and how naming affects garbage collection in Actor systems. The map of Actor name to its state (including its execution stack and the mailbox) can grow unbounded in the absence of an Actor garbage collector. As previous literature suggests [?], Actor garbage collection in distributed settings is an expensive task. Hence this presents an additional barrier towards providing support for distribution and strong mobility in Actor systems. Maps with weak references [?] allow such mapping with the facility that if the Actor name is not reachable outside the map, the JVM garbage collector can remove the entry from the map. Hence, JVM GC effectively doubles as the local Actor GC. But in the presence of distribution and mobility, the names can be shared with remote JVMs and `equals()` method for Actor name object does not check for reference equality. Hence weak maps can not be employed.

In order to mitigate these inefficiencies, we refer to the concepts of 'receptionists' and 'external actors' as described in [?]. We argue that only the receptionists actors need to be added to the tables. Hence the task of maintaining the tables faithfully follows the semantics of Actor operations that update the set of receptionists. We observe that while the semantics for message sends correctly update the set of receptionists and external actors, the semantics for creating and initializing new actors do not consider remote hosts. We provide the correct semantics for actor creation and initialization, as well as semantics for migration in distributed Actor systems.

Using these semantics, we describe and implement techniques that dramatically reduce the space-related inefficiencies. We briefly describe a proof of its correctness. We also describe how the JVM GC doubles as a local Actor garbage collector without weak maps using our approach. We also argue that generic clustering frameworks like Terracotta can not be more efficient since they do not exploit Actor semantics of encapsulation and locality. Evaluation of costs in the context of a JVM-based Actor library suggests that overhead of these semantics is negligible even in the case of a library implementation.

In the next three sections, we discuss the significance of each of encapsulation, distributed execution and strong mobility in Actor systems and what each of them requires from an Actor naming architecture. Next, we present the results from evaluation of a naïve implementation of these requirements. We identify the key reasons of inefficiency and propose optimizations that satisfy the requirements imposed by the properties. Next, we presents our results from evaluating our approach for a number of parallel and concurrent benchmarks. In the end, we conclude with related work in other Actor implementations and discuss future directions for research.

2 Encapsulation

Encapsulation is one of the core principles of object-oriented programming. Preserving encapsulation boundaries between objects facilitates reasoning about safety properties like memory safety, data race freedom, safe modification of object implementation. For example, Java is termed as a memory-safe language because it hides memory pointers behind object references which provide safe access to objects. This turns out to be important in order to preserve object semantics which require access to object state using well-defined interfaces.

In the context of the Actor model of programming, we group two core properties of the model under Actor encapsulation. The first one being that there is no shared state among actors. Secondly, an actor can access the state of another actor only by message-passing.

It is very tempting to ignore these semantics in the quest of an efficient implementation of Actor semantics. The temptation is stronger in the case of a library implementation compared to a language implementation. For example, in order to ensure that an actor is unable to access the state of another actor directly, a language may use an abstraction like mailbox address or a channel but use direct references in the compiled code for efficiency. This is similar to how Java uses object references to abstract away pointers. In an actor library, such abstractions (or indirection) have to be resolved at run-time which is more expensive.

For example, the following code demonstrates violation of one of the core Actor property due to lack of enforcement of Actor encapsulation in the Scala actors library. The main actor, in addition to sending an `enter()` message, executes `enter()` in its own stack. On a multi-threaded, shared memory implementation of the Actor model, this violates the actor semantics that an actor can only update its local state by processing one message at a time. This can lead to an inconsistent state and manifest into an error.

```
import scala.actors.Actor
import scala.actors.Actor._

object sempahore {
    class SemaphoreActor() extends Actor {
        var MAX = 1;
        var num = 0;

        def act() {
         loop { react {
                case ("enter") => {
               enter()
                }
                case ("other") => {
                    System.exit(0);
                }
        }}}
    def enter() {
     if (num  < MAX) {
      // critical section
      num = num + 1;
     }
    }
  }

  def main(args : Array[String]) : Unit = {
     var gate = new GateActor()
     gate.start
     gate ! "enter"
     gate.enter
  }
}
```

