Compilation of a Highly Parallel Actor-Based Language

WooYoung Kim*and Gul Agha†
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {wooyoung | agha}@cs.uiuc.edu

June 23, 1992

Abstract

HAL incorporates a number of high-level language constructs such as the incremental specialization of synchronization constraints to maintain the consistency of actors at run-time, the inheritance of both code and synchronization constraints, and limited reflective capabilities to customize the system with respect to fault tolerance. This paper describes some issues in compiling HAL, and, in particular, three source level transformations used by the compiler for HAL. Two of the transformations translate RPC-style message sending into asynchronous message sending. The third transformation performs code motion to optimize the implementation of replacement behavior. This optimization results in the reduction of object code size as well as execution time.

Keywords: Actor, concurrency, synchronization constraint, inheritance, optimization

*This author is sponsored by a fellowship from Ministry of Education in Korea.

†The work described in this paper has been made possible by generous support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

1 Introduction

A number of efforts have been made to build parallelizing or vectorizing compilers which attempt to extract parallelism from code written in traditional sequential programming languages such as FORTRAN [26, 5, 27]. Others have concentrated on compiling specialized languages which are inherently concurrent, such as functional languages [18] and Data Flow [7]. Unfortunately, such languages are inadequate to model concurrency in a state-based, nondeterministic world [1]. A third language paradigm consists of languages which explicitly manipulate parallelism. Some examples are CSP [9] and Occam [25]. However, CSP and Occam have an unavoidable limitation in the sense that a process in these systems cannot create a new communication channel at run-time or pass it to another process. The configuration of the system is fixed at compile time resulting in a static topology. Our work is related to the last two paradigms. We use the Actor model [1] which unifies the functional and the object-based view of computation, supports parallelism explicitly and provides for a dynamic topology of computational agents. On the other hand, the Actor model differs from the model used in LINDA [11], which separates coordination from computation rather than treats them in a single framework.

We have developed a high-level programming language, called HAL. Our goals in designing HAL are as follows:

Generality: We want our programming language to be truly general purpose. Control parallelism as well as data parallelism should be expressed in HAL in a natural way. Shared objects should be easily described. Furthermore, HAL should be architecture independent to allow efficient execution on both shared and distributed memory machines.

Modularity and Abstraction: To simplify programming, HAL should support data and functional abstraction. In particular, users should be able to specify the abstraction (interface)

independently of its representation (implementation). Furthermore, HAL should support re-usability through language constructs such as inheritance, higher-order functions and reflection.

Efficient Execution: It should be possible to describe arbitrary applications in sufficient detail to support efficient execution on different concurrent architectures. In particular, the same application may require different resource management techniques on different concurrent architectures.

In designing HAL, we use a reflective Actor model to realize those goals. The current paper describes ongoing work on HAL and its compiler. Compiling HAL for efficient execution creates a number of distinct problems. We describe some of these problems, namely, resource management, inherent concurrency, synchronization constraints and distributed objects. These problems stem from the power and the flexibility inherent in actor-based languages.

Resource Management

HAL is designed in such a way that the code of a parallel algorithm can be described in an architecture independent way. However, for efficiency it is often essential to explicitly specify resource management policies for actor placement and load balancing. HAL allows the code for resource management and code for the application to be separately specified. Besides reducing code complexity, this allows both the code of application algorithms and the code for resource management strategies to be reused. The compiler must combine these two kinds of specifications for efficient execution (Section 2.7).

Inherent Concurrency

The Actor model supports internal concurrency as well as explicit parallelism [1]. The internal concurrency is inherently fine-grained. However, almost every conventional processor is too coarse grained to realize the fine-grained computation inherent in the internal concurrency in an efficient way. To achieve better performance, we must have a certain amount of run-time control over grain size. The internal concurrency makes it possible to pipeline behavior changes of an actor. However, the implementation of such pipelining on coarse grained processors carries memory management overhead. By not implementing such pipelining, we have obtained a good performance improvement on current generation medium-grained multicomputers. (Section 3.2).

Representation of method invocations in the form of functional expressions allows the simple specification of control in the program. Unfortunately, it can cause unnecessary loss of concurrency if a function caller blocks until it receives returned function value. The transformation of such representations into asynchronous message passing makes it possible to retain maximal concurrency in the program (Section 3.1).

