

Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages

[WooYoung Kim](#) and [Gul Agha](#)

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, IL 61801 USA

Phone: (217) 244-3087

Fax: (217) 333-3501

Email: {wooyoung, agha}@cs.uiuc.edu

[© Copyright 1995 by ACM, Inc.](#)

Keywords

Concurrent Object-Oriented Programming, Actors, Location Transparency, Migration

Abstract

We describe the design of a runtime system for a fine-grained concurrent object-oriented (actor) language and its performance. The runtime system provides considerable flexibility to users; specifically, it supports location transparency, actor creation and dynamic placement, and migration. The runtime system includes an efficient distributed name server, a latency hiding scheme for remote actor creation, and a compiler-controlled intra-node scheduling mechanism for local messages and dynamic load balancing. Our preliminary evaluation results suggest that the efficiency that is lost by the greater flexibility of actors can be restored by an efficient runtime system which provides an open interface that can be used by a compiler to allow optimizations. On several standard algorithms, the performance results for our system are comparable to efficient C implementations.

- [Introduction](#)
- [The Language and Its Computational Model](#)

- [The Actor Model](#)
- [Programming Abstractions for Communication](#)
- [The Runtime System Architecture](#)
- [Distributed Name Server](#)
 - [Mail Address and Locality Descriptor](#)
 - [Distributed Name Table](#)
 - [Message Delivery Algorithm](#)
- [Remote Actor Creation](#)
- [Implementation of HAL 's Communication Abstractions](#)
 - [Support for Local Synchronization Constraints](#)
 - [Support for Call/Return Communication Abstraction](#)
 - [Compiler-Controlled Intra-Node Scheduling](#)
 - [Collective Scheduling of Broadcast Messages](#)
 - [Minimal Flow Control](#)
- [Performance Evaluation](#)
 - [Performance of the Runtime Primitives](#)
 - [Fibonacci Number Generator](#)
 - [Systolic Matrix Multiplication](#)
- [Related Work](#)
- [Conclusions and Future Work](#)
- [Acknowledgments](#)
- [References](#)

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SC '95, San Diego, CA

© ACM 1995 0-89791-985-8/97/0011 \$3.50

[next](#) [up](#) [previous](#)

Next: [The Language and Up: Efficient Support of Location](#) **Previous:** [Efficient Support of Location](#)

Introduction

We argue that compromising flexibility is not a necessary condition to obtain good execution performance. Specifically, we describe the design of an efficient runtime system for an actor-based language, HAL [15,3], which supports location transparency, placement specification for dynamically created objects, and migration. We have argued that such flexibility is essential for scalable execution of dynamic, irregular applications over sparse data structures [28]. We also provide preliminary performance results showing that the runtime system incurs a tolerable overhead. The distinguishing features of our runtime system are:

- *support for local synchronization constraints:* Synchronization constraints specify the subset of possible states of an object under which the object's methods may be invoked [12]. Synchronization constraints are *local* if they are specified on a per-object basis. The system supports the enforcement of local synchronization constraints to simplify the specification of synchronization requirement necessary for correct execution.
- *efficient implementation of distributed name server:* Support for complete location transparency and object migration increases the complexity of the communication module and degrades execution performance. Our name server is designed to do locality check using only locally available information, providing flexibility with tolerable overhead. The runtime system implements dynamic load balancing using location transparency and migration.
- *latency hiding in remote creation:* A common way to reduce the inefficiency caused by the unpredictable remote creation time in fine-grained multicomputers is *split-phase allocation* [5]. An object is context-switched to another when it requests a remote creation. However, split-phase allocation is not desirable in stock-hardware multicomputers because of their high context switching cost. We

propose an efficient scheme which masks latency in remote object creation without context switching.

- *compiler-controlled intra-node scheduling*: The runtime system employs compiler-controlled stack-based scheduling for local messages inspired by [10,29]. Because our runtime system provides a flexible interface to the compiler, we were able to implement an efficient stack-based scheduling mechanism [22].

We have implemented our runtime system on the CM-5 [32,31]. Primitive operations, such as object creation and message send, have been carefully designed and optimized while maintaining the semantics of the language. The runtime system exposes part of its scheduling mechanism to the compiler so that the compiler can exploit frequently occurring special cases. The runtime system is also designed to optimally utilize the information available at compile-time. Our performance results for primitive operations are comparable to those of other implementations [29,18]. This suggests that the overhead caused by adding more flexibility can be kept tolerable by proper system design and the close interaction between a compiler and its runtime system. In this paper, we describe implementation techniques that enabled us to achieve efficiency.

§ 2 presents relevant abstractions supported in HAL along with its computational model. § 3 discusses the overall architecture of our runtime system and its components. The new techniques implemented in the runtime system are described in detail in § 4 through § 6. Preliminary evaluation results are reported in § 7. We briefly describe related research in § 8.

[next](#) [up](#) [previous](#)

Next: [The Language and](#) **Up:** [Efficient Support of Location](#) **Previous:** [Efficient Support of Location](#)

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [The Actor Model](#) **Up:** [Efficient Support of Location](#) **Previous:** [Introduction](#)

The Language and Its Computational Model

HAL [15,3] is an untyped but statically type-checked, actor-based language. Types are implicit in programs. The compiler infers types for each expression in a program using a constraint-based type inference algorithm [27]. It generates C code as its output. In this section, we briefly review the Actor model of computation on which HAL is based. We also discuss language abstractions relevant to the rest of the paper. For a more detailed description of the language constructs, see [3].

