DEVELOPING THE DISTRIBUTED COMPONENT OF A FRAMEWORK FOR PROCESSING INTENSIONAL PROGRAMMING LANGUAGES

BO LU

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

MARCH 2004

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: **Bo Lu**
Entitled: **Developing the Distributed Component of a Framework for Processing Intensional Programming Languages**

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

[illegible] Dr. E. I. Plotkin	Chair
[illegible] Dr. W. Du	External Examiner
[illegible] Dr. F. Khendek	External to Program
[illegible] Dr. T. Fancott	Examiner
[illegible] Dr. G. Butler	Examiner
[illegible] Dr. J. Paquet	Examiner
[illegible] Dr. P. Grogono	Thesis Supervisor

Approved by [illegible]
Dr. T.D. Bui, Graduate Program Director

APR 02 2004
_____ 2004

Dr. N. Esmail, Dean
Faculty of Engineering & Computer Science

Abstract

Developing the Distributed Component of a Framework for Processing Intensional Programming Languages

Bo Lu, Ph.D.
Concordia University, 2004

Based on a simple non-procedural language with temporal logic operators, Lucid underlies a family of multi-dimensional programming languages based on intensional logic. Intension is a concept rooted in an aspect of natural language called "intensional context", in which the *meaning* of a statement (extension) depends on the *context* in which it is uttered (intension). The implicit temporal feature of Lucid makes it suitable for use as a means of describing dynamic systems. In the past, experiments have been performed and real applications have been developed with programs written in Lucid. However, these systems focused mainly on improving the execution performance of one dialect of Lucid and not address the problem of interpreting variants of Lucid. The GIPSY system is designed to not only process current Lucid variants efficiently but also to be modified easily to accept new dialects of Lucid.

In the thesis, we discuss the essence of executing intensional programming languages using the *eduction* (also called *demand-driven* or *lazy*) execution model; describe experiments with different approaches to interpreting programs written in Lucid; and focuses on execution over a network of processors. We describe the implementation of a prototype for executing Lucid programs in a distributed environment. We also explore the advantages of applying the object concept to distributed systems and describe experiments with these methods. In addition, the thesis includes estimates of the impact of integrating computation functions into the Lucid code and proposes an advanced execution model consisting of self-contained and intelligent clients associated with a meta-level resource management.

Acknowledgements

First and foremost, I would like to thank my esteemed supervisor, Dr. Peter Grogono, who not only guided this work with good advice, insightful comments and constant encouragement but also provided continuous support throughout my Ph.D. studies.

I would like to thank all members in the GIPSY team, especially Dr. Joey Paquet, for their support during these years.

Contents

List of Figures viii

List of Tables ix

1 INTRODUCTION 1

1.1 OVERVIEW 1
1.2 CONTRIBUTIONS 3
1.3 OUTLINE 5

2 LUCID 8

2.1 FUNCTIONAL AND DATAFLOW PROGRAMMING 8
2.1.1 Functional Programming 8
2.1.2 Dataflow Programming 10
2.1.3 Comparison 12
2.2 ORIGINAL LUCID 13
2.2.1 Motivation 13
2.2.2 Lucid Programs 14
2.2.3 Advantages 15
2.2.4 Syntax 15
2.3 LUCID EVOLUTION 16
2.4 LUCID IN GIPSY 17
2.4.1 Types and Constants 17
2.4.2 Operators 18
2.4.3 Identifier and Dimension 21
2.4.4 Variable 22
2.4.5 Eduction 22
2.4.6 Program Examples 23

3 GIPSY SYSTEM 25

3.1 GIPSY MODULES 25
3.1.1 GIPSY Architecture 26
3.1.2 GIPC 26
3.1.3 GEE 27
3.1.4 RIPE 28
3.2 INCREMENTAL DEVELOPMENT 28
3.2.1 Pure and Hybrid Language 28
3.2.2 Data Type and Language Elements 29
3.3 GIPC 30

3.3.1 Function Substitution 30
3.3.2 Non-primitive Operator Reduction 31
3.3.3 Rank Analysis 32
3.3.4 Abstract Syntax Tree and Dictionary 33

4 GEE ANALYSIS 35

4.1 PARALLELISM 36
4.1.1 Data Parallelism 36
4.1.2 Execution Parallelism 38
4.2 DISTRIBUTION 38
4.2.1 Concepts 39
4.2.2 Characteristics 39
4.2.3 Distribution, Concurrency and Parallelism 42
4.2.4 Distribution in GEE 43
4.3 OBJECT APPROACH 45
4.4 DEVELOPMENT MANNER 46
4.4.1 Incremental Development 46
4.4.2 Implementation Language 48

5 GEE IMPLEMENTATION 52

5.1 COMPONENTS 52
5.1.1 IDP 53
5.1.2 IVW 53
5.1.3 IEM 54
5.2 INTERPRETATION 54
5.2.1 Interpretation Rules 55
5.2.2 Evaluation Process 57
5.3 IC CLASS 57
5.4 MULTITHREADING 60
5.4.1 Thread 60
5.4.2 IC Runnable Objects 63
5.4.3 Thread Pool 64
5.5 RMI 65
5.5.1 Distributed Technology 66
5.5.2 GEE Architecture 67
5.5.3 GEE Components 71
5.5.4 Thread Usage 75
5.5.5 Object Serialization 75
5.6 VALUE WAREHOUSE 79
5.6.1 Content 80
5.6.2 Efficiency 80
5.6.3 Value Inquiry 83
5.6.4 Maintenance 83
5.7 INTERFACE BETWEEN GEE AND RIPE 84

6 ANALYSIS AND DISCUSSION 85

6.1 TESTING REPORT 85

6.1.1 Tested Application 85
6.1.2 Testing on a Cluster 87
6.1.3 Testing on a Single Machine 90
6.2 RESULT ANALYSIS 93
6.2.1 Interpretation vs. Compilation 94
6.2.2 Shared-memory and Distribution 98
6.2.3 Internet Distributed Computation 100
6.3 EXECUTION PERFORMANCE 104
6.3.1 Evaluation Model 104
6.3.2 Communication Overhead 106
6.3.3 Divisible Load Scheduling 108
6.4 RESOURCE MANAGEMENT 110
6.4.1 Information 111
6.4.2 Reflection 112
6.4.3 Reflective Resource Management 115

7 RELATED WORK 117

7.1 GLU 117
7.2 IE4 119

8 CONCLUSION AND FUTURE WORK 122

8.1 CONCLUSION 122
8.2 FUTURE WORK 124
8.2.1 Computation Functions 124
8.2.2 Architecture 124
8.2.3 Distributed Value Warehouse 125

BIBLIOGRAPHY 127

APPENDIX: HASH FUNCTION 134

List of Figures

FIGURE 2.1 EXAMPLES OF # AND @ OPERATORS 20
FIGURE 2.2 EXAMPLE OF *FBY* OPERATOR 20
FIGURE 2.3 LUCID STATEMENT 23
FIGURE 2.4 PROGRAM EXAMPLE: GENERATING NATURAL NUMBERS 23
FIGURE 2.5 A SIMPLIFIED HAMMING PROBLEM 24
FIGURE 3.1 GIPSY ARCHITECTURE 26
FIGURE 3.2 PROGRAM AFTER FUNCTIONS HAVE BEEN SUBSTITUTED 30
FIGURE 3.3 REDUCTION OF NON-PRIMITIVE OPERATORS 31
FIGURE 3.4 PROGRAM WITH ONLY PRIMITIVE OPERATORS 31
FIGURE 3.5 RULES FOR RANK ANALYSIS 32
FIGURE 3.6 ABSTRACT SYNTAX TREE 33
FIGURE 4.1 PIPELINE PARALLELISM 36
FIGURE 4.2 DATA PARALLELISM 37
FIGURE 4.3 PART OF THE AST IN FIGURE 3.6 37
FIGURE 5.1 GEE ARCHITECTURE 52
FIGURE 5.2 PROGRAM WRITTEN IN LUCID (THE SAME AS IN FIGURE 3.4) 55
FIGURE 5.3 *IC* CLASS DIAGRAM 58
FIGURE 5.4 CODE GENERATOR 59
FIGURE 5.5 THREAD LIFECYCLE 61
FIGURE 5.6 PSEUDO CODE OF RETRIVING VALUES 63
FIGURE 5.7 EXECUTION ARCHITECTURE 68
FIGURE 5.8 CLASS DIAGRAM OF GEE 71
FIGURE 5.9 PSEUDO CODE OF TAKIGN DEMANDS 72
FIGURE 5.10 PESUDO CODE OF METHOD CAL() 72
FIGURE 6.1 TEMPERATURE DISTRIBUTION FOR A BAR (ONE DIMENSION) 86
FIGURE 6.2 TEMPERATURE DISTRIBUTION FOR A BAR (TWO DIMENSIONS) 86
FIGURE 6.3 SYSTEM CONFIGURATION OF THE CLUSTER 87
FIGURE 6.4 TEST RESULTS USING 1-5 PROCESSORS 88
FIGURE 6.5 RELATIVE PERFORMANCE USING 1-5 PROCESSORS 89
FIGURE 6.6 TEST RESULTS USING 1 AND 2 PROCESSORS 90
FIGURE 6.7 TEST RESULTS OF USING INTERPRETATION 91
FIGURE 6.8 TEST RESULTS OF USING MULTI-THREADING WITH A THREAD POOL 91
FIGURE 6.9 TEST RESULTS OF USING DISTRIBUTION 92
FIGURE 7.1 PARALLEL PROGRAM COMPILATION IN GLU 118
FIGURE 7.2 OVERVIEW OF L4 INTENSIONAL ENGINE 120

List of Tables

TABLE 2.1 STANDARD OPERATORS 19
TABLE 5.1 RELATIONS BETWEEN A NAME AND NUMBER 73

1 Introduction

This chapter introduces the background of the distributed component named GEE that we developed, and highlights the contribution of the thesis and describes its structure.

1.1 Overview

This thesis describes the development of a distributed component called GEE (General Eduction Engine) which is the backend of GIPSY (General Intensional Programming SYstem) that handles an intensional programming language called Lucid.

The original Lucid was a functional dataflow programming language, in which basic data structures were streams of simple values. It used equations to define a network of processors and communication lines, through which the data flowed.

Based on a simple non-procedural language with temporal logic operators, Lucid has been evolved to a multi-dimensional programming language based on intensional logic. Intension is a concept rooted in one aspect of natural languages called "intensional context", in which the *meaning* of a statement (extension) depends on the *context* (intension) in which it is uttered. Lucid appears to be appropriate as a language for reactive programming [Kie1998], software configuration [PW1993], program transformations and formal verification [Raj1995] and so on. Moreover, its implicit temporal feature makes it suitable for use as a means of describing dynamic systems. Finally, it provides a new perspective of a more natural way of understanding problems that have an intensional nature. The advantages of this intensional programming language make it worthwhile for both theoretical research and practical usage. For all of these applications, designing and implementing a system that interprets the language should be fulfilled first to prove the practical applicability of the language.

Experiments or real applications have been performed for executing Lucid programs and exploring the advantages of this language in solving practical problems. Examples include: 1) a method of converting a Lucid program to equivalent C++ source code [Du1994], 2) a hybrid language mixed with Lucid and C to form a complete application of Lucid which is executed in parallel as introduced in GLU [JDA1997], and 3) an eduction engine implemented on the L4 kernel [Jay1999]. However, none of these systems address the problem of interpreting variant Lucids seriously. These variants (e.g. pLucid, Lambda Lucid [WA1985], IndexicalLucid [FJ1991]), which form a Lucid family, were derived from the use of Lucid for different purposes. Moreover, the language has the potential to have more dialects adopting different user-defined intensional operators associated along with its wider usage. Therefore, the adaptability to future changes should be an important factor being considered when designing the language's compiler system.

The GIPSY system [PK2000] designs a framework for processing semantically equivalent variants of the Lucid programming language and addresses the issue of flexibility that was lacking in both GLU and IE4.

The design goals of GIPSY are generality, flexibility and efficiency. GIPSY is designed to process current Lucid variants efficiently, and can be modified easily to future changes. In addition, a friendly interface is required as an integrated component in this system. Similar to GLU, the system adopts Java programming language to perform the concrete computations and uses Lucid to declare the structure information of the relationship between the computations. However, programs written purely in Lucid will be processed at the first stage with future extension kept in minimal.

The workflow of processing Lucid source code can be divided into two major steps: the front-end and the backend. The front-end takes original codes, does syntax checking and semantic checking, and the backend operates on the abstract syntax tree generated by the front-end and computes the desired values.

The GEE implemented in the thesis is the back-end of the GIPSY system, which is object-oriented, adopts an eduction execution model (also called demand-driven or lazy evaluation) to execute Lucid programs over a network of processors.

The GIPSY project was funded by Concordia University and the National Science and Engineering Research Council of Canada and its implementation is currently carried out at Concordia University.

1.2 Contributions

This thesis investigates the essence of executing intensional programming languages using a demand-driven model, designs an execution schema, and implements the execution module by applying Java distribution technologies. Moreover, it explores the advantages of applying object concept on distributed systems and describes experiments with the methods. Finally, it proposes to scale up the distribution to a wider Internet scope.

First, the thesis describes three techniques for interpreting programs written in Lucid: interpretation, multi-threading and distribution. The interpretation is implemented as a first step. It gives insight into more efficient implementations. The multi-threading and distribution solutions are object-oriented and the object concept has different levels. For example, General Eduction Engine (GEE) itself is a module of GIPSY system. If a new version of Lucid is created, only the part that converts graphic expressions into Lucid source code needs to be modified. This assumes that the extension is conservative — that is, no new semantics are required. Also, GEE can be decomposed into the demand propagator and value warehouse. And most importantly, each identifier, which can implement the computation of a variable, is encapsulated in a class. Based on this, several closely related computation units can be combined together to form a bigger unit. In this way, GEE achieves the desired flexibility.

Secondly, the thesis lightens the performance penalty carried by a demand-driven evaluation model by distributing the execution over a network of processors, being equipped with a value cache and providing several optimization methods. The GEE component has to adopt the demand-driven evaluation model to avoid unnecessary or even endless computations. However, the disadvantages are the need for propagating demands and more work is forced to start at runtime. The thesis explores the contribution of dividing and dispensing the whole work on a network of machines. It also embodies a value cache to store computed values that will not need to be recomputed when demanded later on. In addition, it illustrates tasks that can be performed at compile time which can either reduce runtime work or guide the execution in a more organized and efficient manner. Forcing more work to be done at compile time and distributing the tasks wisely help to meet the requirement for efficiency.

Thirdly, the thesis implements an extensible architecture that is suited to the current requirements, easily manageable, and adaptable to future changes. The client/server architecture is adopted with which the server holds resources such as computed value and generated demands and the clients do computations upon the demands retrieved from the server and return results (new demands/values) back to the server. The architecture is extensible in the sense that multiple clients can be added without changing the system. More importantly, the architecture can also be modified with a reasonable effort to be scaled up to a world wide computing environment. Finally, the architecture is client-driven in such a way that each client can propagate demands. This approach avoids the performance bottleneck caused by a central demand generation implemented in GLU and is new for Lucid. In a word, the architecture is good for current implementation and appropriate for a future upgrade.

As a strong complement, the thesis includes estimates of the impact of integrating computation functions and proposes a model of an advanced distribution of the execution consisting of self-contained and intelligent clients associated with a meta-level resource management. In this proposal, each client is able to sense the changes in its environment and adapt it to the changes through object migration. The resource management is performed at a level above the current application so that application developers do not need to worry about the variety of hardware system. Moreover, the application execution can adapt to its environmental changes intelligently.

1.3 Outline

The structure of the thesis is shown below:

Chapter 2 introduces Lucid programming languages as the background of the GEE distribution component, including Lucid programming language. The chapter begins by introducing dataflow and functional programming; shows what Lucid programming language looks like and its evolution; and addresses the problem of Lucid dialects emerging during its development.

Chapter 3 introduces the GIPSY system and its front-end components. It begins by introducing the structure of GIPSY; then describes its design goals. Finally, it presents the major tasks performed by the front-end of the system, which includes non-primitive intensional operators elimination and function reduction. The enclosing of the front-end part illustrates a complete process of interpreting the programming language from the initial source code to the computation of the values.

Chapter 4 analyzes the aspects of the GEE component: parallelism, distribution, object concepts and implementation strategy. The programming language describes data parallelism implicitly which makes it suitable to be executed in a distributed system. The adoption of object concepts makes it convenient to form work pieces of heavier workload, consequently reducing the network traffic. At the end, the chapter explains the rationale of using Java as an implementation language and adopting an incremental development method.

Chapter 5 gives the details of implementation of the GEE component. The component is implemented with three different approaches, including interpretation, multi-threading and distributing. The interpretation approach focuses on the rules of evaluating a Lucid program. The multithreading approach is implemented in two ways: using "wait and notify" and using a thread pool. The distribution approach adopts Java Remote Method Invocation technology.

Chapter 6 discusses the important issues related to the distribution execution of intensional programming languages, which includes result analysis, resource management and performance improvement. The chapter concludes that the interpretation is suitable for understanding how a Lucid program is evaluated; the multi-threading fits the current fine-granulated programs written in Lucid language only; and the distribution is appropriate for a program whose workload is demanded to be divided into pieces each of which takes a long time to work out. The chapter also proposes to manage the resource at a meta-level so that its interference to the application is reduced to be a minimum. Finally, the chapter discusses possible solutions to improve the system's overall performance.

Chapter 7 introduces two tasks performed for intensional language: GLU and IE4.

Chapter 8 concludes the thesis and also describes future work.

2 Lucid

The chapter starts by introducing dataflow programming and functional programming and their implementations. Then, it introduces the Lucid programming language, including the original Lucid, its evolution and appearance in GIPSY.

2.1 Functional and Dataflow Programming

Lucid was originally a dataflow programming language [WA1985]. Its execution adopted a lazy evaluation execution model, which is quite commonly used in functional programming languages. This section introduces dataflow programming, functional programming, and their implementations.

2.1.1 Functional Programming

A quite natural programming methodology that matches the execution process of most computer hardware is imperative programming (example languages are Pascal and C). Following closely the "fetch instruction/update memory" cycle of typical computer architectures, imperative languages emphasize a programming style in which programs execute commands sequentially, use variables to organize memory, and update variables with assignment statements.

The match with the von Neumann computer architecture causes several flaws. One of them is the sequential execution. The languages do not support the simultaneous execution of different parts of a program because each command may depend on variables that could be changed by previous commands. Execution speed is therefore limited by the speed with which individual instructions can be executed. Another is that one must know values of all the variables in order to know the state of a computation.

The design of functional programming is based on the concept of mathematical functions, which are often defined by being separated into various cases, each of which is defined by appealing to function applications. Rather than having variables and assignment statements, functional programming [Hen1980] embodies identifiers that acquire values through parameter binding and the result of one function is immediately passed as a parameter to another function. Consequently, variables (in the sense of values that change during execution) become unnecessary. Rather than using iterative constructs, functional programming adopts recursive functions.

Functional programming takes another large step towards a higher-level programming model. Moreover, it is amenable to formal analysis and manipulation since it is based on mathematical concepts. It is also used for parallel computations due to the fact that functions without side effects can be computed in arbitrary order so that a programmer is freed of a number of problems connected to the synchronization of computations.

A functional program is easier to design, write and maintain and is usually more concise than its imperative counterpart. The disadvantage is that the interpretation task becomes more difficult and the efficiency is harder to achieve.

The implementations of Haskell [Hask] can be used to illustrate the implementation of functional languages.

Haskell is a modern, standard, non-strict, purely functional programming language based on *lambda calculus*. There are a number of implementations available, three popular of which are Hugs, GHC and nbc98, all of which support multiple platforms.