Synchronization Constraints

In a CSP like model, communicating processes must be synchronized whenever they exchange messages with each other. The synchronization is accomplished through busy waits. Synchronous communication is expensive because that a sender cannot send a message until a receiver is ready to accept the message, causing loss of concurrency. In contrast, by using buffering accompanied with asynchronous message passing, we eliminate the need for extraneous synchronization, thereby, retaining the concurrency. Furthermore, asynchronous communication localizes the enforcement of synchronization constraints at the recipient. This is especially important since a requirement of fairness requires the re-evaluation of synchronization constraints with each message and state

change [14, 15].

Distributed Objects

If only one actor is responsible for processing all incoming communications to a large data structure, the actor will be a bottleneck. We avoid the single address bottleneck problem associated with the uniqueness of an actor's address by allowing concurrent access to a large data structure and locking only the relevant portion of the data structure. Our approach follows work on Concurrent Aggregates [12].

In the following section, we discuss some linguistic features of HAL with their semantics. Section 3 discusses the transformations of RPC style message sending and the transformation for code motion to optimize the implementation of replacement behavior. Some research related to our compiler is described in Section 4. The last section provides future research directions and concluding remarks.

2 HAL: A High-level Actor Language

We begin the section with a brief description of the Actor model. We then discusses the specification of synchronization constraints, inheritance, and the specification of replacement behavior. Finally, issues related to memory management, extensions to the asynchronous message passing primitive and reflection capability are addressed in that order.

2.1 The Actor Model

Actors are self contained, independent computational agents that communicate by asynchronous message passing [1]. An actor consists of its mail queue and behavior. It is identified by its unique mail address. The mail queue of an actor buffers incoming communications (i.e. messages). The

behavior of an actor specifies the action performed by an actor in response to a communication. An actor's state is defined by its *acquaintances* (actors whose mail addresses are known to the actor).

All computation in an actor system is carried out in response to communications sent to actors in the system. Specifically, an actor may perform three kinds of actions when it accepts a message:

- it may change its behavior.
- it may send more communications. Communication is asynchronous and point-to-point to an acquaintance. The sender may not be known, but the recipient must be an acquaintance of the sender. The delivery of a message is guaranteed after an arbitrary, but finite, delay (a fairness condition [14]).
- it may create more actors. These actors have their own unique mail addresses which are initially known only to their creator and possibly themselves.

The replacement behavior of an actor is specified through the `become` primitive. Whenever there is no executable `become` primitive in the thread of an actor computation, an identically behaving actor is assumed to be its replacement behavior (by default). Note that communications may contain mail addresses of actors; thus the interconnection topology of an actor system is dynamic.

Building on the basic actor execution model, we have designed a high-level programming language, named HAL [16]. HAL is a descendant of actor languages such as Acore [24], Rosette [30] and ABCL/R [33]. It provides abstractions which facilitate software development. We describe the constructs used for such abstractions below, emphasizing implementation issues. A more detailed description of the language constructs and their motivation may be found in [16, 17]. Note, however, that HAL is an evolving language. New high-level constructs are continually being developed and tested in order to explore ways of simplifying the task of parallel programming while improving

efficiency in execution.

2.2 Specification of Synchronization Constraints

Actor semantics does not require message order preservation. In order to protect the system as well as an actor from possible internal inconsistency due to message order nondeterminism, we need to be able to specify synchronization constraints on actors. Earlier work has established how synchronization constraints can be used for software pipelining [3]. In particular, such pipelining transforms some algorithms, such as the Cholesky Decomposition of a symmetric positive definite matrix, from unscalable algorithms into scalable ones.

We have adopted a philosophy of synchronization constraints proposed in [15]. All constraints are expressed as disabling restrictions. A disabling restriction means whenever any condition associated with a method evaluates to `TRUE`, the underlying system closes the entry point of the method and disables the method. All messages with the method name are not accepted until all conditions evaluate to `FALSE`. Use of disabling restrictions instead of enabling conditions [30, 6] ensures that the synchronization constraints which hold for a superclass also hold for the subclasses which are derived from the superclass. Therefore, programmers can reason about the subclasses in terms of the superclass [15].

Disabling restrictions are specified using the following syntax:

```
(disable <msg-expr> <expr>)

(all-except <msg-expr> <expr>)
```

where <msg-expr> specifies the method name on which the constraint is imposed. Users are allowed to associate more than one synchronization constraint with a method. <expr> is a function of acquaintances (i.e. actor's internal state) and the incoming communication. When <expr> evaluates to `TRUE`, the method specified in <msg-expr> is disabled. If a constraint is

specified with *all-except*, all other methods are disabled except one specified in <msg-expr> . The incoming messages with the method name are kept in the actor. When all conditions associated with a method evaluate to `FALSE`, its entry point is open and the pending messages may be processed.

