

©Copyright by Wooyoung Kim, 1997

THAL: AN ACTOR SYSTEM FOR
EFFICIENT AND SCALABLE CONCURRENT COMPUTING

BY
WOOYOUNG KIM

B.S., Seoul National University, 1987
M.S., Seoul National University, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

Abstract

Actors are a model of concurrent objects which unify synchronization and data abstraction boundaries. Because they hide details of parallel execution and present an abstract view of the computation, actors provide a promising building block for easy-to-use parallel programming systems. However, the practical success of the concurrent object model requires two conditions be satisfied. Flexible communication abstractions and their efficient implementations are the necessary conditions for the success of actors.

This thesis studies how to support communication between actors efficiently. First, we discuss communication patterns commonly arising in many parallel applications in the context of an experimental actor-based language, THAL. The language provides as communication abstractions concurrent call/return communication, delegation, broadcast, and local synchronization constraints. The thesis shows how the abstractions are efficiently implemented on stock-hardware distributed memory multicomputers. Specifically, we describe an experimental runtime system and compiler. The THAL runtime system recognizes and exploits the cost difference between local and remote message scheduling; it transparently supports actor's location independence; and, it implements non-blocking remote actor creation to improve utilization of computation resources. The THAL compiler incorporates a number of analysis and transformation techniques which work hand in hand with the runtime system. Among the techniques are: global data flow analysis to infer type information – the compiler optimizes code for each message send according to the type of its receiver expression; concurrency restoration through dependence analysis and source-to-source transformation; concurrency control with dependence analysis which allows multiple threads to be active on an actor with thread safety, *i.e.* with no interference between the threads. Experiments on a stock-hardware distributed memory multicomputer (CM-5) show that the compiler and the runtime system yield efficiency and scalability on applications with sufficiently large granularity which are comparable to the performance of other less flexible systems.

To my parents, JongSeok Kim and LeeSook Sung

Acknowledgements

I wish thank my parents, JongSeok Kim and LeeSook Sung, for their never-ending love, care, and encouragement. I would not be what I am without them. In particular, their continuous concern about my health is the bedrock of the thesis. I thank my brother, JooYoung, for the care that he has taken of our parents since my leaving my family, that the eldest son is supposed to do. I owe him a lot. I would also like to thank SooKyung Ham who has been my best friend and adviser since 1996 autumn, giving me love, support, and a reason to graduate.

My doctoral committee – Gul Agha, Prithviraj Banerjee, and David Padua – have been invaluable through their advice and comments. I am thankful to my advisor, Gul Agha, who has been my mentor since the very first moment I met him and has treated me like a peer, a friend, and a family member, guiding me to the right direction when I was wondering in my research, and not complaining of my night-owl-like work habit. I also thank Prithviraj Banerjee and David Padua for their kindness in agreeing to the last-minute scheduling of my final examination.

I give thanks to members of the Open Systems Laboratory. In particular, I thank Svend Frølund, Christopher Houck, Rajendra Panwar, Daniel Sturman, Mark Astley, and Shangping Ren, for their suggesting different ways of thinking and their reading my boring papers patiently and returning countless comments.

I thank JaeHoon Kim for his innumerable discussions with me. I would like to express my thank to members of the Champaign-Urbana Korean Catholic Church, including Jin-Woo Bak, NamJung Jung, Han-Sun Yang, Yoo Joong Kim, SangSook Park, JongHoon Lee, Chang-Hyun Kim, and Seung-Han Song, for their friendship and care. I especially thank Hong Seo Ryoo and SoonY Kang for caring me when I was in near-death situation in the summer of 1996.

Table of Contents

Chapter

| | | |
|----------|---|----|
| 1 | Introduction | 1 |
| 1.1 | The Actor model of computation | 2 |
| 1.2 | Contributions | 3 |
| 1.3 | History | 4 |
| 1.4 | Thesis Overview | 5 |
| 2 | Background | 6 |
| 2.1 | Object-Oriented Programming | 6 |
| 2.2 | Concurrent Object-Oriented Programming | 7 |
| 2.3 | Actor-Based Languages | 8 |
| 2.4 | Other COOP Languages | 9 |
| 3 | THAL: A Tailored High-level Actor Language | 11 |
| 3.1 | The Computation Model | 11 |
| 3.2 | Essential THAL Syntax | 13 |
| 3.3 | Groups | 15 |
| 3.4 | Communication Abstractions | 16 |
| 3.4.1 | Concurrent Call/Return Communication | 17 |
| 3.4.2 | Delegation | 18 |
| 3.4.3 | Local Synchronization Constraints | 19 |
| 3.4.4 | Group Communication Abstractions | 21 |
| 3.5 | Dynamic Actor Placement | 25 |
| 3.6 | Related work | 27 |
| 4 | Runtime Support | 28 |
| 4.1 | The Execution Model | 28 |
| 4.2 | The Design of the Runtime Support | 29 |

| | | |
|----------|--|-----------|
| 4.2.1 | Goals | 29 |
| 4.2.2 | The Architecture | 29 |
| 4.2.3 | Architecture of the Virtual Machine | 31 |
| 4.3 | Distributed Shared Message Queues | 32 |
| 4.3.1 | Scheduling with Deferred Message Stack | 33 |
| 4.3.2 | Static Method Dispatch using Deferred Message Stack | 36 |
| 4.3.3 | Dynamic Load Balance | 37 |
| 4.3.4 | Fairness | 37 |
| 4.4 | Distributed Name Server | 37 |
| 4.4.1 | Mail Address | 38 |
| 4.4.2 | Locality Descriptor | 38 |
| 4.4.3 | Distributed Name Table | 39 |
| 4.4.4 | Message Delivery Algorithm | 40 |
| 4.5 | Remote Actor Creation | 44 |
| 4.6 | Implementation of Communication Abstractions | 46 |
| 4.6.1 | Local Synchronization Constraints | 46 |
| 4.6.2 | Groups and Group Communications | 46 |
| 4.7 | Migration | 47 |
| 4.8 | Related Work | 48 |
| 5 | Compiler-time Analysis and Optimization | 50 |
| 5.1 | Type Inference | 50 |
| 5.2 | Transformation of Concurrent Call/Return Communication | 51 |
| 5.2.1 | Join Continuation Transformation: the Base Algorithm | 52 |
| 5.2.2 | Join Continuation Closure | 53 |
| 5.2.3 | Common Continuation Region Analysis | 58 |
| 5.2.4 | Method Fission | 59 |
| 5.2.5 | The Deadlock Problem | 61 |
| 5.3 | State Caching and Eager Write-Back | 65 |
| 5.4 | Local Actor Creation | 68 |
| 5.5 | Related Work | 68 |
| 6 | Performance Evaluation | 70 |
| 6.1 | The Runtime System on the TMC CM-5 | 70 |
| 6.2 | Performance of the Runtime Primitives | 72 |
| 6.3 | Fibonacci Number Generator | 72 |

| | | |
|----------|--|-----------|
| 6.4 | Systolic Matrix Multiplication | 73 |
| 6.5 | Bitonic Sort | 74 |
| 6.6 | N-Queen Problem | 75 |
| 6.7 | Adaptive Quadrature | 76 |
| 7 | Conclusion | 79 |
| | Bibliography | 81 |
| | Vita | 88 |

List of Tables

| | | |
|-----|---|----|
| 3.1 | Timing results of a set of C implementations of the Cholesky decomposition algorithm on a CM-5. | 23 |
| 4.1 | Performance comparison of the two different scheduling mechanisms. | 33 |
| 6.1 | Execution times of runtime primitives. | 72 |
| 6.2 | Execution times of the Fibonacci number generator. | 73 |
| 6.3 | Execution times of a systolic matrix multiplication problem. | 74 |
| 6.4 | Performance of a bitonic sorting problem. | 75 |
| 6.5 | Performance of a 13-Queen problem. | 77 |
| 6.6 | Performance of an adaptive quadrature problem. | 77 |

List of Figures

| | | |
|------|--|----|
| 1.1 | Primitive operations in the Actor model. | 3 |
| 3.1 | The semantic model of THAL. | 12 |
| 3.2 | The essential THAL syntax in BNF. | 13 |
| 3.3 | A bank account program. | 14 |
| 3.4 | An implementation of the N-Queen problem. | 18 |
| 3.5 | Message trajectory in delegation. | 19 |
| 3.6 | An example using delegation. | 20 |
| 3.7 | Tree construction implementations using CCRC and delegation. | 21 |
| 3.8 | An implementation of a systolic matrix multiplication using local synchronization constraints. | 22 |
| 3.9 | An actor implementation of the Gaussian elimination algorithm using the group abstractions. | 24 |
| 3.10 | Placement of sub-matrices in a systolic matrix multiplication. | 26 |
| 4.1 | An abstract view to the THAL runtime system. | 30 |
| 4.2 | The architecture of the THAL virtual machine. | 31 |
| 4.3 | The actor message structure. | 33 |
| 4.4 | An implementation of the Fibonacci number generator. | 35 |
| 4.5 | Scheduling local messages using function invocation. | 36 |
| 4.6 | Scheduling local messages using deferred message stack. | 36 |
| 4.7 | Indefinite postpone | 38 |
| 4.8 | The implementation of the locality descriptor. | 39 |
| 4.9 | Inconsistency in the name tables. | 41 |
| 4.10 | The message send and delivery algorithm. | 42 |
| 4.11 | Inconsistency correction and message forwarding in migration. | 43 |
| 4.12 | Remote actor creation. | 45 |
| 5.1 | The transformation result of <code>twoRsps</code> using continuation actors. | 54 |

| | | |
|------|---|----|
| 5.2 | Extracting continuations from a method with no branch. | 55 |
| 5.3 | Extracting continuations from a method with branches. | 56 |
| 5.4 | Extracting continuations from a method with loops. | 57 |
| 5.5 | The structure of the join continuation. | 58 |
| 5.6 | The coloring algorithm to compute maximal CCRs. | 59 |
| 5.7 | The coloring of an N-Queen implementation. | 60 |
| 5.8 | An incorrect fission result. | 62 |
| 5.9 | Fibonacci number generator with a recursive message send. | 63 |
| 5.10 | An incorrect join continuation transformation. | 64 |
| 5.11 | The transformation result. Each message send is augmented with a reply address. . . | 65 |
| 5.12 | A deadlock example involving indirect recursive message sends. | 66 |
| 5.13 | An algorithm to generate write-back statements for acquaintance variables. | 68 |
| 6.1 | The THAL runtime kernel on the TMC CM-5. | 71 |
| 6.2 | The communication topology of the implementation of the broadcast primitive. . . . | 71 |
| 6.3 | Comparison of performance of Fibonacci implementations with and without dynamic load balancing. | 73 |
| 6.4 | Comparison of performance of Fibonacci implementations on a single Sparc processor. . | 74 |
| 6.5 | Comparison of performance of THAL and Split-C implementations of systolic matrix multiplication. | 75 |
| 6.6 | Comparison of performance of a bitonic sorting problem with different problem sizes. . | 76 |
| 6.7 | Comparison of performance of a 13-queen problem with different placement strategies. . | 78 |
| 6.8 | Comparison of performance of an adaptive quadrature problem with different place- ment strategies. | 78 |

Chapter 1

Introduction

For the past 25 years the computer industry has witnessed a steady increase in computer performance: 18% - 35% per year depending on the class of computers. In particular, performance growth of microprocessors is phenomenal [111, 49, 94, 48, 50, 39, 51, 52, 94]; they have grown in performance by a factor of almost 2 every year. The performance improvement in off-the-shelf microprocessors together with the availability of different kinds of low-latency high-bandwidth interconnects has caused stock-hardware parallel machines to proliferate [65, 122, 134, 66, 60, 32, 77, 31, 110, 8]. Such machines offer a vast amount of computation capability to an extent that we have never dreamed of before.

Indeed, it has been a challenge from the beginning of the parallel computing era to develop a general-purpose programming system which allows users to enjoy the dramatically increased raw performance. Although a number of concurrent programming models have been proposed and actively investigated both in theory and in practice [17, 58, 18, 57, 1], programmers still write their parallel applications in a low-level message passing paradigm [112, 42] or a shared memory paradigm. By almost any measure, massively parallel MIMD machines remain difficult to program.

Because actors [53, 55, 30, 1] (or, concurrent active objects) hide details of parallel execution and present a transparent view of the computation, they provide a promising building block for efficient easy-to-use parallel programming systems. In particular, actors extend sequential object models by abstracting over threads of control along with data and procedures. The encapsulation of both control and data in a single actor makes parallelism and synchronization implicit. Actors specify concurrent computation using asynchronous communication. Use of unique abstract entities called mail addresses to name actors makes the communication location-independent. The encapsulation and the location independence simplify exploitation of data locality as well as enable actor relocation at execution time for scalable execution.

Since the introduction of the Actor model by Hewitt [53] in late 60's, a number of actor-based programming systems have been developed in software on single processor or multiprocessor platforms [88, 121, 56, 13, 16, 89, 11, 70] and implemented directly on silicon [12, 37]. With the availability of low-cost, high-performance microprocessors, it becomes a challenge to implement actor-based programming systems on stock-hardware multicomputers in an efficient and scalable way [119, 117, 28, 75]. It is challenging because actors are inherently concurrent and fine-grained while current-generation microprocessors support coarser-grained, sequential execution semantics. Furthermore, the cost difference between local and remote access is visible to applications on distributed memory multicomputers.

We argue that a key to a successful implementation is to make communication (*i.e.* message sending and scheduling) efficient while retaining the flexibility of actor communication. The thesis experimentally validates the argument by developing compile and run-time techniques on the implementation of an actor-based language, THAL. The language has been designed by adding flavor of sequentiality to the Actor model in controlled ways. As a result, the programmer is given complete control over execution grain size so that she may express the granularity at the most efficient level. Abstracting computation in terms of messages between actors hides architectural details of underlying platforms, improving programmability and portability.

Efficient execution of programs written in the language needs efficient compilation and runtime support. In particular, the extra sequentiality introduced for programmability should be eliminated for the sake of efficiency. In the dissertation we propose implementation techniques for actor primitives as well as message scheduling. We also propose a suite of compilation techniques to remove the sequentiality and restore concurrency in a profitable way. Finally, we evaluate the effectiveness of these techniques using a number of benchmarks on a stock-hardware multicomputer.

1.1 The Actor model of computation

Actors are autonomous components of a system which operate asynchronously. They encapsulate data, procedures to manipulate the data, and a reactive process which triggers local procedures in response to messages received. Because actors are conceptually concurrent and distributed, the simplest form of message passing between them is asynchronous.

A standard way to visualize an actor is to have an active object with a mail queue that is identified with a unique mail address. An actor may send messages to other actors whose mail addresses it knows of. Thus, the communication topology of actors is deterministic at any given instant in time. At the same time, mail addresses may be included in a message – enabling a dynamic communication topology. The uniqueness property of mail addresses provides for a global actor space: an actor can send another actor a message regardless of its current location as long as it knows the receiver’s mail address. It is also the uniqueness property that makes actors location independent.

Computation in actor systems is message-driven. Messages are buffered in the receiver’s mail queue and by default processed first-come-first-served. Processing a message involves triggering a method script. The method execution follows the dynamic data flow specified by the method script without unnecessary synchronization. Such synchronization may be necessary in other programming models due to potential uncertainty in determining real dependencies in sequential control constructs. The model does not enforce any specific constraint on the order of message delivery. In particular, two messages sent by an actor to the same actor may arrive in an order different from their sending order. By implication the Actor model abstracts over possible dynamic routing. Although message arrival order is nondeterministic, all messages are guaranteed eventual reception. The guarantee of delivery is a useful assumption in reasoning about systems where fault-tolerance need not be explicitly modeled at the level of an application.

In response to a message, an actor may *create* new actors, *send* messages to actors, and *change its state* with which it responds to the next message (Figure 1.1). These actions are implemented by extending a sequential language with the following operators:

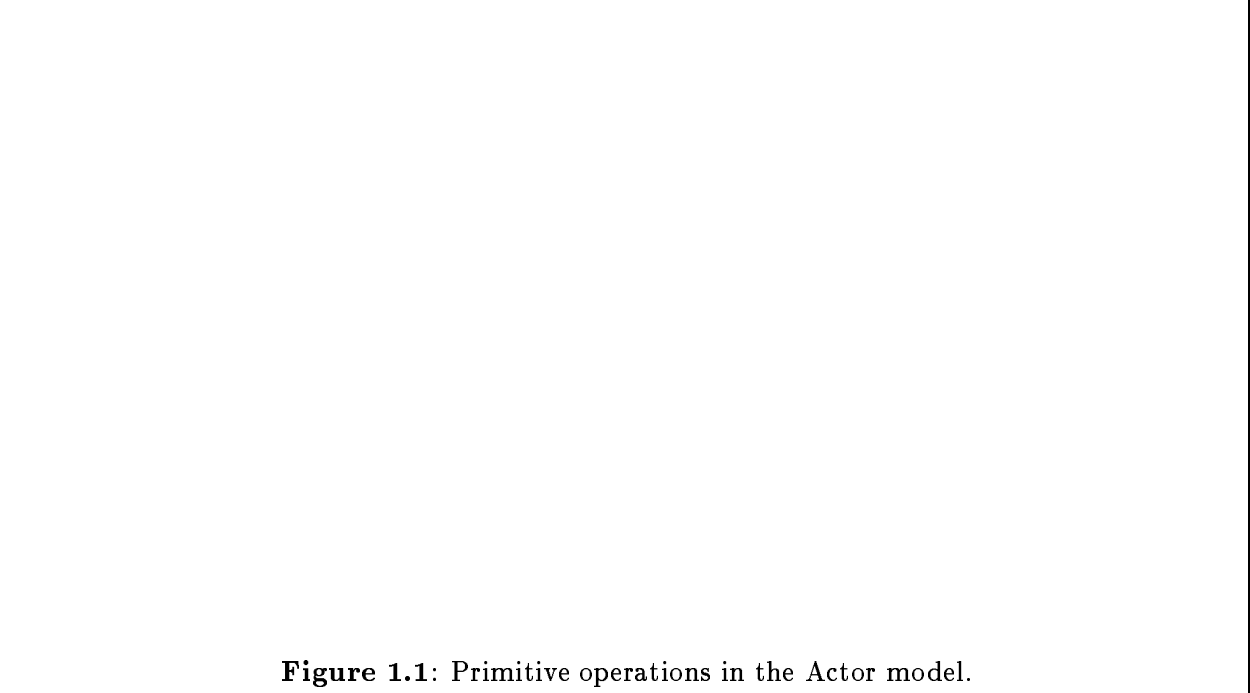


Figure 1.1: Primitive operations in the Actor model.

- **create** takes a behavior description and creates an actor. The operator may take additional initialization arguments.
- **send-to** takes the receiver's mail address and puts the message into its mail queue.
- **become** and **update** take state variables and replace their values with new ones. The former changes the state collectively while the latter does it individually.

State change is *atomic*. Atomic state change serializes message reception [3], thereby offering synchronization on the method boundary. Also, state change may finish before all the other actions in response to a message have been completed. An actor may process the next message without violating the atomic state change requirement as long as the next state is specified. This allows multiple threads to be active on a single actor under a multiple-reader, single-writer constraint [3]. Furthermore, mutually independent actions in a method may be executed concurrently (*i.e.*, *internal concurrency*). Since no state is shared among actors, it is unnecessary to provide hardware or software support for maintaining consistency of shared data.

The Actor model is fairly primitive and abstract so that it may be used to model many different concurrent computing systems. For example, the J-machine [36, 37] is a fine-grained concurrent computer which directly implements the Actor model on its hardware. Moreover, the actor operators form a powerful set upon which to build a wide range of higher-level abstractions [2].

1.2 Contributions

The contributions of the thesis are summarized as follows:

- We designed an actor-based language THAL. The language supports a range of high-level abstractions which help specify frequent interaction patterns in concurrent computation.

- We designed and implemented a message delivery subsystem which transparently supports actor's location independence. Mail addresses are defined to guarantee location transparency while facilitating name translation.
- The design of mail address makes unpredictable the time for remote actor creation. We developed a non-blocking remote actor creation mechanism using locally-allocated globally-unique entities called *aliases* which overlaps remote creation with other useful computation.
- We designed and implemented a message scheduling mechanism using *distributed shared message queue* and *tail calling* which recognizes cost difference in scheduling local and remote messages and exploits it in the scheduling. Part of the implementation is exposed to the compiler to optimize local message scheduling.
- Among the THAL communication abstractions is *concurrent call/return communication* that is analogous to a function invocation abstraction in sequential programming. We designed an analysis technique which identifies independent CCRCs and implemented a source-to-source transformation technique which maximally retains profitable concurrency.

1.3 History

The precursor of THAL is HAL, a high-level Actor language [62, 63] designed and implemented by Christopher Houck. HAL had Lisp-like syntax and focused on remote execution of actors to provide data locality. Its programs were compiled to C and executed on top of CHARM, an architecture-independent runtime system [73]. The language featured inheritance, limited forms of reflection, and synchronization constraints. HAL had been used as a test-bed for experimenting with new language constructs and dependability methods.

After Houck graduated and left the Open Systems Laboratory,¹ the language underwent a series of evolution under the same name. First, join continuation transformation was implemented in the compiler and the abstraction for synchronization constraints were extended [78]. Then, a notion of group was introduced to the language and the communication mechanism was extended with a broadcast abstraction [5]. At this time, the language syntax was changed from Lisp-like one to Modula-like one. Finally, we implemented a runtime system with migration capability. It was initially operational on a network of DEC workstations which was connected by Ethernet and ran ULTRIX V4.2A. Then, the runtime system was ported to the Thinking Machine Corporation CM-5 [79].

During the initial period of experimentation on a CM-5, we found that the versatility of HAL caused an intolerable amount of performance overhead on program execution. The finding led us to drop support for reflection and inheritance. At this time we changed the name to THAL [103],² an acronym standing for Tailored High-level Actor Language. Currently, THAL supports multi-dimensional arrays and interface to other conventional sequential languages, such as C. Its compiler features full support for type inference and a number of optimizations including the common continuation region analysis [80].

¹to work on Mosaic and then Netscape.

²THAL should be pronounced as [tal]. It is a Korean term (탈) for masks that are used in a traditional Korean dance-and-play which is loved and enjoyed especially by common folk.

1.4 Thesis Overview

The rest of the thesis is organized as follows. We summarize some background on our research in Chapter 2. The description of the design and evaluation of THAL are given in Chapter 3. Chapter 4 presents the design and implementation of the runtime support. Compiler-time optimizations are discussed in Chapter 5. We report in Chapter 6 on the performance of the runtime primitives and other evaluation results on a stock-hardware multicomputer. Chapter 7 concludes the thesis with a brief summary and a sketch of future work.

Chapter 2

Background

The roots of actor languages extend to functional and object-oriented programming paradigms. Although both paradigms had been first developed as early as in the late 50's and early 60's, it was not until 70's that the attempt to unify them with a model of concurrency was initiated. Since its introduction by Carl Hewitt, interest in actor languages have intensified through the 80's and 90's – leading to the development of dozens of languages and implementations. In this chapter, we survey some of these programming languages as well as other representative concurrent object-oriented programming languages.

2.1 Object-Oriented Programming

Object-oriented programming paradigm encourages modular design and knowledge sharing (in particular, code reuse). The concept of object-oriented programming has its root in SIMULA [35]. Since then, it has been developed as an important software engineering methodology through a series of breakthroughs in the field of programming language theory and practice.

In object-oriented languages, computation is abstracted as communication between a collection of objects. Each object is an instance of an abstract data type (ADT) (often called *class*) which encapsulates *instance variables* and *methods* that operate on them. An ADT has an external interface and its internals are invisible from the outside; the internals may only be accessed through the interface. Such *encapsulation* or *information hiding* minimizes interdependencies among separately-written ADTs and allows changes to the internals of an ADT to be made safely without affecting its users. Since the interface of an ADT captures the “essential” attributes of the ADT, the user of an object need not be concerned with the ADT itself or its instance objects but only with the abstract interface.

Computation proceeds as objects invoke procedures in other objects. A procedure identifier together with its arguments is called as a message and the process of invoking procedures known as passing messages or *communication*; a message invokes a method (*i.e.* procedure) at the receiving object. However, the nomenclature of communication is misleading in a sense that the model of invocation is closer to procedure activation in imperative languages. Unlike actors, the communication is inherently synchronous and blocking.

In general, which method is to be invoked is not known until the message's dispatch time because a method in an object may share the same name with one in another object. Moreover,

the meaning of a method cannot be determined statically since the method definition may be shared by two or more objects. The semantics of *dynamic binding* may vary from language to language, depending on how the dynamic method lookup is implemented. The semantics are further affected by the knowledge sharing mechanism a language adopts.