Hugs is a small, portable Haskell interpreter written in C runs on almost any machine. It boasts extremely fast compilation, supports incremental compilation, and has the convenience of an interactive interpreter (within which one can move from module to module to test different portions of a program). However, being an interpreter, it does not nearly match the run-time performance of, for example, GHC and nhc98.

GHC consists of a single program what is run with different options to provide either the interactive Haskell interpreter (also known as GHC_i) or the batch compiler.

The nbc98 is a small, easy to install, standards-compliant Haskell 98 compiler. It provides some advanced kinds of heap profiles not found in any other Haskell compiler. It produces medium-fast code, and compilation is itself quite fast. The compiler stresses the saving of space, which is, it produces small programs requiring comparatively little space at runtime (and it is itself much smaller than the other compilers).

The efficiency of program execution can be considerably improved by program transformations or modifications of the execution model using information gained by the static analysis of programs.

2.1.2 Dataflow Programming

As another alternative to the von-Neumann architecture, the dataflow approach was proposed by Karp and Miller [KM1966]. A dataflow program can be represented by a dataflow graph, which represents the ordering of evaluation imposed by data dependencies. Unfortunately, the term "dataflow diagram" is used in a somewhat different sense in system design. In this thesis, "dataflow" always refer to dataflow programming without side-effects.

Dataflow programming imposes the constraint that an expression cannot be evaluated before its operands are evaluated. Rather than having a single thread of control that moves from one instruction to another, demanding data, operating on it and producing new data, the structure has data that flows to instructions, causing evaluation to occur as soon as all operands are available. Therefore, dataflow can achieve concurrency, particularly at the fine-grain level: it finds multiple operations that can be undertaken concurrently within the evaluation of a single expression.

Dataflow computing never became popular. Very few dataflow computers were ever built, and interest in this field has mostly subsided.

Scientific applications often require similar computations across very large data sets, which may represent a connected physical entity. This style of computing is called data-parallel computing with coarse-grain parallelism. That is, the machines work in parallel on different parts of the data, and they only coordinate their activities on occasion.

An example of a language for data-parallel programming is called Canopy [KM1966]. Data in Canopy are represented as records stored on a grid of sites. Grids are dynamically constructed by calls to a runtime routine. Definition routines are provided for many standard topologies, such as three-dimensional meshes, and a programmer may define any desired topology using a more primitive routine. A computation may use several different grids, although using more than one is unusual. Each site in a grid has coordinates and is connected to its neighbors by links. Data records are associated with each site and each link.

2.1.3 Comparison

Dataflow and functional programming have some similarities. Viewing a computation as a dataflow graph leads directly to a functional view of programming. The same as the functional approach, dataflow does not include the notion of variables because there are no named memory cells holding values. In both approaches, computation does not produce side effects.

On the other side, there are some important differences between dataflow and functional programming [KM1966].

First, dataflow graphs have no simple concept of a function that returns a result. One could surround part of a dataflow graph with a boundary and call it a function, where all inbound arcs to the region would be the parameters and all outgoing arcs would be the results. Such an organization could lead to recursively defined dataflow graphs.

Second, functional programming languages rely heavily on recursion, because they do not support iteration. However, dataflows support recursion by building cyclic graphs (equivalent to iteration); an initial token may be placed inside a cycle in order to allow the first iteration to proceed.

Third, the values placed on arcs are all simple values. It is easy to understand integer, real and even Boolean values moving along the arcs, but values of structured types such as records and arrays might take more underlying machinery. Pointer types are most likely out of the question. Most important, function types, which are so useful in functional programming, are not supported by the dataflow graph formalism. Furthermore, functional programming languages allow various evaluation orders. Dataflow evaluators are typically speculative evaluators, since they are data-driven. Computations are triggered not by a demand for values or data, but rather by their availability.

2.2 Original Lucid

The Lucid (original Lucid) programming language [WA1985] was a dataflow programming language designed to experiment with non-vonNeumann programming models by Ashcroft and Wadge at the University of Waterloo in Canada in 1974.

Lucid has evolved to a multidimensional intensional programming language from originally a dataflow programming language. The nature of Lucid is such that it supports parallelism in a straightforward way.

2.2.1 Motivation

The motivation in developing Lucid was to show that conventional programming could be done in a purely declarative language, which has no assignment or *goto* statements. It was based on the hypothesis that real “structured programming“ requires first the elimination of both these features and that only then would program verification become a practical possibility.

The design of Lucid is based on the computation principle of processing the data while it is in motion, “flowing” through a dataflow network. In this model, a dataflow network is built, which is a system of *nodes*, also called filters, connected by a number of communication channels or *arcs*. A filter is not required to produce output at the same rate at which it is consumed but is required to be functional, which implies that the entire history of its output is determined by the entire history of its input. The great advantage of the dataflow approach is that filters interact in a very simple way, through the pipelines connecting them and the internal operation of one filter cannot affect that of another.

2.2.2 Lucid Programs

A program in Lucid is simply an expression, together with definitions of the transformations and data (other than input) referred to in the expression. The transformations are also called filters, which are components through which the data pass and by which they are transformed. The output of the program is the data denoted by the program as an expression.

Expressions in Lucid are really expressions in the mathematical sense and the filters referred in one expression really are functions. In mathematics, the value of a function returns is determined only by the value given to the function, i.e., by the function's argument, and a mathematical function doesn't have a memory. In order to remember values have been computed, conventional programming assigns them to variables. However, in Lucid, the data transformed by a filter is a series of numbers rather than just a single number. Thus, the apparent contradiction between the static nature of conventional mathematics and the dynamic nature of programming is resolved.

2.2.3 Advantages

A Lucid program could be regarded as basically a textual form of a dataflow graph, although multi-dimensional Lucid programs lead to very complex dataflow graphs. A Lucid programmer specifies exactly the meaning of a program by suggesting or indicating the computations to be performed. Therefore, the programmer has the benefits of operational thinking but does not need to worry about details such as in which particular order to perform operations, rates of flow of data, capacities of buffers, protocols and so on. The implicit approach gives the implementations more freedom as well. Also, Lucid allows a programmer to avoid specifying unnecessary sequencing; this gives much greater scope for carrying computations out in parallel. Furthermore, a programmer is not totally restricted to dataflow (and even less to a single dataflow paradigm) but can think in terms of other operational concepts as well. In particular, a programmer can think of some variables as denoting stored (remembered) values that are repeatedly modified in the course of the computation.

2.2.4 Syntax

Besides the fundamentally different semantics (by defining filters that act on time-varying data streams) from a language like C or Lisp, the syntax of Lucid is deliberately designed to be unusual and different in order to prevent programmers from applying procedural-programming habits. The original Lucid supported a very small set of data types: integer, real and symbol. In addition, Lucid employed some techniques from functional programming such as lazy evaluation and lack of side effects. Details of (a version of) Lucid syntax will be illustrated in Section 2.4.

2.3 Lucid Evolution

During the 1980s and 1990s, Lucid evolved from a temporal dataflow language to a multidimensional language based on intensional logic. Moreover, Lucid spawned several additional research areas, two of the most prominent of which are multidimensional programming [Pla2000] and intensional programming [GR1996]. Both of these are currently active research areas in computer science., although the research communities are quite small.

2.3.1 Intensional Logic

Intensional logic is derived from natural languages. In natural languages, the meaning of a statement whose truth value often depends on the context (*intension*) in which it's uttered; the actual truth-value of the statement in the context is its *extension* [Paq1999]. The most commonly used context is *time*. An example statement is "The Prime Minister is Jean Chretien". The statement is evaluated to be either true or false (extension) in a given place (the country of Canada) and time (2002); and the intension of this statement consists of the world and time.

For our purposes, the context can be viewed as a point in a space with multiple dimensions (such as time, place, etc). Suppose we have two dimensions: time ($t=0,1,2,...$) and position ($x=0,1,2,...$). In an imperative language, we might write $f[i,j]$ to denote the value of the variable f at time i and position j. In Lucid, we would write just f but we would evaluate it in the context $[t=i,\ x=j]$. Therefore, the multi-dimensional concept was introduced as a way of representing intensional logic. A variable is extended with a tag consisting of dimension names and values. The tag defines a *context* for the variable.

Intensional programming can be applied in a wide range of areas, including parallel programming, dataflow computation, temporal reasoning, scientific computation, real-time programming, temporal and multidimensional databases, spreadsheets, attribute grammars, and Internet programming.

The use of implicit contexts makes a Lucid program concise, powerful, and close to natural expression but it may also make it difficult to understand and interpret.

2.4 Lucid in GIPSY

Lucid has many variants depending on the basic set of types and data operations. The GIPSY system is intended to support all the existing Lucid variants, and is designed to be able being modified easily to adapt to the future evolution of Lucid. Fortunately, no matter what variant the front-end compiler processes, the back-end should always deal with the same format. In this section, a general variant of Lucid, IndexicalLucid [FJ1991], is used to represent the basic ideas of Lucid.

2.4.1 Types and Constants

Lucid supports these data types: integer, double, and string.

Lucid has no explicit declaration for types of values. Values types are implicit and they are determined at compile time using type analysis of the associated expression. For example:

x = if #.d == 0 then 1.0 else x+1 @.d (#.d-1);

The *x* has a data type of *double* and the explanation is as the followings.

From the Lucid definition, we obtain three type equations:

type (x) = type (1.0) *(1)*

type (x) = type (x+1) *(2)*

type (x) = type (1) *(3)*

From (1), we infer that the type of *x* must be the same as the type of *1.0*, that is, *double*. Since the operands of + must have the same type, we obtain (3) from (2). In order to make (1) and (3) consistent, we must promote the *int* value *1* to the *double* value *1.0*. Thus type analysis yields the results that *x* has type *double* and that *1* should be replaced by *1.0* (We may also note that the replacement can be done by the compiler — a run-time conversion is not needed in this case.)

Lucid provides numeric constants. Instead of being a simple value in conventional languages, a constant is an infinite sequence of values where each value is the same and the value stays the same in the whole context space specified in the Lucid program. For example, constant 3.14 presents the constant sequence <3.14, 3.14, 3.14, ...> in a one-dimensional context.

2.4.2 Operators

Lucid has intensional operators to facilitate the navigation in the context space in addition to the standard operators and if-then-else operators.

Standard Operators

As well as many other languages, Lucid has the standard arithmetic, relational, and logical operators, as shown in Table 2.1. Unlike in conventional programming languages, all these operators can be used with sequences, working term by term. For example, let x = <1,2,3,...> and y = <2,2,2,...>, then x + y = <3,4,5,...>.

Arithmetic Operators	addition	+
	substraction	-
	multiplication	*
	division	/
	modular	%
	negation	-
Relational Operators	greater than	>
	greater than or equal to	>=
	equal	==
	less than	<
	less than or equal to	<=
	not Equal	!=
Logical Operators	and	&&
	or	||

TABLE 2.1 STANDARD OPERATORS

if-then-else Operators

The if-then-else operator takes 3 operands: one Boolean sequence and two base sequences. It returns a new sequence by using each value of the Boolean sequence to select the corresponding value from the first sequence if the Boolean value is true and the corresponding value from the second sequence if it is false.

Intensional Operators

Lucid has two primitive intensional operators: @ is used for intensional navigation; and # is used for querying the concurrent context of the evaluation.

The operator # takes one operand, the name of a dimension, and returns the value of the given required dimension in the current context. This operator allows a programmer to query *part* of the evaluation context rather than the entire context.

The operator @ takes two operands and returns a stream whose values correspond to the first stream indexed by the value of the second stream at each point. Therefore, this operator allows one to change part of the evaluation context.

```
A            =    1  3  5  7  9  11...
B            =    1  2  3  0  1  2 ...
A @.time B   =    3  5  7  1  3  5 ...
#.time       =    0  1  2  3  4  5 ...
```

FIGURE 2.1 EXAMPLES OF # AND @ OPERATORS

Variants of Lucid define other intensional operators over the primitive operators or allow users to define theirs.

The simplest non-primitive operators are *first* and *next*. The operator *first* returns a stream where every element is the first value of its operand sequence. The operator *next* returns a sequence that is identical to the operand sequence except the first element has been removed. The operators *first* and *next* describe the head and tail of a sequence.

Some other frequent-used ones are: *fby*, *wvr*, *upon*, and *asa*, whereas *fby* stands for "follow by"; *wvr* stands for "whenever"; and *asa* stands for "as soon as". Each operator presents a way to navigate in the context space. For example, the *fby* takes two sequences and returns a sequence whose first element is the first element of the first sequence and whose following elements are elements in the second sequence.

```
A             =    1  3  5  7  9  11...
B             =    1  2  3  0  1  2 ...
A fby.time B  =    1  1  2  3  0  1 ...
```

FIGURE 2.2 EXAMPLE OF *fby* OPERATOR

When non-primitive alternatives are used, a Lucid program becomes more concise and close to natural expression. All of the extended intensional operators can be converted to equivalent forms (see Figure 3.3) that use the primitive operators only. For example, the natural number generation program can be written as:

$x = 1\ \mathit{fby}\ x+1$

and $x\ \mathit{fby.d}\ y$ is equivalent to *if #.d == 0 then x else y @.d (#.d-1).* The fact that programs written in Lucid are concise is mainly because of the adoption of these intensional operators that hide the context by making it implicit. It also cause Lucid has some dialects.

2.4.3 Identifier and Dimension

Variables are used in Lucid as they are in mathematics: to give names to expressions.

The left-hand-side of an equation is a variable (in the Lucid sense), whereas the right-hand-side is a term which can be a constant, variable, operation applied to other terms, or a user-defined function whose arguments are other terms. A Lucid program is a structured set of equations where each equation defines a variable. "Variable" here does not have the same meaning as "variable" in imperative languages, but usually denotes a static sequence of values, so from now on we will use the word "identifier" rather than the word "variable".

Associated with each set of equations are one or more user-defined dimensions. These dimensions define a multidimensional context space in which each identifier denotes a scalar value at each point in the space. In principle, a Lucid identifier *T*, with dimensions *time*, *altitude*, and *longitude* could represent the temperature at any point of the earth's surface at any time. A Lucid program computes values of specific identifiers at specific points in the nested multidimensional space that it defines. Computations of each desired value requires the computation of other values at various points as specified by the equations of the Lucid program. GIPSY requires a dimension has values of integer only.

2.4.4 Variable

A Lucid program usually evaluates an identifier at a given context. In case that an identifier is required, we can convert the problem to be evaluating the identifier at each point of the context space. In order to work out the value, values of some other identifiers at other given contexts would be demanded. Thus, the term “an identifier in a certain context” will be frequently referred in the paper. Therefore, we will use the word “variable” to present “an identifier in a certain context”.

2.4.5 Eduction

GIPSY embodies the *eduction* computation model; this is a “lazy evaluation” strategy. In this model, variables are computed only when needed, so there are no superfluous computations. Eduction is also called “call-by-need”, or “demand-driven”, and it differs from “call-by-value” in which arguments are evaluated before they are passed to a function even though the function may not need them. Conceptually, variables in Lucid, such as streams or dimensions, are infinite so programmers can request as many values as they need. Therefore, Lucid must use eduction to avoid useless and even infinite computations.

However, the overhead of eduction is paid for in the form of propagation of demands and storage of the traces leading to the fundamental computations. Stacks of values are used to store the interim values in the model of call-by-value, but call-by-need requires to store the expressions themselves or to use a mechanism that provides the same effect, which may be more complex and time-consuming.

2.4.6 Program Examples

The Figure 2.3 presents a Lucid statement that defines a stream *A*.

```
A = 0 fby.d A+1
```

FIGURE 2.3 LUCID STATEMENT

It can be used in a Lucid program with simply being added a desired value and a *where* clause.

```
A @.d 3
where
        A = 0 fby.d A+1;
end;
```

FIGURE 2.4 PROGRAM EXAMPLE: GENERATING NATURAL NUMBERS

The program is to evaluate the value of stream *A* on the dimension *d* where the value of *d* equals 3. To obtain the value *A @ (d=3)*, the system demands the value *A @ (d=2)*, *A @ (d=1)*, *A @ (d=0)* in turn, where *A @ (d=0)* equals *0* as defined in the equation.

A more complicated example is shown in the Figure 2.5. The program solves a simplified Hamming problem; we will use this as an example to describe the processing of Lucid source code in GIPSY in the following sections.

```
A
where
        dimension d;
        A = 1 fby.d merge.d(2*A, 3*A);
        merge.a(x,y) = if ( xx <= yy) then xx else yy;
        where
                xx = x upon.a (xx <= yy);
                yy = y upon.a (yy <= xx );
        end;
end;
```

FIGURE 2.5 A SIMPLIFIED HAMMING PROBLEM

The example takes two ordered sequences (2,4,6,8, ... and 3,6,9,...) and creates a third ordered sequence: 1,2,3,4,6,8,9,... without duplicates, which is similar to Hamming problem that uses 2, 3, and 5 to generate the sequence 2,3,5,6,8,9,... The *merge* is a non-recursive function that takes two input sequences and returns one ordered sequence.

The above examples illustrate the rule that a variable can be bound to a value only once in Lucid. A Lucid program does not have a concept of visible program states and hence does not have an assignment operator that changes the state. The programming language is also a kind of single-assignment languages that has no side effects. Consequently all operands can be evaluated in parallel. Moreover, the body of a called function can be evaluated as soon as all the arguments have been evaluated. Hence, concurrency does not have to be specified but is performed automatically.

In conclusion, Lucid takes streams as a basic data structure. Streams are described with the *first* and *next* operators, and loops are generated by extracting elements from streams using the *asa* (as soon as) operator. The elements of streams are defined in terms of non-recursive functions of elements of other streams, and/or earlier stream elements. Iteration is built on top of the streams, with iterations of a loop being mapped to elements of a stream.

3 GIPSY System

This chapter describes the GIPSY system, which processes Lucid programming languages family. Then, it presents the front-end in the GIPSY system—GIPC.

3.1 GIPSY Modules

The General Intentional Programming SYstem (GIPSY) is an integrated development framework for generating, compiling, and debugging programs written in the Lucid programming languages and is designed to achieve the goals of generality, flexibility and efficiency.

The GIPSY system uses modularity to achieve flexibility and generality; the distribution of execution and increment of granularity are adopted to resolve the efficiency issue. The system is developed in an incremental manner.

3.1.1 GIPSY Architecture

GIPSY consists of three modules (Figure 3.1): a general intensional programming language compiler (GIPC), a general eduction engine (GEE), and an intensional run-time programming environment (RIPE).

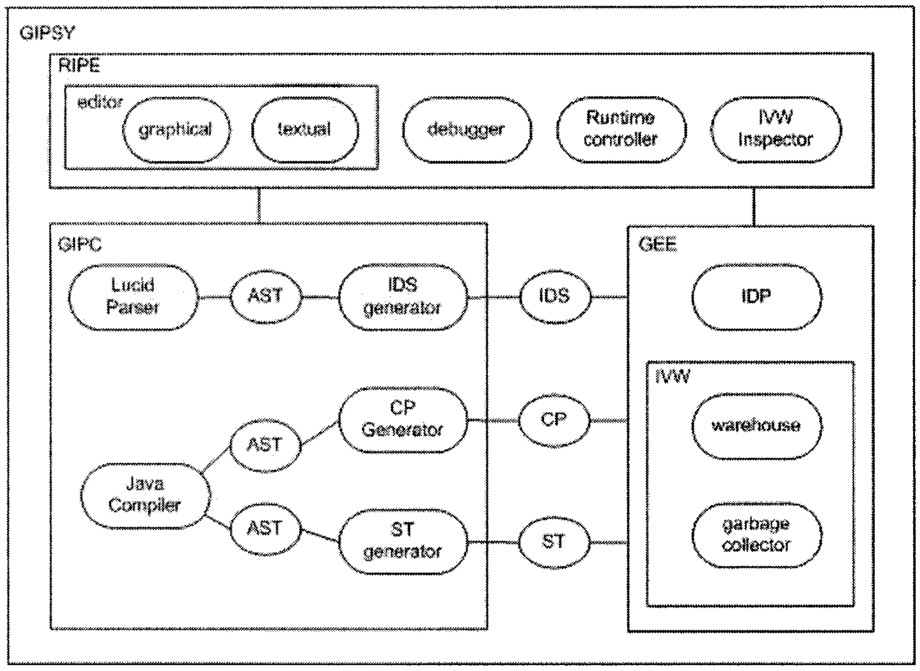


FIGURE 3.1 GIPSY ARCHITECTURE

The "general" is emphasized here because this system is designed to interpret different dialects of Lucid. The modularity of the system is designed to ensure that it will be easy to modify.