A potentially significant source of inefficiency is the need to re-evaluate the constraint expressions of the pending messages that are kept in an actor. In the current implementation, whenever the actor state changes, all constraints associated with the pending messages are re-evaluated. To allow more efficient execution, we are currently experimenting with a scheme which requires the evaluation of only those constraints that depend on the acquaintances which have changed. Specifically, our scheme uses dynamic dependency lists between an actors acquaintance and constraints.

2.3 Inheritance

HAL has been designed to satisfy all of Wegner's qualifications to be classified as an object-oriented language [34] . Class variables are not supported since they imply shared data between actors, thus restricting their autonomy and distribution. HAL allows the inheritance of both code and synchronization constraints. As in Smalltalk [22], any method of the superclass of an actor may be redefined in the definition of the actor. Synchronization constraints, however, may only be strengthened, not weakened [15]. In this sense, inheritance can be viewed as a means of specialization. However, we are investigating whether it is beneficial to allow first-class constraints.

Figure 1 defines a set of stack classes which exemplify the possible usage of inheritance and synchronization constraints. The most basic actor class is `Stack` which has the methods to push and pop one element to and from the stack, respectively. This definition has a constraint `(> count 0)` on `pop` which must hold in order for a pop message to be processed.

The `Bounded-stack` class is a subclass of `Stack`. This class has a constraint on the absolute size of the stack; one cannot make the stack hold more than `max` elements at any time. Since we

```
(defActor Stack (stack count)
    (disable (pop) (< count 0))
    (method (push x)
        (update stack (cons x stack))
        (update count (+ count 1)))
    (method (pop entrypoint continuation)
        (send entrypoint continuation (car stack))
        (update stack (cdr stack))
        (update count (- count 1))))

(defActor Bounded-stack (max)
    (superclass Stack)
    (disable (push) (> count max)))

(defActor Pop2-stack
    (superclass Bounded-stack)
    (disable (pop2) (< count 1))
    (method (pop2 entrypoint continuation)
        (send entrypoint continuation (car stack) (car (cdr stack)))
        (update stack (cdr (cdr stack)))
        (update count (- count 2))))
```

Figure 1: Stack actor classes

have separated the specification of synchronization constraints from the code of the actual method, we only need to state the new constraint without having to redefine the `push` method. Note that more restrictions are imposed on the `push` method in `Bounded-stack` class than in `Stack` class.

Finally, we define the `Pop2-stack` class as a subclass of the `Bounded-stack` class with the ability to pop two elements out of the stack as an atomic action. This behavior is provided through the `pop2` method. Under our constraint specification scheme, the definition of both `pop` and `push` methods can be inherited from the `Stack` class into both of these subclasses. Also, the incremental specialization of synchronization constraints is allowed through inheritance [15].

2.4 State Change

Replacement behavior is computed by the `become` primitive in the Actor model. An actor can *become* an entirely different actor as a result of processing the `become` primitive. However, in many cases, replacement behavior involves only one or two acquaintance changes of an actor.

The `update` primitive gives us a convenient way to specify a replacement behavior with the change of one acquaintance. It has the following syntax:

```
(update <acquaintance-name> <expr>)
```

`update` obeys a single-assignment semantics: there may be only one `become` in any given method invocation. However, more than one `update` can appear if they do not *update* the same acquaintance. Since the effect of replacement behavior is invisible to the current computation of an actor, multiple `update`'s can be viewed as one `become` with several acquaintance changes.

2.5 Message-passing Mechanisms

Besides the asynchronous message sending provided in Actor semantics, HAL provides two more message sending primitives: `ssend` and `bsend`. The former is the message order preserving send primitive, or sequenced send. The latter is the RPC style send primitive akin to Acore's `ask` primitive [24]. Though they have some flavor of synchronous message sending, they are *not* synchronous in the sense that a sender does not need to know whether a receiver is ready to accept its message or not.

`ssend` gives a sender a way to control the sequence of actions which occur at a receiver. `bsend` is just like a function call so that the computation proceeds with the reply from the receiver. `ssend` is implemented by tagging the message sent and reordering messages, if needed, at the recipient's end. `bsend` can be used as a way to synchronize several coordinating actors. The implementation issues related to `bsend` are given in detail in Section 3.1.