-
- [The Actor Model](#)
 - [Programming Abstractions for Communication](#)
-

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Programming Abstractions for Up: The Language and Previous: The Language and](#)

The Actor Model

Actors are independent, concurrent objects which interact through asynchronous communication [1]. In response to a message, an actor may: (*i*) *send* a message asynchronously to the specified actor, (*ii*) *create* an actor with the specified behavior, or (*iii*) *become* a new behavior and be ready to process the next message.

Communication between actors is buffered: incoming messages are queued until the actor is ready to process them. Each actor has a unique mail address which is used to specify a target for communication. Mail addresses may also be communicated in a message, allowing for a dynamic communication topology.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [The Runtime System](#) **Up:** [The Language and](#) **Previous:** [The Actor Model](#)

Programming Abstractions for Communication

The actor primitive operators form a simple but powerful set on which to build a wide range of higher-level abstractions. In this section we describe some simple extensions to the Actor model as well as other abstractions in HAL [3].

The `new` primitive operator is used to specify a single actor creation: it creates an actor of the specified behavior (cf., *class*) and returns a unique mail address. HAL extends the creation abstraction with the `grpnew` operator; `grpnew` is used to create a group of actors with the same behavior template. It returns a unique identifier which may be subsequently used to refer to the group. HAL also extends the asynchronous communication in the Actor model with two communication abstractions: *call/return* [26,37] and *broadcast* [3]. Two operators, `request` and `reply`, implement the call/return communication abstraction; `request` sends a message to the server and blocks the sender while it is waiting for a reply and `reply` sends a result back to the sender of the request. A message sent to all members of a group is broadcast: the broadcast message is replicated and a copy is delivered to each member of the group.

<i>P</i>	256 × 256				512 × 512			
	<i>Seq</i>	<i>BP</i>	<i>CP</i>	<i>Bcast</i>	<i>Seq</i>	<i>BP</i>	<i>CP</i>	<i>Bcast</i>
1	2.56	2.56	2.57	2.57	21.60	21.60	21.61	21.51
4	1.43	1.43	0.73	1.66	12.20	12.22	5.58	13.38
16	0.47	0.42	0.27	0.70	4.02	3.83	1.76	4.99
64	0.29	0.21	0.16	0.46	1.60	1.23	0.90	2.22
256	0.28	0.14	0.14	0.39	1.23	0.77	0.56	1.57

Table 1: Results in msec from a set of C implementation of the Cholesky Decomposition algorithm on the CM-5. *P* is the number of processing elements. Columns *BP* and *CP* represent execution times for the implementations which start the execution of iteration *i+1* before the execution of iteration *i* has completed by only using local synchronization. Columns *Seq* and *Bcast* show the numbers obtained by completing the execution of iteration *i* before starting that of the iteration *i+1*. Implementations of *BP* and *CP* are identical except that the former uses *block* mapping (i.e., $i/(mat\ size/n\ procs)$) and the latter uses *cyclic* mapping (i.e., $i\%n\ procs$).

Non-deterministic message delivery and asynchronous execution in the Actor model necessitates abstractions to specify the order in which messages may be legally processed (i.e., synchronization). However, excessive enforcement of synchronization may erode the performance gained by parallel execution. For example, Table [1](#) compares the effect of local and global synchronizations on the execution of Cholesky Decomposition. [gif](#) Implementations which use local synchronization outperformed those which use global synchronization (cf., data parallelism). HAL supports the modular specification of local synchronization constraints as disabling conditions; this allows the programmer to express the minimal synchronization specification that is required for correct program execution [[12](#)]. HAL communication abstractions can be used to efficiently implement a variety of algorithms [[2,3](#)].

[next](#) [up](#) [previous](#)

Next: [The Runtime System](#) **Up:** [The Language and](#) **Previous:** [The Actor Model](#)

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Distributed Name Server](#) **Up:** [Efficient Support of Location](#) **Previous:** [Programming Abstractions for](#)

The Runtime System Architecture

The runtime system is currently running on the CM-5 and on networks of workstations. This paper describes the implementation of the runtime system on the CM-5. The CM-5 machine is a distributed memory multicomputer which can be scaled up to 16K processors. The machine may be configured in different partitions; a partition has a control processor called *partition manager* and a set of processors called *processing elements*. Each processing element contains a 33 MHz Sparc processor and a network interface chip which supports all accesses to the interconnection network [31].