3.1.2 GIPC

GIPC is designed to take a hybrid language mixed by Lucid and Java, in which Lucid interprets the structure part, and Java specifies functional parts. GIPC is the backbone of GIPSY. It converts a Lucid program into three parts: an intensional data dependency structure (IDS) that describes the dependencies between the computation units involved; a set of sequential threads (ST) that are actually minimum-sized computation units; and a set of intensional communication procedures (CP). All the three parts together are the resource of the intensional demand propagator (IDP) in the GEE components.

However, because the implementation of GIPSY takes an incremental approach (see Section 4.4.1), the input and output of GIPC in the current prototype are quite different from that in the design. First, the current GIPC only deals with programs written in pure Lucid. Therefore, neither CP nor ST is applicable. Moreover, due to the eduction evaluation model adopted in the current implementation with which requires a sequential thread being started only when needed, no computation units can really be predicted before execution, let alone their dependencies. As a result, the output of GIPC in the current implementation is an abstract syntax tree (AST), more precisely, a decorated AST that has some supplemental information, for example, data type of each node.

3.1.3 GEE

GEE is a critical but complex part of GIPSY.

The essential work of GEE is to generate tasks for a particular architecture and environment using the computation model of eduction. These tasks can be executed concurrently.

Moreover, an intensional value data warehouse (IVW) is integrated in GEE to store values of variables that have been computed. When the value of a variable is required later on, it can be taken from the warehouse directly without being recomputed. A garbage collector is also provided to keep the warehouse up-to-date to the current situation. The IVW does not affect results that are eventually computed, but it will often improve performance of the system by eliminating redundant computation.

In addition to the IVW, it would be helpful for GEE to include a subcomponent called intensional environment monitor (IEM) to monitor the execution (although it is not shown in Figure 3.1). At runtime, the monitor watches changes in the environment, analyzes these changes in order to assist the application adapt to the changes safely and consistently. This IEM module is discussed in Section 5.1.3.

3.1.4 RIPE

RIPE is a graphical run-time programming environment. It is an integrated development environment (IDE) for writing and debugging programs written in Lucid. In addition, it can translate a diagram version of the program, provided by the user, into a textual version consisting of Lucid source code and vice versa because theoretically, each Lucid program has a diagrammatic equivalent.

3.2 Incremental Development

The prototype of GIPSY is being implemented currently and its being carried on following phases in which the development of the system has been started with a simple prototype and has then been extended.

3.2.1 Pure and Hybrid Language

In the first place, the prototype deals with programs written in pure Lucid. Then, it should be developed to process a hybrid language embedded with imperative language(s). The imperative language(s) is employed to write functions that perform coarse-grained computations.

There are two major advantages of embedding computation functions. First, it allows reusing legacy codes. More importantly, these functions can be executed on different computers so that the granularity of distribution is increased. As a result, the ratio of time spent on computation to that time on network traffic is increased and a better performance is likely to be achieved.

3.2.2 Data Type and Language Elements

The processing of programs written in pure Lucid can be divided into several phases according to the data type and language elements the prototype can handle.

First, only one data type (*double* is recommended) is handled by the prototype. Then, basic types (*double*, *int*, *string*) should be implemented. At a more advanced stage, user-defined types are supposed to be managed.

The language elements managed have the following increments: at a basic level, *where* expression, point-wise operators, intensional operators (#, @ and non-primitive operators, such as *first*, *next*, *fby*) are managed. Then, user-defined non-recursive functions are handled. Afterwards, user-defined functions with recursion should be able to be processed by the system. The separation of functions with/without recursion is due to the fact that non-recursive functions can be expanded inline and are therefore easier to process. More complexity on describing context (first-class contexts, context expressions) should be considered in the next phase. Also, note that there should not be much need for recursion because, in a well-written Lucid program, iteration is achieved by streams.

The incremental development is partly due to the limited understanding of the Lucid language of most of the participants in the GIPSY project. Also, it ensures that a working prototype can be built up with a small cost and experiments can be added up into the project in a stepwise manner. In addition, the incremented development model reduces the risk of failure.

3.3 GIPC

The GIPC component is the front-end of the compiler, which accepts Lucid source code, generates a parse tree, eliminates functions (if any), reduces non-primitive operators to their primitive forms, analyses the rank of each identifier, and generates an abstract syntax tree and dictionary that will be exported to the GEE.

In this section, a simple example is used to illustrate what GIPC does, and details of its implementation can be found in [Wu2002].

3.3.1 Function Substitution

The Lucid program (Figure 2.5) presents an application that contains a function — *merge*. As all functions are aliases of their underlying expressions, they can be removed from the parse tree on the condition that the language doesn't allow recursive definitions.

The program after functions being substituted is shown in the Figure 3.2.

```
A
where
        dimension d;
        A = 1 fby.d (if (x <= y) then x else y);
        where
                x = 2*A upon.d (x <= y);
                y = 3*A upon.d (y <= x );
        end;
end;
```

FIGURE 3.2 PROGRAM AFTER FUNCTIONS HAVE BEEN SUBSTITUTED

3.3.2 Non-primitive Operators Reduction

By applying the conversion rules (shown in Figure 3.3) to the operator *upon*, a program with only primitive operators (shown in Figure 3.4) is obtained.

```
first.d X      =    X @.d 0
next.d X       =    X @.d ( #.d +1 )
prev.d X       =    X @.d ( #.d-1 )
X fby.d Y      =    if #.d == 0 then X else Y @.d (#.d - 1)
X wvr.d Y      =    X @.d T
                    where
                            T = U fby.d U @.d ( T +1 )
                            U = if Y then #.d else next.d U
                    end
X asa.d Y      =    first.d ( X wvr.d Y )
X upon.d Y     =    X @.d W
                    where
                            W = 0 fby.d if Y then ( W+1) else W
                    end
```

FIGURE 3.3 REDUCTION OF NON-PRIMITIVE OPERATORS

```
A where
        dimension d;
        A = if #.d == 0 then 1 else (if (x <= y) then x else y ) @.d (#.d-1);
        where
                x = 2*A @.d w
                where
                        w = if #.d == 0 then 0 else
                        ( if (x <= y) then w+1 else w) @.d (#.d-1);
                y = 3*A @.d v
                where
                        v = if #.d == 0 then 0 else
                        ( if (y <= x) then v+1 else v) @.d (#.d-1);
        end;
```

FIGURE 3.4 PROGRAM WITH ONLY PRIMITIVE OPERATORS

We can make a few observations about this program:

- The program defines a stream *A*.
- To support the definition of *A*, it defines 4 other identifiers: *x*; *w* to define *x*; *y*; *v* to define *y*.
- It has only one dimension: dimension *d*.

3.3.3 Rank Analysis

Since an identifier is likely to vary according to *part* of the context space rather than having its extension varying throughout to the entire context space, only *some* of the dimensions have an effect on the value of an identifier. The set of dimensions upon which an identifier actually depends is known as the *rank* of the identifier. Since an identifier is defined by an equation, the rank of an identifier is equal to the rank of the right hand side of the equation.

An extreme example of rank is the constant in a Lucid program. A constant has a fixed value no matter what point it stands on. In another word, the change of any of the dimensions in the context space has no effect on the value of a constant.

The rules of analyzing a rank [Dod1996] are in Figure 3.5. In the figure, *Rank* is a function that takes a syntactic form as its argument and returns a set of dimensions. $\oplus$ denotes an arbitrary pointwise operator.

Rank [constant]	≡	ϕ
Rank [#.d]	≡	*{d}*
Rank [E1⊕E2]	≡	*Rank [E1]∪Rank [E2]*
Rank [if E1 then E2 else E3 fi]	≡	*Rank[E1]∪Rank[E2]∪Rank[E3]*
Rank [E1 @.d E2]	≡	*(Rank[E1] - {d})∪Rank[E2]*

FIGURE 3.5 RULES FOR RANK ANALYSIS

Rank analysis is complicated for functions, because a function may be called with arguments of different ranks at different points in the program. The simplest analysis assumes the worst case, and analyses the function with the highest rank argument found in the program.

The result of rank analysis could be used in the optimization. For example, the evaluation of *E1 @.d E2* can be simplified to evaluating *E1* if *E2* doesn't vary on the dimension *d*.

$$E1\ @.d\ E2 \Rightarrow E1 \qquad \text{if } d \notin Rank[E1]$$

Related to this, the rank of the actual parameters to a function call can be used to eliminate save and restore of local dimensions that occur around a function call.

The calculated ranks can also be used to control the variable value cache in the way that each variable is stored based solely upon the dimensions in its rank rather than the entire dimension set.

3.3.4 Abstract Syntax Tree and Dictionary

The GIPC generates an abstract syntax tree (Figure 3.6) for the program in Figure 3.4.

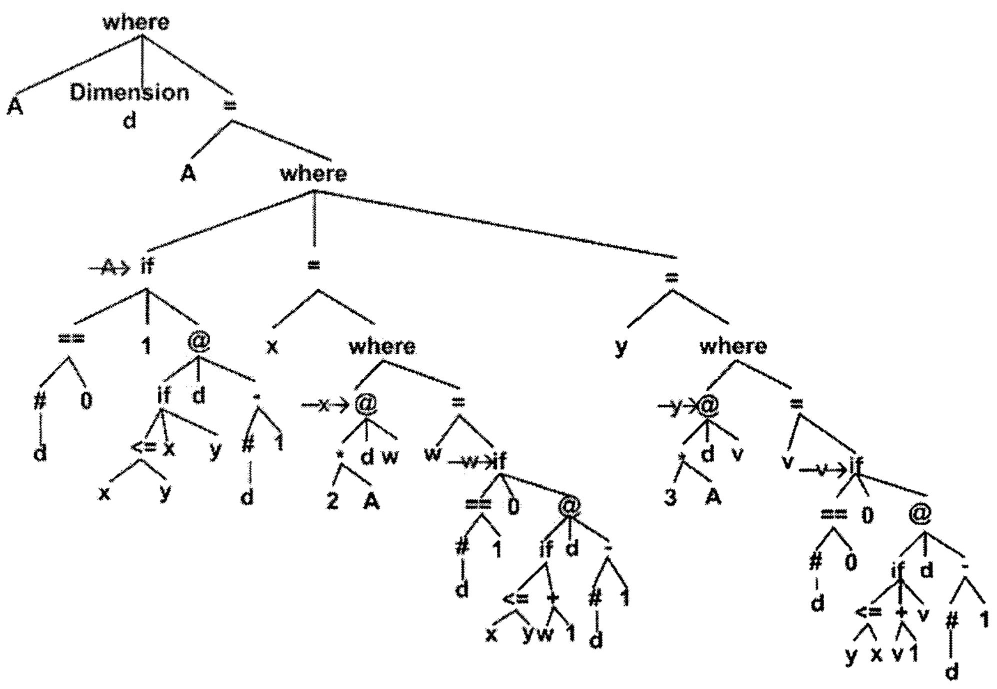


FIGURE 3.6 ABSTRACT SYNTAX TREE

In order to locate an identifier on the abstract syntax tree quickly, the GIPC places a pair consisting of an identifier and its associated access node (marked on the graph) in a dictionary. The dictionary also contains other useful information extracted from the static codes, such as the type (integer or double) and rank of the identifier.

4 GEE Analysis

The general eduction engine, GEE, is the backend in the GIPSY system and is in charge of executing Lucid programs upon the abstract syntax tree generated by the front-end of the system in an efficient way.

GEE is a component of the run-time system that takes charge of generating tasks and executing them. It consists of three sub-components: an executor, a value warehouse and a monitor. When a user demands a computation, GEE propagates demands that are necessary for computing the result progressively until it finds all the computation units needed to obtain the value. The computation of demands and the generation of demands are performed by the same component: executor. GEE also employs a value warehouse to place computed values in order to improve performance. In addition, in an advanced architecture, GEE monitors the execution of the program, adjusting it to ensure efficient and reliable throughput. As an integrated component in the GIPSY system, GEE interacts with the other two components: GIPC and RIPE.

The difficulty of implementing GEE has its roots in the dynamic nature of execution and, moreover, in the variety of possible execution environments. This section analyzes the parallelism existing in programs written in Lucid language, introduces distributed execution, examines the system from an object-oriented approach and discusses how the development could be carried on.

4.1 Parallelism

We proposed to distribute the execution over a network of computers based on the parallelism observed from two aspects: the nature of the language and its execution manner. Not only do the problems within Lucid's application domain have inherent data parallelism but also the language itself supports parallelism in a natural way. Further more, Lucid adopts the demand-driven execution model, in which some execution actions can be carried out at the same time.

4.1.1 Data Parallelism

Lucid applications usually have implicit data parallelism [Jag1995]. For example, a sequence of data could be processed by a "pipeline" of functions where each of the functions can work on a different part of the sequence concurrently. This could be naturally expressed in Lucid as shown in Figure 4.1.

```
a where
        a = 0 fby.x f (b);
        b = 0 fby.x g (c);
        c = 0 fby.x h (d);
        d = 1 fby.x d+1;
end;
```

FIGURE 4.1 PIPELINE PARALLELISM

The computation of successive values of *a* in the *x* dimension has implicit parallelism due to pipelining. While *c* at some point *i* in the dimension *x* is being computed, *b* at point *i-1* is being computed and *a* at point *i-2* in dimension *x* is being computed.

Another kind of data parallelism is that one function can work on different data input so that various data input can be worked at the same time to achieve results. This kind of parallelism is the principle source of massive parallelism in most scientific computation. One example is matrix transposition. If a function is implemented to return the result of the transposition of one input argument, the function will be called repeatedly to obtain each element in the resulting matrix. Lucid is suitable to expresses this idea as illustrated in the following figure.

```
transpose.d0, d1(M) = N
where
        dimension temp ;
        N = realign.tmp, d0(realign.d0, d1, realign.d1, tmp(M)) ;
        where
                realign.a, b(X) = X @.a #.b;
        end;
end;
```

FIGURE 4.2 DATA PARALLELISM

In this example, the transposition of a matrix is done through renaming the dimensions in which the matrix varies. The function *realign* defines the renaming and it can be accessed by all elements of the matrix and does so concurrently.

A simpler example can be observed from the sub-tree for defining the identifier *A*. In this example, values of *x* and *y* can be demanded and worked on simultaneously. However, *if* also imposes a constraint on parallelism; since only one of *A* and *B* will be needed, their evaluation should be delayed until the value of $x \leq y$ is known.

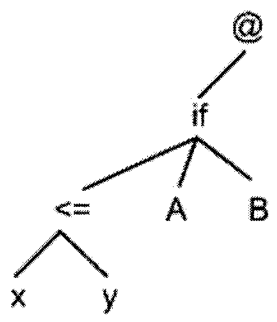


FIGURE 4.3 PART OF THE AST IN FIGURE 3.6

A Lucid program is different from an equivalent procedural program in that the meaning of a Lucid program is inherently parallel unless otherwise constrained, whereas the procedural program is inherently sequential unless parallelism is explicitly identified (which is not even possible in many procedural languages).

4.1.2 Execution Parallelism

The GEE adopts the demand-driven execution model. In this model there are two major types of tasks: demands propagation and computation of demands.

The computation task can be divided into small pieces and be performed using several computers because of a program's internal data parallelism. Moreover, demands can be propagated at the same time as they are being worked on (which are actually individual computation units that consist of the whole computation task). In an advanced execution model, the generation of demands can be carried out on several computers simultaneously.

4.2 Distribution

There are usually two ways to improve the execution efficiency. One way is by using a computer with higher capacity and the other is by distributing the execution over a group of processes/processors that are working collaboratively for the same goal. The second way does not guarantee an improved performance since there is overhead spent on communication. Due to the parallelism in a Lucid program we can try to use the second method to achieve a better performance, obviously having more research values. For this purpose, this section discusses the concept of distribution, its characteristics and application in GEE.

4.2.1 Concepts

There is a trend towards distributed systems today. Applications increase in power, thus increase in number and size; very large applications cannot be handled satisfactorily by a single central processor. At the same time, microcomputers have dropped in cost to a level where most users can afford one. Therefore, it is realistic to consider distributing some components rather than centralizing them. Moreover, putting everything together may not only decrease reliability and security but also increase the response time to end users, or even make the requirement of access unavailable when they need it. These factors cause the trend towards distributed processing.

A distributed system is a collection of computers linked by a network and equipped with distributed system software. The distributed system software enables the comprising computers to coordinate their activities and to share system resources. A well-developed distributed system software gives distribution transparency to the subsystems.

4.2.2 Characteristics

A distributed system has six important characteristics [CDK1994]: (1) resource sharing, (2) openness, (3) concurrency, (4) scalability, (5) fault tolerance, and (6) transparency. These characteristics are not automatic consequences of distribution. Instead, they are obtained only as a result of a careful design and implementation.

Resource Sharing

Resources provided by a computer that is one member of a distributed system can be shared by clients and other members of the system via a network. In order to achieve effective sharing, each resource must be managed by software that provides interfaces which enables the resource to be manipulated by clients. Resources of a particular type are managed by a software module called a resource manager, which performs its job based on a set of management policies and methods.

Openness

Openness in distributed systems is the characteristic that determines whether the system is extendible in various ways. This characteristic is measured mainly by the degree to which new resource sharing services can be incorporated without disruption or duplication of existing services. From the view of software, the openness of a distributed system can be viewed from software extensibility, which is the ability to add new software or modules from different vendors to a distributed system.

Concurrency

Concurrency is the ability to process multiple tasks at the same time. A distributed system comprises multiple computers, each having one or more processors. The existence of multiple processors in the computer can be exploited to perform multiple tasks at the same time. This ability is crucial to improve the overall performance of the distributed system. The software used must make sure that one access to the same resource does not conflict with others.

Scalability

Scalability in distributed systems is the characteristic whereby a system and its application software need not change when the scale of the system increases. Scalability is important since the number of requests processed by a distributed system tends to grow, rather than decrease. In order to handle the increase, additional hardware and/or software is usually needed. A system is said to be scalable if it provides flexibility to grow in size, but still utilize the extra hardware and software efficiently. The amount of flexibility the system has determines the level of scalability it provides.

Fault Tolerance

Fault Tolerance is a characteristic whereby a distributed system provides appropriate handling of run-time errors that occurred in the system. A system with good fault tolerance mechanisms has a high degree of availiability. A distributed system's availability is a measure of the proportion of time that the system is available for use. A better fault tolerance increases availability. To achieve fault tolerance, software is designed to recover from faults when they are detected.

Transparency

Transparency is the concealment of the separation of components in a distributed system from the user and the application programmer such that the system is perceived as a whole rather than as collection of independent components. Due to transparency, local and remote objects can be accessed by identical operations. Objects can be accessed without the information of their location. Several processes can operate concurrently using shared information objects without interference between them. Multiple information objects can be used to increase reliability and performance without knowledge of the replicas. Information objects can be moved within a system without affecting the operation of users or application programs.