2.6 Suicide

When executing a large actor program, many actors may be created as a result of computation. Some of them will never be used again after they have accepted certain communications. Once they process particular communications, they become garbage actors, wasting valuable resources until they are reclaimed by garbage collector. It is beneficial to reclaim the resources which belong to such actors as soon as they become garbage. `suicide` is the primitive which does exactly what we want. It is similar to that used in Cantor [8]. When an actor executes the `suicide` primitive, it frees all the resources which are allocated to it so that the resources can be reused. This primitive can be used by a programmer or the compiler. Readers can find an example for the latter case in Section 3.1.1. Once garbage collection has been fully optimized, use of `suicide` would be discouraged since `suicide` creates unsafe programs.

2.7 Reflection

Reflection is a system's ability to reason about itself and manipulate a causally connected description of itself [28, 23]. Causal connection means that changes to the description have an immediate effect on the described object. Note, however, that the changes go into effect only for the subsequent messages. In general, reflection may be used to customize the implementation of a system from within the language.

Currently, HAL supports the minimal amount of reflection capability necessary to customize the system with respect to fault tolerance [2]. An actor can reify its dispatcher and mail queue (the conceptual entities responsible for sending and receiving the communication, respectively).

```
(Mailq <expr>)

(Dispatcher <expr>)
```

The value of <`expr`> is the mail address of an actor. Once the mail queue of an actor is

reified, all subsequent messages destined to the actor are delivered through its reified mail queue actor. To get a next message to process, the actor notifies its reified mail queue actor by sending a special message. A synchronization constraint is associated with the method in the mail queue actor corresponding to the message, guaranteeing that the message is accepted only when the mail queue actor has one or more messages to deliver. Reification of the dispatcher of an actor causes all the outgoing messages from the actor to be sent through its reified dispatcher actor. These are implemented by using system-generated `put`, `get` and `transmit` messages, respectively.

Since reified mail queue actors and reified dispatcher actors are just ordinary actors, we can use any actor as a mail queue actor or a dispatcher actor. However, our implementation requires that all mail queue actors share a set of common properties such as having `put` and `get` methods, being able to hold the incoming messages in its state, etc. The dispatcher actors share their own common properties too. We enforce these rules by restricting the eligible actors for mail queue actors or dispatcher actors: the run-time system provides the default mail queue/dispatcher actor classes from which all the other mail queue and dispatcher actors inherit, respectively (Figure 2). Users can define a mail queue actor class as follows:

```
(defActor myMailQ (acq1 acq2)
          (superclass Mailq)
          (method (entry1 a b c)
                  :
```

A dispatcher actor can be defined similarly. The compiler enforces all mail queue (dispatcher) actors to have the default Mailq (Dispatcher) actor class at the top of its class hierarchy. Similarly, for the dispatcher actors.

Note that a user does not need to know about the existence of `enqueue`, `dequeue` and `toss` shown in Figure2. `enqueue` adds a message to the queue internal to a mail queue actor. `dequeue`

```
(defActor Mailq (baseActor queue)
    (disable (get) (emptyqueue queue))
    (method (put msg)
            (enqueue queue msg))
    (method (get)
            (let [[msg (dequeue queue)]]
                 (toss msg baseActor))))

(defActor Dispatcher
    (method (transmit msg dest)
            (toss msg dest)))
```

Figure 2: Default mail queue actor class and default dispatcher actor class

returns the message at the front of the queue. `toss` is just a variation of `send`, the asynchronous message sending primitive but it uses an already packed message so that it saves unnecessary unpacking and packing of the message.

These reflection capabilities have been used to carry out an experiments in implementing reusable software fault tolerance protocols [2]. Such capabilities can also be used to separate the code for resource management, such as that for optimal actor placement, from the code for an application.

3 Implementation Issues

In this section, we discuss three transformation techniques used in the HAL compiler. Two transformations allow translation of RPC style send primitives into asynchronous send primitives. The last one is the transformation for code motion to optimize the implementation of `update`/`become` primitive. The transformed programs shown in the figures are written in HAL itself to make it easier to understand the meanings of the transformations.

3.1 Transformation on RPC Style Send Primitive

Most distributed memory multi-processor systems do not support truly synchronous message sending in the sense that a sender and a receiver cannot synchronize with each other when they exchange a message. In order to implement RPC style message sending (Section 2.5) on such architectures, we must be able to express it in terms of ordinary asynchronous message sending. In the following two sections, we discuss the implementation of RPC style message sending in our compiler.