We believe that the two aspects of Actor encapsulation are important for writing large-scale Actor programs. It is difficult to provide semantics for and reason about Actor languages not guaranteeing these encapsulation boundaries. **TODO:** << Private conversation with Mirko regarding providing semantics for Scala, which does not guarantee these two aspects. >>

In this paper, we focus on how to enforce the property *in a library implementation* that an actor cannot directly access methods (or state) of another actor but may only send messages to it. The problem in this respect is that libraries written on top of existing object-oriented languages represent actors as objects, and the host language can not prevent an actor object from directly calling visible methods on another actor object. We believe that the correct approach is to employ indirect addressing through Actor name objects. A new creation and initialization procedure may be required that returns reference to an Actor name object instead of that to an Actor object. When an actor sends a message to the Actor name, the run-time resolves the name to a mailbox and delivers the message.

Fig. 1. Actor node using name table with weak references to support encapsulation.

A simple mechanism is to store the pair of Actor name and mailbox in a hash table. A problem with this approach is that the table can grow unbounded in the absence of an Actor garbage collection mechanism. For example, in Java's implementation of hash tables, each entry in the table would occupy 16 bytes, More importantly though, in the absence of an Actor garbage collection mechanism, an entry in the table means that the actor's state, mailbox and continuation will remain in the memory forever, even if no actor knows its name anymore.

Assuming no support for distributed execution, a more effective solution on platforms with object garbage collection facility, is to use tables with weak references [?]. Maps with weak references facilitate the following desirable characteristic: the object GC is allowed to remove the Actor name and the corresponding mailbox from the table if the Actor name is not reachable except through the weak reference from inside the table. Hence, the object GC effectively doubles as an Actor GC on a single shared-memory node. An Actor system constituting name tables with weak references is shown in Fig 1

3 Distributed Execution

Actor model is based on asynchronous message-passing between independent actors. Hence it provides a uniform model of computation for parallel as well as distributed computing. Apart from programming naturally distributed applications like P2P, sensor networks and Internet computing, distribution is important for parallel performance on scalable multicore architectures. Support for distributed execution is also imperative for mobility.

Distributed execution requires strong references inside the name tables. The reason being that Actors may not be known by any actor inside their node but actors at other hosts may have references to it. Hence, a garbage collector running on the local host may incorrectly remove an entry of an actor from the name table with weak references, and render future messages to the actor from remote actors undeliverable. An Actor system with name tables is shown in Fig 2.

Fig. 2. Actor node with name table to support distributed execution.

4 Strong Mobility

Proponents of Actor model emphasize transparent distribution of Actor programs, making the programs elegant and the task of the programmer less arduous. Actor systems should be able to provide facilities like load-balancing and fault-tolerance while keeping a uniform location-independent naming model for programmers. This in turn separates the concern of “what” from “where” and enables optimizations by the system based on run-time conditions.

Mobility is defined as the ability of computation to move across different nodes. In their seminal work, Fuggetta et al. classify mobility as either strong or weak [?]. Strong mobility is defined as the ability of a system to support movement of both code and execution state. Weak mobility, on the other hand, allows movement of only code. Initial state may be transferrable though in weak mobility.

At the system level weak mobility is essential for achieving load-balancing, facilitating fault-tolerance, reconfiguration. Previous work has shown that location transparency and mobility can be essential for achieving scalable performance, specially for dynamic, irregular applications over sparse data structures [?]. In such applications different stages may require a different distribution of computation. In other cases, the optimal distribution is dependent on run-time conditions such as data and work load. The ability to dynamically redistribute computation requires support for mobility.