Heterogeneity

The presence or absence of the resource sharing, openness, and transparent characteristics explained above influences the heterogeneity of a distributed system.

The existence of resource sharing increases the need for the system to be more open. This is because sharable resources of this system are almost certainly made up of hardware and software from different vendors. Without this openness, these resources cannot be used by clients if they are based on technologies from vendors different from those resources.

Another problem is that hardware and software of various vendors will not be able to be incorporated into the system, especially in cases where legacy systems exist. Having the openness in the system facilitates the mix and match of hardware and software. This allows the system to take advantage of the best features from different products, regardless of who their vendors are.

In addition, transparency determines a distributed system's approach to its heterogeneity. If the transparency is not available, clients will be exposed to the complexities of multiple technologies underlying the system. As a result, the system becomes harder to use and requires a lot of training. This could reduce even eliminate the lure of the distributed systems model completely.

4.2.3 Distribution, Concurrency and Parallelism

Three words appear frequently: concurrent (concurrency), parallel (parallelism), and distribution. They are also used in turn as a contrast to sequential execution. The features of a distributed system have been stated in the above section; however, the concept of concurrency and parallelism haven't been clarified.

Concurrency literally means "at the same time", which refers to the non-sequential semantics of a program. A sequential program has a single thread of control while a concurrent program has multiple threads of control that allow it perform multiple computations in parallel. In order to work together to solve a problem, the multiple threads of controls need to communicate.

The distinguishing attribute of a parallel program is that it is written to solve a program in less time than would be taken by a sequential program. The main goal is to reduce overall execution time. A parallel program can be written using either shared variables or message passing.

Strictly, there cannot be concurrency on a single processor. However, the term is usually used to mean that several threads are "alive" at the same time, even though only one of them is using the processor while other threads are waiting for their turns. A more accurate (although rather out-dated) term is "multi-processing". On the contrast, "parallel processing" is "true" concurrency, in that there are several processors and therefore several threads active at a time.

The communication between multiple threads is realized either by writing and reading shared variables (also called shared-memory) or by sending and receiving messages (if the memory is distributed).

4.2.4 Distribution in GEE

We have discussed the internal parallelism nature of Lucid programs and we have found that a distributed environment is suitable for efficient execution of Lucid programs. The characteristics of distributed systems set up a set of criteria that we must consider when design the distributed execution of a Lucid program. Three important aspects analyzed are: resource sharing, concurrency and transparency.

Resource Sharing

The GEE component has to manage two kinds of resources: application and environment.

The application resources are in terms of tasks that processors work on or interim results that can assist computations. These resources could be represented as: computation units, generated demands and computed values.

Environmental resources consist of the hardware equipped in the system and its dynamic changes if applied. The resources can be static, such as the identity of each host and its capacity etc, or dynamic, such as the CPU usage, memory usage, network traffic and so on. The analysis of environmental resources enables the system to detect the changes in the environment. The major goal of environmental resource management is to achieve a good balanced workload.

Both kinds of resource are shared among executors in the system. All executors should have access to the three kinds of application resources noted above. Demands are working targets; computation units are working bricks and computed values are results that are also essential resource. Similarly, by sharing the environment resource each executor is aware of the situation of the whole system and an individual executor can make decisions on whether and where to migrate tasks when it is over-loaded.

Concurrency

The concurrency feature is presented by the parallelism in the GEE, which has been discussed thoroughly. However, the concurrency implies the risk of conflict, which may happen when two or more processors want to access the same resource. The computation task can be resolved into smaller tasks and each processor works on one or several of them. The tasks are stored in the same place. When processors take tasks from the storage, they may compete for the same task, which is better to avoid in order to prevent redundant computations. The computed values do not have read-write conflict either, because the value of a variable does not change during the execution but can be shared among all processors who want to read. Therefore, it's unnecessary to consider using exclusive lock to prevent "dirty" data.

Transparency

Handling the difference among platforms is definitely a headache. The adopted strategy is to utilize the existing technology to avoid solving the problem from scratch as much as possible. In case the problem can't be avoided, different sets of codes aimed at different platforms should be carefully wrapped in a component and a friendly user interface should be provided to let users configure appropriately. In this way, the difference is hidden from the users and transparency is ensured.

4.3 Object Approach

The object technique, as well as the distribution, is represented in the design of the GEE component.

The object technique is powerful enough to achieve the desired flexibility. The object concept in GEE has different levels, which can be as big as a module or as small as a minimum-sized computation unit.

First, an object can be a module. The GEE system consists of several modules, which are executor, monitor and value warehouse to ensure extendability and expandability. For example, if a new platform is added to GEE, only the part that monitors the environmental changes needs to be modified.

Each module can be decomposed into several sub-modules. For instance, in the monitor part, a sub-module is needed to collect information while another one is needed to analyze the collected information.

An object can be a computation unit, such as a function, that can implement the computation of a variable. Several closely related units can be combined together to form a bigger unit. When the system is running, these units can be treated as one object to be moved together.

4.4 Development Manner

Due to the concise and implicit nature of Lucid and the demand-driven execution model it adopts, the language is more difficult to understand compared with many conventional languages. As a consequence, we chose to develop the GEE component in an incremental manner to help understanding and reduce development risk.

4.4.1 Incremental Development

The development of the eduction engine is performed in an incremental fashion: (1) a program is run with one process on a single processor; (2) multiple processes on one processor are used for executing a program; they cooperate with each other and work for a single goal; (3) multiple processors on the same platform are used; (4) run-time scheduling is enforced. For the integrated IVW component in GEE, we first implement a global warehouse but leave the garbage collection to the second step. Then, a distributed warehouse with a distributed garbage collection should be experimented.

We initially had very limited knowledge of Lucid programming languages, thus a discarded prototype was suitable for the purpose of understanding the Lucid execution. This prototype should be kept as small as possible and be separated from the front-end. After this stage is passed, we should develop a reusable prototype that can grow up to become a real version of implementation.

More concretely, an interpreter should be developed at the beginning to help understand the essence of Lucid execution and it should be experimented with multi-threading technology for understanding parallelism in Lucid. Finally, the execution should be done in a real distributed environment.

These increments are compatible with the increments proposed in Section 3.2. The interpreter runs with a single process; the multi-threading approach is applied with several processes but on a single processor; and the real distribution must be done over a network of processors, which may be installed with heterogeneous systems.

Moreover, an ideal picture of the system is that each computer works efficiently and concurrently for desired value(s). There should be a load balancing among the computers to ensure the adequate utilization of resources. This requires that GEE has a component to monitor the run-time environment of the execution.

Incremental development avoids risks behind a big bang development approach. Complex features found in the distributed systems can be deferred until the initial development is complete while core framework components should be developed first.

4.4.2 Implementation Language

The JAVA programming language is adopted as the implementation language.

Although implementing the project with C++ can probably guarantee more efficient execution, we adopted Java programming language in GIPSY considering the generic advantages of Java, its products of parser generator and distribution technology. Its efficiency problem will be discussed at the end of this section.

Generally, Java is:

- Pure object-oriented;
- Safe, flexible and reusable;
- Portable;
- Easier to develop and debug than C++.

Moreover, Java CC, being a popular parser generator, was selected as the implementation tool [Ren2001] for parsing Lucid source code.

- TOP-DOWN – it allows the use of more general grammars (although left-recursion is disallowed). In addition, with this top-down style, it is easier to debug. It has the ability to parse to any non-terminal in the grammar, and also has the ability to pass values (attributes) both up and down the parse tree during parsing.
- Lexical and grammar specifications are in one file – the lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are both written together in the same file. It makes grammars easier to read (since it is possible to use regular expressions inline in the grammar specification) and also easier to maintain.

- Tree building processor – JavaCC comes with JJTree, which is an extremely powerful tree building preprocessor.
- Extremely customizable – JavaCC offers many different options to customize its behavior and the behavior of the generated parsers. Examples of such options are the kinds of Unicode processing to perform on the input stream, the number of tokens of ambiguity checking to perform, etc.
- Syntactic and semantic *lookahead* specifications –JavaCC generates an LL(1) parser by default and LL(k) where necessary. JavaCC offers the capabilities of syntactic and semantic *lookahead* to resolve shift-shift ambiguities locally at the portions of the grammar that are not LL(1). The parser is LL(k) only at such portions, but remains LL(1) everywhere else for better performance. Shift-reduce and reduce-reduce conflicts are not issues for top-down parsers.
- Very good error reporting – JavaCC error reporting is among the best in parser generators. JavaCC generated parsers are able to clearly point out the location of parse errors with complete diagnostic information.

Finally, Java programming language is adopted widely in concurrent programming and distributed systems [Lea1997, Fox1997] because of the following two reasons. First, Java is an open resource that provides a plenty of distributed technologies that a user can choose, for example, RMI, JavaBeans, Jini, and so on. Second, Java is portable because Java code is translated into a set of bytecodes that can be interpreted in a Java Virtual machine (JVM) located on top of an operating system. Consequently, the Java bytecode can be understood by a computer installed with any architecture or operating system so long as it supports JVM.

Java is safe and flexible in that it has exception checks, dynamic code loading and dynamic method invocations. At the same time, its runtime performance is decreased, which is the basis for most of the criticism.

However, the performance issue is a matter that concerns more about how a Java program is proceed presently. A Java program is typically compiled to universal bytecodes that will be interpreted on the client Java Virtual Machine. Such invocation of Java applications suffers from a performance penalty of between 5 (PC Just In Time Compiler) and 50 (Un-enhanced Interpreter) compared to comparable native compiled C code [Fox1997].

The JIT compiler translates the byte code into native code when a new method is invoked at runtime. Thus, it allows classes to be loaded dynamically. As JIT compilers improve in quality and become generally available, we can expect that this way of proceeding Java programs will improve the performance but it could be still slower than conventional compiled code.

A programmer can build a native Java compiler those compiled code could be expected to have a comparable performance to the executable code produced by the C++ or C compilers because the translation from Java bytecode to native code is done before the start of program execution thus the compilation overhead can be ignored at runtime. Also, it can use expensive optimization. The disadvantage of this static compilation model is it cannot support dynamic class loading and consequently, it cannot take advantage of Java's flexibility and reusability. Also, the portability is lost. Therefore, the solution can be regarded as an argument of Java's efficiency but its usage is not encouraged.

In conclusion, we implement the system in Java because not only does the language have prominent advantages in implementing the parser and the distributed component but also it will hopefully improve the runtime performance from the viewpoint of how to process the implementation language. At the same time, the GIPSY prototype being implemented focuses on developing a framework for a Lucid compiler and ranking efficiency as a non-functional requirement second to its flexibility.

5 GEE Implementation

The GEE employs *eduction* (demand-driven) computation model and uses a value cache to avoid redundant computations. It takes the abstract syntax tree produced by GIPC (see Section 3.3.4), and propagates demands required for working out the desired value. The computed values are placed in the value warehouse, and the system looks up values in the warehouse each time before it calls computation method to avoid redundant computations.

5.1 Components

As shown in the Figure 3.1 (GIPSY architecture), GEE has two basic component, *IDP* (intensional demand propagator) and *IVW* (intensional value warehouse). The Figure 5.1 provides a closer look at this component.

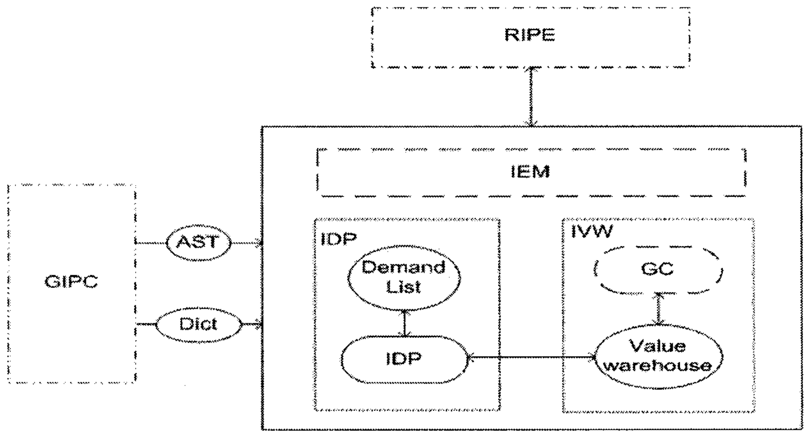


FIGURE 5.1 GEE ARCHITECTURE

The figure shows the inputs of the component are *AST* (Abstract Syntax Tree) and *Dict* (*Dictionary*). The first prototype of GIPSY system deals with programs written in pure Lucid only, therefore, both sequential thread (ST) and communication procedures (CP) for Java functions are not applicable. The GEE component only operates on a decorated AST and this information is placed in the *Dict* data structure. In order to emphasize the abstract syntax tree as a necessary input, the figure 5.1 separates *AST* from *Dict*.

5.1.1 IDP

There are two indispensable tasks in the GEE: demands generation and their execution, however, they are performed by a single role: IDP (intensional demand propagator). IDP has to demand supportive values when it is not able to work out the desired value. Generated demands are stored in the *demands* data structure. IDP takes demands (desired variables) from the *demands*; enquires values for the variables in the *value warehouse*; then does calculations if the value has not been produced. It may generate new demands when calculating the value.

5.1.2 IVW

The IVW (intensional value warehouse) stores computed values. Because IDP enquires the value warehouse each time before it does calculation there would be intensive enquires demanded on the warehouse. Therefore, stored values should be carefully arranged for being quickly retrieved.

Since Lucid tends to be used in large scientific computation, the number of computed values may increase dramatically, the warehouse contains a GC (garbage collector) to discard the less "valuable" data and keep the warehouse at a reasonable size, which is also good for speedy inquiries.

5.1.3 IEM

The GEE is designed to have a component: IEM (intensional execution monitor), which is to check the execution and to watch the workload of processors and the network so that the system can adapt to the changes initiatively to achieve a better work-balancing and reduce the impact caused by failures of individual hosts. More details will be discussed in Section 6.4.

Since this component has not been completed, it has a dash-lined frame in the architecture figure.

5.2 Interpretation

The execution of a Lucid program is hard to understand in comparison to the execution of programs in more conventional languages, let alone to distribute the execution on several machines. Therefore, it is helpful to present the procedure in an interpretive way first. Then, a multithreading approach is presented to help understand the parallel nature of Lucid. Finally, the execution is implemented with a distributed technology, which is also the real method GEE adopts.

5.2.1 Interpretation Rules

GEE takes the abstract syntax tree generated by GIPC. The tree has these types of nodes:

- *Constant*, which has the same value regardless of the context in which it is evaluated.
- *Identifier*, which is defined with a "where" clause (such as identifier *A* in the program shown in the following figure) or on the right-hand-child of its parent "assignment" operator (such as identifier *w*). The access node for an identifier can be retrieved from the dictionary exported by GIPC.
- *Algebraic operators*: arithmetic, relational and logical operators.
- *if*, which has 3 children: "condition" on the left, "then" part in the middle, and "else" part on the right.
- @, which has 3 children: : on the left is the first sequence (which evaluates to an identifier in which we want to navigate in), in the middle is the dimension in which we want to navigate, and on the right is the second sequence (which evaluates to the index we are navigating to in the context space of the identifier, in the dimension identified by the second operand).
- #, which has 1 child: a dimension of the current evaluation context.

```
A where
        dimension d;
        A = if #.d == 0 then 1 else (if (x <= y) then x else y ) @.d (#.d-1);
        where
                x = 2*A @.d w
                where
                        w = if #.d == 0 then 0 else
                        ( if (x <= y) then w+1 else w) @.d (#.d-1);
                y = 3*A @.d v
                where
                        v = if #.d == 0 then 0 else
                        ( if (y <= x) then v+1 else v) @.d (#.d-1);
         end;
```

FIGURE 5.2 PROGRAM WRITTEN IN LUCID (THE SAME AS IN FIGURE 3.4)

The function *eval (expression, context)* is used to evaluate the value of an *expression* in a *context*. The *expression* appears in the program and the *context* is constructed by the interpreter according to the current evaluation context. The value returned by *eval* (*expression, context*) is defined by cases on the kind of expression.

```
(1)   eval (k, c) = k
(2)   eval (i, c) =  eval  (lookup(i), c)
(3)   eval ( x⊕y, c) = eval (x, c) ⊕ eval (y, c)
(4)   eval (if p then x else y, c) = if eval (p, c) then eval (x, c) else eval (y, c)
(5)   eval (e @.d n, c) = eval (e, c[d]←eval(n,c))
(6)   eval (#.d, c) = c[d]
```

The following notes explain each case.

(1) The value of a constant is independent of the context.

(2) The call *lookup(i)* returns the defining expression for the identifier *i* as given by the dictionary.

(3) The evaluator evaluates each operand in the current context and then applies the operator to the results. Since operators do not have side-effects, the operands can be evaluated concurrently.

(4) The evaluator evaluates the condition in the current context and uses the result to decide which of the other operands to evaluate. As usual, evaluation of *x* or *y* is delayed until the value of *p* is known.

(5) A new context in which the value of dimension *d* is *eval(n,c)* is created, then *e* is evaluated in this context.

(6) The index of dimension *d* in the current evaluation context is returned.

An interpreter was implemented according to these rules. It demonstrates that Lucid programs can be executed correctly and the results can be used to validate the correctness of multithreading and distributed approaches. Although the interpreter is inefficient, it provides insight that will lead to more efficient implementations.

5.2.2 Evaluation Process

The process is started by a demand for the value of an identifier in a context. The interpreter applies the evaluation rules (1)-(6) until the desired result has been found. During the evaluation, demands for other identifiers in other contexts (possibly different dimensions and/or with different values) may be needed. For example, the demand *A @.d 3,* given to the natural number generating program, would lead to the evaluation of *A @.d 2, A @.d 1, A @.d 0,* and, finally, *A @.d 3.*

It can be observed that, in a complex computation, some values may be needed more than once. To avoid redundant calculation, the interpreter stores computed values in a cache that is referred to as the *value warehouse.*

It also can be observed that some evaluations can be performed concurrently. For example, to evaluate the condition $x<=y$, the evaluation of identifier x can be done the same time with the evaluation of identifier y, although the evaluation of the comparison itself cannot start until the both values are ready. The saving is trivial in the programs shown here but is significant for larger programs.

5.3 IC Class

The execution schema described in Section 5.2 is helpful for understanding how a Lucid program works, however, it is inherently slow due to interpretation. Moreover, it does not take advantage of the highly parallel nature of large Lucid programs. Therefore, we describe a multithreading approach in Section 5.4 and a distributed approach in Section 5.5 respectively both of which provides acceptable performance.

Rather than operating on the abstract syntax tree directly, both approaches require each identifier in a Lucid program to be converted to a Java class, *IC* Class; the conversion is done by a component called *code generator*.

The identifier class has data members of *context* and *value* and provides a public method that can be called by the engine to calculate the identifier (as defined in the program) at an arbitrarily given context. As a consequence, a demand for an identifier in a context can be presented as an instance (*IC* object, in which *I* stands for "identifier" and *C* for "context") of the associated class possessing a specified context (the data member).

Once the meaning of an *IC* object has been clarified, a demand can be referred as an *IC* object and associated with a value after it is satisfied.

For the demand propagator, the parallel execution of some branches in an abstract syntax tree implies that the generated demands may be run on several processors simultaneously. Moreover, execution of demands can be carried out at the same time as the generation of demands. In a more advanced schema, multiple processors can cooperate on the task of generating demands (see Section 7 for further discussions).

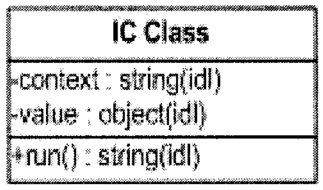


FIGURE 5.3 *IC* CLASS DIAGRAM

Code Generator

The transformation from an abstract syntax tree to the actual code for each identifier class is done automatically with a code generator.