3.1.1 Join Continuation

Recall that `bsend` is like a remote procedure call which returns a result back to the place where the call is made. By lifting into a separated actor the expressions that should be executed only after a `bsend` expression receives a reply. we can implement `bsend` in terms of `send`. Note that the lifted actor is the continuation of the `bsend` expression. Delegation of work to an independent continuation actor makes it possible to avoid unnecessary loss of concurrency in the execution.

```
(send mB (new ActorB a b) x (bsend mC ActorC ...)
                            (bsend mD ActorD ...)
                            (bsend mE ActorE ...))
```

Figure 3: Example expression which uses bsends

Consider the expression in Figure 3. The `send` expression has no effect on the computation of the other parts of the method script except that it should be executed only after it gets all the replies from the `bsend` expressions. Simply making the continuation passing style (CPS) conversion [29, 21] is not satisfactory since it sequentializes the execution of the `bsend` expressions, causing a loss of concurrency.

We create an independent join continuation actor through dependency analyses. The expression in Figure 3 is transformed into the join continuation actor in Figure 4 and the expression in

Figure 5. The join continuation actor is responsible for the execution of the expressions that must be executed only after all replies arrive from the target actors of the `bsend` expressions. When the join continuation actor gets a reply from one of target actors, it checks if all the other values have been received. If so, it executes the assigned expressions. If not, it stores the value in its state and waits until all values are available. The availability of each value is kept in a flag which is associated with the corresponding acquaintance variable. In Figure 4, a join continuation actor is created with the appropriate initial values for its acquaintance variables and the environment in which it executes the assigned expressions. Note that a unique method name corresponding to the entry point of each reply message as well as the join continuation actor is added in the original `bsend` expressions. The definition of receiver actors should be modified accordingly to reflect the change in the sender actor's definition.

```
(defActor JC (acq1 acq2 v1 v2 v3 f1 f2 f3)
    (method  (ep1 x)
        (if (and f2 f3)
            (block (send mB acq1 acq2 x v2 v3)
                   (suicide))
            (block (update v1 x)
                   (update f1 TRUE))))
    (method  (ep2 x)
        (if (and f1 f3)
            (block (send mB acq1 acq2 v1 x v3)
                   (suicide))
            (block (update v2 x)
                   (update f2 TRUE))))
    (method  (ep3 x)
        (if (and f1 f2)
            (block (send mB acq1 acq2 v1 v2 x)
                   (suicide))
            (block (update v3 x)
                   (update f3 TRUE)))))
```

Figure 4: Extracted Join Continuation actor

The transformation given in this section is similar to the transformation done in Acore [24].

```
(let* [[jc (new JC (new ActorB a b) x 0 0 0 FALSE FALSE FALSE)]]
      (send mC ActorC ... ep1 jc)
      (send mD ActorD ... ep2 jc)
      (send mE ActorE ... ep3 jc))
```