We also believe that strong mobility enables declarative exploitation of heterogeneous system resources such as GPUs, DSPs, other task-specific accelerators, high clock frequency cores, (when augmented with discovery services). It leads to elegant programming and simpler control flow.

```
actor MatrixMultiply {
  message start(int N) {
   // Random generation of A, B

   migrate("gpu");

   int [][] C = new int [N][N];
   create <N * N> MultiplyActor()<-start(A, B, C);

   migrate("cpu");
   // print C

  }
}

actor MultiplyActor {
  int id;
  message multiply (int [][] A, int [][] B, int [][] C) {
    int i = id % N, j = id / N;
    C[i][j] = A[i][j] * B[i][j];
  }
}
```

It can also express streaming-code algorithms where the programmer wishes to move code rather than data. In combination with data discovery services, mobile actors can process data in-place. This may be preferable due to privacy reasons or that it is more efficient to move code rather than data. This is quite similar to pipeline processing except that the conceptual model is reversed: transformation actors processing the statically residing data one after the other. Moreover, this model by definition has ownership transfer semantics for messages, which as discussed earlier can be implemented more efficiently.

Mobility is quite natural to the Actor model due to modularity of control and encapsulation. Object-oriented languages may allow mobility at the level of objects but all thread stacks executing methods on the object need to be made aware of this migration. Moreover, when the stack frame require access to the object on a remote node, the execution stack is moved to the remote node, complete the execution of the frame and then migrated back to the original node [?].

A protocol for supporting mobility in Actor systems has been described in [?]. It requires Actor names to be mapped to be mapped to their possible host. Hence, in addition to name table for looking up the Actor state, an Actor system requires another name service that can lookup the host (physical address) of an Actor. An Actor system with name tables is shown in Fig 3.

Fig. 3. Actor node with name table and a name service to support mobility

5 Implementation

We will discuss Actors semantics in the context of a JVM-based Actor library called ActorFoundry v1.0. Actor Foundry v0.1.14 was originally designed and developed at the Open Systems Laboratory by Mark Astley along with Thomas Clausen and James Waldby around 1998-2000 [?]. The goal was to develop a modular Actor library for a new, upcoming object-oriented language called Java. Actor Foundry v0.1.14 has a simple, elegant model in which the control and mailbox are hidden away in the library while the programmer is concerned with an Actor's local state and behavior only **??**. To leverage the actor semantics, the programmer is provided with a small set of library methods as part of the Actor Foundry v0.1.14 API. This includes:

`send(actorAddress, message, args)` Sends an asynchronous message to the actor at specified address along with arguments. The argument message is a String which should match a method name in the target actor behavior, otherwise an exception is thrown at run-time.

`call(actorAddress, message, args)` Sends an asynchronous message and waits for a reply. The reply is also an asynchronous message which either contains the return value or simply an acknowledgement.

`create(node, behavior, args)` Creates a new actor with the given behavior at the specified node. The argument node is optional. In case it is not specified, the actor is created locally. The arguments are passed to the constructor.

`destroy(note)` An explicit termination of the current actor. This is provided as an alternative to an automatic actor garbage collection mechanism.

`self()` Returns the address of the current actor.

`migrate(node)` Requests the run-time to schedule the current actor for migration to a remote node. Actor Foundry v0.1.14 only supports weak mobility, hence the actor is migrated at message processing boundaries only.

`cancelMigrate()` Cancels an earlier request for migration.

Behind the scenes, Actor Foundry v0.1.14 maps each actor onto a JVM thread (1:1 architecture). Messages are dispatched to actors by using the Java Reflection API. This can technically be termed as pattern-matching since the message string is matched to a method name at run-time and the method is selected based on the run-time type of arguments [?]. Though, it is certainly not as expressive as the pattern-matching in Erlang [?] and Scala [?]. Any Java object can be part of a message in Actor Foundry v0.1.14; the only restriction being that the object implements `java.lang.Serializable` interface. All message contents are sent by making a deep copy by using Java's Serialization and Deserialization mechanism.

Actor Foundry v0.1.14 also supports distributed execution of actor programs, location independence and weak mobility [?]. In addition, Actor Foundry v0.1.14 logs system and library events to the local file system.