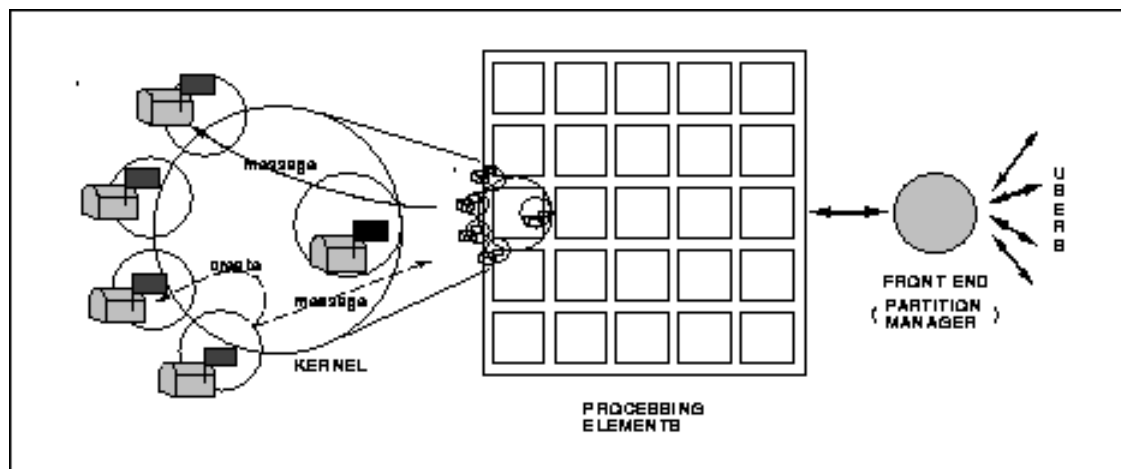


Figure 1: The architecture of the HAL runtime kernel

Figure 1 depicts the overall architecture of the HAL runtime system on the CM-5. The runtime system consists of a *front-end* which runs on the partition manager and a set of runtime kernels which run on the processing elements (i.e., *nodes*). Each kernel contains a system (meta-level) actor named *node manager*. Kernels are implemented as ordinary UNIX processes.

The runtime system is designed to concurrently execute multiple programs on the same

partition; the design minimizes the machine's idle cycles and allows the runtime system to best utilize computational resources. In order to efficiently support sharing, we have designed the kernel to execute all computations on a single address space [23]. The compiler generates executables with load information. Upon execution, the executable is dynamically loaded and integrated into each kernel. The kernel does not discriminate between actors created by different programs. Users are provided with a simple command interpreter which communicates with the front-end to load the executables. In addition to dynamic loading of user's executables, the front-end processes all I/O requests from the kernels running on the nodes.

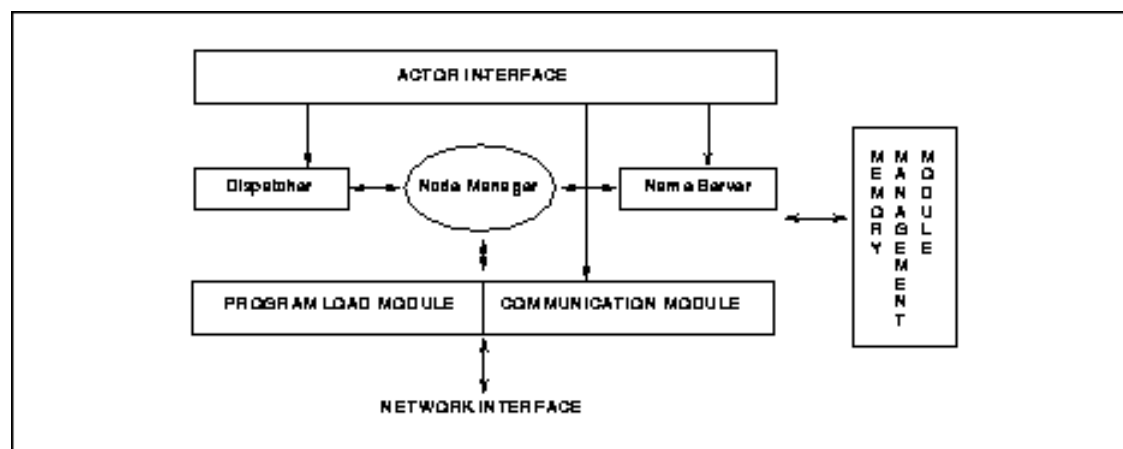


Figure 2: Internal structure of the runtime kernel.

The kernel serves as a passive substrate on which individual actors execute. Because each actor executes kernel functions as part of its own computation, both actor methods and kernel functions may be executed on the same stack assigned to the actor, eliminating the need for context switching between the actor and the kernel. The kernel components and the interaction among them are shown in Fig. 2. On the top level is an *actor interface* which is exported to the compiler. *Communication module* and *program load module* together constitute the kernel's interface to the underlying machine and represent the kernel's machine-dependent part. In between these two layers are *node manager*, *dispatcher* and *name server*.

The *communication module* is built on top of CMAM [35], a messaging layer which provides a transparent view of the underlying architecture; thus, porting the kernel to other platforms is straight-forward as long as a well-defined messaging layer is supported (for example, [34,19,20]). Messages in HAL have some unique properties. In particular, all actor messages have a destination mail address and a method selector. Many of them may also contain a continuation address. These properties are exploited in the implementation of communication module by customizing the CMAM layer to minimize the communication latency. The communication module implements the broadcast primitive in terms of point-to-point communication, using a hypercube-like minimum spanning tree communication structure. It also performs minimal flow control

to reduce network congestion and to implement correct software pipelining.

A *node manager* delivers messages sent by remote actors to local actors, creates an actor (or actors) in response to a creation request from a remote actor and dynamically loads and links a user's executables by calling the program load module. Node managers communicate with each other to maintain the system's consistency and allow dynamic load balancing. A request to a node manager is delivered in the form of a message: upon receiving a request, it steals the processor from the actor that is currently executing, processes the request using that actor's stack frame and subsequently resumes the actor's execution.

The *dispatcher* provides the data structures that are necessary for scheduling actors; the responsibility to actually schedule actors is delegated to individual actors. When an actor completes its execution, it obtains another actor from the dispatcher and yields control to it. This allows the scheduling to be performed without context switching.