Figure 5: The transformed expression

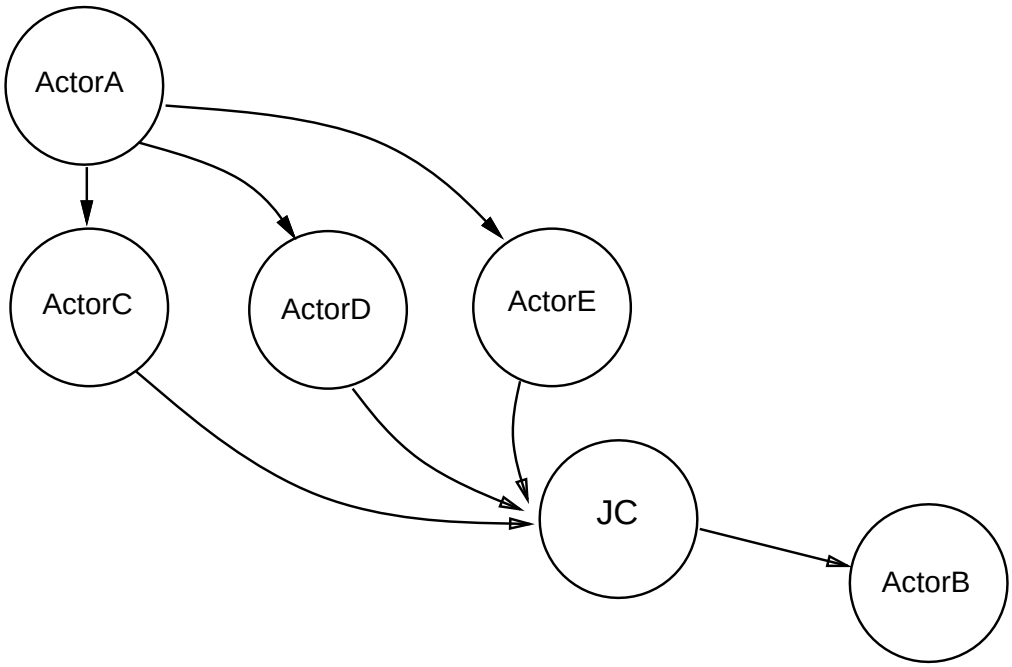

Figure 6: Message trajectory after extracting the Join Continuation actor

However, there are two major differences. First, by associating with each acquaintance variable an explicit flag which holds values received, we eliminate the overhead of system-wide unique value. Second, the `suicide` primitive (Section 2.6) is used to optimize memory reclamation. Because a join continuation actor is defined by the compiler, the compiler knows that the continuation actor will become garbage after it has been used only once. Thus, the compiler includes `suicide` in the definition so that the resources allocated to a join continuation actor can be reused as soon as it becomes garbage. Figure 6 shows the message sending pattern when the transformed program is executed.

```
(defActor Customer (a b c)
    (method (m1 x y)
        :
        (update a (bsend m2 (new Server) x y)
        :
```

(a) Before Transformation

```
(defActor Customer (a b c flag)
    (all-except m1-continuation  flag)
    (method (m1 x y)
        :
        (send m2 (new Server) x y m1-continuation self)
(update flag TRUE)
        : )
    (method (m1-continuation x)
        (update a x)
(update flag FALSE))
    :
```

(b) After Transformation

Figure 7: Transformation with method splitting

3.1.2 Method Splitting

When a `bsend` expression is used with the specification of replacement behavior, we are forced to sacrifice some concurrency in order to maintain the consistency of the actor's state (Figure 7.a). If we allow the sender to proceed before it gets the reply back, the sender may see the old value of the acquaintance rather than a new value when it accepts the next message. In this situation, we cannot delegate the work dependent on the `bsend` expression to a continuation. Rather, we split the method executing the `bsend` expression and add a synchronization constraint to the resulting continuation method (Figure 7). All methods except the desired continuation method are disabled until the reply has been received, guaranteeing a safe `update`.

3.2 Optimization on the Replacement Behavior

3.2.1 Transformation for Code Motion of Replacement Behavior

The semantics of the Actor model dictates that each communication to an actor is processed on its own *version* of the actor's state, allowing pipelining in the execution of the method containing `become` / `update` [1]. This internal concurrency implies that each expression in a method can be executed concurrently and the effect of replacement behavior is visible only to subsequent communications. However, it can be very expensive to utilize all available internal concurrency on conventional multiprocessor architectures. We sacrifice the concurrent execution of each expression of a method while retaining the single assignment semantic of the replacement behavior.

However, simple compilation of HAL program into sequential code preserving its textual order is not sufficient to make the effect of replacement behavior invisible to the current computation. In other words, the `send` expression in the following method definition may use the new value of *balance* where it is supposed to use the old value.

```
(method (deposit amount)
    (update balance (+ balance amount))
    (send deposit LogActor balance amount))
```

A straight-forward solution is to have a copy of an actor's acquaintance list, and perform an `update` on the copy rather than on the original acquaintance list. At the end of the computation, we can simply replace the acquaintance list with the copy. This solution is simple but has a very high overhead since it requires the entire acquaintance list to be copied whenever we execute the method containing `become` / `update` primitives.

Our solution is to be as lazy as possible. The very internal concurrency that gives us the problem also allows us to solve it efficiently. We move `become` / `update` expressions to the end of the computation so that we can perform `update` in place.

The visibility of the effect of `become` / `update` primitives only to the subsequent messages guarantees the operational equivalence between the program before the transformation and the program after the transformation. The use of `bsend`'s need not affect the equivalence since they have already been detached from the corresponding `update`'s.

```
(if ( _A_ )
    (block
        :
        (update  a  x)
        :    )
    (block
        :
        (update  a  y)
        :    ))
(if ( _B_ )
    (block
        :
        (update  c  x)
        :    )
    (block
        :
        (update  c  y)
        :    )))
```