Delegation versus Inheritance: a philosophical debate

There are two common mechanisms that people use to represent knowledge about generalizations they make from experience with concrete examples. The first is based on the idea of abstract sets; the set (or class) abstracts out what one believes is true about all the examples she experienced. The other is to use prototypical objects. One generalizes a concept incrementally as new examples arise by making new analogies to the previous concept that preserves some aspects of the “defaults” for that concept and ignoring others. The traditional controversy between the two gives rise to two mechanisms, inheritance and delegation, for sharing behavior between related objects in object oriented languages.

Implementing the set-theoretic approach to sharing knowledge in object-oriented systems is traditionally done by inheritance. An object called class encodes common behavior for a set of objects. All instances of a class share the same behavior but can maintain unique values for a set of state variables. A class may inherit from other classes. The inheriting class (or *subclass*) may add methods and instance variables to the class. When an object of a subclass receives a message, it tries to respond to it using its own methods. If it fails, it climbs up the inheritance tree to respond to the message.

Delegation implements the prototype approach to sharing knowledge in object oriented systems. It appears in actor languages [88, 135, 29] and several Lisp-based object oriented systems such as Director [72], T [67], Orbit [113], and others. An object shares knowledge with a prototype by simply having the prototype as its acquaintance; the object may also keep its personal behavior idiosyncratic to itself. When it receives a message, it first attempts to respond to the message using the behavior stored in its personal part. If it fails, it forwards (or *delegates*) the message onto its prototypes to see if one can respond to the message.

2.2 Concurrent Object-Oriented Programming

Object-orientation is a useful methodology to attack program complexity; however, it does not address issues of concurrency and distribution. Concurrent objects combine concurrency with object orientation by associating a notion of process or thread with them. They are promising for programming on parallel computers because they hide many details related to parallel execution behind abstract interfaces of objects, thereby allowing programmers to concentrate on algorithm design. Increasing deployment of multiple node systems with high-bandwidth, low-latency interconnects in recent years has been an impetus for active and extensive research on concurrent object-oriented programming languages.

COOP languages differ in how processes are associated with objects. In process-based COOP languages, process and object are two separate notions. An object with a process is said *active*. How much a language distinguishes the two notions determines its flavor. The extent is also reflected on synchronization abstractions that the language provides. By contrast, the distinction between active and passive objects is removed in actor languages because actors are active by definition.

Every actor is associated with a thread; however, the thread makes its presence manifest only when a message is scheduled.

2.3 Actor-Based Languages

The Actor model was first introduced by Carl Hewitt [53], refined by many others [55, 54, 30, 56] and defined in its current standard form by Agha [1]. Particularly, since its introduction many actor-based languages [121, 13, 88, 16, 11, 85, 135, 89, 70, 29] have been proposed for programming concurrent computation.

Act1 [88] was an early actor language which was implemented in Lisp. It supported a number of abstractions which are still found in other contemporary actor-based languages. For example, it used *continuations* to support bidirectional control structure of sending a request and receiving a reply. *Delegation* was used to share knowledge among actors and implement error handling. The language also used *futures* for parallel computation and *serializers* for synchronization. Another actor language with Lisp-based implementation is Acore [89] which even borrowed the syntax of Lisp. Acore was the first language based on the Actor model defined in [1] so that a mutable actor implicitly serializes messages it receives and the expressions in a message handler may be evaluated concurrently.

Cantor [11] is the first actor language that was implemented and executed on multicomputers. The “essential cantor” described in [11] preserved message order between pairs of directly communicating actors. It originally employed dynamic typing which was subsequently replaced with strong typing through the use of type declaration. Cantor version 2.2 also added vectors along with internal iteration.

Another actor language targeted for parallel execution is Plasma-II [85], a parallel extension of Plasma which was the first actor language defined by Carl Hewitt. Plasma-II was designed to be executed on a set of virtual machines distributed on heterogeneous platforms. It allowed programmers to specify distribution of actors and supported broadcast communication abstraction for data parallel style of programming.

The most commercially successful actor-based language up to now is Rosette [125] which was used as a language for the interpreter of the extensible services switch in the Carnot project at Microelectronics and Computer Technology Corporation (MCC). The language continues to be used to provide heterogeneous interoperability for middleware in intranet and enterprise integration software. Rosette is prototype-based and supports inherent concurrency, inheritance, and reflection. Synchronization in Rosette is specified by *enabled set* which defines what methods can be executed under the current state of an actor.

In some ways the Actor model is a very primitive model. Thus, many actor languages extended it to improve their programmability. Along these lines is a family of languages [133, 134, 120] rooted at ABCL/1 [135].¹ Though they differ from one another in detail, all the languages share the core computation model proposed in ABCL/1. Message sending order is preserved between pairs of directly communicating actors, as in Cantor. ABCL/1 supports three asynchronous message sending mechanisms called *now*, *future*, and *past*. The first two have blocking semantics whereas the last is non-blocking. Another actor language, Concurrent Aggregates (CA), extends the Actor model with inheritance and aggregates. An aggregate is a group of actors of the same kind. All

¹ABCL stands for Actor-Based Concurrent Language.

constituent actors share the same name. A message sent to the aggregates is processed by one and only one constituent but which constituent receives the message is left unspecified (*i.e.*, *one-to-one-of-many type of communication*). Unlike the Actor model, every message send in CA expects a reply by default [28, 75].

All of the above-mentioned actor languages have been designed and implemented from scratch. A different approach involves extending an existing sequential object-oriented language with the concurrency semantics of actors. In this approach, actors inherit their actions from a single *Actor* class which wraps sequential objects with actor semantics. Two examples following this approach are Actalk [21] and actra [124] which were built upon Smalltalk-80. Actalk implemented actors by augmenting ordinary Smalltalk objects with asynchronous message passing and message buffering. Actra was implemented by modifying the Smalltalk virtual machine. In contrast to the basic Actor model, communication between actors in Actra was synchronous. Another language following the extension approach is ACT++ [70]; it extended C++ with a class hierarchy which provides the concurrency abstraction of the Actor model.

2.4 Other COOP Languages

The desire to leverage the existing compiler technology motivates implementing a COOP language by extending an existing sequential object-oriented language, such as C++ or Smalltalk, with a notion of process or thread. In particular, given the popularity and portability of C++, a number of COOP languages based on C++ have proliferated [23, 69, 47, 73, 27, 87, 86, 95]. We describe a few examples below.

Compositional C++ (CC++) [69] extends C++ with a number of abstractions for process creation and synchronization. Synchronization is done via special shared variables. COOL [27] is targeted for shared-memory multiprocessors. Invocation of a parallel function creates a thread which executes asynchronously. Threads communicate through shared data and synchronize using monitors and condition variables. Mentat [47] and Charm++ [73] are similar in that both distinguish parallel objects from sequential ones; programmers are required to specify what classes are to be executed in parallel. Mentat objects map one-to-one onto processes in a virtual machine. By contrast, Charm++ requires programmers take the responsibility of mapping of objects onto processing nodes. pC++ [87], C** [86], and pSather [95] are all C++-based COOP languages which are designed to support data parallelism. They differ in how to initiate data parallel execution.

CST [61, 38] and DistributedConcurrentSmalltalk (DCS) [96] are two of many COOP languages which extended Smalltalk-80 [43]. CST supports concurrency using locks, asynchronous message passing, and distributed objects. Distributed objects are similar to aggregates in CA and are equipped with similar communication mechanisms. DCS is an extension of ConcurrentSmalltalk [96] to a distributed interpersonal environment. Concurrency is supported with asynchronous as well as synchronous method call as well as synchronous thread manipulation. DCST allows multiple processes in a single object. Synchronization of the processes may be specified by a *method relation* which defines an exclusive relation between two methods or by a *guard* expression which defines when a method is enabled.

Both Emerald [64] and Orca [116] support encapsulated abstract data types but without inheritance. Furthermore, they have clear distinction of a process and an object. For example, in Emerald, multiple threads of control may be active concurrently within a single object. Synchronization is provided by monitors. Unlike other concurrent languages, communication between

processes is synchronous. Orca implements the shared single address space on distribute memory multicomputers and maintains the coherency by using shared objects and reliable broadcasting. Parallel execution is accomplished by dynamically creating processes on multiple processors.

SOMIW Operating System (SOS) [108] is an object-oriented distributed system which was implemented in C++ on top of UNIXTM. It was designed to be language-independent by adopting a library approach and providing a language-independent interface. SOS supports a notion of groups called Fragmented Objects (FO) and object migration which involves both data and code migration. SOS objects communication with one another using synchronous, asynchronous, or multicast communication. Another language that supports code migration is Java [45] which promotes architectural neutrality, the property of “write-once, run-anywhere.” Java is designed as a simplified derivative of C++ and supports a limited form of concurrency through lightweight threads and remote method invocation. Although object migration is yet to be supported, a programmer may mimic it by remotely creating an object and explicitly forwarding its associated code.

Chapter 3

THAL: A Tailored High-level Actor Language

THAL is a high-level language based on actors; it is a descendant of HAL [63, 5]. THAL allows a programmer to create actors, initialize their behaviors, and send them messages. As the computation unfolds, new messages are generated, new actors are created, and existing actors undergo state change. Data flow and control flow in a program are concurrent and implicit; a programmer thinks in terms of what an actor does, not about how to thread the execution of different actors to ensure a correct (or efficient) order of execution. Although communication in actors is point-to-point, non-blocking, asynchronous, and thus buffered, THAL simplifies programming by providing other forms of communication at the application level.

An important goal of THAL is to provide high performance execution on stock-hardware multicomputers. THAL addresses two important problems to achieve this goal. First, processing nodes of stock-hardware multicomputers have large overhead in utilizing fine-grain concurrency offered by actors. Thus, THAL is designed with the understanding that not all available concurrency in an actor program may be exploited in execution. Specifically, the execution semantics of THAL is defined by systematically introducing sequentiality to the Actor model while preserving the concurrency semantics of actor programs. Second, although naming in actors is location independent, different actor placement strategies result in significantly varying performance characteristics. Actor placement subsumes what is usually termed partitioning and distribution as well as actor migration. THAL makes actor locality potentially visible to programmers to give them explicit control over actor placement. However, programmers still do not need to keep track of the location to send a message to an actor.

3.1 The Computation Model

THAL supports a message-driven model of execution. Message reception by an actor creates a thread on the actor which executes the specified method with the message as its argument. Thus, thread execution is *reactive*. Only message reception can initiate thread execution. Furthermore, thread execution is *atomic* and *finite*. Once successfully launched, a thread executes to completion without blocking. The atomicity requirement allows at most one thread to be active on an actor at any time. As a direct consequence, an active thread is given exclusive control over an actor's state (Figure 3.1). This atomicity is a natural basis upon which a number of synchronization




Figure 3.1: The semantic model of THAL. Each bubble represents an actor which encapsulates a behavior, a mail queue, and a state. A thread is created when an actor processes a message from its mail queue. At most one thread per actor is allowed to be active at any time. Mail queue is not shown in the picture.

mechanisms are built. It also simplifies the task of keeping actors data-consistent. Although a thread execution may make incremental state change, the net result is visible to the outside as if it were done atomically.

Thread execution is guaranteed to terminate in a finite number of steps. An implication of the finiteness is that unbounded loops are not allowed. Even if a programmer unintentionally specifies a potentially unbounded loop the compiler should transform it to a finite one by bounding the number of iterations. Infinite computation, if ever needed, may be expressed alternatively by sending messages to *self*.¹ Finally, method execution is *sequential*: no concurrency inside a method is exploited. The overhead of exploiting internal concurrency is not justified on current generation stock-hardware multicomputers.

The semantics of atomicity and sequentiality allows THAL to support a multi-lingual paradigm to some extent. Specifically, concurrency and synchronization constructs may be used to glue together and coordinate otherwise independent sequential procedures written in different languages. Component procedures of existing sequential application may even be imported. In this way, THAL may facilitate incremental migration of legacy codes to parallel ones. However, not all sequential procedures may be used. To be eligible a procedure must be side effect free: its functionality should be characterized solely by inputs and an output. Furthermore, it should not contain any unbounded loop. The latter requirement is rather demanding in that the compiler may not easily detect the presence of an unbounded loop. Currently, THAL supports C and Fortran interface declarations [98].

¹Explicit sending of a message to *self* does not hurt fairness.

3.2 Essential THAL Syntax

A THAL program consists of a script called `main` and behavior templates. `main` signifies the starting point of program execution. Behavior templates are similar to *classes* in other object-oriented programming languages. A behavior template is used to define the behavior of a new actor. Unlike sequential object-oriented languages, neither global variables nor class variables are provided: actors are encapsulated and they do not share state. This simplifies distribution of actors. It also allows concurrent access to them without any interference. Another characteristic that differentiates THAL from other object-oriented languages is that it lacks support for inheritance, although it could easily be incorporated in the language. In the tradition of actor languages, we prefer to use delegation.

A behavior template (or, simply behavior) is composed of *acquaintance* variables and a set of method definitions.² The method definitions together with the values assigned to the acquaintance variables comprise an actor's state. The former is immutable while the latter is mutable. A behavior may have an optional `init` method. An `init` method is hidden from outside and executed exactly once when an actor is created. The method customizes the creation by specifically prescribing the actor's initial state. An ordinary method specifies a response to a message. A method is defined by an optional local variable declaration followed by a sequence of operations, such as actor creation, state change, and message send. Figure 3.2 summarizes the syntax of THAL.

```

<program>      ::= <behvs>* <main>
<behvs>        ::= behv <behv-id> [<var-decl>] [<init>] <methods> end
<init>         ::= init ( <formal-parameter-list> ) <stmt>+ end
<methods>      ::= method <meth-selector> [<var-decl>] <stmt>+ end
<meth-selector> ::= <meth-name> ( <formal-parameter-list> )
<var-decl>     ::= | <var-list> |
<main>         ::= main [<var-decl>] <stmt>+ end

```

Figure 3.2: The essential THAL syntax in BNF.

Actors are created and initialized using the `new` primitive operator. The operator takes a behavior name and a set of arguments to the `init` method. It may also take an optional location expression which specifies where to create the actor. By default, an actor is created *locally*. State change is incrementally specified with the `update` primitive operator; an update is nothing more than an assignment to an acquaintance variable. Thus, updates encountered in a method execution collectively define the actor's next state with which it responds to the next message.

A canonical example of a bank account program is given in Figure 3.3. “`%%`” starts a comment which extends to the end of the line. The program creates a checking account with an initial balance of \$100 owned by Mark. Two messages are then sent to the account to deposit \$200 and withdraw \$150, respectively. Both acquaintance variables and temporary variables are declared by enclosing a list of identifiers with a pair of vertical bars. `curr_bal` and `owner` are acquaintance variables and `checking` and `mark` are temporary variables. Thus, assignments to `curr_bal` represent *updates* while that to `checking` is a binding.

²A behavior may also have function definitions. Functions are private methods and may not be invoked from outside.


```

behv CheckingAccount
  | curr_bal,owner |      %% acquaintance variable declaration
  init (ib,io)           %% init method definition
    curr_bal = ib;        %% update to curr_bal
    owner = io;           %% update to owner
  end
  method deposit (id)
    curr_bal = curr_bal + id;
  end
  method withdrawal (iw, teller)
    if (iw > curr_bal) then
      teller <- over_drawn(iw - curr_bal);
    else
      curr_bal = curr_bal - iw;
      teller <- done();
    end
  end
  method balance ()
    owner <- balance(curr_bal);
  end
end

main                                %% main script
  | checking,mark,teller... | %% temporary variable declaration
  ...
  checking = CheckingAccount.new(100,mark); %% a binding to checking
  checking <- deposit(200);                %% asynchronous message send
  checking <- withdrawal(150,teller);
end

```

Figure 3.3: A bank account program. Left arrows represent asynchronous message sends. Assignments to *curr_bal* represent *updates* whereas the assignment to *checking* is a binding to a temporary variable.

What makes THAL unique is that it is untyped but strongly type checked at compile time. Indeed, type specification is redundant in variable declaration. Instead, the compiler type-checks a program by analyzing the global data flow and scrutinizing its type consistency. Type information for each variable is inferred as a by-product (Section 5.1).

A left arrow specifies sending an asynchronous message. It was chosen to signify physical transmission of a message; it demarcates a receiver from a message. A method selector and a set of arguments comprise a message. A method selector is either a method name or a variable. A method name is the first class object and may be assigned to a variable or sent in a message. Having the first-class method names allows programmers to manipulate continuations in more flexible ways.

3.3 Groups

A group is a collection of homogeneous actors which have the same method definitions but differ in their states. A group is given a unique name which is shared by all its members. There are a number of computations that may be concisely modeled using groups. Data parallel or SPMD (single program multiple data) computations are the most conspicuous examples among others. Using groups a collection of data is partitioned and distributed among members. Data parallel execution is modeled by broadcasting a message to all the members.

Groups in THAL are based on a simple restrictive model; the goal is to simplify their specification and to provide an efficient implementation of data parallel computations. Two characteristics are important in this respect. First, groups are flat: they may not be nested and may not overlap. Second, membership is static: no members may be added to or removed from a group and the size of a group is fixed at its creation time.

The actor primitives are sufficient to express data parallel computations. Member actors may be created by repeating actor creation. Broadcasting a message is implemented by sending a copy to each member. Nonetheless, using point-to-point communication to specify a broadcast does not make perspicuous the homogeneity of group members; it thus complicates reasoning about program behavior. Besides complicating programmability and reducing readability (refer to Section 3.4.4), an implementation using only actor primitives suffers from at least three sources of inefficiency. First, creation of members by repeated actor creation does not exploit homogeneity of the member actors in memory management. Second, the implementation of the broadcast communication in terms of explicit point-to-point communication increases communication cost; the same message needs to be marshaled and to traverse the network as many times as the number of remote members. Lastly, available network bandwidth may be underutilized because the implementation minimizes involvement of other processing elements, eliminating the opportunity of more concurrent sending.

A group is created using `grpnew` or `clone`. `grpnew` is similar to `new` but takes an additional argument representing the size of the group. `clone` is a specialized `grpnew` in that it places exactly one actor on each processing node. Both operators return a unique group identifier. Group identifiers may be communicated in a message, just like mail addresses. Creation of a group distributes its members across processing nodes; a specific placement may be specified by the programmer or determined by the system. From the programmer's point of view, member actors constitute elements of an ordered collection. A member is specified by qualifying its group identifier with an index expression.

In addition to group identifiers, four pseudo variables are made available to member actors to facilitate naming their groups and peer members. Use of these pseudo variables is legitimate only

when used in member actors; the compiler is responsible for checking their validity. The first one is `mygrp` which refers to the group a member belongs to. `mygrpidx` and `mygrpsize` denote a member's relative position in the group and the group size, respectively. The last one is `mygrpcreator` which refers to the creator of the group. The first three pseudo variables may be used to name peer member actors. Member actors may share information through their creator. They also synchronize and coordinate computations by naming their creator.

Array

Arrays are extensively used in many numerical applications. Specification of computations other than numerical ones may also be greatly simplified using arrays. Arrays are provided in THAL as a degenerate group. Semantically, arrays are viewed as an ordered collection of primitive actors which export only implicit read/write methods. For convenience, a notation to specify multidimensional arrays is provided. For example, an (i,j) -th element of a two-dimensional array `temp` may be represented by `temp[i][j]`.

Before being accessed, an array need be explicitly allocated using the `array` operator. However, the programmer is not allowed to deallocate any array; deallocation is done automatically through garbage collection. The `array` operator takes a list of constants each of which denotes the size of a dimension. The element type need not be specified; it is inferred from the context by the compiler (Section 5.1).

Arrays may be sent in a message. Unlike sending an ordinary group, sending an array causes a copy of the whole array to appear in the destination node if the destination is different from the source. If both nodes are the same, only a handle to the array is sent to the receiver. This rather awkward semantics has to do with access locality. Local access is much cheaper than remote access. When an array is multi-dimensional, a contiguous sub-dimension of the array may be sent. For example, the following message sends are all valid.

```
...
arr = array (3,4,5); %% allocate 3 dimensional array and
                    %% assign it to arr
r1 <- m1 (arr);      %% send the entire array
r2 <- m2 (arr [1][1][1]); %% send the first element
r3 <- m3 (arr [1][1]); %% send the first vector of size 5
r4 <- m4 (arr [1]);   %% send the first plane of size 4x5
r5 <- m5 (arr [1]{1:3}); %% another way to send the entire array
```

3.4 Communication Abstractions

Although point-to-point non-blocking asynchronous message passing is efficient as well as fundamental, it is inconvenient to use in some cases. In this section, we describe three communication abstractions which complement the point-to-point non-blocking asynchronous message passing, namely concurrent call/return communication, delegation, and broadcast.

3.4.1 Concurrent Call/Return Communication

In many cases, a method execution may require information from other actors to complete: such computations may be represented by sending a request and waiting for a reply to continue. We call this *call/return communication*. In the Actor model, call/return communication requires explicit manipulation of continuation actors and synchronization because communication in actors is point-to-point and non-blocking. Both of these characteristics provide an efficient execution model but are insufficient as programming abstractions. Like many earlier actor languages [89, 135], THAL provides an abstract way of specifying call/return communication without requiring the programmer to necessarily manipulate continuations

Call/return communication may be best expressed in the actor paradigm using the concurrent call/return communication (CCRC) abstraction; CCRC directly models the call/return communication by having continuation and synchronization implicit in its semantics. Examples of the concurrent call/return communication abstraction are `ask` [89], `now` [135], `blocking send` [29], and `request` [5].

THAL supports CCRC using two constructs, *request* and *reply*. They are represented with “.” and `reply`, respectively. Execution of a request blocks the sender until a reply is sent back. The sender may be context switched to avoid wasting compute cycles. `reply` is used by the callee to send a result back to the caller. Reply messages to the `nil` actor are consumed by the system and never delivered.

Consider an N-Queen problem which computes the number of different ways to place N queens on an $N \times N$ chess board such that all queens are placed in a safe position, *i.e.*, no two queens are in a row, a column, or a diagonal. An actor implementation may carry out the computation by dynamically creating actors; each actor creates children, waits for results from them, sums results, and sends the sum to its parent actor. With non-blocking asynchronous communication, a programmer need encapsulate into a separate actor both summing the results and sending the sum to its parent and explicitly specify synchronizations. Figure 3.4 illustrates a more succinct alternate implementation [117] using CCRC.

The semantics of CCRC allows concurrent execution of mutually independent message sends. A sender continues to execute even after sending a request as long as the continuation does not need the reply right away. It blocks when it cannot proceed further without the results from the previous requests. Consider a statement:

```
val = add(r1.m1(), r2.m2());
```

Execution of `r2.m2()` does not need the result of `r1.m1()`. As a result, the sender executes the second request as soon as it sends the first, which allows `r1` and `r2` to proceed concurrently. The result from `r2.m2()` may be available even before that from `r1.m1()`. Furthermore, the result of each request may depend on the message reception order at the receiving actor if `r1` and `r2` are indeed the same. The sender blocks before it calls `add` because it requires the two results from the requests.

This is in contrast to remote procedure call (RPC): RPC semantics guarantee that, for any two RPCs in a method, all computations caused by the first RPC are completed before the second call is made (*i.e.*, *no* concurrency). Thus, execution of the above statement with the RPC semantics completes the execution of `r1.m1()` before that of `r2.m2()`. RPC transfers control as well as data to ensure sequential execution whereas CCRC ships data off without sending control.

```

method compute (col,diag1,diag2,maxcol,depth)
...
  initialize the array replies.
  for i = 1, N do
    if (c1 > maxcol) then break; end
    if ((col|c1) == maxcol) then
      sols = sols + 1;
    else
      where = random () % #no_nodes;
      replies[i] = (NQueen.new() on where).
        compute((col|c1),(((diag1|c1)<<1)&maxcol),
          ((diag2|c1)>>1),maxcol,depth+1);
      c1 = ((c1<<1)+c)&~c;
    end
  end
  sums replies[i] into sum
  reply (sum+sols);
end

```

Figure 3.4: An implementation of the N-Queen problem. The method computes the number of the solutions of an N-Queen problem.

3.4.2 Delegation

THAL provides `delegate` (denoted with `!`) as a separate communication abstraction which realizes delegation (Section 2.1). When a sender delegates a message the reply address to the continuation is replaced with the sender's reply address. The latter is used by the receiver as its reply address. As a result, a reply is directly sent to the client (Figure 3.5) (cf. tail recursive optimization). Furthermore, delegating actors (e.g., `Broker` in Figure 3.5) need not block because a reply bypasses the actors *en route* to the client. Note that clients are assumed to send messages using either asynchronous communication or call/return communication.

In addition to delegation, `delegate` may efficiently implement certain communication patterns which frequently arise in many applications. One example is the implementation of exception handling. Exception handlers may be collected and implemented as a system of actors each of which handles a specific exception. The mail address of the receptionist of the system may be known to actors at their creation time or communicated to them in the course of computation. When an exception raises, a message is sent to the receptionist which delegates it to an appropriate handler actor. Another is implementation of a multi-node web server. Requests may be sent to a gateway node which is known to the outside world. The gateway node distributes requests to server nodes taking into account balancing the load among the nodes. Replies are sent directly to clients bypassing the gateway node.