This implementation faithfully preserves Actor semantics like fairness (as fair as the underlying JVM and OS scheduler), encapsulation, location independence and mobility. In order to gauge the performance of Actor Foundry v0.1.14, we implement a small benchmark called Threadring [?] in which 503 concurrent entities pass a token around in a ring 10 million times. Threadring provides a crude estimate of actor creation overhead and stress tests the message-passing and context switch.

For our experiments we use a Dell XPS laptop with Intel Core^{TM} 2 Duo CPU @2.40GHz, 4GB RAM and 3MB L2 cache. The software platform is Sun's Java^{TM} SE Runtime Environment 1.6.0 and Java HotSpot^{TM} Server VM 10.0 running on Linux 2.6.26. We set the JVM heap size to 256MB for all experiments unless specified. Upon running this benchmark, we find it takes about 695s to finish. In contrast, an Erlang implementation takes about 8s while a Java Thread implementation takes 63s. See Figure 4 for a full comparison. This

Fig. 4. Threadring Performance - Actor Foundry v0.1.14 compared with other concurrent languages and libraries

shows that a faithful but naïve implementation of Actor semantics can make the execution overhead of Actor programs extremely high, making Actor Foundry v0.1.14 feasible only for coarse-grained conccurent or distributed applications. We observe a similar execution overhead for SALSA and Actor Architecture. On the other hand, Kilim and Scala actors library perform an order of magnitude faster. JavaAct performs much better than Actor Foundry v0.1.14 but does not come close to either Kilim or Scala in terms of efficiency.

5.1 Continuations based Actors

Prior experience with ThAL language [?] suggests that continuations based actors would provide significant improvement in terms of creation as well as context switch overhead. We integrate Kilim's light-weight Task class and its bytecode post-processor ("weaver") into ActorFoundry v1.0. This transformation presents two challenges.

To begin with, the transform would not work when messages were dispatched using Java Reflection API. The reason being that the weaver is unable to transform Java library code and this prevented the continuations to be available in the actor code. To overcome this, we generate custom reflection for each actor behavior. It finds the matching method by comparing the message string to method's name and the type of message arguments to method's formal arguments' type. Once a match is found, the method is dispatched statically. As a pleasant side-effect, it is more efficient than Java Reflection.

Secondly, the transform also require introducing a scheduler for ActorFoundry v1.0 which is aware of cooperative, continuations-based actors. The scheduler employs a fixed number of JVM threads called worker threads. All worker threads currently share a common scheduler queue. Each worker thread dequeues an actor from the queue and call its continuation. Actors are assumed to be cooperative; an actor continues to run until it yields waiting for a message. In order to increase locality and reduce actor context switches, an actor can process multiple messages while being scheduled. On the other hand, this can cause starvation in the system. Scheduling is message-driven only. An actor is put on the scheduler queue if and only if it has a pending message.

With this implementation, the running time for Threadring comes down to about 267s.

5.2 Message-passing semantics

We profile the execution to identify further performance bottlenecks. Profiling makes it apparent that the deep copying of message contents is by far the biggest bottleneck. The faithful implementation of message-passing semantics means that message contents are deep-copied using Java's Serialization and Deserialization mechanism, even for immutable types. We disable deep-copying for some known immutable types. These included `java.lang.Integer`, `java.lang.String` and others. This further brings down the running time of Threadring to 30s. The reason behind this significant improvement is that actors in Threadring pass a token which is basically an integer representing the remaining number of passes.

We add two new methods to ActorFoundry v1.0 library: `sendByRef()` and `callByRef()` to overcome this inefficiency. These methods allow the programmer to explicitly declare transfer of ownership semantics for messages. This enables the run-time to implement such message efficiently on shared-memory systems by passing a reference.

5.3 Naïve Implementation

Distribution and Location-Independent Addressing In Actor Foundry v0.1.14, distributed execution is supported by means of a Transport layer. With its extensible architecture, either UDP or TCP based transport layer can be used. A node manager can communicate with another node manager by requesting its actor address through a lookup service called Yellow Page Service. The prerequisite is that the service is running at the target node at the specified port.