The only thing that distinguishes one *IC* class from another is the implementation function *run* while other parts are the same except the name of a class is in a form of string "IC" appended with the index for the associated identifier.

A recursive function *trans* is used to generate the *run* method according to the node types. The *trans* function has two parameters: one is the node, and the other is the indent space. Its major part is a *switch* statement that produces relative text code for associated node type. The piece of code for translating @ node is shown in the following figure.

```
1:   void trans ( SimpleNode expr, int indent ) throws Exception
2:  {
3:       ....
4:       switch ( expr.id )
5:       {
6:        ....
7:        case JJTAT:                                              // @ node
8:             dim = ( (SimpleNode) expr.children[MIDDLE] ).ID ;  //dimension
9:             code.append ( "cont [" + dim + "] = " );           //produced code
10:            trans ( (SimpleNode) expr.children[RIGHT], indent + 1 );
11:            trans ( (SimpleNode) expr.children[LEFT], indent + 1 );
12:       break;
13:         ....
14:      }
15:       .....
16:  }
```

FIGURE 5.4 CODE GENERATOR

The operator @ is used to change the current dimension using its right-side operand and evaluate its left-side operand in the new context. On an AST tree, the node with the type of @ has 3 children with its right-side operand on the right side, effective dimension in the middle and the left-side operand on the left side.

Therefore, in the above figure, the current dimension is fetched to the variable *dim* on line 8. Then, on line 10, the code of its right child is generated to update the context. Finally, the code of its left child is generated to evaluate the node's left-side operand on line 11.

As a by-product, the code generator becomes middleware between the GIPC and GEE, thus, the interference between the two component is reduced to minimum.

5.4 Multithreading

Because each *IC* object can represent an identifier on a certain context, it is reasonable to assign each *IC* object a thread that the objects can be executed concurrently.

5.4.1 Thread

A thread is a single sequential flow of control within a program. Multiple threads in a single program can perform different tasks and run at the same time. Although a thread is similar to a real process in that a thread and a running program are both a single sequential flow of control, a thread is considered as *lightweight* because it runs within the context of a program and takes advantage of the resources allocated for that program and the program's environment.

A Java thread object can be in one of the following states:

- *Created*: when it's first created
- *Terminated*: when it can not be restarted and is eligible for garbage collector
- *Running*: currently executing
- *Runnable*: waiting to be scheduled
- *Suspended*: non-runnable, can not be scheduled by JVM

Figure 5.5 shows the life cycle of a thread.

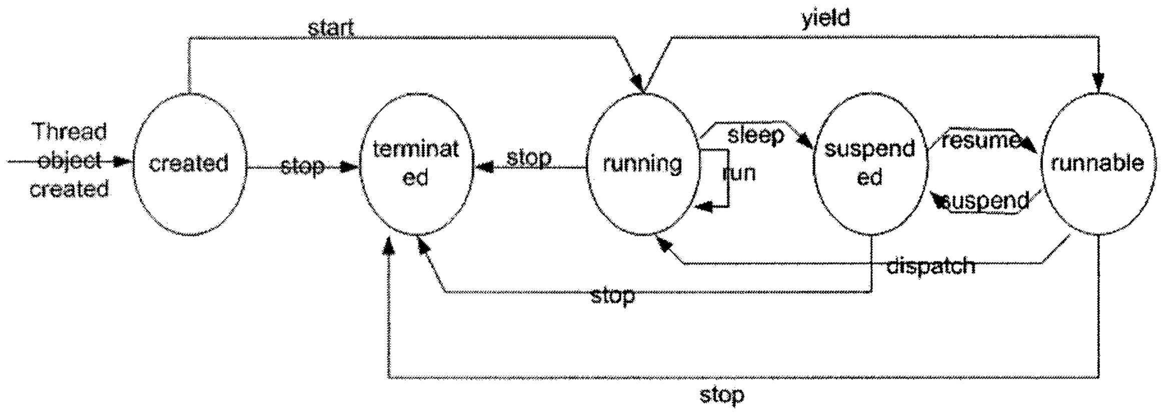


FIGURE 5.5 THREAD LIFECYCLE

A thread is an empty object when it is *created.* After it is started up, it is assigned resources and its *run* method is called up to perform a certain task. A running object can be suspended either actively by calling the *sleep* method or passively blocked by I/O. A thread can *yield* the right of execution to one of the other threads. A thread can be forced to terminate by calling a *stop* method.

A thread inherits *wait* and *notify/notifyAll* method from its parent class *Object.* The *notify* method wakes up a single thread that is waiting on this object's monitor. The *notifyAll* wakes up all threads that are waiting on this object's monitor, while the *wait* method causes current thread to wait until another thread invokes the *notify* method or the *notifyAll* method for this object.

Although threads are claimed to run concurrently, in practice, it is usually not the case. Most of the computer has a single CPU, so there's actually only one thread running at a time. However, each of the threads is assigned a time slice to execute in turn hence gives an illusion of concurrency. Execution of multiple threads on a single CPU, in some order, is called *scheduling.* The Java supports a deterministic scheduling algorithm known as *fixed priority scheduling.* The algorithm schedules threads based on their priority relative to other runnable threads.

When a thread is created, it inherits its priority from the thread that created it but the priority can be modified at any time after its creation. Thread priorities are integers, in which bigger numbers imply higher priorities. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. Only when that thread stops, yields, or becomes not runnable for some reason will a lower priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.

The Java runtime system's thread scheduling algorithm is also *preemptive* that if at any time a thread with a higher priority than all other runnable threads becomes runnable, the runtime system chooses the new higher priority thread for execution. The new higher priority thread is said to *preempt* the other threads.

At any given time, the highest priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower priority thread to avoid starvation. For this reason, use priority only to affect scheduling policy for efficiency purposes. It is important not to rely on thread priority for algorithm correctness.

There are two ways to create a new thread of execution. One is to declare a class as a subclass of the class *Thread.* This subclass should override the *run* method of *Thread.* An instance of the subclass can then be allocated and started. The other way is to declare a class (e.g. *myThread*) that implements the *Runnable* interface. The *myThread* class then implements the *run* method. An instance of *myThread* can then be allocated, passed as an argument. The two methods are equivalent.

5.4.2 IC Runnable Objects

As proposed in Section 5.3, each identifier can be translated to an *IC* class, which contains a *context* attribute to specify an identifier in a context. Moreover, *IC* object may be executed at the same time to achieve the desired value. Therefore, each *IC* object is designed as a thread in this multithreading approach.

In order to indicate whether a value has been worked out, the *IC* class includes a flag *isReady*. All *IC* objects are placed in an array, each of which is indexed by a unique *IC* code.

When a value is demanded, the system creates an *IC* thread associated with it, and starts it right away. A demand is likely to request other values, and if so, it suspends, waiting to be notified if there are any values worked out by other threads. After being woken up, the thread checks the value it needs and either continues calculating if it's available or goes back to the *wait* status again if it hasn't been ready yet.

```
public synchronized int getValue (int h)
{
    while ( !entries[h].isReady() ) // the desired value is not ready
    {
        try
        {
            wait();
        }
        catch (InterruptedException e)
        {
            ...
        }
    }
    entries[h].setReady();          // set the ready flag
    notifyAll();                    // entries[h].notifyAll(), notify all waiting threads
    return entries[h].getValue();   // return computed value
}
```

FIGURE 5.6 PSEUDO CODE OF RETRIEVING VALUES

It is likely that there is more than one thread expecting the same value, and we expect only the exact processors that are waiting for the value can be notified with maintaining a list of waiters for the value. Unfortunately, Java does not provide a way of doing selective notification: it provides only *notify* for a single process and *notifyAll* for all waiters. Therefore, we can only notify each of the waiters individually. As an easier solution, the implementation invokes the *notifyAll* method. The method is synchronized to prevent more than one access.

5.4.3 Thread Pool

Each thread imposes a certain amount of overhead on spawning a new thread and the situation becomes worse on a heavily loaded processor, for this reason, the model of *wait and notify* cannot be accomplished if the system does not have enough resource to allow as many threads as needed. Unfortunately, Lucid is likely being used in massive scientific computations in which large amount of demands are needed. Hence, a thread pool is introduced which holds a certain given number of threads and assigns the task to each of these threads. As the threads finish with old tasks, new ones are assigned. This makes a program use a fixed number of threads, rather than continually creating new threads.

Implemented in Java, a thread pool contains an array of *workerThread* objects, each of which is a thread. The number of *workerThread* is determined by a programmer and can be adjusted according to the system resource. When a new *threadPool* object is created, all *workerThread*s it contains are initially paused. A *workerThread* object starts (not being created) when a task arrives and stops when finishes executing the task. If there are more tasks than available *workerThread*s, the tasks wait in a queue until one or more *workerThread*s are freed. Any class that implements the *Runnable* interface can be tasks and they are placed in another array, being picked up by an available thread. The picked-up order has big impact on the performance of a program that the data dependency between *IC* objects is preferably to be examined carefully at compile time to ensure those can be worked out have priority of being executed first. This corresponds to the data dependency analysis that was originally proposed for the compiler.

An empty task array does not guarantee that the execution finishes. The thread pool must also be checked to make sure there are no active threads that are executing tasks previously assigned. So the termination of the ultimate computing task is determined by two factors: all threads being idle and nothing in the task array.

The "thread pool" implementation is likely to be more efficient than the "unlimited threads" implementation because the overhead of creating and destroying threads is saved with assigning tasks to existing threads except the target program is rather small that only a few threads are needed. Our measurements confirmed that the thread pool implementation did in fact run faster than the unlimited threads implementation.

5.5 RMI

Experimenting with multithreading provides a means to understand the parallelism of Lucid programs. A Lucid program runs on multiple processes that are assigned by a single processor. However, this is not distributed processing. The next step is to run a program on several processors.

This section introduces the implementation of GEE with the RMI approach. It includes the comparison of distributed technologies provided by Java, the architecture of GEE and its software components.

5.5.1 Distributed Technology

Although many distributed technologies are available, given that the front-end of the compiler uses *Java CC* to construct the Lucid parser and Java is used as the implementation language in current GIPSY, the selection was done in the scope of distributed technologies provided by Java™.

Sun offers both Java RMI (Remote Method Invocation) and Java IDL (Interface Definition Language) for distributed computing.

The Java RMI [Sun] allows an object running in one Java Virtual Machine (JVM) to invoke methods of an object running in another JVM. It provides a simple model for distributed computation with Java objects. These objects can be either new Java objects or simple Java wrappers around an existing API. So RMI offers powerful features for remote object distribution in Java. The model has the following advantages: it

- is simple and easy to use;
- provides seamless remote invocation on objects in different VMs;
- is designed for reliable distributed applications;
- preserves the safety and security provided by the Java runtime system;
- can realize the callbacks from servers to clients.

Different from Java RMI, Java IDL provides a standard Java mapping from the CORBA IDL specifications that benefits applications involving other languages. In respect that the project is within the Java domain, RMI was chosen.

Another technology available is Java Jini™, which enables all types of services and devices to work together in a community organized without extensive planning, installation, or human intervention. Jini technology allows the lines to blur between what is hardware and what is software by representing all hardware and software as Jini technology-enabled "services," accessible either directly or through surrogates written in the Java programming language. In a distributed system of Jini technology-enabled services, these programs interact spontaneously, enabling services to join or leave the network with ease.

Jini has service producers, service consumers, and a service registry. It provides the natural infrastructure to support Web Services, which are reusable, componentized Web applications.

Jini is at a generation that communications are done at a level of "participant to participant" [Sun2001], which is marked by the merging of networks, enabling any client in any of the networks to find services offered by any participants in any other network.

Jini offers "network plug and play" that a device or a software service can be connected to a network and announce its presence, and clients that want to use such a service can then locate and call it to perform tasks. Although including new devices is not required in GIPSY, adding a software service flexibly would be attractive. However, this is not demanded under the current design and using Jini would increase the complexity of

GIPSY. Moreover, Jini is still at an experimental stage. Therefore, we think Jini is not suitable for our project at this stage.

5.5.2 GEE Architecture

The GEE prototype adopts a client-server model. Inherited from RMI technology, it can be layered into: application, proxy, remote reference, and transport layer (Figure 5.7).

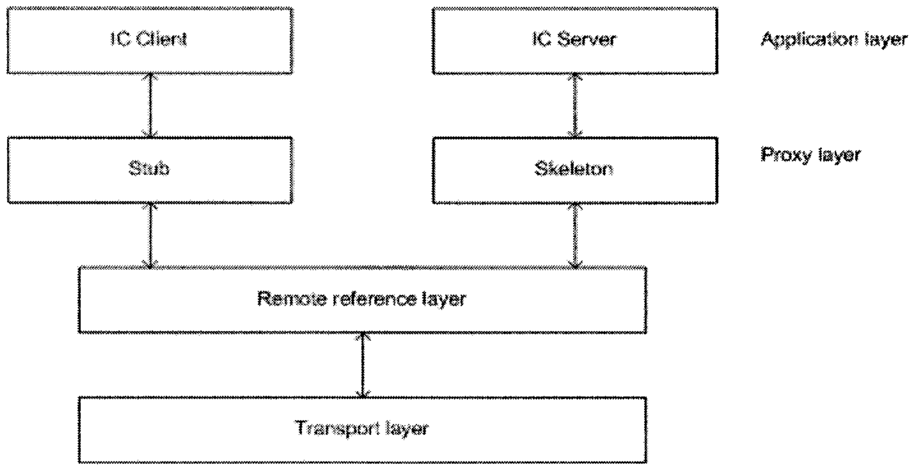


FIGURE 5.7 EXECUTION ARCHITECTURE

In this execution architecture, the *IC Server* keeps resource of demands and computed values whereas the *IC Client* contains all identifier objects. The IC client takes demands from the server, does calculations, returns computed values (back to the server) if they are available or generates new demands. Multiple clients can be started concurrently. In the current implementation, all clients are responsible to a single server, and do not talk to each other.

Application Layer

Both the *IC server* and *client* are at an application layer. The *IC Server* application implements remote interfaces used by the *client*: *valueHouse* (for computed values) and *demandList* (for propagated demands). It also exports the objects whose methods can be invoked remotely. In order to do this, it registers itself with the *rmi registry* where the *IC Client* looks up to get the reference to remote objects. The architecture is client-driven that a client takes demands from the server, does calculations and returns results to the server while the server is passive, being a resource holder.

A remote object registry is a bootstrap naming service that a RMI server uses on a host to bind remote objects to names with which clients can find remote objects and invoke their methods. The client then casts references as remote interfaces and applies methods.

Proxy Layer

The *Stub* and *Skeleton* are on the proxy layer. The stub is the clients' proxy for remote objects, which initiates a call to remote objects by calling the remote reference layer. It has to marshal arguments to a marshal stream obtained from the remote reference layer before informing the remote reference layer that the call should be invoked. When a value is returned, it unmarshals the returned value or an exception (if occurred) from a marshal stream and informs the remote reference layer that the call is complete. In contrast, the *Skeleton* is the server's proxy for the remote object. It unmarshals arguments from the marshal stream then dispatches actual methods. It also marshals returned values of the call or an exception (if one occurred) onto the marshal stream.

Remote Reference Layer

The remote reference layer is responsible for carrying out a specific remote reference protocol that is independent of the client stubs and server skeletons. It deals with the lower-level transport interface. Various invocation protocols can be used at this layer, for example, unicast point-to-point invocation, invocation to replicated object groups, support for persistence reference to remote objects and replication and reconnection strategies. This layer transmits data to the transport layer via abstraction of a stream-oriented connection.

Transport Layer

The transport layer is in charge of connections. After it sets up connections to remote address spaces, the layer monitors whether connections are alive or not and manages connections. It also listens for incoming calls and sets up one connection for each incoming call. Moreover, it locates the dispatcher for the target of a remote call and passes the connection to the dispatcher. At the same time, it maintains a table of remote objects that reside in the address space.

Distributed Object Model vs. Normal Object Model

The distributed Java object model shares a few things with the normal object model. For example, a remote object reference can be passed as an argument or returned as a result in any method invocation whether it is local or remote. A remote object can also be cast to any remote interface supported by the implementation. The Java *instanceOf* operator can be used to the remote interfaces supported by a remote object. However, the distributed Java object model is different from the normal object model in that clients interact only with remote interfaces and not with the implementation classes of the remote objects; and clients have to deal with additional exceptions and failure modes when invoking methods on an object remotely.

In the current implementation, the server acts as a passive resource holder being accessed by clients. However, in some cases, a server may need to make a remote call to a client. Examples include progress feedback, time tick notifications, warning of problems, etc. To accomplish this, a client must also act as a server. This can be achieved using callbacks.

The architecture (Figure 5.7) presents a single client and a single server. It can be extended to have more than one client without any changes. Each client is the same, capable of executing any functions and (at a first stage) we assume that they are equally powerful. The clients connect to server actively to find out if there are any demands expecting to be processed. For a specific client, it visits the demand list after it finishes executing a demand. And a client finishes when the demand list becomes empty. Therefore, clients are always busy, which ensures a balanced workload.

5.5.3 GEE Components

Figure 5.8 shows the class diagram of GEE.

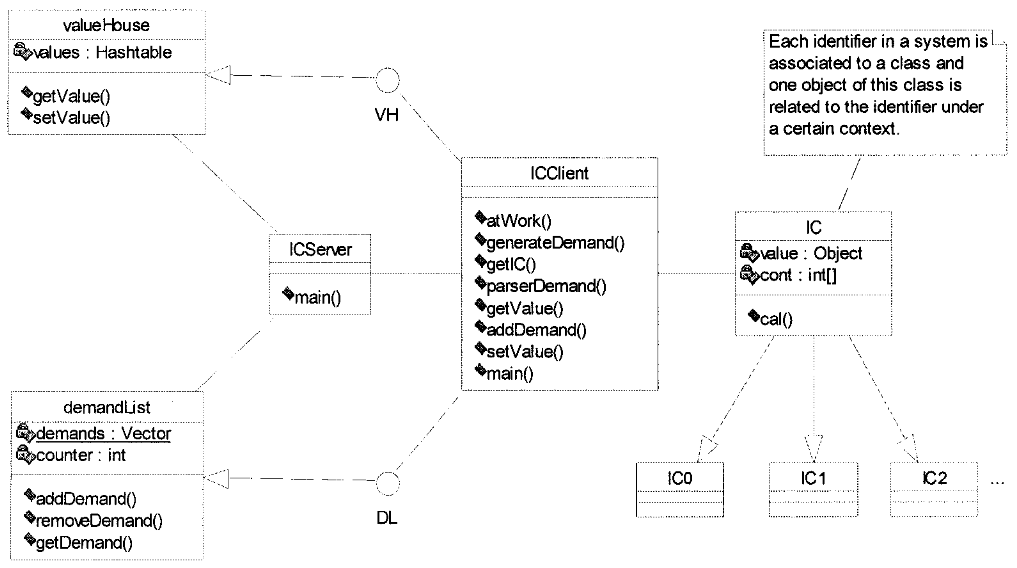


FIGURE 5.8 CLASS DIAGRAM OF GEE

In this diagram:

- *VH* and *DL* are interfaces, representing value warehouse and demand list respectively.
- The class *valueHouse* implements the interface *VH*; it has two synchronized methods: *getValue* for retrieving values from the value house and *setValue* for adding values into the warehouse.
- In the value warehouse, the values are currently arranged in a hashtable.
- The class *demandList* implements the interface *DL* and has 3 methods: *addDemand*, *getDemand* and *removeDemand*. The *addDemand* is used for inserting demands into a demand list *demands*; *getDemand* is for retrieving one demand from the list; (if the retrieved demand can be resolved) the method *removeDemand* is invoked for removing it from the list.