The tree construction phase in an actor implementation of the Barnes-Hut algorithm [97] illustrates advantages of delegation over CCRC. The phase begins with each body sending its coordinates to the root. The message climbs down the partially constructed tree and the body is added

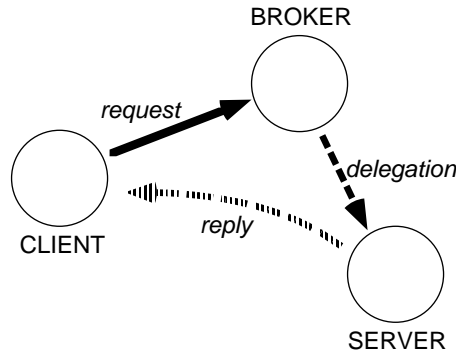


Figure 3.5: Message trajectory in delegation. A client sends a request to a broker. It does not matter whom it receives a reply from. The broker delegates a message to a server. The server sends a reply to the given destination. It does not know the reply goes to the original client.

appropriately. For a body to be sure, it must be notified of its addition. Figure 3.6 shows the definition of `add_body` which implements the tree construction phase using delegation.

CCRC offers an easier mechanism for implementing the multi-party communication: CCRC implements the multi-party communication in terms of point-to-point communication without requiring explicit specification of synchronization and continuation. However, an implementation using CCRC incurs unnecessary overhead. For example, the inner tree nodes involved are unnecessarily blocked. Since the root is also blocked it becomes a bottleneck and the tree construction is completely serialized. Furthermore, replies are unnecessarily passed through inner nodes; they never use the reply but just forward it (Figure 3.7(a)).

These disadvantages translate into advantages of using delegation. First, unnecessary reply communication is eliminated since a reply is now directly sent to its destination body. Message traffic is reduced and bodies are notified much earlier. Second, continuation allocation by tree nodes is avoided. Finally and most significantly, the inner tree nodes need no longer be blocked, thereby allowing multiple additions to proceed concurrently (Figure 3.7(b)).

3.4.3 Local Synchronization Constraints

The sender of a message and its recipient operate asynchronously in actor communication. Thus, the sender may not know if the recipient will be in a consistent state in which it can logically respond to an incoming message. This problem is addressed in some process-oriented languages with input guards on synchronous communication [24, 92, 136, 57]: the recipient refuses to accept a message from a given sender (or a specific channel) until it is in a state in which it can process that message. Thus, the sender must busy-wait until the recipient is ready to accept the message. The net result is potentially inefficient execution – both because communication traffic is increased and because computation and communication do not overlap. In THAL, a programmer may

```

method add_body (... , body,...)
  compute quadrant
  if (type[quadrant] == nil) then
    type[quadrant] = BODY;
    child[quadrant] = body;
  elseif (type[quadrant] == BODY) then
    temp = child[quadrant];
    child[quadrant] = Node.new (...);
    type[quadrant] = NODE;
    child[quadrant].add_body (... , temp,...);
    child[quadrant] ! add_body (... , body,...);
  else
    child[quadrant] ! add_body (... , body,...);
  end
end
end

```

Figure 3.6: An example using delegation.

specify local synchronization constraints; the processing of incoming messages not satisfying these constraints is delayed until such time when the state of the actor changes to allow their satisfaction.

Proper synchronization is essential for efficient as well as correct execution of a concurrent program. Consider an actor implementation of the Cholesky Decomposition algorithm for dense matrices. For a given symmetric positive definite matrix A of size $n \times n$ the algorithm computes a lower triangular matrix L , of size $n \times n$ such that $A = LL^T$ [44]. In the implementation, each row of a matrix is abstracted as an actor and the matrix itself is represented as a group of actors. Factorization proceeds as row actors send messages to actors representing lower rows. Because actors operate and communicate asynchronously, messages corresponding to different iterations may be in transit at the same time. As a consequence, an actor i on processor P_i may send an actor r a message $m_{k,i}$ for iteration k after an actor j on processor P_j has sent r a message $m_{k+1,j}$ for iteration $k + 1$. Even if $m_{k,i}$ is sent before $m_{k+1,j}$, $m_{k,i}$ may take a longer path and arrive at r later than $m_{k+1,j}$. Thus, it is necessary to specify the synchronization on message reception to process messages in the correct order.

In general, using global synchronization results in suboptimal performance. Table 3.1 supports the argument by comparing performance results from a set of C implementations of the Cholesky Decomposition algorithm on the CM-5: implementations using local synchronization exhibit better performance than those using global synchronization. The results show that proper synchronization is essential for efficient as well as correct execution of concurrent programs.

In THAL, synchronization necessary for correct execution is specified using *local synchronization constraints*. Synchronization constraints are a language construct to specify a subset of an actor's states under which the specified method of the actor may be invoked [127, 70, 40]. Unlike input guards in conventional process oriented languages [58], they do not cause a sender to wait until such time when the recipient is in a state in which it can process the message. Thus, synchronization constraints ensure maximal overlap of computation and communication. Synchronization constraints are *local* if they are specified on a per actor basis. By postponing the processing of

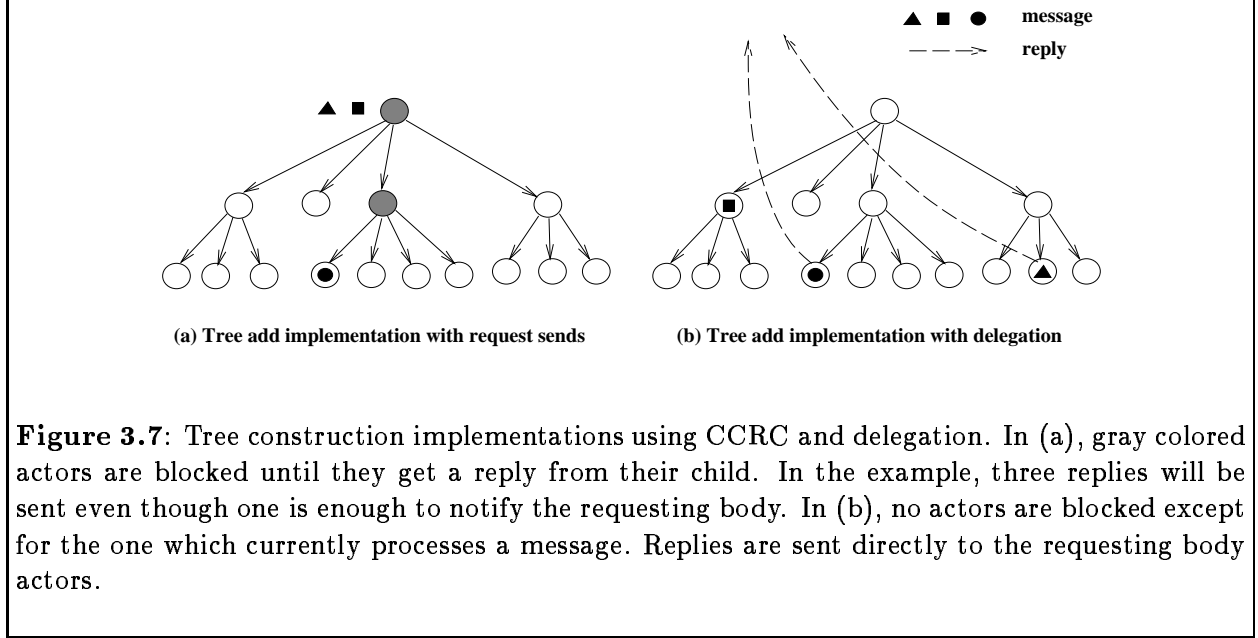


Figure 3.7: Tree construction implementations using CCRC and delegation. In (a), gray colored actors are blocked until they get a reply from their child. In the example, three replies will be sent even though one is enough to notify the requesting body. In (b), no actors are blocked except for the one which currently processes a message. Replies are sent directly to the requesting body actors.

certain messages, local synchronization constraints enforce the correct order on message processing and guarantee data consistency of the receiver.

A local synchronization constraint is specified using a `restrain` expression:

```
restrain msg-expr with ( bool-expr );
```

where *msg-expr* denotes a message pattern consisting of a method name and formal arguments and *bool-expr* is a boolean expression over acquaintance variables and the method arguments. Processing a message which matches *msg-expr* is delayed if *bool-expr* evaluates to *true*. Such synchronization constraints are called *disabling* constraints. These constraints may be specified separated from their corresponding method definition [91]. Such separation facilitates code reuse [40]. Making synchronization constraints a disable condition and having them separated from corresponding method definitions are to avoid interference with inheritance [40] and a legacy from HAL.

Figure 3.8 shows an example illustrating the use of local synchronization constraints. The example is a THAL implementation of a systolic matrix multiplication algorithm known as Cannon's algorithm [82]. Systolic algorithms employ synchronized data movement in lock step. However, our implementation does not use any global synchronization (e.g., *barrier*); instead, the correct order of execution is enforced by using local synchronization constraints only.

3.4.4 Group Communication Abstractions

THAL provides two mechanisms for group communication: *broadcast* and *point-to-point asynchronous message passing*. Sending a message to a group using a group identifier or *mygrp* denotes broadcasting. Semantically, the message is replicated and a copy is delivered to each member. Point-to-point communication among member actors is expressed by naming individual member actors. Some systems, such as [29, 25], provides one to one-out-of-many type of communication mechanism [26] which sends a message to an indeterminate representative member. We found that


```

behv SystolicMatrixMultiplication
| result, subright, subbelow, next_iter, left, up |
restrain fromright (sm,i) with (i ~= next_iter);
restrain frombelow (sm,i) with (i ~= next_iter);
init ()
    next_iter = 1;
    ...
end
method fromright (submatrix, iter)
    ...
    next_iter = next_iter + 1;
    left <- fromright (submatrix, next_iter);
    up <- frombelow (subbelow, next_iter);
end
method frombelow (submatrix, iter)
    ...
    next_iter = next_iter + 1;
    left <- fromright (subright, next_iter);
    up <- frombelow (submatrix, next_iter);
end
    ...
end

```

Figure 3.8: An implementation of a systolic matrix multiplication using local synchronization constraints. Each message has its intended iteration number which is compared against the receiver's next iteration number to maintain the correct processing order.

| | 256 × 256 | | | | 512 × 512 | | | |
|----------|------------|-----------|-----------|--------------|------------|-----------|-----------|--------------|
| <i>P</i> | <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 | 2.559 | 2.567 | 2.572 | 2.572 | 21.597 | 21.603 | 21.613 | 21.512 |
| 4 | 1.429 | 1.430 | 0.726 | 1.662 | 12.192 | 12.223 | 5.583 | 13.383 |
| 16 | 0.471 | 0.422 | 0.271 | 0.696 | 4.016 | 3.833 | 1.755 | 4.985 |
| 64 | 0.294 | 0.211 | 0.160 | 0.448 | 1.602 | 1.225 | 0.895 | 2.217 |
| 256 | 0.282 | 0.135 | 0.137 | 0.386 | 1.230 | 0.767 | 0.558 | 1.569 |

Table 3.1: Timing results of a set of C implementations of the Cholesky decomposition algorithm on a CM-5. The unit is msec. *P* is the number of processing elements. Columns *Seq* and *Bcast* represent the implementations which employed global synchronization, thereby completing the execution of one iteration before starting the execution of the next. The column *Seq* used barrier synchronization provided in the CM-5 Active Message layer while the column *Bcast* used global synchronization implicit in a CMMD broadcast primitive which was implemented on the CM-5 broadcast network. Columns *BP* and *CP* have execution times from the implementations which overlap the execution of different iterations by using local synchronization. Implementations of *BP* and *CP* are identical except that the former uses *block* mapping and the latter uses *cyclic* mapping. Communication between row actors was implemented using a minimum spanning tree-like broadcast mechanism built on top of the CM-5 Active Message (CMAM) layer in *Seq*, *BP*, and *CP*. A vendor-provided CMMD broadcast primitive was used for *Bcast*.

such a communication pattern does not occur in actor computations frequently enough to justify the cost of its implementation, and thus, did not support it.

Figure 3.9 illustrates the use of the group abstractions. The methods `iterate` and `eliminate` implement the *i*-th iteration of the Gaussian elimination algorithm to solve a linear system $Ax = b$. The *i*-th iteration gets started by normalizing the *i*-th row of *A* and the *i*-th element of *b*. Then, it eliminates the *i*-th column below the *i*-th row by broadcasting `eliminate` message.

Some applications, especially a number of matrix applications, have a successive computation structure; they step through a sequence of computation stages over their data elements. When the data elements in these applications are abstracted in terms of a group of actors, some member actors become idle in a predictable manner while others actively take part in the computation. For example, in the implementation of the Gaussian elimination algorithm, the *i*-th row does not participate in the computation until the back substitution begins after the *i*-th normalization. Therefore, copies of `eliminate` messages which are broadcast by actors representing *i*+1-th through *N*-th rows are simply discarded if they are received by one of actors representing the first through the *i*-th rows.

The unnecessary delivery of a broadcast message may be avoided by controlling the scope of the message. Initially, the scope of a broadcast message is the entire group. A member places itself out of the scope by executing the `resign` operator. `restore` is a global operation and resets the scope. By controlling the scope the local scheduling cost on processing node *p* is reduced from $\kappa \times N_p$ to $\kappa \times AN_p$ where κ , N_p , and AN_p are the scheduling cost per broadcast message, the number of member actors in *p*, and the number of active member actors in *p*, respectively.

```

behv Row
| rowidx, rowA, eltB, nexttiter |
restrain eliminate (iter,inRow,inB) with (iter==nexttiter) ;
...
method iteration ()
| i |
for i = rowidx+1 to N do
  rowA[i] = rowA[i] / rowA[rowidx] ;
end
eltB = eltB / rowA[rowidx] ;
rowA[rowidx] = 1;
if (rowidx == N) then
  mygrp.restore () ;
  mygrp <- backsubst (rowidx, eltB) ;
else
  resign ();
  mygrp <- eliminate (rowidx, rowA, eltB) ;
end
end
method eliminate (iter, inRow, inB)
...
if (iter == mygrpidx) then
  self <- iteration ();
end
...
end

```

Figure 3.9: An actor implementation of the Gaussian elimination algorithm using the group abstractions.

3.5 Dynamic Actor Placement

Performance of programs on parallel computers depends largely on the time spent in communication. In general, the cost associated with communication is a function of the proximity between a sender and a receiver. The cost of a remote message send is much higher than that of a local one. Although communication latency in current generation parallel computers is roughly independent of the distance between a sender and a receiver, sustained bandwidth between two nodes under busy traffic may vary considerably as a function of the physical distance between the two nodes. As a consequence, one may favor actor placements resulting in less remote communication. On the other hand, over-emphasizing locality may hurt scalability because scalable execution often requires actor placement which balances load across processing nodes. Optimal placement is one that harmonizes locality and load balance.

Such optimal placement is often both application and architecture specific. Specifically, it may depend on data structures that an application employs, on how input data are partitioned, and/or on network and processor characteristics in an architecture. Some applications may even have multiple phases each of which has a different communication pattern, and thus, has a different optimal data distribution. For those applications, migration of data and computation may result in a more efficient and scalable execution.

Programmers may specify application-specific distribution and placement strategy using annotations as well as migration. Placement is specified by annotating a `create` expression with an `on <location>` phrase where `<location>` is an expression that evaluates to a processor identifier. We represent a processor identifier by a positive integer. For multi-phase applications with different optimal placement for each phase, programmers may migrate actors before entering a new phase. Migration is triggered by sending an actor a `migrate` message with a piece of location information.

Figure 3.10 illustrates the use of placement. Consider the systolic matrix multiplication $C = A \times B$. Each matrix is divided into small equal-sized square blocks that are distributed over a square grid of processors. Matrices A , B , and C are implemented as an actor group with a different placement. The matrix C is simply overlapped onto the processor grid. The matrix A is row-skewed before it is overlapped; *i.e.*, sub-blocks in the i -th row are cyclic-shifted to the left $i-1$ times. The matrix B is similarly distributed in a column-skewed manner, *i.e.*, sub-blocks in the i -th column are cyclic-shifted upward $i-1$ times.

Note that placement functions are defined outside the behavior definitions of matrix actor A and B . In this way, the behaviors may be combined with different placement functions by simply changing the annotation. Also, the placement functions may be reused with other behavior definitions. However, the separate specification of placement function makes it difficult to specify member-specific placement. Use of a meta variable `@grpidx` alleviates the difficulty. The compiler extends the function's interface so that the function can take a member index as a formal parameter. The runtime system instantiates the variable with each group member index at the function's invocation time.

More often than not, a placement strategy leads to different performance results on different architectures. Thus, when an application is migrated from one platform to another, different placement strategies may be used to improve performance. Note that placement affects only performance; correctness of an implementation is orthogonal to placement. Thus, in order to reuse algorithm specification in porting an application, placement specification needs to be separated from algorithm specification. Different placement strategies may be specified in a meta language and kept in a library. A programmer may write her application in an architecture-independent

```

function f_row_skew (gridsize)
  | r_sub_1, c |
  r_sub_1 = (@grpidx-1) / gridsize;
  c = @grpidx & (gridsize-1);
  if (c == 0) then c = gridsize; end
  c = c - r_sub_1;
  if (c <= 0) then c = c + gridsize; end
  r_sub_1 = r_sub_1 * gridsize + c;
  return (r_sub_1);
end

function f_column_skew (gridsize)
  | r_sub_1, c |
  r_sub_1 = (@grpidx - 1) / gridsize;
  c = @grpidx & (gridsize - 1);
  if (c == 0) then c = gridsize; end
  r_sub_1 = r_sub_1 - (c - 1);
  if (r_sub_1 < 0) then
    r_sub_1 = r_sub_1 + gridsize;
  end
  r_sub_1 = r_sub_1 * gridsize + c;
  return (r_sub_1);
end

main
  | c, sqrt_partition_size |
  sqrt_partition_size = sqrt (#no_nodes);
  c = SystolicMatrixMultiplication.clone ();
  MatrixLeft.clone (c)
    on f_row_skew (sqrt_partition_size);
  MatrixRight.clone (c)
    on f_column_skew (sqrt_partition_size);
end

```

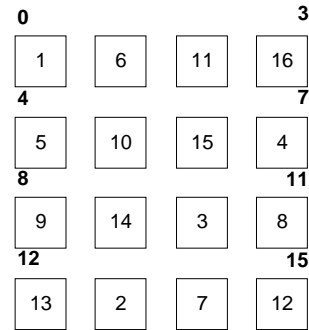


Figure 3.10: Placement of sub-matrices in a systolic matrix multiplication. Sub-matrices abstracted as group members are placed on 4×4 processor grid. Bold face numbers denote processor numbers and numbers in squares represent member actors which are placed in column-skewed manner.

way, and then, choose the best placement strategy for the application on a given architecture from the library and combine the two to obtain an efficient implementation. Such a modular specification methodology for actor placement has been developed in [102] on top of the THAL placement constructs.

3.6 Related work

THAL owes much of its linguistic features to HAL which was, in turn, influenced by other actor languages, such as Acore [89], ABCL/1 [135], and Rosette [126]. The language featured inheritance, limited forms of reflection, and synchronization constraints; the first two features are not supported in THAL for the sake of performance. HAL had been used as a test-bed for experimenting with new language constructs and dependability methods.

A general framework for modeling groups may be found in the ActorSpace paradigm [25] which provides group abstractions as a key component of its semantics. ActorSpace adopts a *pattern-based* model of communication. Messages may be sent to groups of actors which are identified with patterns. Groups in THAL are a passive container as in the ActorSpace model, but have only flat structures. Concurrent Aggregates (CA) [29] also supports a notion of groups but with a one-to-one-out-of-many type of communication where a message sent to a group (*i.e.*, an *aggregate*) is processed by a unspecified member. pC++ [87, 20] has a notion of groups similar to that of CA but its use is limited to data parallel computation.

The communication abstractions supported in THAL are found in other actor languages with different names. For example, CCRC appears in Acore as `ack`, in ABCL/1 as `now`, and in CA as `blocking send`. Both ABCL/1 and CA support delegation, too. However, broadcasting in THAL seems to be a unique feature. In particular, ABCL/1 has no notion of groups. CA does not have any particular support for broadcasting; they can be explicitly expressed by repeatedly sending a message to each of the group members. Synchronization constraints in THAL is also unique in that they are expressed as a disabling condition. Synchronization constraints in THAL are modeled after the work in [40]. Synchronization constraints in Rosette [127] are specified by using enabled sets. The ABCL family also has some provision for specifying synchronization constraints [90].

THAL is one of a few languages which make object locality visible to programmers. The placement primitives have been used as a basis for modular specification of partitioning and distribution strategy in [102]. The ABCL family provides a way to specify actor placement similar to ours, but does not support object migration. CA also supports user-specified object placement to some extent but the runtime system largely takes responsibility to control over object placement. Charm++ [73] provides programmers with only partial control over object placement. For example, migration of parallel objects is not allowed. Emerald [64] allows object migration as well as code migration. In particular, Emerald objects have fine-grained mobility. Programmers may use a range of primitives to control object mobility, such as object location, object movement, and object `fix/unfix`. Currently, THAL supports simpler object mobility; actors may migrate but behavior definitions must be available in the destination node. Behavior definitions of an application are broadcast to all processing elements upon its execution.

Chapter 4

Runtime Support

In general efficient execution of actor programs requires efficient runtime support. The runtime support implements machinery for actor execution as well as the actor primitives (Section 1.1). It also collaborates with a compiler to provide high-level abstractions. This chapter describes the design and implementation of such runtime support for THAL. Specifically, we describe the design philosophy of the runtime support and its organization. The description is followed by detailed discussion on the implementation of its components. We also present how the runtime support realizes high-level communication abstractions as well as how it supports migration.

4.1 The Execution Model

Actor computation unfolds as actors communicate each other. The communication is asynchronous, which requires a message be buffered at the receiver because it may be busy processing another message. The messages in the queue are processed one after another. The scheduling does not assume any particular order to pick up the next message to process though the first-come-first-served order would be the most natural as well as the easiest-to-implement choice.

Processing a message involves method invocation. The invocation creates a light-weight thread which carries out the method execution. Threads are guaranteed to execute to the completion without blocking. Non-blocking execution is particularly important in stock-hardware multicomputers where context switching is an expensive operation. The atomic execution gives a thread exclusive control over the actor state. The exclusiveness, however, does not preclude concurrent execution of multiple threads on an actor as long as they do not interfere with each other, *i.e.*, all the changes made by a thread to an actor can be seen to others as if they were done instantly.

Ideally, an actor is allocated its own processing element. In reality, processing elements are a scarce resource even on a massively parallel multicomputer, considering the fine-granularity and the resulting multiplicity of actors during typical executions. Thus, it is necessary for the runtime support to provide a sharing mechanism. It needs to be *fair* to keep an actor from monopolizing the processor, although the semantics does not specially prescribe how to implement the fairness. The decision is left to individual implementations. Now, upon completion of a thread execution control is transferred to the scheduler, which yields the control to the next actor.

4.2 The Design of the Runtime Support

4.2.1 Goals

The runtime support has been designed with three goals in mind: performance, modularity, and portability. First, the runtime support closely interacts with a compiler to achieve performance goal. It relies on information inferred by the compiler to avoid redundant computation. Also, it makes fine-grained access to its internals available to the compiler so that the compiler may utilize them to generate efficient code. Secondly, related functions are placed together in modules in ways that minimize inter-dependence between the modules. The modular design allows easy maintenance and facilitates component-based enhancement where implementation changes to a function are contained in the enclosing module without affecting others. Furthermore, the design naturally puts machine-dependent functions together in a module and thus makes porting to other platforms relatively straight forward.

4.2.2 The Architecture

The runtime system may be viewed as a network of virtual machines completely connected by some communication medium (Figure 4.1, which are mapped to physical processing elements, *i.e.* processor-memory pairs. The virtual machine approach makes opaque architectural difference existing in different platforms, thereby providing application portability. It also simplifies placement specification of actors.

Single Address Space Design

The runtime support has been designed to concurrently execute multiple programs from different users. What makes it unique, however, is it makes them share the same address space [81]. Supporting a multi-user environment with a single instance of a runtime support precludes the library approach, where a compiled code is linked to a runtime library before execution. In the library approach, an executable is self-contained and does not share any address space with others. Thus, the same runtime support is replicated in each executable and is loaded on each execution, wasting resources. Instead, we use dynamic loading. The runtime support runs on the background. A compiler is assumed to generate executables with load information. Upon execution, an application is dynamically loaded and integrated into the runtime support. (From now on, we use the term “runtime system” and runtime support, interchangeably.)