[next](#) [up](#) [previous](#)

Next: [Distributed Name Server](#) **Up:** [Efficient Support of Location](#) **Previous:** [Programming Abstractions for](#)

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Mail Address and](#) **Up:** [Efficient Support of Location](#) **Previous:** [The Runtime System](#)

Distributed Name Server

When an actor sends a message to another actor, the following sequence of actions occurs: the sender first consults the local name server to get the receiver's location information. Then, the sender executes a low-level communication primitive to send the message off. On the receiving node, the node manager consults the name server to locate the receiving actor and delivers the message. Name translation in a sender's node is required even when the recipient is on the same node. We describe below how this generic message send mechanism is efficiently implemented while supporting location transparency. § [6.3](#) describes an optimization which can be used to eliminate local message sends in some cases.

-
- [Mail Address and Locality Descriptor](#)
 - [Distributed Name Table](#)
 - [Message Delivery Algorithm](#)
-

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Distributed Name Table](#) **Up:** [Distributed Name Server](#) **Previous:** [Distributed Name Server](#)

Mail Address and Locality Descriptor

Each actor is uniquely identified by its mail address which represents its *locality* in the computational space. A mail address is implemented as a pair of real addresses, `<birthplace, address>`, where `birthplace` represents the node on which the actor is created and `address` represents the memory address of a *locality descriptor*. An actor's locality descriptor contains information about the actor's current locality. Specifically, if the actor is local, it has a reference to the actor. On the other hand, if the actor is remote, it contains the remote node address as well as the memory address of the actor's locality descriptor on the remote node. In addition, it may have the actor's *alias* (§ 5). A locality descriptor is allocated and assigned to an actor when it is created.

The use of locality descriptors and the use of real addresses in mail addresses allow for an efficient implementation of the generic message send mechanism. To send a message, the sender consults the local name server. If the location information is available in the name server, the message is sent using the information. Otherwise, a locality descriptor is allocated and the message is sent to the node on which the receiver actor was created, using the information encoded in the mail address. (For the moment we assume that actors never migrate. A more general solution with actor migration is given in § 4.3.) The memory address of the locality descriptor in the receiving node is sent back to the sending node and cached in the newly allocated locality descriptor. Subsequent messages to the receiver actor are sent with the cached address, making name table look-up in the receiving node unnecessary.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Message Delivery Algorithm](#) **Up:** [Distributed Name Server](#) **Previous:** [Mail Address and](#)

Distributed Name Table

Each kernel maintains its own (local) name table, and name translation from a mail address to the location information is performed by consulting the local name table only; i.e., it does not require inter-processor communication to get a receiver's actual location. Name tables are implemented as hash tables whose entries are actor locality descriptors.

We relax consistency requirements in order to make the name server more efficient. Inconsistency may be introduced when an actor is migrated. This is based on our assumption that migration is a relatively infrequent event and that those actors that have moved are more likely to migrate again before they are sent a message. Thus, location information for remote actors is a ``best guess." Two name tables may not even agree with each other on an actor's location information. Instead of maintaining stringent consistency, we relax the requirements and provide a mechanism to correct the inconsistency whenever the correction is needed.

```
if mail address is in my local table then
  if the actor is local then
    deliver the message
  else if it is on a known remote node (note: the actor may no longer be there) then
    send the message to the node
  else if the location is not known then
    send the message to the birthplace of the actor
  else if it is on a remote node but a forwarding information request has been sent then
    enqueue the message and hold it until the actual location is known
  else
    error
else
  if its birthplace is me then
    error
  else
    send the message to its birthplace
    (a) Sender Actor

if mail address is in my local table then
  if the actor is local then
    deliver the message
  else if it is on a known remote node (note: the actor may no longer be there) then
    enqueue the message and send a forwarding information request
  else if the location is not known then
    it will never happen
  else if it is on a remote node but a forwarding information request has been sent then
    enqueue the message and hold it until the actual location is known
  else
    error
else
  error
    (b) Receiving Node Manager
```

Figure 3: Message send and delivery algorithm

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Remote Actor Creation](#) **Up:** [Distributed Name Server](#) **Previous:** [Distributed Name Table](#)

Message Delivery Algorithm

Actors may migrate, and an actor's migration history is kept in locality descriptors for the actor. Since location information of remote actors is only a best guess, a message may be sent to a node from which the receiver has already migrated. If a node manager is requested to deliver such a message, it forwards the message using the history information kept in its local name table. However, instead of forwarding the entire message the node manager sends a special forwarding information request (FIR) message to locate the actor. The FIR message is relayed until it reaches the actor. When it reaches the actor the location information is propagated back along the forward chain with the locality descriptor's memory address. Once the location is known, the original message is sent directly to the node where the receiver resides. Meanwhile, all node managers in the forward chain update their name table with the new information.

When a node manager receives a request to deliver a message to an actor, it may have already sent an FIR message to locate the actor. It is unnecessary for the node manager to send another FIR message; thus, it puts off the message delivery until the receiver's location is known. In order to further reduce message traffic due to migration the memory address of an actor's locality descriptor in the new node is cached in its birthplace node as well as in the old node. Fig. 3 summarizes the message send and delivery algorithm.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Implementation of HAL](#) **Up:** [Efficient Support of Location](#) **Previous:** [Message Delivery Algorithm](#)

Remote Actor Creation

Recall that a locality descriptor is allocated in the node in which an actor is created.

The memory address of this locality descriptor is used to define the newly created actor's mail address. An actor which requests a remote creation must wait until a new actor is created and its mail address is returned from the remote node. (Note that *split-phase allocation*, preferable on fine-grained multiprocessors, may not be desirable on stock-hardware multicomputers because of high context switching overhead.)

We use *aliases* to hide the remote creation latency with no context switching. An alias is another entity which uniquely identifies an actor. An actor's alias can be used interchangeably with its mail addresses. The use of aliases is based on the observation that an actor which sends a remote creation request may continue the rest of its computation (i.e., its *continuation*) as long as it can uniquely identify the newly created actor. Note that actors created in response to local creation requests don't have aliases.