The client takes demands from the demand list until it is empty.

```
while (demandList != null )
{
        icObj = getIC ( demandList );
        icObj.cal();
}
```

FIGURE 5.9 PSEUDO CODE OF TAKING DEMANDS

Each IC class has a *cal()* method to calculate the desired value, whose pseudo code is shown in the following figure.

```
public void cal ()
{
    if (value_icObj2 is needed)                                   // another value is needed
    {
            icValue2 = valueWarehouse.getValue (icObj2);  // lookup the value
            if ( icValue2 == null );
            {
                    demandList.addDemand (icObj2);          // add a demand
            }
            else
            {
                    icValue = ...;                          //compute according to definition;
                    valueWarehouse.setValue (icObj, icValue); // add the value
                    demandList.removeDemand (icObj);          // remove the demand
            }
    }
}
```

FIGURE 5.10 PSEUDO CODE OF METHOD CAL()

One successful scenario of the process (the enquired value exists) is demonstrated.

1. The client obtains a demand from the demand list.
2. The client calls its *cal()* method to calculate the desired value.
3. The client needs another value and looks it up in the value house.
4. The value exists so the client continues the computation and achieves a value.
5. The client adds the achieved value into the warehouse, and removes the demand it has satisfied.

The steps repeat until the demand list becomes empty.

IC String

As stated before (Section 5.3), an instance of a class *IC* has the meaning of an identifier in a context, and it can be cited as: (1) exchanged information between a client and server, (2) a demand, and (3) a key of value house. It is demanded to design a suitable data structure, preferably consisting of a primitive data type, to replace the object. Thus, the simple data structure can be speedily transmitted over net, quickly generated, easily stored, and rapidly matched to an *IC* instance and for inquiries performed in the value warehouse. For easy understanding, a *String (IC string)* is employed in the current implementation.

First of all, identifiers in a Lucid program are assigned small sequential numbers. The same thing is applied to dimensions but a new sequence is declared. The numbers shown in Table 5.1 are issued to the expressions (*A*, *x*, *y*, *w*, *v* are identifiers, whereas *d* is an dimension) in the example *simplified-Hamming* code of Figure 5.2.

Name	Number
A	0
x	1
y	2
w	3
v	4
d	0

TABLE 5.1 RELATIONS BETWEEN A NAME AND NUMBER

The numbering is done by GIPC and code generator serves as a layer separating GEE from GIPC; therefore, names are never referred after code generation. The code generator produces a class *IC0* for identifier A, *IC1* for identifier x, so on and so forth.

Secondly, the context for each identifier should be considered. As stated in section 2.4.3, an identifier is likely to vary according to *part* of the context space, called *rank*. Not only identifiers can vary on different dimensions, but also they can vary on a different number of dimensions.

The IE4 compiler [Jay1999] uses a meta-demand-list method and only includes dimensions in the rank of an identifier. Thus different identifiers will in general have different dimension lists. This solution is likely to be slow because lists have to be traversed and it takes time to decode which dimension a value is associated to.

Another solution is to include all dimensions for each identifier, regardless of whether a dimension is effective to the identifier or not. In this solution, an *ic string* can be designed as "*id: val_0: val_1: val_2...val_n*", which starts from an identifier number, followed by a sequence of values of each dimension, which are separated by ":". As an example, the *IC string* for *A @.d 4* is "*0:4*". The example is simple because the program has only one dimension. Suppose a new dimension t (numbered as 1) is introduced to the program and A does not vary on the *t* dimension, the representation of A is as "*0:4:?*", in which "*?*" is the value of dimension *t*, The "*?*" also implies the value on this dimension doesn't matter.

For simple problems, most identifiers will probably have much the same set of dimensions therefore storing all of the dimensional should not be a large overhead. For complex problems, it is possible that there are many dimensions but that most identifiers only need one or two of them. In this case, storing all dimensions in every key might waste a lot of space, cause more time to retrieve, and result in a bigger volume of network traffic.

5.5.4 Thread Usage

In a RMI system with multiple clients, the JVM on the server makes the usage of thread technique transparent to a programmer. It executes calls originating from different clients in different threads; executes some calls originating form the same client in the same thread and others in different threads. In a word, it makes no guarantees with respect of mapping remote object invocations to threads (other than the calls from different client VMs).

5.5.5 Object Serialization

In the distributed system, variables (IC objects) are transmitted over the net in the form of demands or queries of values. Moreover, since GIPSY is designed to take functions (written in Java) to do the computation part, it is necessary to discuss the transforming of objects on the net, which could also be helpful for the implementation of communication procedures.

The transparent transmission of objects from one address space to another is supported through marshalling/unmarshalling arguments. Marshalling is transforming an object stored in the memory to a format suitable for being stored in an external storage. Then, the formatted data is transmitted over the network. Later on, unmarshalling is operated to restore the object. It has to be ensured that the receiver understands the received data completely and correctly.

Marshalling Problems

Three major problems may occur with marshalling: (1) basic data types may be represented differently on different platforms; (2) complex data types may be implemented differently by the run-time environment of programming languages; (3) different kinds of data types may be mixed.

First, basic data types may have different data representation formats on different platforms. For example, HP, IBM and Motorola 68000 systems store multi-byte (e.g. long integers) values in *big-endian* order, while Intel 80x86 and DEC VAX systems store them in *little-endian* order. The *big-endian* model stores the high-order byte at the starting address while the *little-endian* stores the low-order byte at the starting address.

Secondly, a user may require transforming not only various but also complicated data types: numbers (both integers and floats), strings, lists, records, hash tables and even self designed ones.

Thirdly, this data may be a mixture of several different kinds of data types and a programmer does not know exactly, at what time, what kind of information will be manipulated, and how this data is mixed, and what architecture/OS will be used.

As a result, data may have to be converted when a system transfers object request parameters and results from one host to another. This transformation requires type information therefore it has to be done before request parameters are marshaled.

Solutions

There are three principle ways [Emm2000] that the problem can be resolved: (1) the data conversion can be performed implicitly by middleware that utilizes a standard data representation; (2) it can be resolved explicitly at the application level by declaring the type of data in a different way; (3) it can be resolved by the platform on which the middleware executes.

In the first solution, the middleware performs the mapping of data representations specific to a host or programming languages to standard data representations.

Several different forms of standard data representations have been proposed and are being used with different object-oriented middleware. As an example, the General Inter-ORB Protocol of the CORBA standard defines a Common Data Representation (CDR) that is used for those object requests that cross the boundary object request broker between the two requests. The CORBA CDR defines two mappings for atomic types that represent numbers. The first one is for *big-endian* representations and the second one is for *little-endian* representations. CORBA implementations declare during request marshalling whether they use *little* or *big-endian* representation. This has the advantage that the computation intensive reversing of byte orders is not necessary when requested data is sent between two *little-endian* or two *big-endian* hosts.

Resolving the data heterogeneity by a middleware requires the middleware to understand the construction of the complex data structures that are transmitted during requests, which some application developers are not in favor of. In this case, the domain-specific languages may be used to structure complex application data.

XML (eXtensible Markup Language) is an example used for exchanging structured information over the Internet. By definition, any XML document is a stream consisting only of ISO 10646 characters. This definition facilitates the exchange of data in a heterogeneous setting. In order to facilitate the exchange of other atomic data types, such as integers and floating-point numbers, distributed components transform them to and from a Universal Character Set (UCS) representation.

The data heterogeneity problem can also be resolved by introducing a platform above the operating system that is not concerned with distributed communication but entirely with interpretation of code and data in a standardized form. Java Virtual Machine is such an example. The JVM specification resolves data heterogeneity by precisely standardizing the representation of Java's atomic data types, type construction primitives and object references. Hence all data that are passed as parameters to Java remote method invocations have the same presentation, by specification, even if they are executed within two separate virtual machines.

Java Object Serialization

Java accomplishes the job of marshalling/unmarshalling in terms of object serialization. Object serialization provides a program with the ability to read or write a whole object to and from a raw byte stream. It allows Java objects and primitives to be encoded into a byte stream suitable for streaming to some type of network or to a file-system, or more generally, to a transmission medium or storage facility. Objects may be placed into external streams or byte arrays by implementing the *Serializable* or *Externalizable* interface, or by invoking methods of *ObjectStream*.

A detailed description of Java object serialization can be found at "Java object serialization specification" [Sun1997] and a few points are highlighted as the followings:

- The information of an object: class, fields, and data are represented separately.
- A string is coded in UTF and contains up to 65535 symbols.
- All primitive data written by classes are wrapped into blocks, which are stored into block-data records.
- Serializing Java objects is designed to maintain the Java object type and safety.
- Currently, the Java serializer which converts a Java object to a byte stream supports only these kinds of objects: *byte*, *char*, *double*, *float*, *integer*, *long*, *short*, *boolean*, *array*, and *object*.
- Java serializer uses a common way to represent a basic computer data (numbers, strings, float numbers).
- The formats of serialized data depend on JVM and serializer implementation; they are practically useless without Java environment. The size of a stream would be increased at the expense of the service information and the time of processing a serialized stream would also be increased outside the Java environment.

- The Java serializer has the advantages of checking and verifying loaded code (which are guaranteed by JVM).

5.6 Value Warehouse

The value warehouse keeps track of the computed values along with their associated identifiers and contexts. Adopting the value warehouse is likely to improve the system performance.

The first draft of GEE did not include a value warehouse, and it was soon proved to be intolerably slow.

5.6.1 Content

The warehouse contains pairs of a variable and its value.

5.6.2 Efficiency

Since warehouse inquiries are very frequent, the warehouse must be very efficient so that the performance of the whole system is satisfactory.

One way of achieving efficiency is to arrange the warehouse as a *hashtable*, which maps *IC strings* (key) to *values* (value).

A *hashtable* stores data in an array. The index into the array for a given element is calculated by applying a hash function to a key that is all or part of the inserted elements. A *collision* occurs if two keys hash to the same index. In this case additional elements with the same hash value are chained off in a linked list from the initial element, which is one of several ways of handling collisions. In this chained-off way, if a large number of collisions occur, a hash table degenerates into what is essentially a linked list. A well-chosen hash function results in a low incidence of collision; consequently, results in a high performance. Increasing the size of a hash table also results in a low incidence of collision. Under light to medium load the hash table performs very well.

Java provides class *Hashtable*, which has two parameters that affect its performance: *initial capacity* and *load factor*. The *initial capacity* is the number of buckets in the hash table is created. In the case of a collision, a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, its capacity is increased by calling the *rehash* method.

The implementation of a value warehouse can be easily achieved by declaring a variable of the Java class *Hashtable*, and its performance can be tuned. However, it would be more powerful and flexible to design a hash function mapping from an *IC string* to a unique number and declare the number as the index of an array. Consequently, the access to the associated value would become extremely efficient.

One possible solution is to restrict the number of identifiers (the number could of cause be configured) and give the value of each dimension a fixed maximum length. For example, *A @.d 4* can be coded as " 000 0004" if the number of identifiers is limited to be less than 1000 and the value of dimensions can not exceed 10000. This code would be unique and easily generated but the number representation for the "?" mark should be carefully chosen. However, if the warehouse is designed as an array indexed by the numbers generated in this way directly, a sparse array will be resulted in and lots of space will be wasted. And it might even not be practical to restrict the length of the values of dimensions although the number of identifiers can be known through the front-end compiler and there are existing algorithms dealing with sparse arrays.

Therefore, we have two choices: one is to design a hash function and the other is to use a sparse array and shrink it properly. Because the second approach needs to restrict the number of identifiers and dimension values, we focus here on designing a hash function.

The hash function: *H(x)* takes an input *IC string*: *x* and produces a hash code. The basic requirements for this hash function are:

- Input strings have varied lengths although they include all possible dimensions because the integer value of a dimension may have different number of digits.
- The output hash code has a fixed length; the best sizes are powers of 2. A conventional hash function uses the *mod* operation of a prime to limit the output size. However, *mod* should be avoided in *H(x)* because it is comparatively slow.
- The returned hash code should have a reasonable length that the resulted array will not cost a lot of space.
- *H(x)* should be free of collision, or have a very low rate of collision. This requires that data should be dispersed as randomly as possibly across the hash table to minimize the chances of a collision.
- *H(x)* should be relatively easy to compute for any given x.

Considering the IC strings are made up from small integers, a simple way is to add up the integers and (if necessary) then restrict the result into a scope. Based on this thought, a tiny program is developed with randomly generating 3 integers and adding them together. The test result shows the collision rate is unbearably high 18%, with an average rate of 10%.

Here we introduce a method designed by Bob Jenkins [Jen1996] that hash a variable-length key into a 32-bit value: *ub4-hash(k,length, initval)*, where ub4 is the type of unsigned 4-byte integers.

The hash function works well in GEE. It costs much less collision than the above add-up method (claimed to be one in 2^{32}). It's fast too. The operation adopted is shiftting and mixing is done on three 4-byte registres rather than on 1-byte.

The hash function is general and can deal with keys being character strings, numbers, bit-arrays, or anything else because it treats keys bit by bit without taking into account their representation. It also handles keys with variable lengths. These are definitely strong points for a general hash function. For our needs, it would be even better to tailor the function: IC strings consist of integers and have fixed lengths. Therefore, we modified the function and its complete source code is shown in the Appendix. The function maintains the low-collision rate and the efficiency of the original one.

5.6.3 Value Inquiry

When an input inquiry matches a key exactly, the warehouse returns the matched value.

In addition to this kind of absolute match, the intensional warehouse may support a partial match that happens when an input inquiry has fewer dimensions than the key defined. In other words, the value in the warehouse is correct for all values of the missing dimensions. For example, the warehouse has an value A @.(d =4, t = 1) = 2. The warehouse returns 2 for an inquiry of A @ (d=4). Since the rank is a set of all effective dimensions for a variable, any request for a dimension outside its context space should be disregarded.

5.6.4 Maintenance

Since Lucid is intended for large-scale scientific computation, it is likely that massive calculations are needed that will probably cause the value warehouse to grow fast. Therefore, the value warehouse needs to be swept periodically. Common strategies include removing the earliest generated values, the least used, or the ones that take the shortest time to be worked out. In addition, the dependency information obtained by static analysis of the source code could also be used: values could be discarded when they are no longer needed for other computations. These strategies could also be combined. Whichever strategy or what combination is used, it requires GEE to supply with additional information, such as the time a value is produced, the number of times a value is requested, how long a value needs to be worked out.

5.7 Interface between GEE and RIPE

The interface between GEE and GIPC is quite straightforward in that GIPC feeds GEE with an *abstract syntax tree* and a *dictionary* and expects nothing to be returned by the GEE.

The interface between GEE and RIPE is comparatively flexible. Taking into account of what to be displayed graphically or what interference can be imposed from a console, the interaction between GEE and RIPE can be extremely complex. However, exchanged information can be ranked according to importance.

The graphical representation of the generated demands and/or contents in the warehouse is definitely required to verify if the result is correct. On the contrast, the interference from the environment other than for the purpose of debugging a program is not recommended since it would affect the performance greatly.

6 Analysis and Discussion

This chapter reports the testing results, reviews the three approaches adopted in GEE implementation, analyzes their results and addresses two important issues: performance and resource management. Moreover, it discusses the possibility of scaling up of the distribution into the worldwide web and a way to balance workload among hosts.

6.1 Testing Report

We used a bar temperature application written in Lucid to test the implemented prototype on both a cluster of PCs and a single machine. Testing results are reported in this section.

6.1.1 Tested Application

The experimented program finds an approximate temperature distribution for a bar that is heated at one end using relaxation. Its one- dimensional version is shown in the following figure.

```
temp @.T Tmax @.X Xmax
   where
      dimension T
      dimension X
      temp = if #.T == 0
               then
                  if #.X == 0
                  then Tinit
                  else 0
              else A * px + B * x + A *nx
      px = if #.X == 0 then 0 else temp @.T (#.T-1) @.X (#.X-1)
      x  = temp @.T (#.T - 1)
      nx = if #.X == Xmax then 0 else temp @.T (#.T-1) @.X (#.X+1)
      Tmax = 100
      Xmax = 100
      Tinit = 100
      A = 0.4
      B = 2 * A - 1
```

FIGURE 6.1 TEMPERATURE DISTRIBUTION FOR A BAR (ONE DIMENSION)

The application can also be easily made bigger to a square of 2 dimensions (Figure 6.2).

```
temp @.T Tmax @.X Xmax @.Y Ymax
   where
      dimension T
      dimension X
      dimension Y
      temp = if #.T == 0
               then
                  if #.X == 0 && #.Y == 0
                  then Tinit
                  else 0
               else      A * ny + A * px + B * x + A *nx +  A * py
      x  = temp @.T (#.T - 1)
      px = if #.X == 0 then 0 else temp @.T (#.T-1) @.X (#.X-1)
      nx = if #.X == Xmax then 0 else temp @.T (#.T-1) @.X (#.X+1)
      py = if #.Y == 0 then 0 else temp @.T (#.T-1) @.Y (#.Y-1)
      ny = if #.Y == Ymax then 0 else temp @.T (#.T-1) @.Y (#.Y+1)
      Tmax = 100
      Xmax = 100
      Ymax = 100
      Tinit = 100
      A = 0.2
      B = 4 * A - 1
```

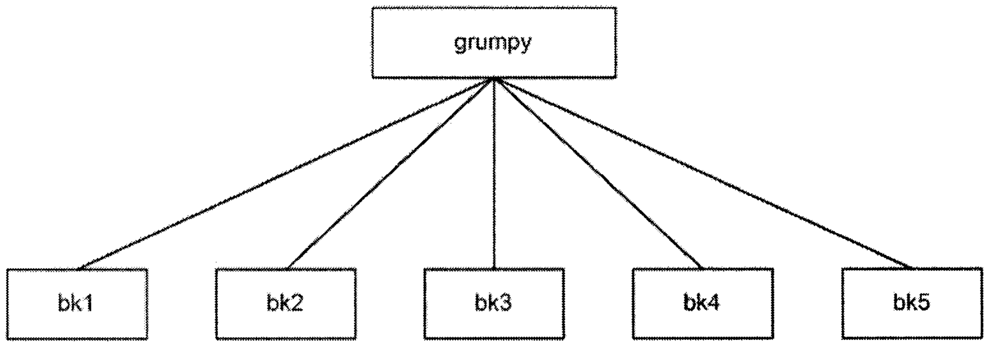
FIGURE 6.2 TEMPERATURE DISTRIBUTION FOR A BAR (TWO DIMENSIONS)

The one-dimensional version is used in the following tests.

6.1.2 Testing on a Cluster

We execute the bar temperature program (one dimensional version) on a cluster of computers to compare the performance of using different numbers of processors working on the same task.

The cluster has one server, named *grumpy*, and five processors, *bk1*, *bk2*...*bk5*. The system configuration of the cluster is shown in the following figure.



Component	Model and Specification
grumpy	Intel(R) Xeon(TM) CPU 2.66GHz
	512 MB RAM
bk1	Intel(R) Pentium(R) 4 CPU 1.60GHz
	1.2 GB RAM
bk2	Intel(R) Pentium(R) 4 CPU 1.60GHz
	512 MB RAM
bk3	Intel(R) Pentium(R) 4 CPU 1.60GHz
	1.2 GB RAM
bk4	Intel(R) Pentium(R) 4 CPU 1.60GHz
	512 MB RAM
bk5	Intel(R) Pentium(R) 4 CPU 1.60GHz
	1.2 GB RAM
Software	
Software	Linux version 2.4.18-14

FIGURE 6.3 SYSTEM CONFIGURATION OF THE CLUSTER

Figure 6.4 shows the testing result of using a cluster of processors. The numbers on X-axis represent the integer values of T and X; and the Y represents the time spent on computing the value under the corresponding context, which is with a unit of second.