The design offers several advantages. The most conspicuous one is that it minimizes the system’s idle cycles and maximizes system throughput. The design reduces cycles spent in context switching as each thread uses a smaller amount of execution context. The cycles which may be wasted otherwise are productively used to process messages from different programs. Network throughput increases because, unlike *gang scheduling* [123], context switch occurs between lightweight threads, and thus, there is no possibility of packet loss during context switch and no need to flush the network. Finally, the design eases the operating system’s burden for fair scheduling in an environment where processors may have the dual responsibility of executing sequential and parallel applications, as in networks of workstations.

There are, however, two difficulties associated the design. One is possible running out of available address space. It seems unlikely to happen, though, because most current-generation micro-

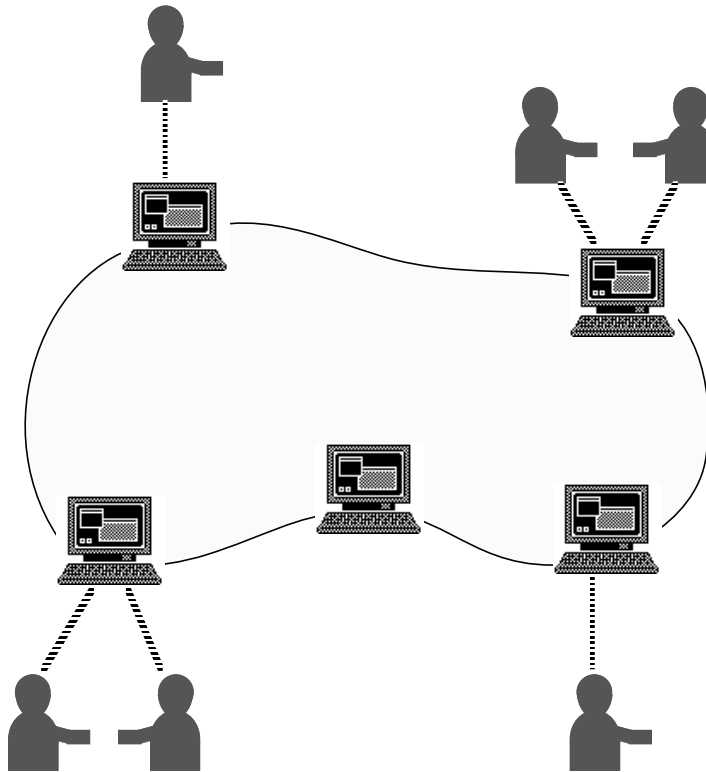
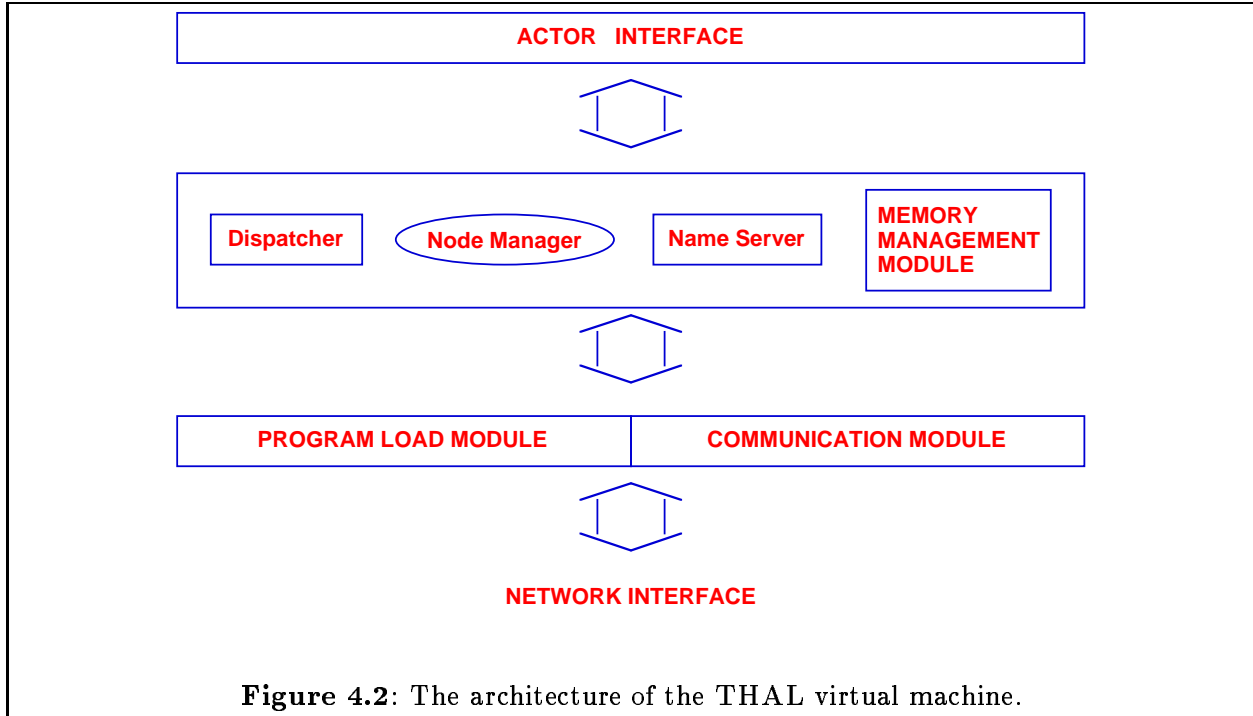


Figure 4.1: An abstract view to the THAL runtime system. Each terminal icon stands for a virtual machine. Mapping from virtual machines (VMs) to processing elements (PEs) may differ from one implementation to another, although running a VM on a PE is most common. The PEs may not be completely connected physically but users see the system as if they are.



processors adopted a 64-bit architecture which provides up to $2^{64} \approx 1.8 \times 10^{19}$ bytes of virtual address space.¹ The other has to do with security. Exceptions raised by individual actors, whether they are benign or malicious, must be caught and handled gracefully; the runtime system must be tolerant to them and should not fail. THAL employs strong type checking at compile time and most type related errors are caught before execution. Furthermore, memory access is strictly compiler-controlled. Thus, there is only one possible security breach left: assignment to array elements. A compiler may generate a bound check for an array reference or the runtime system may implement a fault isolation technique similar to [132]. Either technique is yet to be implemented.

4.2.3 Architecture of the Virtual Machine

A virtual machine (VM) implements an execution environment for actors and provides supporting machinery. It mostly serves as a passive substrate for active actors: they don't actively take part in actor computation. Only functions that a VM performs actively are those related to remote requests, such as delivering messages sent by remote actors and creating actors upon remote requests. Others are executed as a part of caller's execution. A VM exports a well-defined interface and restricts access to its internals only through the interface. The encapsulation makes the VM generic in that any program written in a language with actor-like semantics may be compiled and run on the runtime system.

The components of the virtual machine and their interaction are shown in Figure 4.2. On the top level is an *actor interface* which is exported to the compiler. The *communication module* and the *program load module* together constitute the kernel's interface to the network and are the only architecture-dependent modules. In between the two layers are the *node manager*, the *dispatcher* and the *name server*.

¹To our best knowledge, no current-generation implementation supports a full 64-bit address space.

The *communication module* and the *program load module* may be built on top of any well-defined low-level messaging layer, such as TCP/IP, Active Messages [130, 106], and Fast Messages [76]. They make actors an illusion of the completely connected network. Moreover, the hierarchical organization offers the runtime system some degree of network independence, and thus, portability.

The *node manager* delivers messages from remote nodes, creates actors in response to remote requests, and dynamically loads and integrates user executables into the runtime system. Also, the node managers in the virtual machines communicate with each other to maintain the system's consistency and to dynamically load-balance the system. A request to a node manager is delivered in the form of a system-level message. When a request is posted, it steals cycles from the currently active actor, processes the request within the actor's execution environment, and then, resumes the actor's execution (minimal context switch). The node manager is the only active component in a virtual machine.

The *dispatcher* provides only data structures necessary for actor scheduling. Actual scheduling is delegated to individual actors. The delegation allows scheduling to be done on an individual actor's current execution context, making unnecessary context switching between the actor and the node manager. The *name server* contains data structures for translating a mail address to physical location information and implements location transparency. The following two sections describe in more detail the implementations of the two components.

4.3 Distributed Shared Message Queues

Actors are autonomous; each may have its own computation power and process messages with a scheduling policy. In actor implementations on stock-hardware multicomputers, however, processors are a scarce resource and actors on a processing element are required to share the processor. Thus, most actor systems adopt a hierarchical scheduling mechanism. In the scheduling mechanism, the scheduler in a processing element selects an actor and yields control over the processor to the actor. The actor processes one of its messages and returns the control. Although fair and simple to implement, scheduling actors to process their messages is redundant, considering that it is not actors but messages that abstract computation.

The redundancy may be removed by extending message structure with a reference to a receiver (Figure 4.3) and unifying the two separate schedulings into *message-only* scheduling. In the message-only scheduling, all individual mail queues (which either already exist or will exist) are combined into a single message queue and shared by all actors. The queue is distributed over the processing nodes. Messages delivered to an actor are enqueued in the sub-queue in the node (thus, the name *distributed shared message queue* (DSMQ)). Eliminating actor scheduling simplifies the scheduler implementation. It also simplifies actor creation. Table 4.1 compares two scheduling mechanisms, one scheduling both actors and messages and the other scheduling messages only.

Figure 4.3 shows the definition of the message structure. A message consists of two parts, message header and message arguments. `receiver` holds a reference to the receiver of a message. To reduce message frame allocation overhead, the runtime system maintains a pool of message frames. `actualsize` contains message frame size which is used to allocate/deallocate a message frame from the pool. `msgsize` represents effective message size *i.e.* the size of the message header plus the argument size of a message. The effective message size is used to send the message off the node. Since messages delivered to an actor are now scattered in the DSMQ, sending them along with the actor at its migration time would be overly expensive. Instead, messages are forwarded

```

typedef struct _actor_msg_header {
    struct _actor_message *next;
    Method                method;
    struct _actor *receiver;
    int                   msgsize;
    int                   actualsize;
    int                   *locale;
    int                   packer;
    int                   dummy;
} ActorMsgHeader;

typedef struct _actor_message {
    ActorMsgheader hdr;
    int             data [0];
} ActorMessage;

```

Figure 4.3: The actor message structure.

| | message-only | | hierarchical | |
|------------------|----------------------------|-----------|--------------|-----------|
| | μ sec | cycle | μ sec | cycle |
| local creation | 8.04 | 265 | 11.52 | 380 |
| remote creation | 5.83 (20.83 [†]) | 192 (687) | 5.83 (23.68) | 192 (781) |
| lsend & dispatch | 0.45/5.67 | 15/187 | 8.10 | 267 |
| rsend & dispatch | 9.91 | 327 | 15.26 | 504 |

Table 4.1: Performance comparison of the two different scheduling mechanisms. *lsend* and *rsend* stand for local send and remote send, respectively. Local send and dispatch time does not include the time for locality check. Times obtained for the THAL runtime system are measured on TMC CM-5 by repeatedly sending a message with no argument. Each CM-5 node hosts a 33 MHz Sparc processor. [†] The local execution of remote actor creation in THAL takes 5.83 μ sec while the actual latency is 20.83 μ sec.

when necessary. The forwarding is enabled by use of `locale` which points to the `actorDefPtr` field of the receiver's locality descriptor, which is nullified when the actor migrates. When a message is about to be processed the presence of the receiver is examined by dereferencing `locale`. If not present, the message is forwarded to the receiver. Marshaling and sending arrays in a message poses a problem when forwarding the message. The compiler rearranges message arguments, puts array arguments after all simple ones, and store encoded packing information in `packer` (Section 4.7)/

4.3.1 Scheduling with Deferred Message Stack

Recognizing cost difference between remote and local message sending offers a substantial amount of performance gain, especially in fine-grained COOP languages [119, 75]. To exploit the cost difference in local message scheduling, the runtime system divides the message queue in a node into two sub-queues, *local* and *remote*, and distinguishes scheduling local messages from scheduling

remote messages. To reduce message frame allocation time, local messages are allocated not from the message pool but from a separate LIFO buffer. Distinguishing local and remote messages and giving higher scheduling priority to local messages simplify queue management and reduce message scheduling cost.

To reduce overhead further, part of message scheduling is exposed to the compiler. For each send statement the compiler inserts a runtime locality check to see if the receiver actor is *local* i.e. it is on the same node and is ready to execute. Depending on the result, the compiler generates two versions of message sending code. One is specialized for local message sending and the other is a generic one that is used when the locality check fails. Furthermore, both message frame allocation and argument marshaling are done in user code to eliminate redundant memory copy of message arguments from user space to kernel space.

Methods execute to completion without blocking. Non-blocking execution precludes the use of immediate dispatch of a local message with function invocation [119, 75] because function invocation implicitly blocks the sender of the message. The message is scheduled in the local message queue and its dispatch is deferred until the sender finishes its execution. (The local message queue is named *deferred message stack (DMS)* because it holds deferred local messages and behaves like a stack.) Upon completion, the sender dispatches the next message with its receiver by tail-calling the specified method. Since control information need not be stored upon method dispatch, the amount of memory for control information is bounded.

Local message scheduling with DMS (SDMS) is a little more expensive compared to that with function invocation (SFI) because the former requires explicit message frame allocation and argument marshaling which are implicit in the latter. On the other hand, the SDMS allows more concurrent message sending in a method and thus is more scalable. Consider a method which sends two messages where the first is local and the second is remote. The SDMS defers the dispatch of the first message so that the second message is sent right after scheduling the first one. The two messages are dispatched concurrently; the second may even be dispatched before the first one. This is not the case when using the SFI. The scheduling suspends the method execution at the time of sending the first message until the receiver finishes its computation and returns the control. The blocking semantics keeps the second message from being sent concurrently.

Another advantage of the SDMS has to do with load balancing. Consider the implementation of the Fibonacci number generator in Figure 4.4. The program places a subtree of the computation tree on a processing node. Since the computation tree is lop-sided, the execution of the program eventually develops severe load imbalance. Notice that the `compute` method of the behavior `Fib_Local` sends two local messages. The SDMS schedules one in the DMS while the other is being processed. The messages in the DMS are available for dynamic load balancing, resulting in more scalable execution. On the other hand, the SFI may not generate the second message in advance and thus not support dynamic load balance. The SDMS may be thought of as a compromise between locality and scalability.

The atomic method execution semantics, which allows no more than one active thread on an actor, interferes with the SFI and may cause a deadlock when two actors send messages in a mutually recursive manner. Other systems [119, 75] which employ the SFI avoid deadlock by providing a stack unwinding mechanism along with *futures*.

Figure 4.5 shows snapshots of a stack and a message queue when function invocation is used to schedule local messages. Suppose actors S_1 , R , and S_2 are on the same node. Before S_2 sends a message to S_1 , it examines if S_1 is indeed local (Figure 4.5.(3)). The check fails because S_1 has yet

```

behv Fib
  method compute (n,dist)
    if (n <= 1) then
      reply (1);
    else
      if (dist > 1) then
        reply ((Fib.new()).compute(n-1,dist/2) +
              (Fib.new() on (#myvpn+dist)).compute(n-2,dist/2));
      else
        reply ((Fib_Local.new()).compute(n-1) +
              (Fib_Local.new()).compute(n-2));
      end
    end
  end
end

behv Fib_Local
  method compute (n,dist)
    if (n <= 1) then
      reply (1);
    else
      reply ((Fib.new()).compute(n-1) + (Fib.new()).compute(n-2));
    end
  end
end

main
  print ((Fib.new()).compute(33,#no_nodes/2));
end

```

Figure 4.4: An implementation of the Fibonacci number generator.

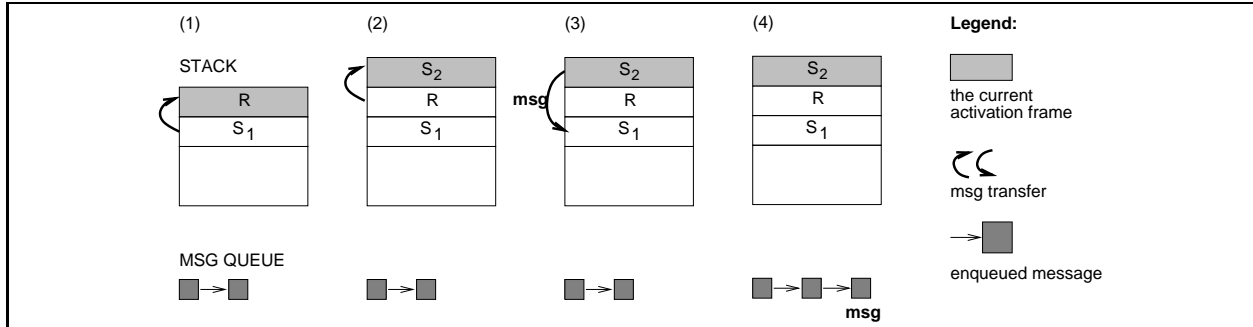
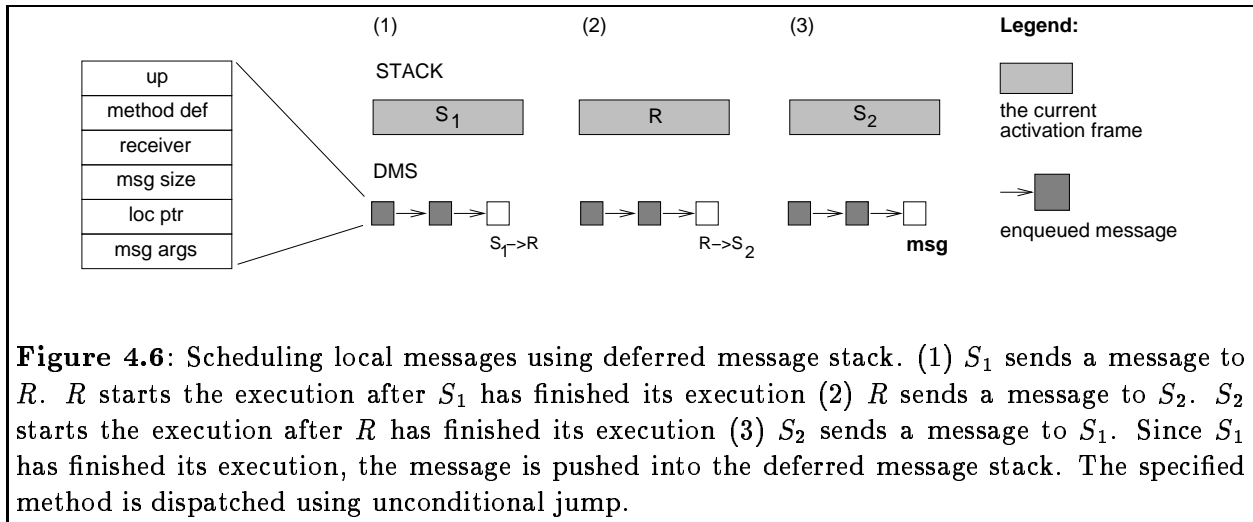


Figure 4.5: Scheduling local messages using function invocation. (1) S_1 sends a message to R . R starts execution immediately. (2) R sends a message to S_2 . S_2 starts execution immediately. (3) S_2 sends a message to S_1 . Since S_1 is running, the message is sent using the generic message send mechanism. (4) The message is enqueued in the local message queue.



to finish its method execution. S_2 is forced to send its message using the more expensive generic send. Note that function invocation may be used for the message from S_2 to S_1 if S_1 has finished its method execution before the message is sent.

Figure 4.6 has snapshots of a stack and a DMS for the same example but with the SDMS. It also shows the components of a message frame. The `up` field points to a message frame right above the message frame and `loc ptr` holds the memory address to the receiver actor's location information. Each message frame in the DMS encapsulates the complete information for the method execution. The encapsulation allows a scheduler to migrate messages for load-balanced scalable program execution. A scheduler may migrate the bottom-most message in the DMS and its receiver actor.

4.3.2 Static Method Dispatch using Deferred Message Stack

The meaning of a message depends on the receiver in (concurrent) object-oriented programming languages. Type dependent method dispatch necessitates the method table lookup at the receiver

when processing a message. The cost is significant especially in actor languages because their method execution is fine-grained. The situation gets worse if the language supports inheritance and the class hierarchy gets deeper.

The dynamic method lookup may be bypassed if, at compile time, the type of the receiver can be known to be unique throughout all possible executions. The type information may be obtained through a global data flow analysis. We extended the type inference presented in [100, 101] (see Section 5.1) to infer types for expressions in a THAL program. The kernels integrate application programs into their address space so that each program is located in the identical space in all the kernels. Thus, a function address has the same meaning wherever it is accessed. If the method selector in a message send expression has a unique meaning throughout the entire program execution, we replace the method selector with the corresponding function address.

4.3.3 Dynamic Load Balance

Even when programmers may specify their application-specific load balancing policy, load imbalance may develop as computation unfolds. Multiprogramming supported in the THAL runtime system should alleviate the impact of load imbalance on the system throughput. To further improve processor efficiency and scalability, the runtime system supports dynamic load balancing. As a virtual processor runs out of messages to process, it randomly selects a target virtual processor and steals a message from it (*random polling*). The target processor looks for a candidate message from its remote message queue and then its local message queue and sends it to the requesting processor. For a message to be migrated, its target actor should not be engaged in any computation at that moment. When the message is migrated, it takes the receiver along with it. Other messages to the actor are subsequently forwarded.

4.3.4 Fairness

One problem with the SDMS is that processing certain messages may be indefinitely postponed in a finite computation. Figure 4.7 has a non-strict program which terminates even though one of its subcomputations does not. Fair message scheduling eventually dispatches the `done` method, making the computation halt and the actor be reclaimed. The `halt` primitive cannibalizes the actor and reclaims its resources. However, the SDMS indefinitely postpones the dispatch of the `done` method.

We use a *counter* to guarantee the eventual message delivery in the Actor model. Recall that any method can only execute finitely in THAL. At first, the counter is set to 0. It is incremented upon each method dispatch. If the value of the counter exceeds a predefined value, the actor yields control and schedules the bottom-most message of the DMS. The counter is then reset to 0. The counter is also reset when the DMS becomes empty. It never decrements, though. Non-strict computation in a CPS transformed method can be similarly handled with an additional overhead for restoring concurrency. We have assumed that the non-strict computation occurs relatively infrequently.

4.4 Distributed Name Server

Sending a message requires the receiver's current locality be known to the sender. The actor's whereabouts are abstractly represented by its mail address. It is this abstraction that provides


```

behv InfiniteFinite
  method send_twice ()
    self <- send_twice ();
    self <- send_twice ();
  end
  method done ()
    halt;
  end
end

main
  | inFinite |
  inFinite = InfiniteFinite.new ();
  inFinite <- send_twice ();
  inFinite <- done ();
  inFinite <- send_twice ();
end

```

Figure 4.7: Indefinite postpone. With a fair scheduler, the program terminates even though one of its subcomputations does not terminate. A fair scheduler eventually dispatches `done` method, making the computation halt and reclaiming the actor. `halt` cannibalizes the actor and reclaims its resources. However, the static method dispatch may indefinitely postpone the dispatch of `done` method.

actors with *location transparency*. However, a receiver's abstract location must be translated to a physical location before the first byte of a message is injected into the network. The name server manages data structures and exports access routines for name translation. We describe in this section how a mail address is defined to facilitate name translation while guaranteeing location transparency. The implementation of the distributed name server in the runtime system is also discussed in the section.

4.4.1 Mail Address

An actor is uniquely identified with a mail address which represents its locality in the computation space. The entities used to define a mail address determine the efficiency of name translation as well as the degree of location transparency. Often, the two are found to be conflicting requirements. On one hand, the use of location-dependent entities tightly coupled with actors offers efficient name translation at the expense of location transparency. On the other hand, location-independent entities allow location transparency but increase name translation time.

To meet both requirements well, we define mail addresses using location-dependent entities which are loosely-coupled with actors. A mail address is composed of two parts. One part is the address of a creator node – its *actor's birthplace*. The location dependency enables efficient name translation. The other is the memory address of a *locality descriptor* which contains the actor's location information. Decoupling of the mail address of an actor from its physical location makes it location independent and relocatable.

4.4.2 Locality Descriptor

The name server keeps actors' locality information in locality descriptors. The implementation of a locality descriptor is shown in Figure 4.8. If an actor is local, the locality descriptor contains the memory address of the actor in the `actorDefPtr` field. Otherwise, the locality descriptor contains the remote node address (`location`) and the memory address of the actor's locality descriptor on the remote node (`ALD_cache`). A locality descriptor is allocated when an actor is created. It is also

```

typedef struct _actor_locality_descriptor {
    struct _actor_locality_descriptor *fwd;
    int status;
    ActorAddress actorAddr; /* 2 words */
    int location;
    Actor *actorDefPtr;
    struct _actor_locality_descriptor *ALD_cache;
    ActorMessage *hold_list;
    struct _back_prop_task *reverse_forward_chain;
    int dummy; /* for double word alignment */
} ActorLocalityDescriptor;

```

Figure 4.8: The implementation of the locality descriptor.

allocated at message sending time if the receiver's locality descriptor is not found at the sending node.