Figure 8: Before the transformation to optimize replacement behavior

Simply moving `update`'s to the end of the method is not enough when a program has more than one `if` expression (Figure 8). We split each `if` into two `if`s, one with no `update` and the other with `update`'s only. All `if` expressions without `update`'s are placed before all `if` expressions with `update`'s only. Splitting `if` expression makes two copies of the same `if` condition expression. Unnecessary evaluation of the same conditions can be saved by introducing a `let` binding (Figure 9). Nested `if` expressions are not difficult to transform. `let` bindings introduced during splitting inner `if` expressions are promoted to the outer block so that the outer `if` expression is split as before.

```
(let* [[ra ( _A_ )] [rb ( _B_ )]]
      (if ra
          (block ... )
          (block ... ))
      (if rb
          (block ... )
          (block ... ))
      (if ra
          (block (update  a  x))
          (block (update  b  y)))
      (if rb
          (block (update  c  x))
          (block (update  c  y)))))
```

Figure 9: After the transformation to optimize replacement behavior

3.2.2 Performance Comparison

Table 1 shows the execution time of different number of `update`'s when we compiled an actor program with 6 acquaintance variables using the earlier version [16] and the current version of the HAL compiler. The current version uses the optimization which is mentioned in Section 3.2.1. The reason why the earlier version of HAL takes much more time to perform `update`'s is that `update`'s are implemented in the straight-forward way by copying the acquaintance list of the actor.

	number of updates	1	2	3	4	5
Execution time (unit: cycle)	unoptimized update	285	304	323	342	361
	in place update	17	26	35	44	53
method size (unit: byte)	unoptimized update	32	33	33	33	33
	in place update	17	21	24	26	27

Table 1: Execution time for updates

Table 1 also shows the code size per method in the executable object file. In all cases, optimized update yields less code size. Currently, HAL programs run on the top of the CHARM programming system which is a C based system [20]. CHARM has been implemented on both shared and distributed memory machines. The total execution time for this simple example was about 0.0527

sec. Over 90% of the time was spent to execute CHARM kernel code, showing that the start-up overhead of CHARM is very high. In order to further improve performance, we are implementing our compiler on a lower-level run-time system, namely, CHOICES [10], an object-oriented distributed operating system, which is known to be more efficient.

4 Related work

In this section, we briefly compare our work with several previous efforts to build actor-based parallel languages and their compilers. Note that the first three languages, Act, Cantor and Acore, do not support inheritance and do not provide explicit constructs for specifying synchronization constraints.

Act: It compiles the Scripter language into code which can be executed on a simulator for the Actor model, called Apiary, written on Symbolics 3600 Lisp machines. The Act compiler changes most system generated message sending to lisp function calls [13].

Cantor: Programs written for the Cantor system [8] can run on either sequential machines or Intel iPSC series machines. All communication is done through asynchronous message passing.

Acore: C. Hewitt's group at M.I.T. implemented the Acore compiler on the Apiary operating system [24]. It allows for synchronous message passing as well as asynchronous message passing.

Rosette: Developed at MCC in collaboration with one of the authors, Rosette [30] runs on a uni-processor virtual machine. It supports synchronous and asynchronous message passing. Synchronization constraints are specified through *enabled sets* [31] which specify the methods that may be invoked by the next message. Enabled sets are mixed in with the code so that Rosette cannot support incremental specialization of synchronization constraints.

ABCL/R: ABCL (Actor Based Concurrent Language) provides for asynchronous, synchronous and future based message passing. ABCL also allows the specification of synchronization constraints. One difference of HAL and ABCL is that ABCL does not support inheritance [35].

5 Current Status and Future Research Direction

Some practical examples show the performance advantage of using fine-grained inherent concurrency in the Actor model for distributed execution on multicomputers [3]. However, to design an actor-based language which is easy to use and machine independent, we need to add several linguistic extensions to the basic Actor model. For example, the ability to easily describe synchronization constraints in a high-level, abstract and reusable way is indispensable in a high-level concurrent language: such constructs guarantee the consistent and reliable execution of parallel algorithms. HAL supports the specialization and factorization of both code and synchronization constraints using inheritance. Other features such as sequenced send, RPC style send, and `update` are added to make the language a general and easily programmable one.

The HAL compiler generates a program written in CHARM [20] as its output. Optimization on the `update` primitive is implemented in HAL. We are implementing the transformation to extract a join continuation actor. Optimization on synchronization constraints is going to be added in near future. Active research is being done to incorporate distributed data structures in our compiler.

A fair amount of effort has been made to preserve the concurrency which is naturally expressed in HAL. Currently, the run-time system for HAL does not support garbage collection. Distributed garbage collector [19, 32] should be incorporated in near future. Finally, the compiler should be able to provide the run-time system with information on load balancing and locality control using modular specifications given by the user [3].