Aliases have the same structure (i.e., `<birthplace, address>`) as ordinary mail addresses. However, `birthplace` represents not the node where the actor was created, but the node where the creation request was issued. The node address where the actor is created is also encoded in `birthplace` along with type information. The encoded information may be used in subsequent message sends. For example, if an actor sends a message using an alias and the corresponding locality descriptor is not found in the (local) name table, the message is forwarded to the node where the actor was actually created with the assumption that the receiver has not migrated.

When an actor issues a remote creation request it allocates an alias and sends the alias to the remote node along with the request. As soon as the last packet of the request is injected into the network the sender resumes the execution of its continuation. The node manager which receives a remote creation request creates an actor with an ordinary mail

address and registers the actor in its local name table with the received alias. Meanwhile, the locality descriptor's memory address is sent back to the requesting node to be cached (as background processing). The execution of a remote creation with no initialization message completes within 5.83 μ s whereas the actual creation takes 20.83 μ s.

[next](#) [up](#) [previous](#)

Next: [Implementation of HAL](#) **Up:** [Efficient Support of Location](#) **Previous:** [Message Delivery Algorithm](#)

WooYoung Kim
Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Support for Local Up](#) **Up:** [Efficient Support of Location](#) **Previous:** [Remote Actor Creation](#)

Implementation of HAL 's Communication Abstractions

The runtime system implements high-level communication abstractions in terms of low-level primitives and data structures hidden from the compiler. However, it exposes part of its scheduling mechanism to the compiler so that the compiler can exploit frequently occurring special cases [[10,22](#)].

-
- [Support for Local Synchronization Constraints](#)
 - [Support for Call/Return Communication Abstraction](#)
 - [Compiler-Controlled Intra-Node Scheduling](#)
 - [Collective Scheduling of Broadcast Messages](#)
 - [Minimal Flow Control](#)
-

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Support for Call/Return](#) **Up:** [Implementation of HAL](#) **Previous:** [Implementation of HAL](#)

Support for Local Synchronization Constraints

The enforcement of local synchronization constraints is implemented using an auxiliary mail queue (i.e., *pending queue*). When an actor dispatches a message, it examines whether the corresponding method is enabled. If it is disabled, the message is put into the actor's pending queue. Whenever an actor completes its method execution, it examines whether or not it has pending messages. If it does, it dispatches the pending messages one by one before it schedules the next actor.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Compiler-Controlled Intra-Node Scheduling](#) **Up:** [Implementation of HAL](#)

Previous: [Support for Local](#)

Support for Call/Return Communication Abstraction

The HAL compiler transforms a `request` send to an asynchronous send and separates out its continuation through dependence analysis [21,3,22]. Message sends which have no dependence among them are grouped together to share the same continuation. Such continuation have a deterministic behavior: as soon as all the expected replies are received, they execute the computation specified at their creation time and never receive further messages. The runtime system distinguishes `reply` messages and implements continuations using *join continuations*, thereby efficiently exploiting their deterministic behavior.

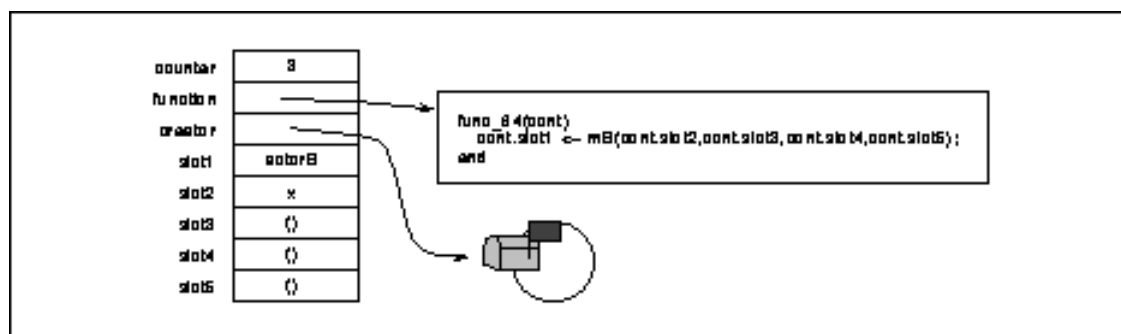


Figure 4: The structure of join continuation. The code segment pointed by `function` represents the compiler-generated continuation of a message send. When executed, it sends to `actorB` a message `mB` with values in `slot2` through `slot5` as its arguments.

A join continuation has four components, namely `counter`, `function`, `creator` and a set of argument `slots` (Fig. 4); `counter` contains the number of empty slots to be filled with subsequent replies. As soon as one slot is filled, it is decremented by one. When it becomes zero the function pointed by `function` is invoked with the continuation as its argument; the function implements the continuation of a `request` message send. Some argument `slots` are filled with values that were already known when the join continuation was created; others represent empty slots which will be filled with subsequent replies. Finally, `creator` points to the actor which created the join continuation; it is used to notify the actor of the completion of continuation if necessary.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Collective Scheduling of Up](#) **Up:** [Implementation of HAL](#) **Previous:** [Support for Call/Return](#)

Compiler-Controlled Intra-Node Scheduling

The large performance difference between the generic message send mechanism and function invocation justifies the use of runtime locality check to enable static method dispatch for scheduling local messages [29,18,22]. The difficulty in using static method dispatch lies in the fact that the method to be invoked may not be determined statically because of late binding and type-dependent method dispatch in object-oriented languages.