Using locality descriptors, a generic message send mechanism may be implemented as follows. During a message send, the sender composes a message and consults the name server. If the location information is locally available, the message is sent using the information. Otherwise, a locality descriptor is allocated and the message is sent to the node where the receiver was created. Note that the location information is encoded in the mail address. No inter-processor communication is required to get the receiver's location information. (For the moment we assume that actors never migrate. A more general solution with actor migration is given in Section 4.4.4.) 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 while the message is delivered. Subsequent messages to the same receiver are sent with the cached address, obviating name table lookup at the destination node.

4.4.3 Distributed Name Table

The name server is consulted every time a message is sent, either locally or remotely. Thus, the name server should be distributed to not be a bottleneck. This is done by having local copies of the name server manage their own local name table independently of each other. Each local name table is implemented as a hash table whose entries are pointers to locality descriptors. To avoid inter-processor communication in the name translation process, each local name table may have its own copies of actors' locality descriptors. As a result, name translation from mail address to location information is performed by consulting only the local name table.

Allowing multiple locality descriptors for an actor implies local name servers collaborate in order to maintain the consistency of the name tables. Note that inconsistency arises only when actors migrate. The most straight forward mechanism to maintain consistency is broadcasting: a virtual processor involved in the migration broadcasts the actor's new location information. Since broadcasting requires participation of all virtual processors, it typically wastes compute cycles because only a few virtual processors will have a locality descriptor for the migrating actor. To reduce migration cost and give better resource utilization, we relax the consistency requirement

and update only two name tables: one at the source node and one at the birth place node if the two nodes are different.

Figure 4.9 illustrates how incorrect location information arises in name tables. In the example, the actor was originally created in node 3 and then migrated to node 1, to node 2, and to node 4. Since only the name tables at the source node and at the birthplace node are updated, when the actor moved to node 4 the location information in the name table in node 2 becomes out of date and inconsistent.

Allowing inconsistency in the name tables makes location information for remote actors only “best guess.” A name server cannot tell if information in its local name table is up to date; at best, it believes that an actor is still there when it sends it a message. If migration occurs infrequently the guess may be correct most of the time. Indeed, it is our underlying assumption for the implementation that user specified migration is a relatively infrequent, bursty event. Thus, instead of maintaining stringent consistency requirement, we allow inconsistency in the name table and provide a mechanism to correct the inconsistency.

4.4.4 Message Delivery Algorithm

The message send and delivery algorithm is summarized in Figure 4.10. The algorithm is straightforward except for the inconsistency correction mechanism. We describe the algorithm in detail.

Actors may migrate, and an actor’s migration history is kept in its locality descriptors. Since location information for a remote actor 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 a message but finds the receiver has already moved, it forwards the message using the history information kept in its local name table.

Consider the two examples in Figure 4.11. The top one illustrates a situation where a receiver’s location information is not found in the local name table upon message send. Even though the receiver has a migration history of node 5 \rightarrow node 1 \rightarrow node 2 \rightarrow node 3, the sender has no means of knowing it. Since no location information is available at all, the message is sent to the receiver’s birth place, from which it is forwarded to node 3. Recall that an actor’s birth-place node always has up-to-date location information.

The one at the bottom depicts a more complicated situation. The actor *A* was first created on node 5. After it moved from node 5 to node 1 and to node 2, another actor in node 4 sent *A* a message. Thus, node 4 thinks *A* is on node 2. Then, *A* moved again to node 3. Although the name tables on nodes 2 and 5 (*A*’s birth place) are updated, those on nodes 1 and 4 are not. Then an actor on node 4 (it may be the same actor which sent the previous message) sends a message to *A*. Since the name server guesses that *A* is on node 2, the message is sent to node 2 and forwarded to node 3 from the node.

The forwarding process consists of three phases. In the first phase, the requesting node manager sends a special forwarding information request (FIR) message to locate the actor. The FIR message is relayed until it reaches the receiver. Then, the receiver’s locality information, *i.e.*, the node number and the memory address of the receiver’s locality descriptor, is back-propagated along the forward chain (`reverse_forward_chain` in Figure 4.8). Meanwhile, all node managers in the forward chain update their name tables as the message passes by. The shaded arrows in Figure 4.11 denotes the information flow. Once the receiver’s location is known, the original message is sent directly to the node.

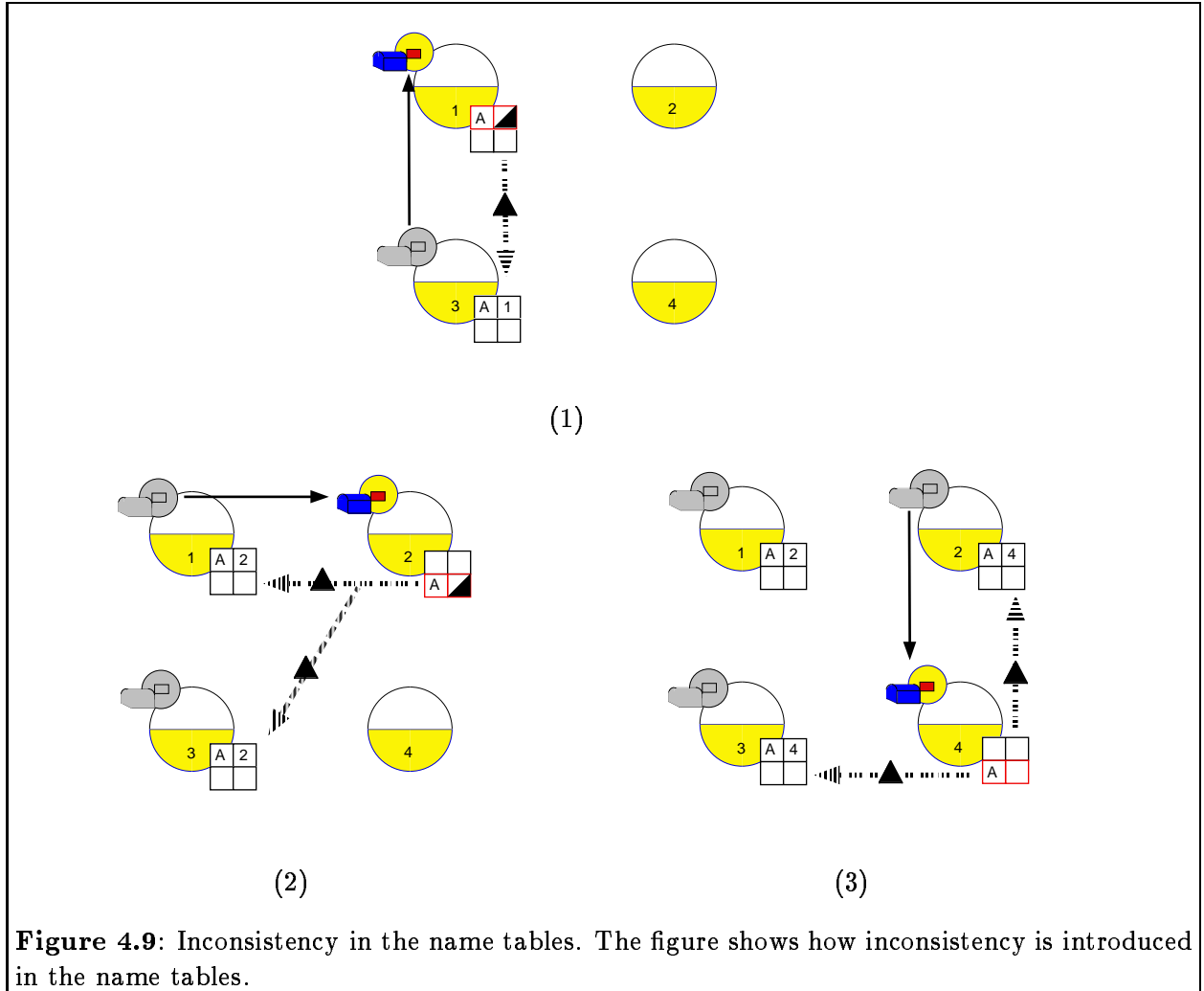


Figure 4.9: Inconsistency in the name tables. The figure shows how inconsistency is introduced in the name tables.

```

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 not 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 not 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 4.10: The message send and delivery algorithm.

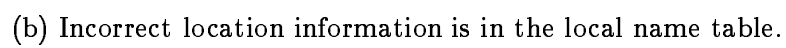
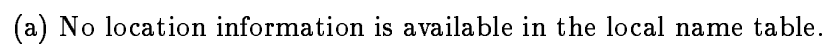


Figure 4.11: Inconsistency correction and message forwarding in migration.

When a node manager is requested to deliver a message to an actor, it may have already sent an FIR message. Since it is unnecessary for it to send an FIR message again, it simply enqueues the message in the hold list (`hold_list` in Figure 4.8) and waits until the receiver's location is known. When the information is available, it updates the name table, forwards the messages in the hold list to the node, and relays the information back to the virtual processors which wait for the information.

4.5 Remote Actor Creation

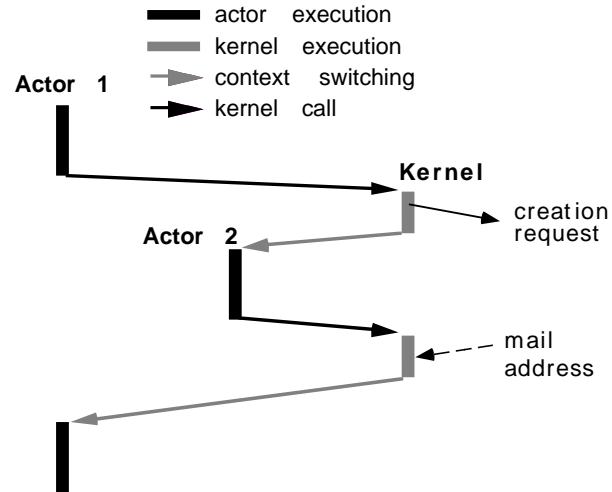
A mail address is allocated and assigned to an actor at its creation time. The use of an entity which is coupled with an actor in defining its mail address requires it be allocated at the node where the actor is placed. This implies that an actor requesting remote creation must wait until the mail address is returned. In consequence the remote creation time varies unpredictably as traffic in the network or load at the remote node changes.

The unpredictable remote creation time makes *split-phase allocation* (Figure 4.12.a) desirable on platforms with hardware support for context switch; context switch to another from a thread requesting remote creation while the latter waits for the mail address to be delivered, effectively hides the latency and saves idle cycles [9, 34]. However, it is less desirable in stock-hardware multicomputers where context switch is very costly (e.g., 52 μ sec in the TMC CM-5).

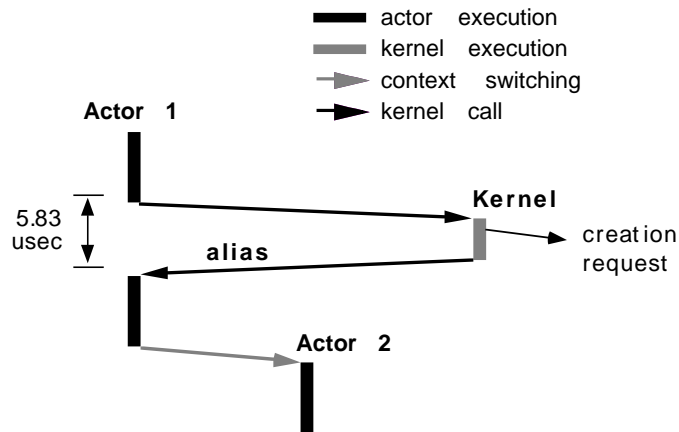
We use *aliases* instead of relying on context switch. An alias is a locally allocated clone of a mail address, which is equally capable of uniquely identifying an actor. An actor's alias may be used interchangeably with its mail address. The use of aliases is based on the observation that an actor requesting remote creation may continue with its remaining computation as soon as it gets a handle that uniquely identifies the newly created actor. Note that actors created as results of local creation requests need not have aliases.

An alias has the same structure (*i.e.*, `<birthplace,address>`) as a mail address. However, `birthplace` now represents not the node where the actor was created, but the node where the creation request was issued. The address of the node on which the actor is created is also encoded in `birthplace`. The encoded information may be used when an actor sends a message using an alias and the locality descriptor is not found in the name table. The sender sends the message to the receiver's birthplace node with the assumption that it has not migrated. Having the same structure allows us to implement message sending with aliases without incurring any additional cost.

When an actor issues a remote creation request it allocates an alias and sends it along with the request. As soon as the last packet of the request is injected into the network the alias is returned to the sender which then continues its execution with it. On the receiving end, the node manager creates an actor with an ordinary mail address and registers the actor in its local name server with the received alias (Figure 4.12.b). Meanwhile, the locality descriptor's memory address is sent back to the requesting node and cached there (on the background). Our measurements show that local execution of remote creation with no initialization argument completes within 5.83 μ s whereas the actual creation (*i.e.*, time taken from request sending to completion of the creation at the receiving node) takes 20.83 μ s.



(a) remote creation using split-phase allocation.



(b) remote creation using alias.

Figure 4.12: Remote actor creation.

4.6 Implementation of Communication Abstractions

THAL provides flexible high-level communication abstractions. These abstractions must be implemented efficiently by the compiler and the runtime system. We describe how the runtime system interacts with the compiler to implement the abstractions. Runtime support for CCRC is discussed in Section 5.2 in conjunction with the compile-time transformation.

4.6.1 Local Synchronization Constraints

Synchronization constraints are bound at compile time to methods that they constrain. The runtime system allocates an auxiliary queue called *pending queue* to an actor upon its creation to enforce local synchronization on the actor. When a message is dispatched, the runtime system evaluates the synchronization constraints of the target method. If any of them evaluates to true, the method is disabled for the moment and the message is enqueued in the actor's pending queue. If none of them evaluates to true, the method is enabled and the dispatch is granted. Since a synchronization constraint is a function of actor state and message arguments, some messages in the pending queue may become enabled when an actor changes its state. Before dispatching the next message in the DSMQ, the actor re-evaluates synchronization constraints for each pending message and attempts to consume them.

When a message is dequeued for processing, the synchronization constraints of the specified method are evaluated. A positive evaluation result means the method is disabled under the actor's current state and the message is put into the actor's *pending queue* [79]. Recall that synchronization constraints are a function of actor state and message arguments. When an actor changes its state, some messages in its pending queue may become enabled for processing. Before yielding control over computation resources, an actor re-evaluates synchronization constraints for each pending message and digests the enabled ones.

Synchronization constraints have been used to maintain the correct iteration order in our experiments. Our experience is that pending queues are usually short. Few messages are delivered out of order and the cost associated with re-evaluating synchronization constraints of pending messages is kept tolerable. However, if a long pending queue may develop in an application, re-evaluating synchronization constraints for each pending message every time an actor changes its state would be expensive. In this case, dependencies between state variables and synchronization constraints should be analyzed at compile time so that only those pending messages for methods that might be enabled are re-examined if they may be processed. The current compiler does not implement such an analysis.

4.6.2 Groups and Group Communications

The group abstractions in THAL provide additional opportunities for optimization. The flat structure of a group and the homogeneity of member actors often allow efficient memory management. The broadcast communication may be implemented more efficiently than by using point-to-point message passing. The runtime system takes advantage of the opportunities and provides efficient support for the group abstractions.

First, the runtime system allocates member actors in a node in a contiguous block. The continuity of member actors in the address space enables the compiler to generate efficient collective

scheduling code because the next actor may be accessed via simple pointer arithmetic. Since member actors belong to the same group, they share common attributes, such as group size and creator. These attributes are collected in a group descriptor and a copy is kept in each node to allow efficient access. To further facilitate the access, each copy is allocated with the same local address in each memory. Among the shared attributes is a distribution map which records locations of members. This map contains a pointer to a member actor if it is local; else, the address of a processing element.

Conceptually, a message sent to a group is replicated and a copy is delivered to each member (*i.e.* broadcast). In consequence, the cost associated with broadcast is $O(N)$ where N is the group size. To reduce the cost the runtime system implements broadcast with two phases: *explicit* and *implicit*. In the explicit phase, each processing node is delivered one copy of the message. The optimal implementation should be architecture-dependent; it may be directly implemented using a broadcast facility available in the underlying architecture or it may be simulated using a point-to-point communication. We sketch our implementation of the explicit phase on the TMC CM-5 in Section 6.1. It has a binary tree-like communication structure and a cost of $O(\log P)$ where P is the number of processing nodes in the system. When a broadcast message arrives at a node, the node manager delivers it to the first member actor in the node. The compiler puts scheduling code at the end of the method, which schedules the next member actor with the message. Thus, only the first member in a node receives a broadcast message; the others simply reuse the message (*i.e.* the implicit phase). The compiler ensures that the content is intact.

Send-to-member shares the same syntax with ordinary message sending but has a member actor as a receiver. Since the exact location information for member actors is locally available in the distribution map, the locality check and the point-to-point message send for a member actor are implemented differently from those for ordinary actors to take advantage of the locally available information. A direct consequence of having different implementations is that their interfaces to the compiler are different. For the compiler to use type-dependent interfaces correctly, it must distinguish group communications from ordinary communications. It also should generate methods invoked on groups differently than those invoked on individual actors. The compiler analyzes the global data flow of a program and collects type information about methods and receivers. This type information is used to translate the program correctly.

4.7 Migration

The communication module supports location transparency. So, the remaining implementation question on migration is when and how to migrate actors. During its life, an actor may be in one of three states: *ready*, *running*, and *blocked*. To minimize the context that must be transmitted with an actor, the runtime system migrates actors only when they are in the ready state. When an actor receives a migration request, a flag is posted if it is either running or blocked. Actual migration is put off until the actor completes its processing of the current message.

Use of the DSMQ may scatter messages delivered to an actor through the message queue. This complicates migrating messages along with the actor. Instead, only the actor is moved. The messages are forwarded subsequently as the absence of the receiver is detected when they are dispatched.

Supporting migration would be easier if arrays were not available in the language. However, THAL allows arrays to be part of an actor's state, hence migratable. It even allows sending arrays

in a message. A naive solution would be to make programmers supply a marshaling function for each actor and message [73]. Unfortunately, the solution may fail to work with dynamic load balancing that programmers are not aware of. Our solution is to let the compiler keep sufficient information available to the runtime system so that it can easily deduce relevant information for state and message packing.

The compiler rearranges acquaintance variables/message arguments so that it places array arguments after all simple arguments. Then, it generates packing information composed of the number of arguments and the number of simple arguments and puts them in the `packer` field (see Figure 4.3). Each array object is associated with a size information which is accessed by the runtime system. For example, consider a message send statement

```
recv <- m0 (arr0, sim0, arr1, sim1, arr2, sim2);
```

Suppose `simi` are all simple integers and `arri` are all arrays of real numbers. The types are supposed to be inferred from the enclosing context. The compiler takes the arguments, shuffles them, and generates the following C structure.

```
struct msg0 {
    int    sim0;
    int    sim1;
    int    sim2;
    float *arr0;
    float *arr1;
    float *arr2;
} Msg0;
```

Of course, the formal parameters in the corresponding method interface are also reordered following the same rule. Then, the compiler generates packing information of (6,3) meaning there are a total of 6 arguments and the first 3 are simple. When the message must be forwarded, the runtime system sees the packing information in `packer` and packs the message accordingly. Note that strings are treated as one-dimensional arrays of characters.

4.8 Related Work

Among other systems the implementations of ABCL/onAP1000 [119, 120] and the Concert system [28, 75] have influenced the design of THAL most.

ABCL/onAP1000 adopted an encapsulation approach to implement its runtime system. For example, the runtime system determines whether to use the stack-based or the queue-based scheduling mechanism for local messages. In consequence, the message scheduling mechanisms are hidden from the compiler, hampering the generation of codes which efficiently utilize scheduling information available at execution time. By contrast, our runtime system exposes part of its scheduling mechanism for the compiler to generate executables that use the scheduling information to choose where to schedule local messages.

Objects in ABCL/onAP1000 are identified with a unique mail address, as in our system. Although actor placement is put under programmer control, the use of location-dependent addresses

to favor fast locality check undermines object mobility. We believe that language support for dynamic object relocation (*i.e.*, *migration*) is crucial in load balanced and scalable execution of many dynamic, irregular applications.

The runtime support of the Concert system provides the compiler with a flexible interface, as in our system. Both systems make cost of runtime operations explicit to a compiler, thereby enabling the compiler to perform a range of optimizations. The main difference is in the extent of location transparency they support. Aggregates in the Concert system are located at the same memory address on each node [74]. This location dependence limits aggregates' mobility, making it difficult to load-balance in dynamic, irregular computation. Concert objects other than aggregates are allocated in a global space and subject to global name translation. THAL's locality check uses only locally available information which is made possible by our name management scheme which works efficiently with migration.

Threaded Abstract Machine (TAM) supports multithreading at instruction level [34]. It 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. In TAM, a thread executes to completion once successfully initiated, like our method execution. Furthermore, quasi-dynamic scheduling allows the compiler to exploit temporal locality existing among logically related threads. Such temporal locality is utilized in our runtime system with coarser grain by collectively scheduling messages broadcast to a group of actors.

Cilk [19] is a C-based runtime system for multithreaded parallel programming. The Cilk language is defined by extending C with an abstraction of threads in the explicit continuation-passing style. A Cilk program is a collection of Cilk procedures, each of which is broken into a sequence of threads. As in TAM, each Cilk thread is a nonblocking C function and runs to completion without waiting or suspending once invoked. Although the decision to break procedures into separate nonblocking threads simplifies the runtime system, writing programs in the explicit continuation-passing style is onerous and error-prone. In our system, programmers are allowed to use blocking asynchronous communications and the compiler translates them away into explicit continuation-passing style code (Section 5.2).

Chapter 5

Compiler-time Analysis and Optimization

We have introduced sequentiality to enhance programmability. However, excessive sequentiality should be removed from a program for efficiency before it is executed on parallel computers. Our compiler optimizations focus on restoring useful concurrency. First, the compiler infers types for each expression. The type information is used to reduce message scheduling cost and automatic memory management. The compiler also transforms `request` expressions into asynchronous sends with appropriate synchronization. The constraint of serialization of message processing in the Actor model requires only one thread be active on an actor. This is enforced by locking/unlocking the actor. Note that multiple threads may be active on actors unless they interfere with each other. The compiler uses data flow analysis to unlock actors as early as possible so that multiple threads may be concurrently running on an actor.

The compiler translates method definitions in a user program to a set of C functions. Acquaintance variables of a behavior are collected in a C structure definition to provide the runtime system with size information for actor allocation. These are compiled using a C compiler, linked together into an executable, and loaded into the runtime system to execute on parallel computers.

5.1 Type Inference

Untyped languages allow rapid prototyping and parametric polymorphism [107]. They have better programmability and encourage code reuse. On the other hand, typed languages have the advantage of type safety. All possible misuses of variables are checked against their type declarations at compile time and/or execution time. The resulting static type checking removes many run-time type checks, guaranteeing efficient execution. Type inference is a compile-time analysis which provides untyped languages with the advantages of static typing. By having the THAL compiler infer types for expressions and verify their consistency, programmers may enjoy untyped languages' programmability and typed languages' reliability and efficiency at the same time [93, 109, 100].

The THAL compiler uses an extension of the constraint-based type inference developed by Palsberg and Schwartzbach [100]. In the algorithm, types are defined as a set of classes. A type variable is assigned to an expression and type relations between expressions are represented as a set inclusion between the corresponding type variables called *type constraint*. The algorithm derives a

set of type constraints by examining statements and expressions in each method. It derives another set of type constraints from actual/formal parameter relations in message send expressions. Then, the algorithm tries to iteratively solve the two sets of type constraints. Existence of the smallest solution means the program is type safe. If none exists, the program is rejected as type unsafe.

The implementation is similar to that of [99]; it incrementally builds a trace graph and successively refines the solution. A trace graph represents all possible message sends in any program execution. Incremental construction of a trace graph and successive refinement of the solution make complexity of the implementation polynomial. If a partial solution does not satisfy any single constraint, the program is rejected as type unsafe. If the current solution does not improve the previous solution, the solution is the smallest solution and the program is accepted as type safe. The reconstructed type information is used to bypass the method lookup process in method dispatch.

5.2 Transformation of Concurrent Call/Return Communication

Concurrent call/return communication (CCRC) provides programmers with an easy way to express *remote function invocation*. The convenience comes from its blocking semantics, which requires a form of context switch. However, a naive implementation of CCRC which context-switches a sender whenever it sends a request would make it less attractive on stock-hardware multicomputers because their context switch costs are high (e.g., 52 μ sec on CM-5).

A technique used in a number of systems [117, 75] to implement CCRC-like abstractions is one using *futures* [68]. Futures are a promise for a value; they are a place holder for a value which is yet to be defined. In the future-based implementations, sending a request message creates a future which is immediately returned as the result of the request. Execution continues until the result is actually used (*i.e.*, the future is *touched*). If the value is available, execution proceeds with the value. Otherwise, it is blocked until the value is available.

Although simple to implement, a future-based implementation still makes CCRC a potential performance bottleneck because it relies on context switching. Consider an N-Queen implementation in Figure 3.4. Suppose an actor creates N children in response to a `compute` message. An implementation using futures would allocate N futures which are assigned to `replies[i]` for i ranging from 1 to N . The next expression to be evaluated is a summation; here the execution blocks on a future whose value is yet to be defined. In the worst case, $2N$ context switchings are required to finish the computation (*i.e.* 2 for each touch). Furthermore, a separate mechanism is needed which resumes an actor blocked on a future when its value is available. Out-of-order arrival of replies does not help because futures are touched sequentially conforming to the order of appearance of the requests in a method.

In contrast to the future-based implementations, we employ a compiler-oriented approach [78, 5]. The compiler transforms a method containing CCRCs in such a way that a programmer would write a method with the same semantics if CCRC is not available. The compiler analyzes data dependence among requests in a method to identify those that may be executed concurrently. It isolates computation simultaneously dependent on the requests (*i.e.* a *join continuation* [1]) into a continuation actor and translates the requests to non-blocking asynchronous message sending expressions. A unique reply address represented as a triple is appended to each expression as an additional message argument. Use of non-blocking asynchronous message passing allows us to avoid expensive context switching. Instead, the continuation actor caches only the context needed to execute the join continuation.

5.2.1 Join Continuation Transformation: the Base Algorithm

The transformation exploits *functional parallelism* existing in evaluation of an expression; subexpressions of an expression may be evaluated in any order because actor state does not change in between their evaluations. In particular, there is no control dependence between argument expressions of a function invocation or a message send expression. Note that the control dependence does not exist in the Actor model; only data dependence does. It appears in THAL because we introduce sequentiality to method execution for execution efficiency and implementation convenience.

Let Δ_m denote the data dependence relation of a method m and Γ_m denote the control dependence relation of m . Each of them defines a partial order between evaluations of any two expression in m . Further, let R_m be a set of all requests in m . We define *request send partition* (RSP), a subset of R_m , to be a set of requests which are mutually independent. Borrowing the set inclusion notation, for any $r_i, r_j \in \text{RSP}$, $(r_i, r_j) \notin \Delta_m$, $(r_j, r_i) \notin \Delta_m$, $(r_i, r_j) \notin \Gamma_m$, and $(r_j, r_i) \notin \Gamma_m$. Thus, an RSP is a set of requests from a statement which are mutually data independent. In the method