Acknowledgments

We would like to thank other members of Open Systems Laboratory at the University of Illinois at Urbana-Illinois, in particular, Svend Frolund and Daniel Sturman for their constructive suggestions for designing HAL, and Raju Panwar and Chirstian Callsen for careful reading the draft of this paper. We also appreciate Chris Houck, a former member of OSL, for his work on designing and implementing the earlier version of HAL.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A Linguistic Framework for Dynamic Composition of Fault-Tolerance Protocols. Technical Report UIUCDCS-R-92-1730, University of Illinois at Urbana-Champaign, April 1992.

[3] G. Agha, C. Houck, and R. Panwar. Distributed Execution of Actor Systems. In *Proceedings of Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, 1991.

[4] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesly, 1986.

[5] R. Allen and K. Kennedy. Automatic Translation of Fortran Program to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9, 1987.

[6] P. America and F. van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *OOPSLA '90*, pages 161–168, October 1990.

[7] Arvind and D. Culler. *Annual Reviews in Computer Science*, chapter Dataflow Architecture, pages 225–253. Annual Reviews Inc., 1986.

[8] W. Athas and C. Seitz. Cantor User Report Version 2.0. Technical Report 5232:TR:86, California Institute of Technology, Pasadena, CA, January 1987.

[9] S. Brookes, C. Hoare, and A. Roscoe. A Theory of Communicating Sequential Processes. *Communications of the ACM*, 31:560–599, July 1984.

[10] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987. IEEE.

[11] N. Carriero and D. Gelernter. Linda in Context. *Communication of the ACM*, 32(4):444–458, April 1989.

[12] A. Chien. *Concurrent Aggregates: An Object-Oriented Language for Fine-Grained Message-Passing Machines*. PhD thesis, MIT, July 1990.

[13] P. de Jong. Compilation into Actors. *SIGPLAN Notices*, 21(10):68–77, October 1986.

[14] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[15] S. Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In *Proceedings of European Conference on Object-Oriented Programming*, 1992. (to appear).

[16] C. Houck. Run-Time System Support for Distributed Actor Programs. Master's thesis, University of Illinois at Urbana-Champaign, January 1992.

[17] C. Houck and G. Agha. HAL: A High-level Actor Language and Its Distributed Implementation. In *21st International Conference on Parallel Processing (ICPP '92)*, August 1992. (to appear).

[18] P. Hudak. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[19] D. Kafura, D. Washabaugh, and J. Nelson. Garbage Collection of Actors. In *OOPSLA '90*, pages 126–134, October 1990.

[20] L. Kale. *The CHARM(3.0) Programming Language Manual*. University of Illinois, February 1992.

[21] D. A. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, February 1988. YALEU/DCS/RR-632.

[22] W. LaLonde and J. Pugh. *Inside Smalltalk*, volume 1. Prentice Hall, 1990.

[23] P. Maes. computational reflection. Technical Report 87-2, Vrije University. Artificial Intelligence Laboratory, 1987.

[24] C. Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.

[25] D. May, R. Shepherd, and C. Keane. Communicating Process Architecture: Transputer and Occam. In P. Treleaven and M. Vanneschi, editors, *Future Parallel Architecture*, pages 35–81. Springer-Verlag, 1986. LNCS 272.

[26] D. Padua and M. Wolfe. Advance compiler optimizations for supercomputers. *Communication of the ACM*, 29(12):1184–1201, December 1986.

[27] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung, and D.A. Schouten. The Structure of Parafrase-2: an Advanced Parallelizing Compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 423–453. The MIT press, 1990.

[28] B. C. Smith. Reflection and semantics in a procedural language. Technical Report 272, Massachusetts Institute of Technology. Laboratory of Computer Science, 1982.

[29] G.L. Jr. Steele. RABBIT: a compiler for SCHEME. AI Memo 474, MIT, May 1978.

[30] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An Object Oriented Concurrent System Architecture. *SIGPLAN Notices*, 24(4):91–93, 1989.

[31] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA*, 1989.

[32] N. Venkatasubramian. Hierarchical Memory Management in Scalable Parallel Systems. Master's thesis, University of Illinois at Urbana-Champaign, 1991.

[33] T. Watanabe and A. Yonezawa. *ABCL An Object-Oriented Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass, 1990.

[34] P. Wegner. Dimensions of Object-Based Language Design. Technical Report CS-87-14, Brown University, July 1987.

[35] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.