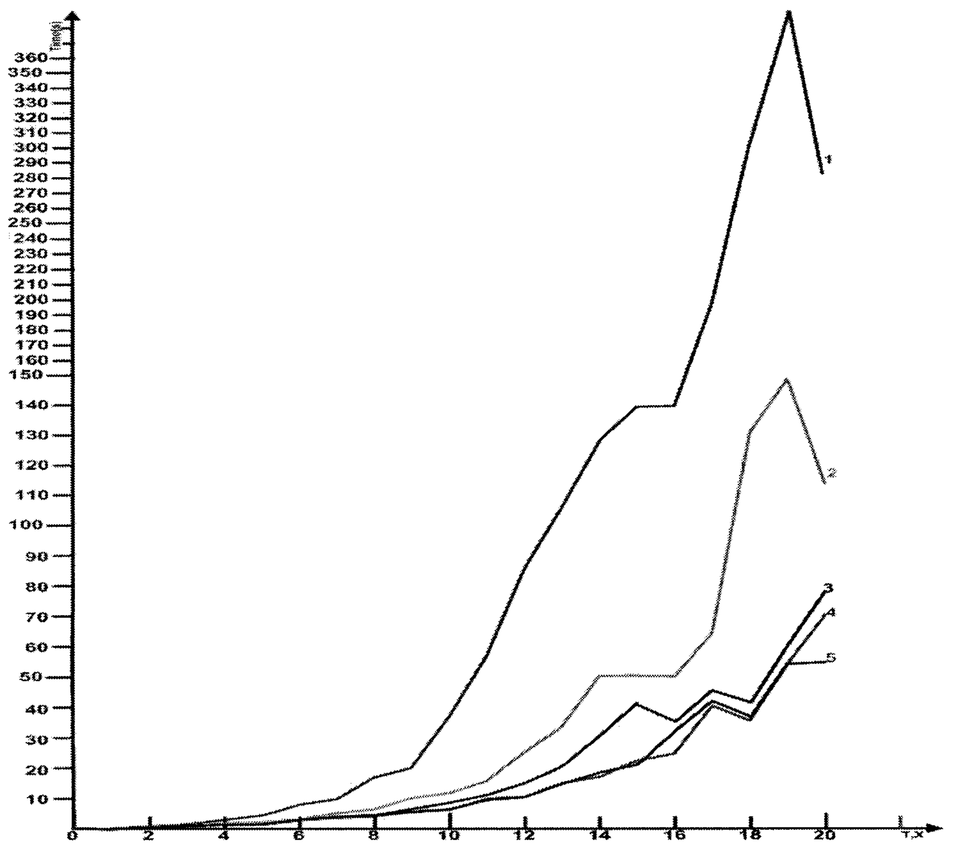


FIGURE 6.4 TEST RESULTS USING 1-5 PROCESSORS

Each curve is tagged with a number, which presents the number of processors that is in use. For example, the curve 1 shows the testing results of using 1 processor; curve 2 is of using 2 processors, so on and so forth.

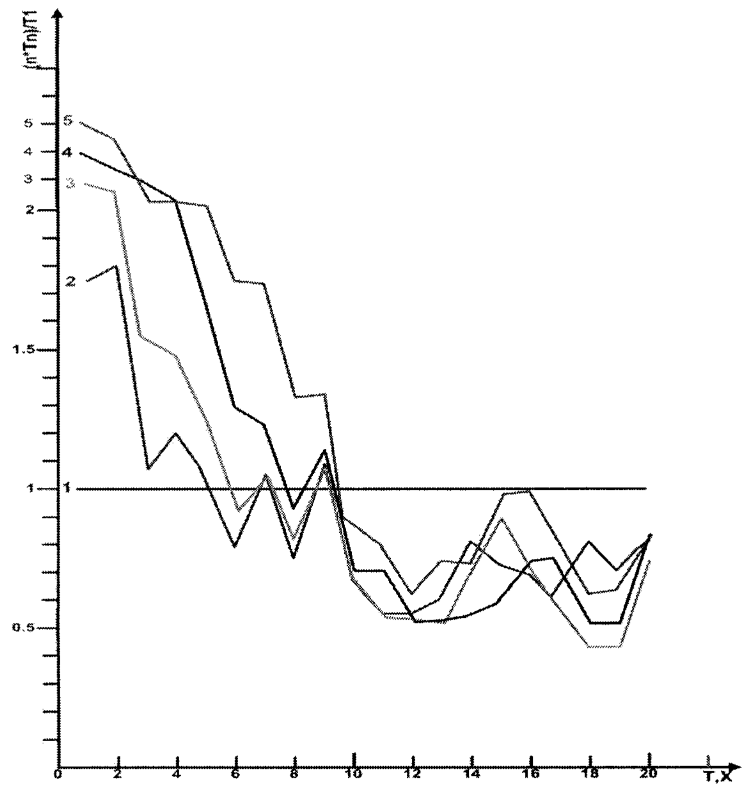


FIGURE 6.5 RELATIVE PERFORMANCE USING 1-5 PROCESSORS

The same results used in Figure 6.4 are expressed in Figure 6.5 but in a different way. The integer numbers on X-axis are the T and X ranges and the Y-axis represents the "relative performance", measured as *(n * Tn)/T1*, where *Tn* is the time taken by *n* processors and *T1* the time taken by *1* processor. Each curve is also tagged with a number at the beginning place of a curve, which presents the number of processors that is in use.

If there was no overhead, *(n * Tn)/T1* should be a horizontal line at height *1* because *n* processors should work *n* times faster than *1* processor. Normally, the value will be less than 1, and gives a measure of efficiency. However, we can observe that when the values of T and X are small, the relative performance is more than 1. This is because the results can be computed very quickly in these cases. The work is done even before more processors start to share the work. A closer look of using one processor and two processors is show in Figure 6.6.

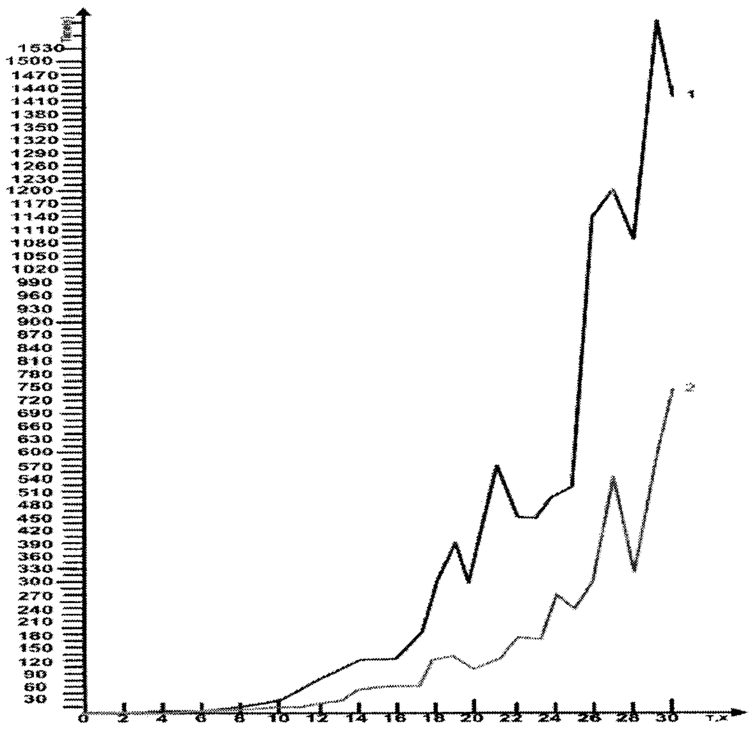


FIGURE 6.6 TEST RESULTS USING 1 AND 2 PROCESSORS

6.1.3 Testing on a Single Machine

We used one PC to run the application implemented with the three prototypes: interpretation, multi-threading (with a thread pool) and distribution.

The single processor is of Intel Pentium 4 CPU 2.00 GHz, 256 MB RAM and equipped with Microsoft Windows 2000.

The testing results are separately shown in the following figures (Figures 6.7, 6.8, and 6.9) because their performance varies seriously. The time unit used in the interpretation is millisecond while the units in the other are second.

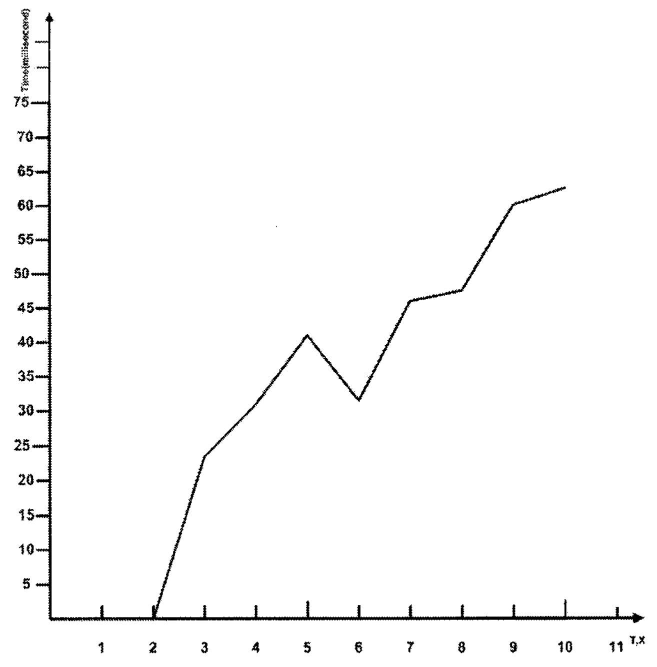


FIGURE 6.7 TEST RESULTS OF USING INTERPRETATION

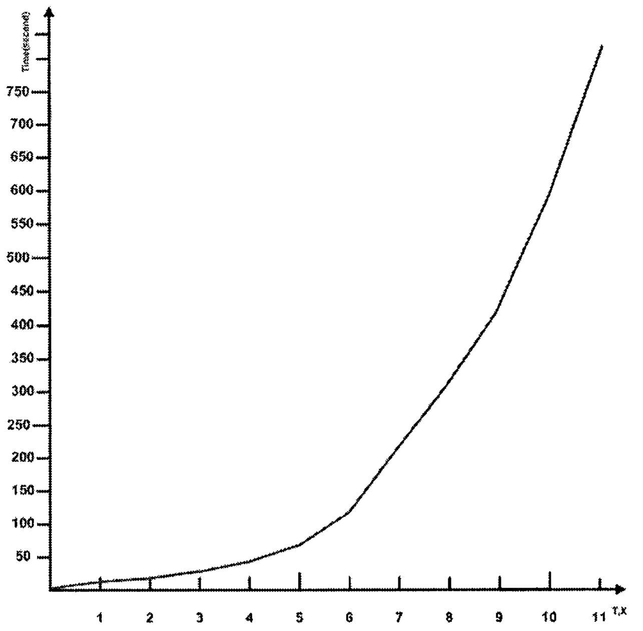


FIGURE 6.8 TEST RESULTS OF USING MULTI-THREADING WITH A THREAD POOL

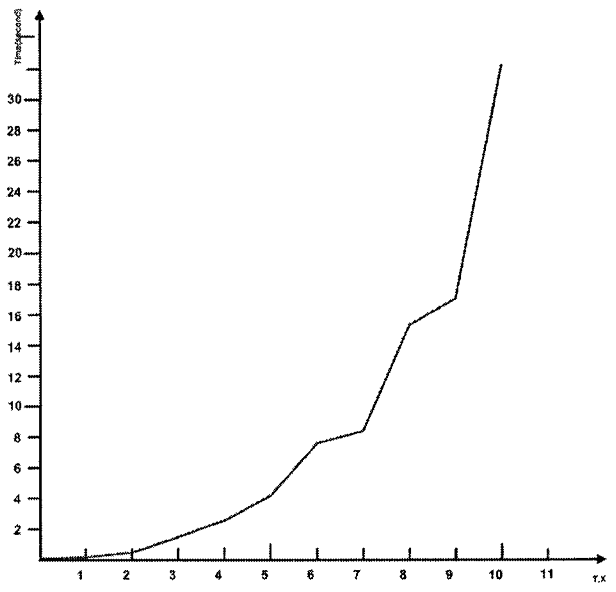


FIGURE 6.9 TEST RESULTS OF USING DISTRIBUTION

The distribution prototype is tested on this single machine, on which two terminals are started.

The result shows the interpretation is the most efficient for this application, and the multi-threading with a thread pool is the least efficient while distribution stands in the middle.

This outcome seems in conflict with the theoretical analysis we did before, which indicates the interpretation prototype should be inherently slow and worse than the multi-threading and distribution. We analyze the reasons are as followings. First, the design of distribution and multi-threading mainly aims to present the ideas so that efficiency is not carefully considered. This is especially overlooked in the design of the thread-pool management, which limits the number of threads simply by starting and terminating threads frequently when the number reaches a pre-defined fixed integer and this number is not optimized at all. Consequently, slow implementations based on these design are unavoidable. Secondly, time is wasted on big volume of exchanged information for distribution and on pool management for the multi-threads. Finally, in the tested application, values of variables *px*, *x* and *nx* all depend on the values of the variable *temp* thus the application has high data-dependency which is not very suitable for being distributed. Therefore, for this application and with the current implementations, we received the above results.

We also find the system runs out of memory when calculating T and X have values of 151 when using the interpretation approach and the performance of the multi-threading is not stable. We suspect it is due to how the running environment manages the resources, including memory and threads.

6.2 Result Analysis

The GEE component interprets programs written in Lucid on top of the AST generated by GIPC. Whether the generated Java code is further processed by being translated to Java Byte Code or being compiled by a Java compiler (see Section 4.4.2) then executed is out of the scope of this thesis.

The interpretation has been performed with three different approaches: interpreted, multi-threaded and distributed. The interpretive approach operates on the input syntax tree in a straightforward way, while the other two approaches translate the tree's branches (that define identifiers) into objects.

With a closer look to the three approaches, we distinguish the multi-threaded and distributed approaches from the interpretive one according to the difference between interpretation and compilation. Moreover, the multi-threaded approach is distinguished from the distributed according to the different communication methods adopted: memory sharing and message exchanging.

6.2.1 Interpretation vs. Compilation

The execution of a program on a machine written in *high level programming languages* needs a "translator" since the only language a computer understands is the so called *machine language*. A machine language is composed by a set of basic operations whose execution is implemented in the hardware of the processor. On the contrast, high-level programming languages provide a higher and machine-independent level of abstraction than the machine language and are therefore more adapted for a human-machine interaction. But this also implies that there must exist a sort of translator between a high level programming language and a machine language. There exist two kinds of *translators*: interpreter and compiler.

An *interpreter* is a *program* that implements or simulates a *virtual machine* using a base set of instructions of a *programming language* as its *machine language*. In other words, an interpreter is a *program* that implements a library containing the implementation of the basic instruction set of a *programming language* in *machine language*. An interpreter reads statements of a program, analyzes them and then executes them on the virtual machine or calls the corresponding instructions of the library.

Different from an interpreter, a *compiler* translates a program from the language in which it was written, into machine code, which is also called *object code*. This machine code is then (usually) written out to an executable file. When the file is run, the computer executes the machine-code instructions sequentially written in the file.

A *compiler* predicts the static behavior of a program and separates *execution* from *translation* of a program. With compilation, the *source code* is translated only once. Also the *object code* is *machine-dependent*, i.e. a *compiled* program can only be executed on a machine for which it was compiled, whereas an *interpreted* program is not *machine-dependent* and the *machine-dependent* part is in the interpreter itself.

An interpreted program usually runs more slowly than a compiled program because it has quite a high overhead to translate a program line by line and repeats the same work for the same statement that has to be executed several times, e.g. statements inside a loop. However, because the original program is still being worked on, debuggers are easier to write and no separate executable file need be stored.

The advantages of compiling are that codes run quickly because they are already in machine-code form and no translation is required while the program is being executed. Likewise, because machine-code is fairly compact, the resulting executable program can often be small. The disadvantages are that compiling a program can be somewhat lengthy since the entire program must be compiled. Also, the original program is no longer connected to the machine-code, and mapping the machine-code back to the original program can be very difficult (therefore, writing a debugger becomes difficult). An example of this is that run-time errors in a compiled program are usually very general and often don't indicate where the error occurred. This is because information like line numbers are lost during compilation. In addition, processors from different vendors usually have different machine codes and thus require completely different compilers.

The third kind of translators can be regarded as a combination of interpretation and compilation that tries to combine the advantage of more effective execution of compiled code and the advantage of machine-independence of interpreted code. This concept is used by the Java programming language and JIT compilation (see Section 4.4.2) for example.

In this combination method the *source code* of a program is translated by a compiler to a sort of *object code*, also called *byte code* that is then executed by an interpreter implementing functions of a *virtual machine* that uses this *byte code*. The *byte code* execution is faster than the interpretation of the *source code*, because the major part of the source code analysis and verification is done during the compilation step. But the *byte code* is still *machine-independent* because the machine-dependent part is implemented in a *virtual machine*.

The interpretation method used in GEE follows a typical interpretative way. In this approach, the interpreter traverses an abstract syntax tree and interprets nodes in the tree by taking appropriate actions for each note it encounters and using the result of node evaluation to determine the next node to evaluate recursively.

The approach may work efficiently on small and simple applications but is generally slow because it explains the meaning of each node every time it encounters even if it might have met the same kind of nodes without recording the work it has done before.

Further, it is difficult to apply distribution with the interpretation approach. For example, it can be analyzed what parts in a recursive program can be executed concurrently. In the evaluation rule:

$$eval\,(x \oplus y,\, c) = eval\,(x,\, c) \oplus eval\,(y,\, c),$$

$eval(x,c)$ is evaluated independently from $eval(y,\, c)$, thus the two can be executed in parallel. The realization of parallelism might not be practical because there may be too much concurrency. Each statement above creates two processes, one for each function call. This will lead to a large number of processes if the depth of recursion is great; and perhaps too many of them are executed in parallel. A solution to this problem is to prune the recursion tree and switch from using concurrent recursive calls to sequential recursive calls when there are enough processes. Because the recursive parallelism is not the topic of this thesis and wrapping identifiers into classes is object-oriented that provides more space to discover the feasibility and advantages of distributed computations, we did not go along this direction further once we got enough understanding of how a Lucid program works using this approach.

However, a Lucid interpreter might be useful as the basis for a debugging tool.

Instead of operating on the syntax tree, both the multi-threaded and distributed implementations convert each identifier to an object. Thus, the work of translating according to different node types is pushed forwarded to be done at the compile time and is being performed only once. In this sense, the two approaches are considered as being done in a compiling way. However, since the system adopts a lazy evaluation that requires a value to be evaluated only when needed that the evaluation relies heavily on the runtime system. As a consequence, the compiler is not allowed to predict many things so that the programs compiled with these two approaches are probably less efficient than conventional compiled programs adopting eager evaluation strategy. It is also a reason that the execution wouldn't be very efficient. A solution for this problem is proposed in the performance improvement part of Section 6.3.1.

Since the execution is highly coupled with the translation of the program, it is difficult to separate one from the other as designed in the initial proposal.

The initial design proposes to separate the execution to two stages. First, only demands are propagated according to the computation units and data dependencies that are indicated on the syntax tree. The run-time architecture is not considered at this stage. Second, information about the run-time architecture is used to build a system-dependent module that is then integrated with the computation part to form a complete execution schema. This is an application of the principle of "separation of concerns". The application is developed in a stepwise manner, only one concern being handled at a time.

The initial execution schema looks attractive. Unfortunately, it is very hard to do due to the fact caused by lazy evaluation that many computations cannot be determined until at execution time. To handle various runtime architectures separately and dynamically, a resource management module (see Section 6.4) could be adopted at a meta-level. In this model, a better flexibility is achieved.