```
method twoRsps ()
  r0 <- m0 (r1.m1 (r2.m2 ()), r3.m3 (r4.m4 ()));
end
```

`r1.m1` and `r3.m3` are in the same RSP and `r2.m2` and `r4.m4` are in the same RSP. The join continuation transformation (JCT) proceeds by recursively isolating a continuation of an RSP into a separate actor.

JCT begins with partitioning R_m into RSPs. Before the compiler begins the partitioning, it promotes each request to an assignment statement with a fresh temporary variable. After being modified, the method `twoRsps` looks as follows:

```
method twoRsps ()
(1)   t2 = r2.m2 ();
(2)   t1 = r1.m1 (t2);
(3)   t4 = r4.m4 ();
(4)   t3 = r3.m3 (t4);
(5)   r0 <- m0 (t1, t3);
end
```

With the transformed method m' , the compiler computes the def-use chain DUC_{d_m} [6] for each definition d ¹ in m and constructs data dependence relation $\Delta_{m'}$. Using $\Delta_{m'}$, the compiler shuffles the requests promoted from the same statement in such a way that requests in the same RSP are placed together without violating the partial order relation defined by Δ_m . Note that requests in the same statement may be evaluated in any order. After the requests are partitioned into RSPs lines (2) and (3) are exchanged:

¹We use the terms *definition* and *assignment* interchangeably though definition subsumes assignment.

```

    method twoRsps ()
(1)    t2 = r2.m2 ();
(3)    t4 = r4.m4 ();
(2)    t1 = r1.m1 (t2);
(4)    t3 = r3.m3 (t4);
(5)    r0 <- m0 (t1, t3);
    end

```

After the partitioning is done the method is recursively split on each RSP. The input to this phase is the flow graph of a method. It is a directed graph $FG = (V, E)$ where $V = \{b \mid b \text{ is a basic block in the method}\}$ and $E = \{(b_i, b_j) \mid b_j \text{ can immediately follow } b_i \text{ in some execution sequence}\}$ [6]. One node is distinguished as the *initial* node: it is the block whose leader is the first statement of the method.

The compiler traverses FG in a topological order starting from the initial node. As soon as it encounters an RSP, it splits the enclosing basic block at the point right after the RSP. It pulls out the portion of the flow graph which is dependent on requests in the RSP and encapsulates it into the behavior of a separate continuation actor. The continuation actor has slots to hold replies until all are available. It also keeps the execution context at the moment of sending the request messages in the RSP. The compiler guarantees that only part of the context is cached in the continuation actor, which may actually be accessed during the continuation execution. Thus, it implements the semantics of context switching with minimal overhead.

A statement creating the join continuation actor with the necessary context is placed before the RSP. All request send statements in the RSP are transformed to non-blocking asynchronous send statements, each of which is appended with a set of arguments that collectively identify the unique entry point to the continuation actor. The arguments consist of the mail address of the continuation actor and a method name. After separating out remaining RSPs in the method the compiler applies the split process recursively to each of extracted continuations. The transformation result of the method `twoRsps` are shown in Figure 5.1. Figure 5.2 through Figure 5.4 illustrate how to split methods with different control structures.

5.2.2 Join Continuation Closure

The behavior of a continuation actor is deterministic: as soon as all the expected replies are received, it executes the specified computation and cannibalizes itself. We exploit the deterministic behavior and optimize continuation actors using a structure called *Join Continuation Closure* (JCC) and specialized reply messages.

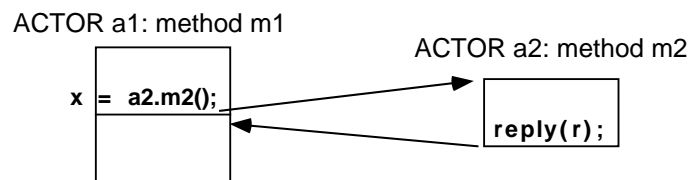
A JCC consists of four components, namely *counter*, *function*, *creator*, and *argument slots* (Figure 5.5). *Counter* contains the number of empty argument slots yet to be filled. A reply message fills the specified slot and decrements the counter. As soon as all slots are filled, *function* implementing the join continuation is invoked with the JCC as its argument. The argument slots which are filled at the creation time are reserved for the execution context used in the continuation execution. With JCC, a reply address is defined by a triple $\langle \text{processor number}, \text{continuation address}, \text{slot address} \rangle$. The discussion of the use of *creator* which represents the actor that sent the request messages is deferred to Section 5.2.4.


```

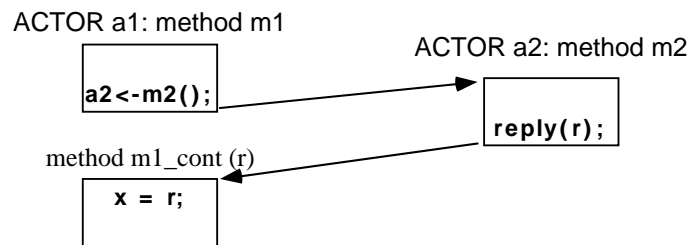
method twoRsps ()
  c1 = C1.new (r1,r3,r0);
  t2 <- r2.m2 (c1, m2t2c1);
  t4 <- r4.m4 (c1, m4t4c1);
end
behv C1
| r1c1, r3c1, r0c1, t2c1, t4c1, cnt |
init (i1,i2,i3)
  r1c1 = i1;    r3c1 = i2;    r0c1 = i3;
  t2c1 = nil;   t4c1 = nil;
  cnt = 2;
end
method m2t2c1 (r)
  | c2 |
  if (cnt == 0) then
    c2 = C2.new (r0c1);
    t1 <- r1c1.m1 (t2c1, c2, m1t1c2);
    t3 <- r3c1.m3 (t4c1, c2, m3t3c2);
  else
    cnt = cnt - 1;
    t2c1 = r;
  end
end
method m4t4c1 () ... end
end
behv C2
| r0c2, t1c2, t3c2, cnt |
init (i)
  r0c2 = i;
  t1c2 = nil;   t3c2 = nil;
  cnt = 2;
end
method m1t1c2 (r)
  if (cnt == 0) then
    r0c2 <- m0 (t1c2, t3c2);
  else
    cnt = cnt -1;
    t1c2 = r;
  end
end
method m3t3c2 (r) ... end
end

```

Figure 5.1: The transformation result of twoRsps using continuation actors.

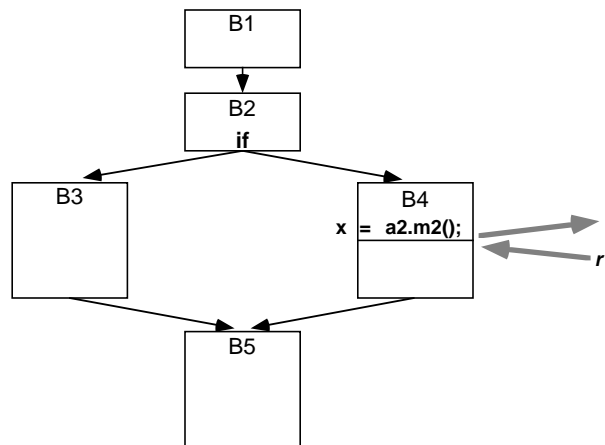


(a) Before the transformation

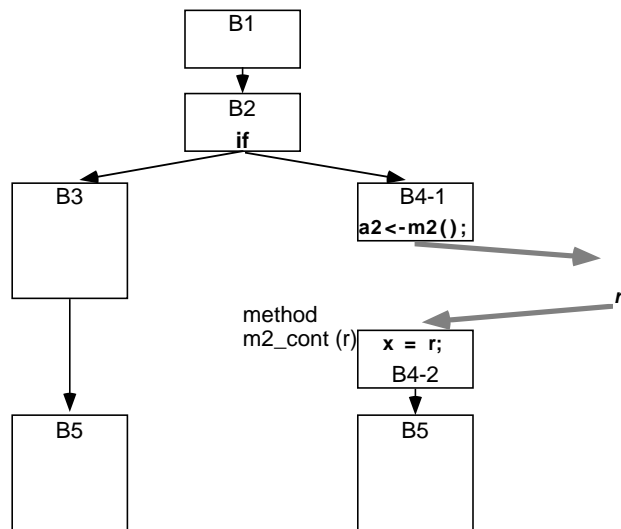


(b) After the transformation

Figure 5.2: Extracting continuations from a method with no branch.

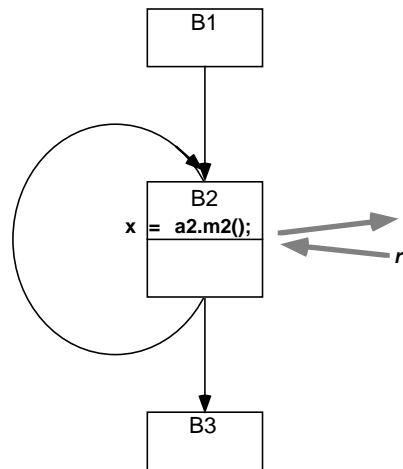


(a) Before the transformation

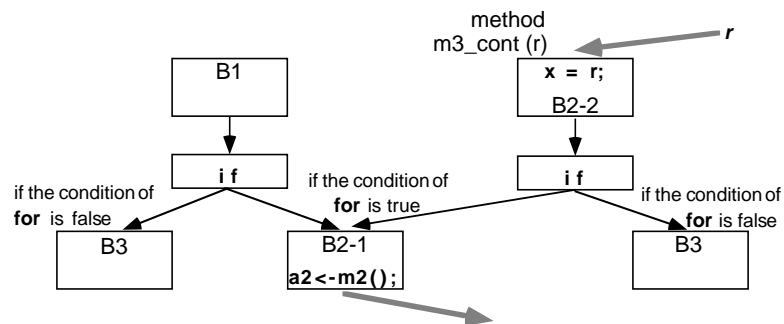


(b) After the transformation

Figure 5.3: Extracting continuations from a method with branches.



(a) Before the transformation



(b) After the transformation

Figure 5.4: Extracting continuations from a method with loops.

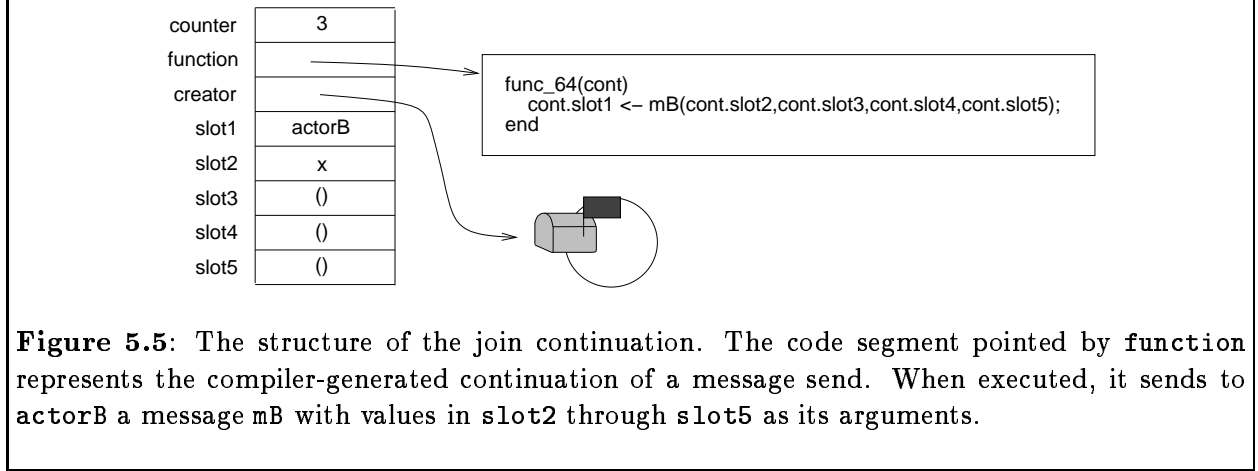


Figure 5.5: The structure of the join continuation. The code segment pointed by **function** represents the compiler-generated continuation of a message send. When executed, it sends to **actorB** a message **mB** with values in **slot2** through **slot5** as its arguments.

5.2.3 Common Continuation Region Analysis

The algorithm described above can only exploit functional parallelism which is inherent in the evaluation of function arguments with the CCRC. Consider a code fragment for the N-body example. The algorithm is not general enough to recognize that **request** sends in the **if** blocks are indeed independent of each other and can be executed concurrently. In this section, we present a more general transformation framework based on a data dependence analysis. The proposed technique is sufficiently general so that it can identify a set of **request** send statements which can share the same join continuation across statement boundary. The transformation technique is based on a simple observation: any two **request** sends can be executed concurrently if no data dependence exists between the two **request** message sends.

As before, the transformation is divided into a *partitioning* phase and *split* phase. The input to the partitioning phase is an abstract syntax tree $AST = (AV, AE)$ of a method. As in the algorithm discussed in Section 5.2.1, we assume that all **request** sends have been lifted to assignment statements and there are no expressions containing embedded request sends. Since all relevant information is at the statement level, we abstract away expressions in each statement and assume that AV is a set of all statements in the method. **if** statement and **for** statement represents **if** conditional and **for** loop header, respectively.

Given an AST for a method, we define a *common continuation region* at $s \in AV$ (CCR_s) to be a subtree rooted at s such that all **request** send statements it contains may share the same join continuation. A *maximal* CCR is defined to be a CCR which is not contained in any other CCRs in the AST. Formally, a CCR is the set $S = (SV, SE)$ where $SV \subseteq AV$, $SE \subseteq AE$ if for any request send statement r in a method m , the cardinality of set $\{(r, s) \in \Delta_m \mid s \in S\}$ is 1.

During the partitioning phase, we partition the **request** message send statements in a method into maximal CCRs. The partitioning algorithm itself can be described as a tree coloring (Figure 5.6). We color subtrees in the AST using three colors, *white*, *grey* and *black* which represent a subtree with no **request** send statement, a CCR, and a subtree which has at least one **request** send statement but is not a CCR, respectively. First, we color a leaf node grey if it is a **request** send statement; otherwise, we color it white. For each internal node, we color it black if at least one of its children has **request** send statement(s) but is not a CCR (*i.e.*, if it has at least one black child node). If a subtree has no **request** send (*i.e.*, all of its children are white), we color it white. If neither case applies, the subtree must have some white children and some grey children. We

INPUT: the abstract syntax tree $AST = (AV, AE)$ of a method.

OUTPUT: the abstract syntax tree with CCRs marked.

ALGORITHM:

For each $s \in AV$, let $Child_s = \{c \mid (s, c) \in AE\}$ and let $color_s$ denote a color assigned to s .

1. Color each leaf node with *white* or *grey* using the definition of CCR for leaf nodes.
2. For each $s \in AV$, color s with
 - *black* if $\exists c \in Child_s, color_c$ is *black*.
 - *white* if $\forall c \in Child_s, color_c$ is *white*.
 - Let $D = \{s \mid \text{for } c \in Child_s, color_c \text{ is } white \wedge r \in R_c \wedge s \in AV_s - AV_c \wedge r \mapsto s\}$ where R_c is a set of **request** send statements in a subtree rooted at c . Color s with *grey* if $D = \emptyset$. Otherwise, color it with *black*.

Figure 5.6: The coloring algorithm to compute maximal CCRs.

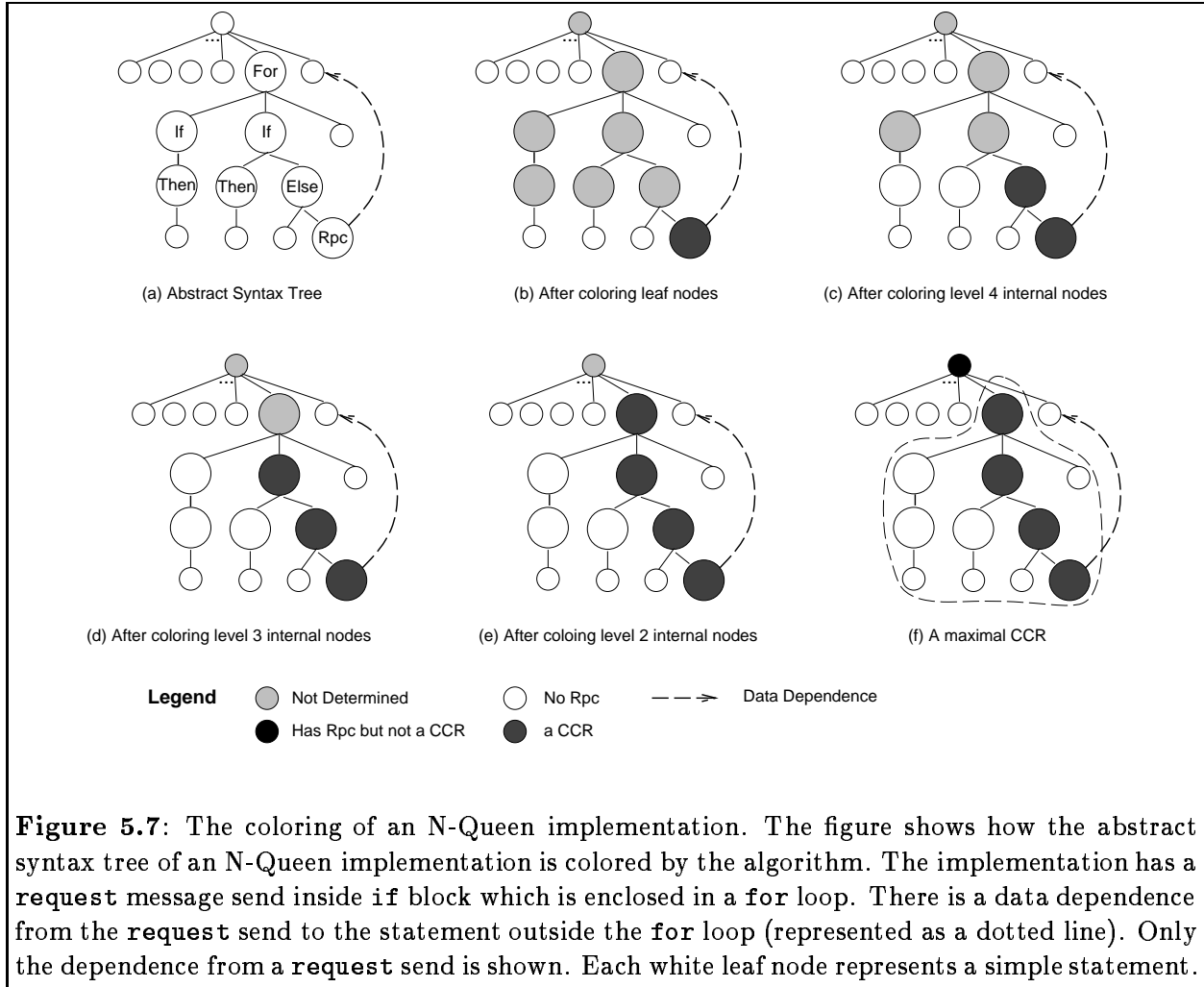
determine if there is data dependence from any **request** send statement to any other statement in the subtree rooted at the node except one from which the *request* send is pulled out. If there is no such data dependence, we make it a grey node. Otherwise, we make it a black node. Finally, we merge into a single CCR consecutive maximal CCRs that have the same parent node in the AST and have no data dependence between them. Figure 5.7 illustrates how maximal CCRs are determined for a given AST.

Split phase is similar to the one described in Section 5.2.1. After all **request** sends are partitioned into maximal CCRs, we compute for each maximal CCR the maximum number of reply slots that may be used. If the conservative estimate of the number cannot be determined² or if the compiler finds that splitting at the CCR is not profitable, the corresponding subtree is changed to non CCR and join continuations are extracted for the smaller CCRs in the subtree. The method is split at the point right after each maximal CCR and a separate object is allocated to encapsulate the continuation to the CCR. Then, all the **request** sends are transformed into non-blocking asynchronous sends with an additional statement which counts the number of messages that are actually sent. Finally, a statement is appended to the CCR to adjust the number of replies that the continuation object must wait before it executes the specified continuation. If nodes in a maximal CCR are all leaf nodes (*i.e.*, all are **request** send statements), the overhead associated with CCRs is not justified and we use the algorithm in Section 5.2.1 to separate out the join continuation.

5.2.4 Method Fission

JCT restores useful concurrency by separating out join continuations from an original computation. However, indiscriminate application of JCT may result in incorrect computation. Consider a method which sends a request message and defines the actor's next state with the result. The next state is only partially defined; the sender should not process messages until the reply arrives and

²E.g., a subtree corresponding to the CCR may represent two branch if statement that contains **for** loops.



execution of the continuation completes. However, applying JCT blindly may cause the actor to process the next message in an inconsistent state.

For example, the result of a request in Figure 5.8 is used to define the value of acquaintance variable `a`. The result of blind application of JCT is shown in Figure 5.8.b. The result from the message send is erroneously assigned to a compiler-generated temporary variable.

For a class of methods which modifies the actor's next state using a result from a request, we still apply JCT but with more care. As before, the continuation is separated into JCC. But, this time the actor locks itself right after sending the request (the compiler inserts an additional lock statement). The continuation is dispatched by the runtime system through the JCC regardless of the actor's state. Note that the sole purpose of locking the actor is to keep it from processing subsequent messages prematurely. Execution of continuation unlocks the actor using the `creator` field in JCC.

5.2.5 The Deadlock Problem

Consider a program in Figure 5.9. The program generates the Fibonacci sequence. The figure also has the expected message trace when $n = 3$. Suppose a naive implementation is used which simply blocks a sender on each request. Under the atomic method execution the program cannot be executed without causing deadlock. The shaded actor is blocked after sending its very first message to `self` and unable to progress because it cannot process the message. A future-based implementation is no better than the naive implementation [118].

Some systems replace a message send to self with a function call [28, 75]. It may be successful in avoiding deadlock but may not be used with the atomic method execution because the replacement may alter the meaning of the program incorrectly, as shown in Figure 5.10. Further, it serializes the sender's computation eliminating the possibility of dynamic load balancing. In this example, JCT helps avoid deadlock by allowing the sender to continue executing the remaining computation independent of the result of the request after sending a request (Figure 5.11).

Nonetheless, the deadlock problem is a potential difficulty that limits the usefulness of the CCRC abstraction. Although the compiler detects and avoids many spurious deadlocks using data dependence analysis and the join continuation transformation, it is in general not feasible to prevent all deadlock situations by using compile-time analysis and source-level transformation. For example, Suppose the following code fragment has been executed with behaviors in Figure 5.12:

```
left = DeadLockLeft.new ();
right = DeadLockRight.new();
left <- setup (right);
right <- setup (left);
left <- deadlock ();
```

The last message send statement causes a deadlock if the method execution is *atomic*. The `left` actor blocks upon the `request` send to the `right` actor and vice versa.

Even the JCT could not break a deadlock if it was caused by an indirectly recursive `request` send whose result defines the actor's next state. In general, it is impossible to break such deadlocks with compile-time transformation.