The compiler uses a constraint-based type inference [27] to determine the recipient's type (i.e. class) for each message send. If the compiler can infer the unique type for the recipient it generates code for static method dispatch with locality check as well as code for the message send [22]. The runtime system provides the compiler with the locality check routine which is part of the generic message send mechanism. The routine additionally checks if the recipient actor is in a state in which it is enabled to process the message. If the compiler infers more than one type for the recipient, it generates code to obtain the function pointer to the method as well. The runtime system makes available to the compiler the method lookup routine which is part of its message dispatch mechanism.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Minimal Flow Control](#) **Up:** [Implementation of HAL](#) **Previous:**
[Compiler-Controlled Intra-Node Scheduling](#)

Collective Scheduling of Broadcast Messages

Messages broadcast to a group are delivered to all members of the group. The broadcast primitive is implemented on top of the CMAM layer using a hypercube-like minimum spanning tree communication structure. By distinguishing broadcast messages and exposing the implementation of groups to the compiler, broadcast messages are scheduled in a manner similar to the *quasi-dynamic* scheduling in Threaded Abstract Machine (TAM) [10]. The compiler uses flow analysis information [27] to provide information for the runtime system to collectively schedule a broadcast message.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Performance Evaluation](#) **Up:** [Implementation of HAL](#) **Previous:** [Collective Scheduling of](#)

Minimal Flow Control

Since Active Messages are not buffered, a three-phase protocol is required to send bulk data from one node to another [35]. We have learned from the implementation of Cholesky Decomposition algorithm (§ 2.2) that without flow control, software pipelining may not be correctly implemented because of the three-phase protocol. The runtime system supports minimal flow control for sending messages of large size to guarantee the correct implementation of software pipelining. A node manager controls sending the acknowledgment for a bulk data transfer request to the requesting node so that only one such transfer is active at a time. The support for flow control reduces packet back-up in the network, improving network performance as well as processor efficiency. For example, without flow control the *pipelined* version of Cholesky Decomposition did not deliver the expected performance (Table. 1).

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Performance of the](#) **Up:** [Efficient Support of Location](#) **Previous:** [Minimal Flow Control](#)

Performance Evaluation

In this section, we present results of our preliminary evaluation of the runtime system. The runtime system was written using C and the assembly language. The part written in C was compiled using the GNU C compiler with -O3 option. The part written in the assembly language was compiled using the GNU assembler. Our compiler generates C code which is compiled using the GNU C compiler with -O3 option.

-
- [Performance of the Runtime Primitives](#)
 - [Fibonacci Number Generator](#)
 - [Systolic Matrix Multiplication](#)
-

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Fibonacci Number Generator](#) **Up:** [Performance Evaluation](#) **Previous:** [Performance Evaluation](#)

Performance of the Runtime Primitives

Table 2 summarizes the execution time of the runtime primitives. As mentioned earlier, the use of *aliases* allows the local execution of a remote actor creation takes 5.83 μ sec whereas the actual latency is 20.83 μ sec. The locality check is done using only locally available information and completes within 1 μ sec for the locally created actors. The performance of the runtime primitives is comparable to that in other systems [29,18] (Table 3).

	Local Creation	Remote Creation	Local Send/Dispatch	Remote Send/Dispatch	Locality Check
time	8.01	5.83(20.83)	0.45 - 5.67	9.91	1.00

Table 2: Execution time of runtime primitives (unit: μ sec)

	Local invocation		Remote invocation	
	μ sec	cycle	μ sec	cycle
ADCL/AP1000 (25 MHz)	2.3	58	8.9	221
Concert (33 MHz)	3.7 ⁺	122	7.67-16.3	254-528
HAL (33 MHz)	1.45 [†]	48	9.91	327

Table 3: The comparison of comparable method invocation costs. All numbers are minimum values. ₋ and ₊ are the sum of the time for locality check and the time for function invocation.

The overall performance of a runtime system on stock-hardware distributed memory multicomputers depends on the efficiency of its communication module. In order to assess the efficiency of the communication module we have implemented two benchmark programs, Fibonacci number generator and a systolic dense matrix multiplication algorithm [24] and measured the execution time on different CM-5 partitions.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Systolic Matrix Multiplication](#) **Up:** [Performance Evaluation](#) **Previous:** [Performance of the](#)

Fibonacci Number Generator

Number of Processors	1	2	4	8	16	32
<i>Without DLB</i>	55.5	32.3	20.8	12.9	7.94	4.96
<i>With DLB</i>	60.7	37.9	24.1	8.98	3.84	1.87

Table 4: Execution times (seconds) of the Fibonacci computation with and without dynamic load balancing.

Although the Fibonacci number generator is a very simple program, it is extremely concurrent: executing the Fibonacci of 33 results in the creation of 11,405,773 actors. Moreover, its computation tree has a great deal of load imbalance. Table 4 shows the execution results when using a dynamic load balancing scheme. Since Fibonacci actors are purely functional, actor creations were optimized away. Receiver-initiated random polling scheme [25] is used for dynamic load balancing. As a point of comparison, executing the Fibonacci of 33 using the Cilk system [6] takes 73.16 seconds on the same Sparc processor and an optimized C version completes in 8.49 seconds.