The client node can cache the manager's address locally. This address can now be used to send requests for creating actors remotely and migrating actors to the target node. This is where the power of location-independent addressing comes into play. Remote actor creation is a blocking operation that returns a location-transparent actor address. An actor on the client node can send send further messages to this actor on the remote node. It can also share this address with other actors and in turn they can send messages to this address without ever worrying about its physical location.

In case a local actor migrates to a remote node, other actors that know this actor's address can continue to send messages to this actor. This location-independent addressing is enabled by using a protocol based on name tables, similar to the one discussed in [**?**].

Mobility

5.4 Optimizations

6 Performance

Figure 5 compares the performance of optimized implementation of ActorFoundry with Scala, Kilim, SALSA and Actor Architecture. Figure 6 provides a similar comparison for Chameneos-redux benchmark. **TODO:** << Need to compare the size/memory footprint of the run-time, each actor and each message to understand cache-related effects. >>

Fig. 5. Threadring Performance

Fig. 6. Chameneos-redux Performance

We implement distributed versions of Threadring, Chameneos-redux, Naïve fibonacci and Red-Black ordering. We present the results of running these on a cluster of 1 to 4 nodes.

7 Related Work

8 Discussion and Future work

Discuss Erlang, ThaL and Terracotta here.

Erlang does not support strong mobility but has fault-tolerance aspect (node restart).

References

1. Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.
2. Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
3. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.
4. Edward A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, 2003.
5. The e language. *http://www.erights.org/elang*, 2000.
6. Jacob Lee. Parley. *http://osl.cs.uiuc.edu/parley/*, 2007.
7. Jonathan Sillito. Stage: exploring erlang style concurrency in ruby. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 33–40, New York, NY, USA, 2008. ACM.
8. S. Srinivasan and A. Mycroft. Kilim: Isolation typed actors for Java. In *Procedings if the European Conference on Object Oriented Programming (ECOOP)*, 2008.
9. Mark Astley. *The Actor Foundry: A Java-based Actor Programming Environment.* Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99.
10. Myeong-Wuk Jang. *The Actor Architecture Manual.* Department of Computer Science, University of Illinois at Urbana-Champaign, March 2004.
11. William Zwicky. Aj: A systems for buildings actors with java. Master's thesis, University of Illinois at Urbana-Champaign, 2008.
12. P. Haller and M. Odersky. Actors That Unify Threads and Events. *LECTURE NOTES IN COMPUTER SCIENCE*, 4467:171, 2007.
13. David Wentzlaff and Anant Agarwal. The case for a factored operating system (fos). Technical Report MIT-CSAIL-TR-2008-060, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, Oct 2008.
14. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.
15. Nicolas Zea, John Sartori, and Rakesh Kumar. Servo: a programming model for many-core computing. *SIGARCH Comput. Archit. News*, 36(2):28–37, 2008.
16. Rajendra Panwar and Gul Agha. A methodology for programming scalable architectures. In *Journal of Parallel and Distributed Computing, vol. 22 pp 479-487*, 1994.
17. Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. In *in Y. Bekkers and J. Cohen (editors), International Workshop on Memory Management, ACM SIGPLAN and INRIA, St. Malo, France, Lecture Notes in Computer Science, vol. 637, pp 134-148, Springer-Verlag, September*, 1992.

18. Ole Agesen and Alex Garthwaite. Efficient object sampling via weak references. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 121–126, New York, NY, USA, 2000. ACM.
19. Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(01):1–72, 1997.
20. Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
21. T. Walsh, P. Nixon, and S. Dobson. As strong as possible mobility: An Architecture for stateful object migration on the Internet. *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France)*, 2000.
22. WooYoung Kim and Gul Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, New York, NY, USA, 1995. ACM.
23. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM.
24. Open Source. The computer language benchmarks game. *http://shootout.alioth.debian.org/*, 2004-2008.
25. WooYoung Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.