```

behv Customer
| a, b, c |
method m1(x,y)
  %% computation 1
  a = (Server.new()). m2(x,y);
  %% computation 2
end
...
end

```

(a) Customer behavior

```

behv Customer
| a, b, c |
method m1(x,y)
  %% computation 1
  Server.new() <- m2(x,y);
end
...
end

```

```

func_28 (cont)
| t |
t = cont. slot1;
%% computation 2 with t
%% having replaced a.
end

```

(b) An incorrect transformation result

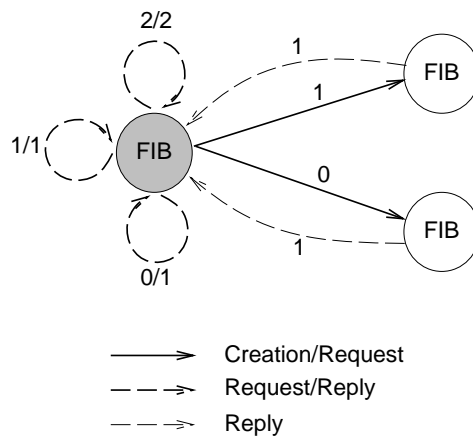
Figure 5.8: An incorrect fission result. The result from the request send m2 defines instance variable a's new value. The application of join continuation transformation alters the meaning of method m1.

```

behv Fib
  method compute(n)
    if (n == 0) then reply(0);
    elseif (n == 1) then reply(1);
    else reply(self.compute(n-1) + (Fib.new()).compute(n-2));
    end
  end
end

```

(a) An implementation



(b) Message trace graph. Associated numbers denote the argument passed.

Figure 5.9: Fibonacci number generator with a recursive message send.

```

behv WriteAndSendPlusOne
  | value |
  ...
  method read_and_add1()
    reply (value+1);
  end
  method write(to,n)
    to <- send_plus_one(self.read_and_add1());
    value = n;
  end
end
(a) Before

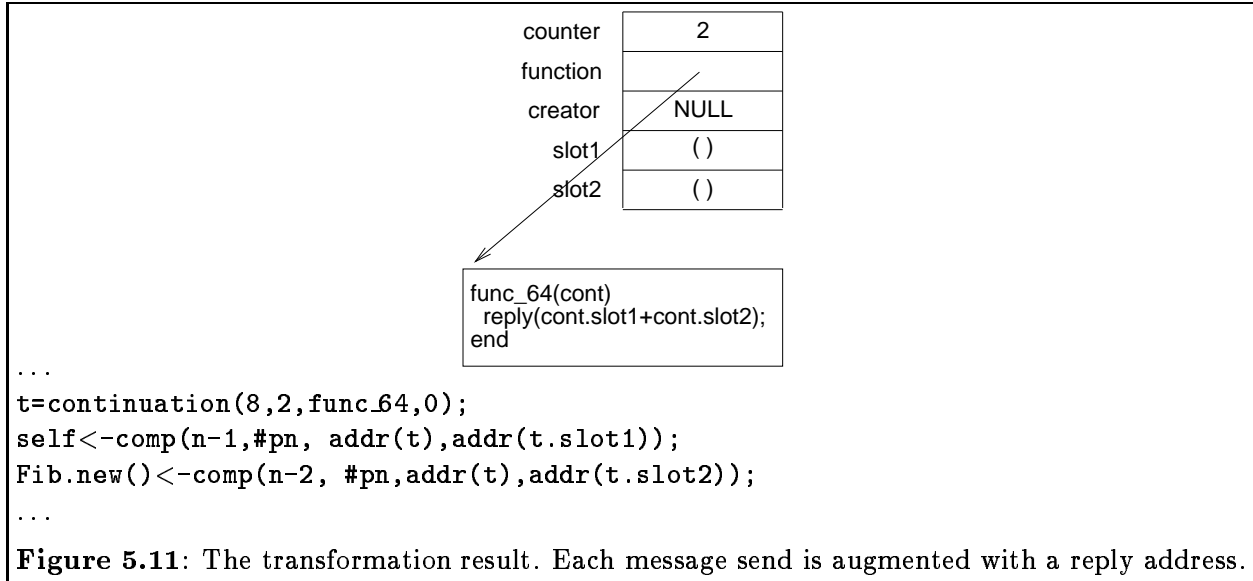
```

```

behv WriteAndSendPlusOne
  | value |
  ...
  method read_and_add1()
    reply (value+1);
  end
  method write(to,n)
    to <- send_plus_one(read_and_add1());
    value = n;
  end
end
(b) After

```

Figure 5.10: An incorrect join continuation transformation. The transformation of a message send to `self` to a function call may alter the meaning of the program. (a) Method `read_and_add1` is meant to read the new value of `value` which is to be written in method `write`. (b) The transformation makes method `read_and_add1` read the old value of `value` incorrectly.



At best, a compiler may conservatively alert programmers of the possible presence of deadlock. The THAL compiler is able to do so by using a form of global flow analysis to construct a call graph and checking whether or not the **request** sends form a (potential) cycle. The compiler constructs a message trace graph [99] for a given program and detects a cycle that contains a **request** send whose result is used to define the next state of the actor. The trace graph can represent only a subset of all possible message trace. Thus, detecting deadlock using a trace graph can be no more than an *approximation*. But it is *safe*: if a program may indeed encounter deadlock, the compiler gives a warning.

The JCT tends to generate many small methods, especially for those programs that use **request** sends extensively. This does not necessarily degrade the execution performance because compiler-generated methods are dispatched only by continuations and need not be included in a method lookup table.

5.3 State Caching and Eager Write-Back

In actor computation a method is given exclusive access to the actor when executed. If an actor is executing a method, the next message is not scheduled until the actor's next behavior is fully defined. The semantics of atomic method execution is realized by making **become** an instant operation and by creating an anonymous actor with the same behavior when **become** is executed. The actor becomes ready to process the next message as soon as it executes **become**. The anonymous actor carries out the rest of the computation. The use of anonymous actors allows for concurrent execution of methods belonging to the same actor in the Actor model.

THAL implementation creates a thread to carry out a method execution (Chapter 4). In such implementations, atomic method execution may be realized by locking the actor and caching into the thread a subset of the actor's state which may be accessed during the method execution. No further messages are processed while an actor is locked. A method is executed using the local copy of the state in the thread (*i.e.*, *state caching*). At the end of the method execution the acquaintance variables that might be modified is written back. Notice that such implementations serialize the

```

behv DeadLockLeft
  | right,result,value |
  init ()
    right = nil; result = 0; value = 0;
  end
  method setup(a)
    right = a;
  end
  method deadlock()
    result = right.dead();
  end
  method lock()
    value = value + 1;
    reply (value);
  end
  ...
end