WooYoung Kim
Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Related Work](#) **Up:** [Performance Evaluation](#) **Previous:** [Fibonacci Number Generator](#)

Systolic Matrix Multiplication

M	P	2	4	8	16
256		1.06	0.31	0.12	0.05
512		8.40	2.37	0.69	0.23
1024		72.78	12.51	4.94	1.46

Table 5: Execution times of systolic matrix multiplication (unit: seconds). All results were obtained by executing the program with $M \times M$ matrix on $P \times P$ processor array.

The systolic matrix multiplication algorithm involves first skewing the blocks within a square processor grid, and then, cyclicly shifting the blocks at each step. No global synchronization is used in the implementation. Instead, per actor basis local synchronization is used to enforce the necessary synchronization. The local block matrix multiplication is implemented using the assembly routine used in [9]. The execution times are shown in Table 5. The results are comparable to the results given in [9]. For example, the performance peaks at 434 MFlops for 1024 by 1024 matrix on 64 node partition of the CM-5.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Conclusions and Future](#) **Up:** [Efficient Support of Location](#) **Previous:** [Systolic Matrix Multiplication](#)

Related Work

Many efficient software-based and hardware-based implementations for fine-grained concurrent object-oriented programming languages have been reported in literature [14,36,29,18]. Our work relies on software technologies without any special hardware support; it is more closely related to the software-based implementations [29,18]. Our implementation supports fine-grain object-level concurrency and thus differs from research in coarse-grained COOP languages such as [7,11,13,16]. However, the techniques we use in the implementation of a name server and remote object creation may also be used in the implementation of coarse-grained COOP languages.

The implementation of ABCL/onAP1000 [29,30] uses an encapsulated runtime system. For example, the runtime system determines whether to use the stack-based or the queue-based scheduling mechanism for local messages. By contrast, our runtime system and the compiler interact more closely to generate executables which choose the mechanism to use to send local messages. Objects in ABCL/onAP1000 are identified with a unique mail address, as in our case. However, the use of location-dependent mail address limits an object's mobility. On the other hand, our runtime system guarantees location transparency through the indirection provided by locality descriptors. Furthermore, the use of locality descriptor allows us to use aliases to mask remote creation latency.

Our work and the Concert system [8,18] is similar in that the runtime system provides the compiler with a flexible interface. The two runtime systems make cost distinctions explicit for runtime operations to enable a variety of optimizations by the compiler. The difference between the two systems lies in the extent of location transparency they support. Aggregates in the Concert system are located at the same address offset on each node [17]. This location dependence limits the aggregates's mobility, making it difficult to balance load in dynamic irregular computations. Objects other than aggregates are allocated in a global space and subject to global name translation. Our locality check

uses only locally available information thanks to our name management scheme which works efficiently with migration.

TAM defines an extension of a hybrid dataflow model with a multilevel scheduling hierarchy where synchronization, scheduling and storage management are explicit and under compiler control. It supports instruction-level multithreading. On the other hand, our runtime system implements the Actor model of computation with user-specified synchronization, location transparency and object-level concurrency. TAM ensures that all enabled threads for an activation frame are scheduled consecutively. The set of such threads is called *quantum*. This quasi-dynamic scheduling allows the compiler to exploit temporal locality existing among logically related threads. Such temporal locality is utilized in our system by collectively scheduling messages broadcast to a group of actors of the same type.

[next](#) [up](#) [previous](#)

Next: [Conclusions and Future](#) **Up:** [Efficient Support of Location](#) **Previous:** [Systolic Matrix Multiplication](#)

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [Acknowledgments](#) **Up:** [Efficient Support of Location](#) **Previous:** [Related Work](#)

Conclusions and Future Work

We have described a runtime system for an actor-based programming language HAL . The preliminary results of the runtime system are promising. Moreover, the performance of our runtime system on dense systolic and regular dynamic computations is competitive with that obtained by implementations using less flexible systems. However, we need to do more thorough evaluation with a wider range of realistic applications to find potential performance bottlenecks in irregular, sparse computations.

The use of locality descriptors to support location transparency has the advantage of supporting an efficient garbage collection scheme. We are investigating the possibility of using a real-time garbage collector, such as the one described in [33].

Recently, networks of workstations with fast interconnect network have drawn more and more attention as the potential work force for high performance concurrent computing [4]. Our runtime system allows dynamic loading and linking of application programs and can support the concurrent execution of multiple programs from different users. We are investigating ways to reconcile such hardware platforms and our runtime system in a multi-user environment.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

[next](#) [up](#) [previous](#)

Next: [References](#) **Up:** [Efficient Support of Location](#) **Previous:** [Conclusions and Future](#)

Acknowledgments

The research has been made possible by support from the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195).

We would like to acknowledge Mark Astley, Patricia Denmark, and Rajendra Panwar for their careful review of the draft of this paper. We thank Thorsten von Eicken for allowing us to use his assembly routine for matrix multiplication. We also thank the UIUC NCSA for use of their CM-5 machine.

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995

next [up](#) [previous](#)

Up: [Efficient Support of Location](#) Previous: [Acknowledgments](#)

References

- 1
G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- 2
G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3--14, May 1993.
- 3
G. Agha, W. Kim, and R. Panwar. [Actor Languages for Specification of Parallel Computations](#). In G. E. Blelloch, K. Mani Chandy, and S. Jagannathan, editors, *DIMACS. Series in Discrete Mathematics and Theoretical Computer Science. vol 18. Specification of Parallel Algorithms*, pages 239--258. American Mathematical Society, 1994. Proceedings of DIMACS '94 Workshop.
- 4
T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team. [A Case for NOW \(Networks of Workstations\)](#). *IEEE Micro*, 15(1):54--64, February 1995.
- 5
Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. In *4th International DFVLR Seminar on Foundations of Engineering Sciences*, pages 61--88, 1987. LNCS 295.
- 6
R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, 1994.