behv DeadLockRight
  | left |
  init (a)
    left = nil;
  end
  method setup(b)
    left = b;
  end
  method dead()
    reply (left.lock() + 1);
  end
  ...
end

```

Figure 5.12: A deadlock example involving indirect recursive message sends.

method executions which may change the actor's state as well as serialize the state changes. They allow for only the concurrent execution of read-only methods.

Such serialization is unnecessarily stringent and prevent even safe overlap in executions which might be profitable in some cases. Suppose an actor executes a method m_1 which modifies some acquaintance variables and then uses them without further modification. When another actor wants to read some of the actor's acquaintance variables (say the corresponding method is m_2), the implementation cannot dispatch m_2 until the actor have finished the execution of m_1 . This is because the new values are written at the end of the method execution. If the values are written back as soon as they are defined (or, at least as soon as the last value to the modification is defined), the actor may be released even before the end of the method. m_2 may be dispatched while m_1 is executing and both actors may proceed concurrently.

The input to the analysis is the control flow graph of a method whose nodes are a basic block. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [6]. A basic block may have one or two successor blocks, *jump-on-true* (JOT) and *jump-on-false* (JOF). JOF is not defined for a *simple* block; only *if* block and *for* block have two successor blocks.

For each basic block acquaintance variables whose values are newly defined in the block are collected into a set called *Updates* (U_B). The set is upward propagated so that each block has a set of acquaintance variables whose values may be defined along an execution path starting from the block to the end of the method (*Reachable Updates* (RU_B)). More concisely,

$$RU_B = U_B \cup RU_{JOT_B} \cup RU_{JOF_B}^3$$

Using these two sets, the compiler generates the write-back statements on the fly as it generates codes for a basic block. After the generation of the last update statement ⁴ in each basic block, the compiler generates write-back statements for acquaintance variables whose values are finalized (*i.e.* $U_B - (RU_{JOT_B} \cup RU_{JOF_B})$). The rest (*i.e.* $U_B \cap (RU_{JOT_B} \cup RU_{JOF_B})$) are passed to its successor blocks as leftover (LO_{JOT_B} and LO_{JOF_B}).

More specifically, for a block with two successor blocks, LO_B is passed to both of its successor blocks. If the basic block is a single branched *if*, it is made double branched by adding a ghost block as its *JOF* successor before passing LO_B down.

For a simple basic block which has only one successor block, the compiler generates $LO_B - RU_B$ before generating any statement in the block. After the generation of the last update statement in the basic block, write-backs for $U_B - RU_{JOT_B}$ are generated and $U_B \cap RU_{JOT_B}$ is passed as leftover.

Consider an update statement enclosed in a *for* loop. If we simply generate write-backs as described thus far, the acquaintance variable will be unnecessarily written for the number of loop iterations. It would be more efficient to put all the write-backs outside the loop and let the updates be done on local variables so that the write-back occurs only once, The write-backs for the acquaintance variables whose values may be defined in a *for* loop body are not generated in the loop body but the set of variables are passed contained in LO . They are generated in the block which follows the *for* loop in the input program. The generation algorithm is summarized in Figure 5.13.

³If *JOF* is not defined, JOF_B is an empty set.

⁴An update statement is an assignment statement to an acquaintance variable.

INPUT: Control flow graph of a method whose nodes are basic blocks

OUTPUT: Code generation as side effects

ALGORITHM:

for each basic block B , compute U_B and RU_B .

for each basic block in the topological order

1. generate leftover write-back statements.
2. after generating the last update statement in the basic block,
 - if the statement is an IF statement
 - if enclosed in neither a FOR loop block nor a single branch IF statement, generate write-backs for $LO_B - RU_{JOT_B}$
 - else, pass its leftover to both jump-on-true and jump-on-false successor blocks
 - if the statement is a FOR header, pass its leftover to the jump-on-false successor block
 - if neither of the above holds and if not enclosed in a FOR loop, generate write-backs for $U_B - RU_{JOT_B}$ and let $LO_{JOT_B} = U_B \cap RU_{JOT_B}$

Figure 5.13: An algorithm to generate write-back statements for acquaintance variables.

5.4 Local Actor Creation

An actor may send a message to any actor as long as it knows the receiver's mail address independent of its physical location. Thus, actor creation in stock-hardware multicomputers should involve more than simple allocation of a chunk of heap space; it must involve location-transparent mail address allocation, heap space allocation and possibly actor initialization. Support for location transparency adds quite an overhead to actor creation. Fortunately, not all actors need such full-fledged creation. For example, functional actors which have no acquaintance variables need not be created at all. Certain local actors will never receive messages from remote actors and need not have location transparency. To get better performance, a cheaper implementation may be used when creating these actors. The compiler uses the def-use analysis and the constant propagation [6] to identify such actors. Specifically, the compiler examines each creation expression to see if the following conditions are to be satisfied: (i) both the creator and the createe never migrate, (ii) the creator does not export the mail address of the createe to the third party actors, and (iii) the createe does not export its mail address to third party actors. If the compiler ascertains that the conditions will hold during the execution, it generates the codes that exploit the information.

5.5 Related Work

A number of type inference mechanisms for object-oriented programming languages have been proposed [115, 46, 100, 22]. In particular, the type inference in the THAL compiler is implemented using a constraint-based type inference algorithm [100]. The implementation is similar to that of [99] but is extended to infer types for groups and member actors. A more detailed discussion on constraint-based type inference for object-oriented programming languages may be found in [101]. A

similar constraint-based type inference mechanism was implemented on the Concert system [28, 75] and is presented in [104]. The implementation iteratively traverses a global flow graph of a program to refine type information it gathers.

The discussion of join continuation in the context of the Actor model appears in [1]. Extracting join continuation through a source-to-source transformation was attempted in other actor-based languages as well [89, 62]. Also, a similar transformation technique for explicitly message-passing programs was presented in [59]. The common continuation region analysis extends the base join continuation transformation (Section 5.2.1) to restore concurrency across loop boundary by using data dependence analysis [7, 14, 15].

Chapter 6

Performance Evaluation

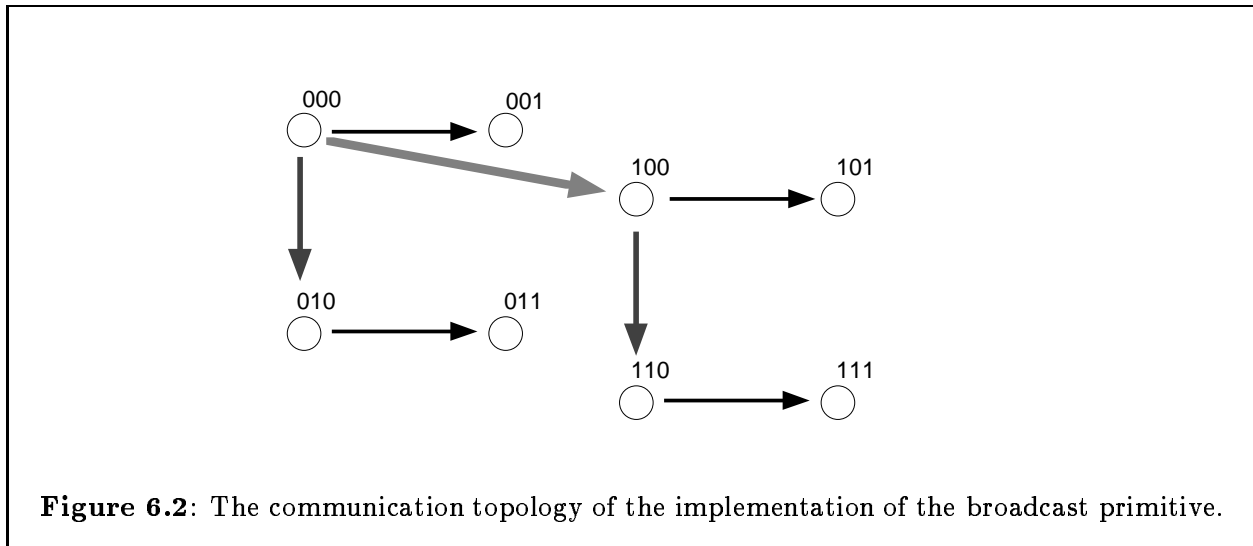
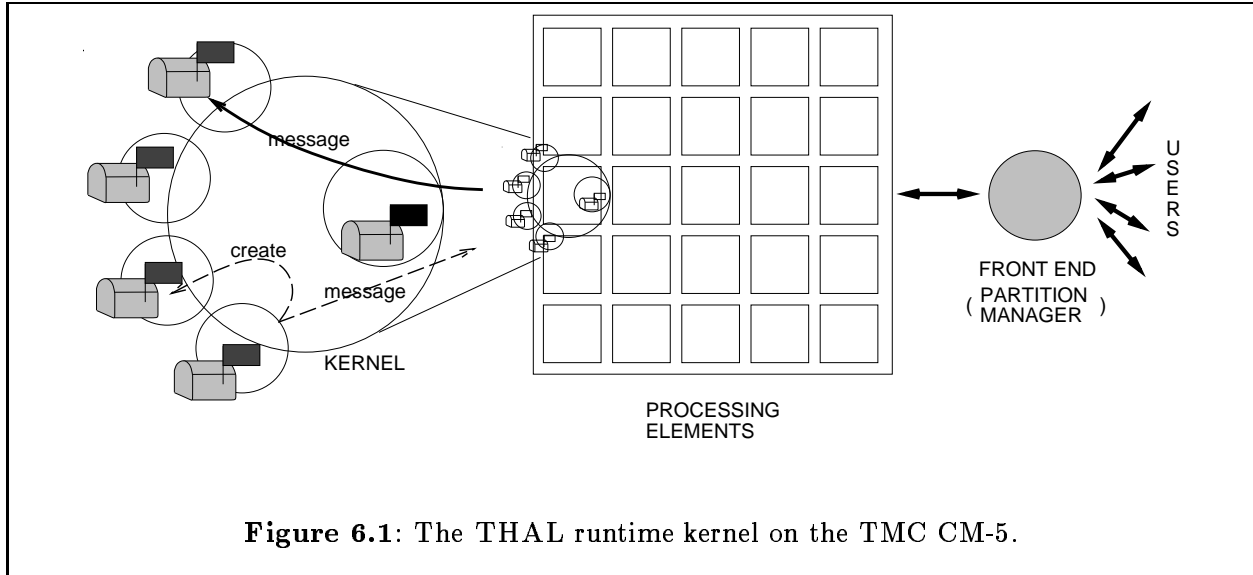
The runtime system has been running on the TMC CM-5 and porting it to other platforms, such as Cray T3D/T3E, SGI PowerChallenge array, and networks of workstations, is in progress. This section presents implementation details and evaluation results of the TMC CM-5 version. Most of the runtime system on the CM-5 is written in C and only part of scheduling code that implements tail-call is written in the assembly language. The part written in C was compiled using the GNU C compiler with -O3 option and the assembly part was compiled using the GNU assembler. The compiler takes a THAL program and generates C code which is compiled using the GNU C compiler with -O3 option.

6.1 The Runtime System on the TMC CM-5

The TMC CM-5 is a distributed memory multicomputer which may 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 *processing elements*. Each processing element hosts a 33 MHz Sparc processor and a network interface chip. The network chip supports all accesses to the interconnection network. The CM-5 has three kinds of networks: the data network, the control network, and the diagnosis network. The data network is responsible for application data motion and connects the processing nodes in the fat tree topology. The other two CM-5 networks have binary tree topologies. See [122] for additional details on the three CM-5 interconnection networks.

Figure 6.1 shows the virtual architecture defined on the TMC CM-5. The runtime system consists of a *front-end* which runs on the partition manager and a set of virtual processors which run on the processing elements (*i.e.*, *nodes*). Kernels are implemented as ordinary UNIX processes. Users are provided with a simple command interpreter which communicates with the front-end to load the executables. In addition to dynamic loading, the front-end processes all I/O requests from the kernels.

The communication module is implemented with a veneer layer on top of CMAM [131]. All THAL messages have a destination actor address and a method selector. The layer exploits these properties to minimize communication overhead. The broadcast primitive is implemented in terms of point-to-point communication on the CM-5 data network, using a hypercube-like minimum spanning tree communication structure (Figure 6.2). Although the CM-5 has a broadcast facility using the separate control network, it is not available in the CMAM layer. Moreover, simulating



broadcast on the data network was more efficient when sending bulk data because the bandwidth of the data network is much higher than that of the control network [84].

Since Active Messages are not buffered [131], sending bulk data from one node to another requires a three-phase protocol. The source sends size information and the destination acknowledges with a buffer address. Then the source sends data without any concern about overflow. However, because Active Messages are scheduled immediately at the destination, the unconstrained three-phase protocol may develop multiple outstanding bulk data transfers at the destination, increasing contention at the network interface and packet back-up in the network and degrading network performance. In order to avoid the problems the runtime system controls the flow of bulk data transfer. A node manager acknowledges bulk data transfer requests in a first-come-first-served manner so that only one request is outstanding at any time. Flow control reduces packet back-up in the network, improving network performance as well as processor efficiency.

| | THAL | | ABCL | | Concert | |
|------------------|-----------------|-----------|-----------------|--------|-----------------|-------|
| | μsec | cycle | μsec | cycle | μsec | cycle |
| platforms | CM-5 (33 MHz) | | AP1000 (25 MHz) | | CM-5 (33 MHz) | |
| local creation | 8.04 | 265 | 2.1 | 69 | N/A | N/A |
| remote creation | 5.83 (20.83) | 192 (687) | N/A | N/A | N/A | N/A |
| locality check | 1.00 | 33 | 0.12 | 3 | 11.70 | 390 |
| lsend & dispatch | 0.45/5.67 | 15/187 | 2.18/9.6 | 55/240 | 0.12/8.44 | 4/277 |
| rsend & dispatch | 9.91 | 327 | 8.9 | 223 | 7.67 | 252 |

Table 6.1: Execution times of runtime primitives. Local send and dispatch time does not include the locality check time. Times are measured by repeatedly sending a message with no argument. Others are cited from the previously published papers. [†]The local execution of remote actor creation in HAL takes 5.83 μsec while the actual latency is 20.83 μsec .

6.2 Performance of the Runtime Primitives

Table 6.1 summarizes execution time of the THAL runtime primitives. As mentioned earlier, the use of aliases makes it possible to complete a local execution of remote creation in 5.83 μsec where the actual latency is 20.83 μsec . A locality check is done using only locally available information and completes within 1 μsec for locally created actors. The performance of the runtime primitives is comparable to that of other systems [119, 75]. The runtime system also supports two primitives to implement the call/return communication abstraction: continuation creation and reply. Continuation creation with two slots, one empty and one filled, takes 2.27 μsec and deallocation takes 0.75 μsec . Sending a reply locally takes 2.76 μsec and sending it remotely takes 9.26 μsec .

Below we present performance results of five benchmarks: namely the Fibonacci number generator, a systolic matrix multiplication, a bitonic sorting problem, an N -Queen problem, and an adaptive quadrature problem. The Fibonacci number generator is used to examine overhead of the message layer of the runtime system as well as effectiveness of dynamic load balancing. The systolic matrix multiplication example shows that the THAL system delivers performance comparable to less flexible systems when execution granularity is sufficiently large. The bitonic sorting problem also shows that the runtime system supports scalable execution. The last two examples are presented to demonstrate the effects of different placement strategies. For each application, we put a brief problem statement, evaluation results, and analysis.

6.3 Fibonacci Number Generator

The Fibonacci number generator computes the n -th number in the Fibonacci sequence using the recurrence relation, $Fib(0) = 1, Fib(1) = 1, Fib(n) = Fib(n-1) + Fib(n-2)$ (Figure 4.4). Although it is a very simple program, it can be used to measure overhead of the messaging layer in the runtime system takes because each Fibonacci actor's thread length is quite small. Figure 6.4 compares performance of THAL versions computing Fibonacci of 33 on a single node of TMC CM-5 with that of an optimized C version on a single Sparc processor. The C version completes in 8.49 seconds. As a point of comparison, computing $Fib(33)$ in the Cilk system [19] on a single Sparc processor takes 73.16 seconds.




Figure 6.3: Comparison of performance of Fibonacci implementations with and without dynamic load balancing.

The Fibonacci program is extremely concurrent: naively computing $\text{Fib}(33)$ creates 11,405,773 actors. However, the THAL compiler optimizes away actor creations since Fibonacci actors are purely functional. The computation tree of the Fibonacci program has a great deal of load imbalance. Table 6.2 and Figure 6.3 compare two execution results with and without dynamic load balancing (DLB). A receiver-initiated random polling scheme [83] is used for dynamic load balancing. As Figure 6.3 shows, the version with DLB performs worse on partitions of a small size due to the overhead for extra book-keeping. However, it eventually outperforms the version without DLB as the size increases.

6.4 Systolic Matrix Multiplication

The systolic matrix multiplication algorithm, also known as Cannon's algorithm [82], uses N^2 processors where N is a natural number. To compute $C = A \times B$, each matrix is divided into N^2 square blocks and matrix A is row-skewed and matrix B is column-skewed. Then, A , B , and C are placed on the square processor grid (Figure 3.10). At each step of the execution, node P_{ij} performs

Figure 6.4: Comparison of performance of Fibonacci implementations on a single Sparc processor.

| 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 6.3: Execution times of a systolic matrix multiplication problem. (unit: seconds). All results were obtained by executing the program with a $M \times M$ matrix on a $P \times P$ processor array.

local matrix multiplication with blocks $A_{P_{ij}}$ and $B_{P_{ij}}$. Then, $A_{P_{ij}}$ s are cyclicly shifted to the left, and $B_{P_{ij}}$ s to the upward. After N iterations, the result is in matrix C .

Unlike usual systolic implementations, no global synchronization is used to make computation march in lock-step fashion. Rather, per-actor-basis local synchronization is used to simulate the barrier synchronization. Local block matrix multiplication is implemented using the same assembly routine used in [33]. Table 6.3 shows the execution times of a THAL implementation on TMC CM-5. Results are comparable to those given in [33] (Figure 6.5). For example, the performance peaks at 434 MFlops for a 1024 by 1024 matrix on a 64 node partition of a CM-5. The results show that, despite its flexibility, our implementation is as efficient as other more restrictive low-level ones when granularity is sufficiently large.

6.5 Bitonic Sort

A *bitonic sequence* is a sequence of elements $\langle a_0, a_1, \dots, a_{n-1} \rangle$ with the property that either (1) there exists an index i , $0 \leq i \leq n-1$, such that $\langle a_0, \dots, a_i \rangle$ is monotonically increasing and

Figure 6.5: Comparison of performance of THAL and Split-C implementations of systolic matrix multiplication.

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|--------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 32768 | 3.103 | 1.458 | 0.694 | 0.333 | 0.171 | 0.093 | 0.056 | 0.038 | 0.029 |
| 65536 | 7.084 | 3.279 | 1.593 | 0.789 | 0.387 | 0.206 | 0.114 | 0.070 | 0.047 |
| 131072 | 15.83 | 7.376 | 3.578 | 1.758 | 0.920 | 0.459 | 0.246 | 0.143 | 0.085 |

Table 6.4: Performance of a bitonic sorting problem. (unit: seconds). A block of elements in a node is sorted using the sequential heap sort algorithm.

$\langle a_{i+1}, \dots, a_{n-1} \rangle$ is monotonically decreasing, or (2) there exists a cyclic shift of indices so that (1) is satisfied. We define *bitonic merge* as a process of rearranging a bitonic sequence into a sorted sequence. Then, *bitonic sort* is defined as an algorithm which sorts a sequence of numbers by repeatedly merging bitonic sequences of increasing length. In the experiments, a group of actors is created using `clone`: thus, one actor per processor. Array elements were distributed evenly over the member actors. As a result, the number of elements assigned to an actor decreases as the number of processors increases. Each actor independently sorts a block of elements using the sequential heap sort algorithm before the block bitonic sorting begins. Synchronization is enforced by using local synchronization constraints. Timing results using different numbers of elements are shown in Table 6.4; Figure 6.6 shows their scalability.

6.6 N-Queen Problem

The goal of the N -Queen problem is to place N queens on an $N \times N$ chess board such that no two queens are placed on the same row, column, or diagonal. The version used in the experiment computes the number of solutions of a given configuration [117]. Table 6.5 has the timing results

Figure 6.6: Comparison of performance of a bitonic sorting problem with different problem sizes.

with different placement policies when $N = 13$. The first column shows the results obtained with random placement while others have the numbers with the subtree-to-subcube placement. In the subtree-to-subcube cases, actors are randomly placed until the depth reaches D . A subtree-to-subcube placement with a smaller depth places more actors locally. Note that the execution times on a single processor are different. Although the same program was used, the random version followed an execution path which visited local message sending with a more general interface because receiver's locality is not known until the message sending time. By contrast, some of messages destined to local actors are known at compile time in the subtree-to-subcube versions and the compiler uses a more efficient interface for these message sends. Thus, the performance difference in the local scheduling mechanisms (refer to Table 6.1) is reflected in the single processor performance numbers.

The subtree-to-subcube versions show better scalability as well as better performance than the version using random placement (Figure 6.7). Scalability characteristics differ even among the subtree-to-subcube versions, demonstrating the importance of placement in performance. As a point of comparison, a C implementation without any object allocation takes 8.05 seconds.

6.7 Adaptive Quadrature

Evaluation of integrals is called *quadrature*. An automatic quadrature algorithm takes as inputs a function f , an interval $[a, b]$, and an accuracy request ϵ and produces a result Q and an error estimate E . The algorithm recursively divides all subintervals until it meets the accuracy request ϵ . A sequential, globally adaptive quadrature technique [71] saves computation time by starting integral evaluation from coarser subintervals and repeatedly dividing the subinterval with the largest local error estimate until the accuracy request is met.

| PEs | Random | D = 7 | D = 6 | D = 5 |
|-----|--------|-------|-------|-------|
| 1 | 105.4 | 91.23 | 90.70 | 90.35 |
| 2 | N/A† | 59.02 | 53.96 | 52.37 |
| 4 | 118.2 | 33.09 | 27.91 | 27.07 |
| 8 | 68.69 | 17.38 | 14.10 | 13.69 |
| 16 | 36.39 | 9.120 | 7.289 | 7.102 |
| 32 | 18.93 | 4.610 | 3.898 | 3.392 |
| 64 | 10.94 | 2.610 | 2.040 | 1.653 |
| 128 | 5.884 | 1.318 | 1.092 | 0.843 |
| 256 | 3.260 | 0.628 | 0.595 | 0.476 |

Table 6.5: Performance of a 13-Queen problem. (unit: seconds)

| | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|-----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| depth = 3 | 184.0 | 107.8 | 46.78 | 27.33 | 16.89 | 11.97 | 5.604 | 3.369 | 2.577 |
| depth = 4 | 183.7 | 94.05 | 49.43 | 27.01 | 16.68 | 9.944 | 5.765 | 3.194 | 1.864 |
| depth = 5 | 183.8 | 95.66 | 52.06 | 27.94 | 16.50 | 9.091 | 5.797 | 3.028 | 1.738 |
| Random | 183.5 | 96.12 | 52.44 | 27.66 | 17.11 | 9.282 | 5.445 | 2.921 | 1.952 |

Table 6.6: Performance of an adaptive quadrature problem. (unit: seconds)

A parallel version of the algorithm may be implemented using a master-worker configuration. First, the master creates a worker on each processing node, divides the input interval into equally spaced subintervals, and assigns one to each worker. Each worker is responsible for computing the integral estimate and the error estimate on its subinterval. If the error estimate is sufficiently small, it returns its integral estimate to the master as the result. Otherwise, it refines the integral estimate by creating child workers, dividing its subinterval, and assigning one to each child. Then, it waits for results, sums them, and returns the sum to the master. In our experiment, the following function was integrated over the interval from $[0.001, 32]$ using an initial subinterval length of 0.1 and the error bound of 10^{-4} .

$$10^6 + |10^6 \times \frac{\sin(0.5/x)}{x}|$$

The N -Queen example demonstrates that different placement strategies may exhibit different performance characteristics. This example shows that it is not always the case. In adaptive quadrature the work assigned to a processing element may dynamically increase as the computation proceeds. The subtree-to-subcube placement strategies cause more remote actors to be created and more remote messages to be sent as the cutoff depth increases. As in the N -Queen problem, versions which scheduled more computations locally performed better (Table 6.6). However, the difference is not significant (Figure 6.8).

Figure 6.8: Comparison of performance of an adaptive quadrature problem with different placement strategies.

Chapter 7

Conclusion

The thesis shows that communication in actor programming which involves both sending and scheduling messages, can be efficiently and scalably supported on stock-hardware distributed memory multicomputers. In particular, we have developed a number of run-time implementation techniques and compile-time analyses for flexible actor communication abstractions and applied them to an actor-based language, THAL.

The language supports flexible high-level communication abstractions, such as concurrent call/return communication, delegation, local synchronization constraints, and broadcast. The runtime system implements an efficient message delivery subsystem which supports location transparency, a remote object creation mechanism which allows remote creation to overlap with local computation, and a scheduling mechanism which recognizes and exploits the cost difference in local and remote message scheduling. In particular, the scheduling mechanism enables the runtime system to implement dynamic load balancing. The compiler uses global data flow analysis to infer types for expressions. The inferred type information is used to optimize message scheduling. The compiler also implements a join continuation transformation to restore concurrency lost in specifications. It compiler analyzes local data flow in a program to enable multiple threads active on an actor with thread safety.

Preliminary experiment results are encouraging. Specifically, the performance of primitive operations, such as actor creation and message sending, are comparable to those of other systems. Although the performance of fine-grained benchmark programs is worse than that of implementations in less flexible systems, our system yields comparable or better performance, on benchmarks with sufficiently large granularity.

The implementation techniques of the runtime system and the compile-time analyses developed in the thesis may serve as a basis on which to implement high-level actor-based systems efficiently. Such systems include: multi-object synchronization and coordination in distributed environment [41], meta-level specification of interaction policies between distributed components [114], synchronization between distributed objects with real-time constraints [105], visualization of coordination patterns in concurrent algorithms [10]. All of these systems are based on asynchronous objects and thus are modeled with actors. Although the systems support different high-level linguistic abstractions, they share a property that the abstractions may be implemented in terms of primitive actor operators.

The work addressed in the thesis may be extended in a number of directions. First, the THAL language need to support more advanced abstractions to further improve programmability and

re-usability. Among the abstractions are inheritance, reflection, and exception handling. These abstractions are necessarily added to the language in ways that maintain efficiency of the language. Furthermore, the high-level, modular abstraction mechanism for actor placement developed in [102] may be incorporated in the language. Lastly, the runtime system's modular design allows an automatic memory management scheme [128, 129] to be easily plugged in. Automatic memory management is necessary for the runtime system to guarantee location transparency and execution security by obviating user-level memory management.

The advent of low-cost off-the-shelf interconnects opens the possibility of networks of workstations as tomorrow's economic workhorses. The compilation techniques and the runtime support are independent of the underlying platforms, and thus, may easily be adapted to such platforms. Intelligent agents on the world wide web (WWW) for data mining and/or distributed processing roam from node to node to achieve their goals. By definition, they require migration capability as well as transparent naming. Inherent location independence of actors and their ability to migrate may be used to implement the intelligent agents efficiently.

Bibliography

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] G. Agha. Supporting Multiparadigm Programming on Actor Architectures. In *Proceedings of Parallel Architectures and Languages Europe, Vol. II: Parallel Languages (PARLE '89)*, pages 1–19. Espirit, Springer-Verlag, 1989. LNCS 366.
- [3] G. Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.
- [4] 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.
- [5] 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.
- [6] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
- [7] J. Randy Allen. "Dependence Analysis for Subscripted Variables and Its Application to Program Transformations. Ph.D. dissertation, Rice University, Dept. Mathematical Sciences, April 1983. (UMI 83-14916).
- [8] 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.
- [9] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *4th International DFVLR Seminar on Foundations of Engineering Sciences*, pages 61–88, 1987. LNCS 295.
- [10] M. Astley and G. Agha. A Visualization Model for Concurrent Systems. *Information Sciences*, 93(1–2):107–132, August 1996.
- [11] W. Athas and C. Seitz. Cantor User Report Version 2.0. Technical Report 5232:TR:86, California Institute of Technology, Pasadena, CA, January 1987.
- [12] W. Athas and C. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, pages 9–23, August 1988.
- [13] G. Attardi. Concurrent Strategy Execution in Omega. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 259–276. MIT Press, Cambridge, MA, 1987.
- [14] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Press, 1988.
- [15] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic Program Parallelization. *Proceedings of IEEE*, 81(2):211–243, February 1993.
- [16] B. M. Barry, J. Altoft, D. A. Thomas, and M. Wilson. Using Objects to Design and Build Radar ESM Systems. In *Proceedings of OOPSLA '87*. SIGPLAN, ACM, 1987.

- [17] F. Baude and G. Vidal-Naquet. Actors as a Parallel Programming Model. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science*, pages 184–195, 1991. LNCS 480.
- [18] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. International Series on Computer Science. Prentice Hall, 1990.
- [19] 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.
- [20] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Supercomputing '93*, pages 588–597, 1993.
- [21] Jean-Pierre Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In S. Cook, editor, *European Conference on Object-Oriented Programming (ECOOP'89)*, pages 109–129, Nottingham, U.K., July 1989. Espirit, Cambridge University Press, United-Kingdom. British Computer Society Workshop Series.
- [22] Kim B. Bruce. Safe Type Checking in a Statically-Typed Object-Oriented Programming Language. In *Twentieth Symposium on Principles of Programming Languages*, pages 285–298. ACM Press, January 1993.
- [23] P. A. Buhr, G. Ditchfield, R. A. Strooboscher, B. M. Younger, and C. R. Zaranke. $\mu C++$: Concurrency in the Object-Oriented Language C++. *Software - Practice and Experience*, 22(2):137–172, February 1992.
- [24] A. Burns. *Concurrent Programming in Ada*. Cambridge University Press, 1985.
- [25] C. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.
- [26] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.
- [27] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):13–26, August 1994.
- [28] 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.
- [29] A. A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. MIT Press, 1993.
- [30] W. Clinger. Foundations of Actor Semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, May 1981.
- [31] Convex Computer Corporation, Richardson, Texas. *Convex Exemplar Architecture*, November 1993.
- [32] Cray Research, Inc. *Cray T3D System Architecture Overview*, March 1993.
- [33] 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.
- [34] 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.
- [35] O.-J. Dahl and K. Nygaard. SIMULA 67 Common Base Proposal. Technical report, NCC Doc., 1967.
- [36] W. Dally. *A VLSI Architecture for Concurrent Data Structures*. Kluwer Academic Press, 1986.
- [37] W. Dally. *The J-Machine: System Support for Actors*, chapter 16, pages 369–408. M.I.T. Press, Cambridge, Mass., 1990.

- [38] W. J. Dally and A. A. Chien. Object-Oriented Concurrent Programming in CST. In G. Agha, P. Wegner, and A. Yonezawa, editors, *The ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 28–31, San Diego, USA, September 1988. The ACM SIGPLAN, ACM Press.
- [39] J. H. Edmondson, P. Rubinfeld, R. Rreston, and V. Rajagopalan. Superscalar instruction Execution in the 21164 Alpha Microprocessor. *IEEE micro*, 15(2), April 1995.
- [40] 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. LNCS 615.
- [41] Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.
- [42] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.
- [43] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.
- [44] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.
- [45] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Comptuer Company, 1995.
- [46] Justin O. Graver and Ralph E. Johnson. A Type System for Smalltalk. In *Seventh Symposium on Principles of Programming Languages*, pages 136–150. ACM Press, January 1990.
- [47] 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.
- [48] L. Gwennap. 620 Fills Out PowerPC Product Line. *Microprocessor Report*, 8(14), October 1994.
- [49] L. Gwennap. UltraSparc Unleashes SPARC Performance. *Microprocessor Report*, 8(13), October 1994.
- [50] L. Gwennap. P6 Underscores Intel's Lead. *Microprocessor Report*, 9(2), February 1995.
- [51] L. Gwennap. Pentium Is First CPU to Reach 0.35 Micron. *Microprocessor Report*, 9(4), March 1995.
- [52] L. Gwennap. Digital's 21164 Reaches 500 MHz. *Microprocessor Report*, 10(9), July 1996.
- [53] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
- [54] C. Hewitt and R. Atkinson. Specification and Proof Techniques for Serializers. *IEEE Transactions on Software Engineering*, 5(1), January 1979.
- [55] C. Hewitt and H. Baker. Laws for Communicating Parallel Processes. In *IFIP Conference Proceedings*, 1977.
- [56] C. Hewitt and P. de Jong. Analyzing the Roles of Descriptions and Actions in Open Systems. In *Proceedings of the national Conference on Artificial Intelligence*. AAAI, August 1983.
- [57] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series on Software Engineering. McGraw-Hill, 1995.
- [58] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [59] J. G. Holm, A. Lain, and P. Banerjee. Compilation of Scientific Programs into Multithreaded and Message Driven Computation. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 518–525, Knoxville, TN, May 1994.
- [60] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proceedings of Hot Interconnects*, August 1993.

- [61] W. Horwat. Concurrent Smalltalk on the Message Driven Processor. Master's thesis, MIT, May 1989.
- [62] C. Houck. Run-Time System Support for Distributed Actor Programs. Master's thesis, University of Illinois at Urbana-Champaign, January 1992.
- [63] C. Houck and G. Agha. HAL: A High-level Actor Language and Its Distributed Implementation. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.
- [64] N. Hutchinson, R. Raj, A. Black, H. Levy, and E. Jul. The Emerald Programming Language REPORT. Technical Report 87-10-07, University of Washington, October 1987.
- [65] Intel Corporation. *Paragon User's Guide*, 1993.
- [66] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. An Architecture of Highly Parallel Computer AP1000. In *Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 13–16, May 1991.
- [67] J. Rees et. al. The T Manual. Technical report, Yale University, 1985.
- [68] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501–538, 1985.
- [69] K. Mani Chandy and Carl Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In G. Agha and P. Wegner and A. Yonezawa, editor, *Research Direction in Concurrent Object-Oriented Programming*, chapter 11, pages 281–313. The MIT press, 1993.
- [70] D. Kafura, M. Mukherji, and G. Lavender. ACT++: A Class Library for Concurrent Programming in C++ Using Actors. *Journal of Object-Oriented Programming*, 0(0):47–62, October 1993.
- [71] D. Kahaner, C. Moler, and S. Nash. *Numerical Methods and Software*. Prentice Hall, 1989.
- [72] K. Kahn. *Creation of Computer Animation from Story Descriptions*. PhD thesis, Massachusetts institute of Technology, 1979.
- [73] 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).
- [74] V. Karamcheti. Private Communication, 1994.
- [75] 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.
- [76] 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.
- [77] R. Kessler and J. Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *Proceedings of COMPCON*, pages 176–182, 1993.
- [78] 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.
- [79] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Proceedings of Supercomputing '95*, 1995.
- [80] W. Kim, R. Panwar, and G. Agha. Efficient Compilation of Call/Return Communication for Actor-Based Programming Languages. In *Proceedings of HiPC '96*, pages 62–67, 1996.
- [81] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architectural Support for Single Address Space Operating Systems. In *Proceedings of ASPLOS V '92*, pages 175–186, 1992.
- [82] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Inc., 1994.

- [83] 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.
- [84] T. T. Kwan, B. K. Totty, and D. A. Reed. Communication and Computation Performance of the CM-5. In *Proceedings of Supercomputing '93*, pages 192–201, 1993.
- [85] G. Lapalme and P. Sallé. Plasm-II: an Actor Approach to Concurrent Programming. In G. Agha, P. Wegner, and A. Yonezawa, editors, *The ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 81–83, San Diego, USA, September 1988. The ACM SIGPLAN, ACM Press.
- [86] J. Larus. C*: A Large-Grain, Object-Oriented, Data Parallel Programming Language. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 326–340. Springer-Verlag, 1993. LNCS 757.
- [87] J. K. Lee and D. Gannon. Object-Oriented Parallel Programming Experiments and Results. In *Proceedings Supercomputing 91*, pages 273–282, 1991.
- [88] H. Lieberman. Concurrent Object-Oriented Programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, chapter 16, pages 9–36. MIT Press, Cambridge, MA, 1987.
- [89] C. Manning. ACORE: The Design of a Core Actor Language and its Compiler. Master's thesis, MIT, Artificial Intelligence Laboratory, August 1987.
- [90] S. Matsuoka, K. Taura, and A. Yonezawa. Highly Efficiency and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages. In *ACM OOPSLA '93*, 1993.
- [91] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1993.
- [92] 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.
- [93] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
- [94] MIPS Technologies, Inc. *Product Overview: R10000 Microprocessor*, October 1994.
- [95] S. Murer, J.A. Feldman, C.-C. Lim, and M.-M. Seidel. pSather: Layered Extensions to an Object-Oriented Language for Efficient parallel Computation. Technical Report TR-93-028, ISCI, December 1993.
- [96] T. Nakajima, Y. Yokote, M. Tokoro, S. Ochiai, and T. Nagamatsu. DistributedConcurrentSmalltalk: A Language and System for the Interpersonal Environment. In G. Agha, P. Wegner, and A. Yonezawa, editors, *The ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 43–45, San Diego, USA, September 1988. The ACM SIGPLAN, ACM Press.
- [97] A Hierarchical $O(n \log n)$ Force-Calculation Algorithm. J. Barnes and P. Hut. *Nature*, 324(4):446–449, 1986.
- [98] Open Systems Lab. University of Illinois at Urbana-Champaign. *THAL Programmer's Manual. Version 1.0*, May 1997.
- [99] N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making Type Inference Practical. In *Proc. ECOOP'92*, pages 329–349. Springer-Verlag (LNCS 615), 1992.
- [100] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proc. OOPSLA '91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161. ACM Press, October 1991.

- [101] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
- [102] R. Panwar. *Specification of Resource Management Strategies for Concurrent Objects*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
- [103] R. Panwar, W. Kim, and G. Agha. Parallel Implementations of Irregular Problems using High-level Actor Language. In *Proceedings of IPPS '96*, 1996.
- [104] J. Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, August 1996.
- [105] S. Ren. *Modularization of Time Constraint Specifications in Distributed Real-Time Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
- [106] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of IPPS '95*, 1995.
- [107] R. Sethi. *Programming Languages. Concept & Construct*. International Series on Software Engineering. Addison Wesley, second edition, 1996.
- [108] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An Object-Oriented Operating System – Assessment and Perspectives. *Computing Systems*, 2(4):287–337, Fall 1989.
- [109] O. Shivers. Data-flow Analysis and Type Recovery in Scheme. In P. Lee, editor, *Topics in Advance Language Implementation*, pages 47–87. M.I.T. Press, 1991.
- [110] Silicon Graphics. *POWER CHALLENGEarray Technical Report*, 1996.
- [111] J. E. Smith and S. Weiss. PowerPC 601 and Alpha 21064: A Tale of Two RISCs. *IEEE Micro*, 14(3), June 1994.
- [112] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.
- [113] L. Steels. An Applicative View of Object Oriented Programming. AI Memo 15, Schlumberger-Doll Research, March 1982.
- [114] D. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.
- [115] Norihisa Suzuki. Inferring Types in Smalltalk. In *Eighth Symposium on Principles of Programming Languages*, pages 187–199. ACM Press, January 1981.
- [116] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.
- [117] K. Taura. Design and Implementation of Concurrent Object-Oriented Programming Languages on Stock Multicomputers. Master's thesis, The University of Tokyo, February 1994.
- [118] K. Taura. Private Communication, 1995.
- [119] 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.
- [120] 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.
- [121] D. G. Theriault. Issues in the Design and Implementation of ACT2. Technical Report AI-TR-728, MIT Artificial Intelligence Laboratory, June 1983.

- [122] Thinking Machine Corporation. *Connection Machine CM-5 Technical Summary*, revised edition edition, November 1992.
- [123] Thinking Machine Corporation. *CMMD Reference Manual Version 3.0*, May 1993.
- [124] D. A. Thomas, W. R. LaLonde, J. Duimovich, and M. Wilson. Actra - A Multitasking/Multiprocessing Smalltalk. In G. Agha, P. Wegner, and A. Yonezawa, editors, *The ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 87–90, San Diego, USA, September 1988. The ACM SIGPLAN, ACM Press.
- [125] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The Extensible Services Switch in Carnot. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2), May 1993.
- [126] 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.
- [127] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *OOPSLA Proceedings*, 1989.
- [128] N. Venkatasubramanian, G. Agha, and C. Talcott. Scalable Distributed Garbage Collection for Systems of Active Objects. In *Proceedings of the International Workshop on Memory Management*, pages 441–451, St. Malo, France, September 1992. ACM SIGPLAN and INRIA, Springer-Verlag. Lecture Notes in Computer Science.
- [129] N. Venkatasubramanian and C. Talcott. A MetaArchitecture for Distributed Resource Management. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE Computer Society Press, January 1993.
- [130] 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.
- [131] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of International Symposium of Computer Architectures*, pages 256–266, 1992.
- [132] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of Fourteenth ACM Symposium on Operating System Principles*, pages 203–216, December 1993.
- [133] T. Watanabe and A. Yonezawa. A Actor-Based Metalevel Architecture for Group-Wide Reflection. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 405–425. Springer-Verlag, 1990. LNCS 489.
- [134] 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.
- [135] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.
- [136] S. E. Zenith and D. May. *Occam 2 Reference Manual*. INMOS Ltd and Prentice-Hall, 1988.

Vita

WooYoung Kim was born on October 23, 1964, in Pusan, Korea. As being a son of a military officer, he moved many times in his elementary and middle school days. He graduated from YoungDong Middle School in February, 1980 and from HwiMoon High School in February, 1983, both in Seoul.

In March, 1983, WooYoung enrolled in the Engineering College of Seoul National University where he majored in Computer Engineering. He was awarded a scholarship from the Chung-O Foundation throughout the four years of the study in the school. As a senior, he won a second place prize with ChaeRyung Park in a student paper contest sponsored by the Korean Information Science Society in 1987. Upon receiving his Bachelor of Science Degree with Magna Cum Laude from the Seoul National University in February 1987, he continued his graduate study in the same department. During his tenure, he served as a Research Assistant in the Programming Language and Artificial Intelligence Laboratory under the direction of Dr. YoungTaek Kim and a Teaching Assistant in the same department. He received his Master of Science Degree in February, 1989.

After serving 6 month-long military duty as a trainee officer, WooYoung enrolled in the Ph.D. program in the Department of Computer Science at the University of Illinois at Urbana-Champaign. During his tenure at Illinois, he served as a Research Assistant in the Open Systems Laboratory under the direction of Dr. Gul Agha and a Teaching Assistant in the same department.