7

R. Chandra, A. Gupta, and J. L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):13--26, August 1994.

8

A. Chien, V. Karamcheti, and J. Plevyak. [The Concert System - Compiler and Runtime Support for Efficient Fine-Grained Concurrent Object-Oriented Programs](#). Technical Report UIUCDCS-R-93-1815, University of Illinois at Urbana-Champaign, Department of Computer Science, June 1993.

9

D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. [Parallel Programming in Split-C](#). In *Proceedings of Supercomputing 93*, pages 262--273, 1993.

10

D.E. Culler, A. Sah, K. E. Schauer, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of ASPLOS*, pages 166--175, 1991.

11

F. Bodin and P. Beckman and D. Gannon and S. Yang and S. Kesavan and A. Malony and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of Supercomputing '93*, pages 588--597, 1993.

12

S. Frølund. [Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages](#). In O. Lehrmann Madsen, editor, *ECOOP'92 European Conference on Object-Oriented Programming*, pages 185--196. Springer-Verlag, June 1992. Lecture Notes in Computer Science 615.

13

A. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic Object-Oriented Parallel Processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):33--47, May 1993.

14

W. Horwat. Concurrent Smalltalk on the Message Driven Processor. Master's thesis, MIT, May 1989.

15

C. Houck and G. Agha. [HAL: A High-level Actor Language and Its Distributed Implementation](#). In *Proceedings of th 21st International Conference on Parallel*

Processing (ICPP '92), volume II, pages 158--165, St. Charles, IL, August 1992.

16

L. V. Kale and S. Krishnan. [CHARM++: A Portable Concurrent Object Oriented System Based On C++](#). In Andreas Paepcke, editor, *Proceedings of OOPSLA 93'*. ACM Press, October 1993. ACM SIGPLAN Notices 28(10).

17

V. Karamcheti. Private Communication, 1994.

18

V. Karamcheti and A. A. Chien. [Concert -- Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware](#). In *Proceedings of Supercomputing '93*, November 1993.

19

V. Karamcheti and A.A. Chien. [A Comparison of Architectural Support for Messaging on the TMC CM-5 and the Cray T3D](#). In *Proceedings of International Symposium of Computer Architecture*, 1995.

20

K.E. Schauer and C.J. Scheiman. [Experience with Active Messages on the Meiko CS-2](#). In *Proceedings of IPPS '95*, 1995.

21

W. Kim and G. Agha. [Compilation of a Highly Parallel Actor-Based Language](#). In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 1--15. Springer-Verlag, 1993. LNCS 757.

22

W. Kim and G. Agha. A Scalable Implementation of Communication Abstractions in Actor Programming. in preparation, 1995.

23

E.J. Koldinger, J.S. Chase, and S.J. Eggers. Architectural Support for Single Address Space Operating Systems. In *ASPLOS V '92*, pages 175--186, 1992.

24

V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., 1994.

25

V. Kumar, A. Y. Grama, and V. N. Rao. Scalable Load Balancing Techniques for Parallel Computers. Technical Report 91-55, CS Dept., University of Minnesota, 1991. available via ftp `ftp.cs.umn.edu:/users/kumar/lb_MIMD.ps.Z`.

26

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

27

N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making Type Inference Practical. In *Proc. ECOOP'92*, pages 329--349. Springer-Verlag (LNCS 615), 1992.

28

R. Panwar and G. Agha. [A Methodology for Programming Scalable Architectures](#). *Journal of Parallel and Distributed Computing*, 22(3):479--487, September 1994.

29

K. Taura, S. Matsuoka, and A. Yonezawa. [An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers](#). In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, pages 218--228, May 1993.

30

K. Taura, S. Matsuoka, and A. Yonezawa. [ABCL/f: A Future-Based Polymorphic Typed Concurrent Object-Oriented Language - Its Design and Implementation](#). In G. E. Blelloch, K. Mani Chandy, and S. Jagannathan, editors, *DIMACS. Series in Discrete Mathematics and Theoretical Computer Science. vol 18. Specification of Parallel Algorithms*, pages 275--291. American Mathematical Society, 1994. Proceedings of DIMACS '94 Workshop.

31

Thinking Machine Corporation. *Connection Machine CM-5 Technical Summary*, revised edition edition, November 1992.

32

Thinking Machine Corporation. *CMMD Reference Manual Version 3.0*, May 1993.

33

N. Venkatasubramaniam, G. Agha, and C. Talcott. [Scalable Distributed Garbage Collection for Systems of Active Objects](#). In *Proceedings International Workshop*

on *Memory Management*, pages 441--451, St. Malo, France, September 1992. ACM SIGPLAN and INRIA, Springer-Verlag. Lecture Notes in Computer Science.

34

T. von Eicken, A. Basu, and V. Buch. [Low-Latency Communication over ATM Networks Using Active Messages](#). *IEEE Micro*, 15(1):46--53, February 1995.

35

T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. [Active Messages: a Mechanism for Integrated Communication and Computation](#). In *Proceedings of International Symposium of Computer Architectures*, pages 256--266, 1992.

36

M. Yasugi, S. Matsuoka, and A. Yonezawa. [ABCL/onEM-4: A New Software/Hardware Architecture for Object-Oriented Concurrent Computing on an Extended Dataflow Supercomputer](#). In *ICS '92*, pages 93--103, 1992.

37

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

WooYoung Kim

Tue Aug 15 20:46:00 CDT 1995