6.2.2 Shared-memory and Distribution

The multi-threaded approach in this thesis uses shared-memory method. The shared-memory method can be used in one process or multiple processes (usually 2) that share one memory. In this approach, there is only one process in which multiple threads run: a single address space is assigned to this process and all threads work within the space, communicating with shared variables.

In a distributed architecture, processors have their own private memory and they interact using a communication network rather than a shared memory. Hence, processes cannot communicate directly by sharing variables. Instead, they have to exchange messages with each other.

The Shared-memory model is commonly used on shared-memory machines on which variables can be directly accessed by each process. In contrast to shared-memory, the distributed memory model can be found in distributed-memory multi-computers as well as networks of machines. In addition, hybrid combinations of shared-memory and network architectures are sometimes employed, such as a network containing workstations and multiprocessors.

The Shared-memory model can also be used on distributed-memory machines if the machine is supported by a software implementation, which is called a distributed shared memory (DSM). DSM is a technology with which processes have an illusion of accessing a single shared memory though the physical memory is distributed. Processes access DSM by reading and updating what appears to be ordinary memory within their address space. However, an underlying runtime system ensures that processes executing at different computers observe the updates made by another and this is done transparently.

DSM frees a programmer from the concerns of message passing when writing applications that might otherwise have to use. It is primarily a tool for parallel applications, and distributed applications, and groups of applications in which individual shared data items can be accessed directly. Yet, in general, it is less appropriate in a client-server system, where clients normally regard server-held resources as abstract data and access them by requests (for reasons of modularity and protection) although servers can provide DSM that is shared between clients. Even with the application of DSM, message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM system also manages replicated data: each computer has a local copy of recently accessed data items stored in DSM for the sake of access speed.

6.2.3 Internet Distributed Computation

Although the current predominant use of internet is informaton retrieval, due to the fact that many Internet hosts have a low CPU usaguage most of the time, in a long term, the unused power will probably become a resource to make Internet play a significant role in distributed computing.

It was postulated in [MRKB2003] that the combination and extension of web servcies, peer-to-peer (P2P) systems and grid computing provides the foundation for Internet Distributed Computing (IDC).

A *Web Service* [W3C2002] is a software application identified by a URL, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet-based protocols. In other words, web services are self-contained and loosely-coupled small applications that can be located and used.

P2P provides a distributed computing paradigm in which participating machines usually act as equals that both contribute to and draw from the resources shared among a group of collaborating resources nodes. P2P has been used for applications such as file backup, content distribution, and collaboration — all without requiring centralized control and management.

Grid computation extends conventional distributed computing by facilitating large-scale sharing of computational and storage resources among a dynamic collection of individuals and institutions. The term *grid* comes from the notion of computing as a utility and it is analogous to a power grid that is a pool of resources aggregated to meet various input demands without users' awareness of or interest in the details of the grid's operation.

Several projects have sprung up that use the Internet for distributed computing. Anyone who uses the Internet can participate in lending their computer's free time to help solve a complex problem. The most popular distributed computing project is Distributed.net [Dis] which tries to break encryption. Another project is the Great Internet Mersenne Prime Search, which seeks to discover very large prime numbers and participants in PiHex [PiH] are using their collective computing power to calculate the forty trillionth bit of pi. A project called SETI@home [SET] uses distributed computing to search for extra-terrestrial intelligence by analyzing data captured by the world's largest radio telescope. To participate, a PC user downloads a program for the project that he/she wants to work on. The program works on a portion of its task — whether it's helping to compute pi, cracking encryption, or whatever — then the result (which may take days to compute) is submitted (either automatcially or manually done by the PC user) to a central Internet server.

Complexities in security and discovery infrastructure while preserving local autonomy have been proved to be significant barriers for building distributed applications that exceeded organizational boundaries. Lack of adequate management tools also limited the scale and adaptability of distributed systems.

The maturity of Internet and its prospected increasement of popularity have led the GIPSY people to consider the necessisity and feasibility of scaling up the distribution of GEE execution over the Internet.

It would be beneficial to run a Lucid application by hundreds of thousands of computers in a world-wide environment. First of all, the problem with which a Lucid application deals may be very large, requiring a lot of computing power to solve hence it might not be worked out by any one computer or person or a small group of computers and users in a reasonable amount of time. Moreover, GIPSY is a research project, which surveys current availabe technologies and experiements them in an attempt to benefit both the development of the technology and the progress of the GIPSY project. Consequently, integrating IDC in GEE component has research values. Finally, the GIPSY project is intended to become an open source whose all resource (paper, presentation, source code, etc.) are accessible (read only) by public. Therefore, with including IDC as an experimental research direction, more people would be interested and have a chance to get involved. This would be helpful to expand its usage and indirectly encourage the use of the Lucid programming language. In a word, running a Lucid application on computers spread out on Internet would have an effect of broadcasting GEE and GIPSY.

However, not all applications are suitable to be put in IDC. For example, some applications require a tight time limit for solving a problem. It might be the case IDC computation is more efficient, however, there's such possibility that potential sources are not usable because computing resources on internet are not well oragnized, most of which are provided by volunteers whose availability cannot be ensured. Also, applications require high security are not appropriate for being performed in IDC. Although advanced secure technology may ensure the security of hosts and transmission, data still has higher risk of being “stolen” on internet than in an isolated environment. GIPSY is a research project in which the secuirty issue is not well addressed currently. As a result, only restricted applications can run in IDC.

Not only executable applications are restricted, GEE also needs to be refined or modified to fit into a much broader execution environment. First of all, a computing task should be divided carefully that each piece must have reasonable workload that usually takes a long time to work out. Internet cannot offord high-speed data transimission compared with a local LAN and the overhead of communication would consume the execution time saved from using multiple machines. The divison of task can be done by wrapping more work into bigger units or analyzing features of an application and optimizing its algothrim to present more data parallelism. Also, the risk of delayed/non-answer must be considered. If the desired result has not been returned within a given time limit, the task should be assigned to other contributors. Moreover, computed values should be downloaded to each host along with the task package and value caches are installed locally. Therefore, values inquires need not to be performed remotely. Finally, IDC requires the GEE component to consider the varieties of Internet hosts. GEE should have profound knowledge about various kinds of machines, at least, the popular ones. A installation module must be build with which a user can choose one proper platform.

6.3 Execution Performance

Lucid programs should take less time to write (for experienced programmers) because they are in general more concise than programs written in most other languages (such as C). Therefore, programmer performance cannot be neglected especially in this age of cheap processors and expensive programmers.

Besides considering the performance of writing Lucid programs, there are three major factors affect the performance of executing Lucid programs in a distributed way. First, since the adopted lazy execution delays the computations as late as possible, more work is being pushed to the runtime. Secondly, the "weight" of each computation unit has critical influence on the communication overhead. Thirdly, the workload should be kept balanced in order to fully utilize the resource of each participating machine. The overall performance is expected to boost up if these factors are carefully considered.

Considerations should focus on the features of Lucid languages, but not some specific kinds of problems; moreover, considerations should be based on the general execution process but not a specific situation. In addition, there's a tradeoff between efficiency and flexibility because more work is pushed to the front-end in order to ensure efficiency whereas more work should be determined at runtime to ensure the flexibility. Being a prototype of GEE, this thesis addresses the flexibility more than the efficiency.

6.3.1 Evaluation Model

The *eduction*, or *lazy evaluation* model comes in handy when an expression is expensive or impossible to evaluate and may not need to be evaluated at all. It is also useful for recursively defining infinite data structures. Since each level of recursion is evaluated only when it is needed, data is only generated when it is consumed and the evaluation of the data structure can terminate when the consumption is completed.

Unfortunately, *lazy evaluation* usually carries a performance penalty for small programs and simple implementations. To support *lazy evaluation* suspended calculations have to be built. These suspended calculations take time to create, consume memory (i.e., may make garbage collection slower), and have to be detected and evaluated when encountered.

The opposite of lazy evaluation is eager evaluation, or call-by-value, in which expressions are evaluated as they are encountered (i.e. in the order in which they occur in the program text.) Under call-by-value, there is a simple correspondence between a primitive operation and its realization on the machine. We represent the state of a computation as a stack of nested activation frames, each containing the local data for a procedure call. As a result, it is relatively easy to produce an efficient implementation of a strict language using eager evaluation.

Modern compilers use *strictness analysis* [SRM1997, SR1995] for optimizing sequential and parallel evaluation of lazy functional programming languages. It is a well-know technique to discover those portions of the code that must be executed, and these portions are implemented with *call by value* to avoid the overhead of closure creation.

In order to simplify the compilation process, many compilers for higher-order languages use the continuation-passing style (CPS) transformation [FSDF1993] in a first phase to generate an intermediate representation of the source program. The aspect of this intermediate form is that all procedures take an argument that represents the rest of the computation (the "continuation"). The advantage of CPS is there is no need for a stack avoiding overhead because values enter and leave the stack too frequently (because functions never return). Also, every intermediate value of a computation is given a name (possible corresponding to a machine register). This allows an easy translation to machine code. As a disadvantage, the replacement considerably increases the size of programs.

Although adopting the lazy evaluation execution model, GEE can have some "eagerness" with pushing a few things forward to the compile time similar as proposed in [Mae2002]. For example, instead of taking demands in an ad hoc manner, it can give higher priority to identifiers placed on leaf nodes of the abstract syntax tree because their values are usually needed ahead of the nodes placed at a higher layer. Such guidelines can be used at runtime. Also, static analysis on an AST can discover some identifiers that are frequently used by others. These identifiers can be attempted to compute before others. In addition, results of rank analysis are good resources for optimization of the execution (see Section 3.3.3).

6.3.2 Communication Overhead

The communication overhead is another obstacle for improving performance. Though the cooperation (usually in the form of message passing) is definitely necessary in a distributed system, we can reduce the communication overhead in two ways, one is to reduce the transmission and the other is to reduce the size of information transmitted over the net when the transmission is unavoidable.

Increasing the workload of distributed tasks and distributing the value cache can reduce the demand for transmitting dramatically.

The granularity of each computation unit has apparent influence on the communication overhead.

Computation units could be very tiny pieces of work that can be done concurrently. Other computations are such that the results of step one are necessary input to start step two and the results of step two are for carrying on step three and so on. These computations can not be broken down into as small of work units. Those things that can be broken down into very small chunks of work are called fine-grained and those that require larger chunks are called coarse-grain.

If the chunk of work to be done is so small that it takes comparatively too long to send the work and receive results to another CPU then it is quicker to have a single CPU doing the work. Also, if the chunk of work to be done is too big then it is better to be spread out over machines in order to get the result more quickly.

A computer system with multiple processors in a single machine can handle very fine-grained problems well because communications are performed internally by shared-variables, while a system built from computers distributed over the Internet can handle only coarse-grained problems. Computer systems centered around a small high-speed local network can handle problems somewhere in the middle.

The data parallelism described in Lucid is fine-grained in that the parallelism is achieved at a low-level, such as the comparison of two elements. However, Lucid allows a programmer to define functions, consequently, grains becomes coarser. Another way to increase the granularity is achieved with including calculation functions as illustrated in GLU [JDA1997]. In GLU, calculation functions were written in C considering two factors: Lucid was regarded as being more suitable for declaring the relationship between the functions and there was a large amount of legacy C programs that had been implemented for the purpose of computing. A programmer can assign the workload of individual functions. Therefore, the granularity is manageable.

GIPSY can use a strategy similar to that of GLU with processing a mixture of Lucid and some other imperative language (e.g., JAVA) to fulfill the calculation to granulate parallelism expressed by Lucid. However, at the current stage, only pure Lucid programs are processed.

Another way to reduce the communication is achieved by distributing the value cache. Instead of having a central value warehouse, the values can be kept on each host. Consequently, a host looks for a value locally and needs to consult remote central value house only when the local one doesn't have the value it needs. This could also be required by the system since there might have a huge amount of values generated during the process that each host might not be able to keep the whole set but only part of it and one host has a value warehouse that holds all up-to-date values.

6.3.3 Divisible Load Scheduling

Divisible Load Theory (DLT) [VR2003, VGR2003, VGMR1996] theory is a mathematical tool created to allow tractable performance analysis of systems incorporating communication and computation issues, as in parallel and distributed processing.

A key feature of the theory is that it uses a linear mathematical model. It is assumed that computation and communication loads can be partitioned arbitrarily among a number of processors and links respectively. A continuous mathematical formulation is used. In addition, there are no strong precedence relations among the data so that load can be arbitrarily assigned to processors and links in a network.

This method is well suited for modeling data-intensive and data parallel computational problems. It also sheds light on architectural issues related to parallel and distributed computing. Moreover, the theory of divisible load scheduling is fundamentally deterministic.

A typical divisible load model allows one to account for model parameters such as heterogeneous processors, link speed, computation and communication intensity. For instance, the time to process the load at the *i*th processor is typically $\alpha_i\omega_i T_{cp}$ where α_i is the fraction of the total load assigned to the *i*th processor, ω_i is the inverse speed of the *i*th processor and T_{cp} is the computation intensity of the load. Similarly, the time to transmit a load fraction α_i on the *i*th link is $\alpha_i\chi_i T_{cm}$ where χ_i is the inverse speed of the *i*th link and T_{cm} is the communication intensity of the load.

The DLT theory has been applied to a variety of applied problems including monetary cost optimization, matrix calculations [GK1998], multimedia scheduling and image processing [VR2002]. There has also been experimental work and a proposal for dynamic real time scheduling [Gho2002]. The approach is particularly suitable to process very large linear data files as in signal processing, image processing, and experimental data processing. In general, DLT is applicable to situations where large amount of CPU time is in demand due to a very heavy computational requirement.

The GEE component can make use of DLT to optimize the overall processing time so long as the data dependencies can be taken care. Following the DLT theory, both computing time and channel transmission time are modeled linearly. Then, continuous time modeling is invoked.

6.4 Resource Management

The RMI approach adopts a "task pool" which can be accessed by all processors who take part in the job. The processors take tasks from the pool and produce: (1) no task; (2) one task; or (3) many tasks. The produced tasks will be returned and placed in the pool. The parallel part of the program finishes when not only the task pool is empty but also all processors are idle. This method makes it easy to balance the workload between all processes.

The GEE component is functioning well with adopting the task pool model. However, in order to ensure an overall performance for a more complicated application and obtain a robust system at runtime, GEE should embody a monitor module. The module is used to collect runtime resource information, check whether the usage of the system or an individual machine has exceeded a threshold, and (if it does) takes proper steps to ameliorate the situation without interrupting the execution.

Managing resources increases the complexity of a system and may slow down the execution. However, since distribution implies possible failure of hosts and connections that the risk has to be considered. In this section, an intelligent adaptation is proposed that requires an individual host being aware of its own execution environment as well as that of the whole system.

The design of the monitor component has to fulfill a clear separation of concerns between the application functionality and the adaptation processes. All the code necessary to make the application aware of the execution environment as well as the code that defines the adaptation actions are encapsulated inside the component. Moreover, the reflection technology is applied to fulfill the required flexibility.

6.4.1 Information

There are two kinds of information: static or dynamic. The static information is collected only once (usually) at the beginning of the execution. On the contrast, the dynamic information needs to be checked on a regular basis during the execution.

The static information includes important hardware capacity parameters of each host in the network and the computation objects written in an application.

Hardware parameters include:

- Location (IP address)
- CPU capacity
- Memory
- Disk storage;
- Network transmission speed.

Information of computation objects includes:

- Object name
- Number of objects.

During the runtime, some valuable information needs to be tracked, which includes:

- Correspondence between an object and its location (we assume objects located on one host can migrate to other hosts)
- Network traffic.

Each host has:

- CPU usage
- Memory usage.

The dependencies between the computation objects can be easily read from the dictionary exported by GIPC and obtained before the execution starts. In contrast, it is more difficult to obtain the hardware information of each host and link capacity. The reason is that a distributed system implies a system consisting of various kinds of platforms, each of which provides different means to access its hardware configuration. For example, the system call, *getProcessTimes*, maybe used to get the CPU usage for Win32 platform; and the system call, *getrusage*, can be used as a starting point to get the CPU usage for a UNIX platform.

Java does not provide an API to access hardware information directly, but provides Java Native Language (JNL) that a programmer can use C to access hardware information then insert the chunks written in C into a Java program.

The client-server approach presents a model of one server and multiple clients. The model is centrally managed that all clients are responsible to the server. It is relatively easy to manage but the whole system stops working when the server is shut down for some reason.

6.4.2 Reflection

The reflection technology would be suitable to be used in the resource management module.

In programming languages, reflection appears as follows [BGW1993]: "Reflection is the ability of a program during its own execution. There are two aspects of such manipulation: *introspection* and *intercession. Introspection* is the ability for a program to observe and therefore reason about its own state. *Intercession* is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification."

Reification must be distinguished from first-class entities. When an entity is first-class in a language, it means that this entity can be created at run-time, passed as actual parameters to procedures, returned as result of a function and stored in a variable. All that is needed is to provide programs with values of a suitable abstract data type defined by the language.

[Mae1987] summed up a view of reflection on computation system.

- A computational system is something that reasons about and acts upon some part of the world, called the domain of the system.
- A computational system can be causally connected to its domain. This means the internal structures of a system and the domain it represents are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other.
- Reflection is the process of reasoning about and/or action upon oneself.
- A reflective system is a system that incorporates structures representing aspects of itself. The sum of these structures, the self-representation, makes it possible for the system to answer questions about itself and support actions on itself.

- A meta-system is a computational system that has its domain as another computational system, called its object-system. A meta-system has a representation of its object-system in its data. Its program specifies meta-computation about the object-system and is therefore called a meta-program.
- A programming language is said to have a reflective architecture if it recognizes reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly.

The concept of reflection fits most naturally in the spirit of object-oriented programming: abstraction. An object is free to realize its role in the overall system in whatever way it wants to. Thus, it is natural to think that an object performs computation not only about its domain but also about how it can realize this (object-) computation.

In object-oriented programming, two kinds of reflection exist with different purposes: Structural and Computational.

Structural reflection is obtained by *metaclasses*, which define the behavior and structure of their classes. Changes to a *metaclass* influence its class and all objects of the class. Using structural reflection it is not possible to make adaptations to individual objects because adaptations apply to all objects of a class. Also, structural reflection adaptations must be used at compile time because a class needs to be defined and compiled before its instances can be created.

For computational reflection, an object is closely related to its *metaobject*. Each individual object can have a *metaobject* attached and objects of the same class can be attached to *metaobjects* of different classes. Furthermore, it is possible to dynamically remove or attach *metaobjects*. A *metaobject* monitors and affects its base level object depending on its base level object's state. A base level object and its *metaobject* are coordinated through a causal connection. A causal connection implies that any changes made in a metaobject are immediately reflected in the base level objects' state and behavior.

Reflection does not directly contribute to solve problems in the external domain of the system. Instead, it contributes to the internal organization of the system or its interface to the external world. Its purpose is to guarantee the effective and smooth functioning of the object-computation.

Java is a programming language supporting reflection. Its reflective ability is called the refection API. However, the Java's ability is almost restricted to introspection, which is the ability to introspect data structures used in a program such as a class. Its ability to alter program behavior is very limited; it only allows a program to instantiate a class, to get/set a field value, and to invoke a method through the API.

6.4.3 Reflective Resource Management

The GEE monitor consists of several information collectors and one central analyzer. Each host is responsible for collecting its own information on a periodic basis in order to keep track of the state of the execution environment in which the application is running. A collector keeps track of the dynamic dependencies between the component and other system or application components. By maintaining an explicit representation of those dependencies, it is possible to guarantee runtime consistency. All collected information is forwarded to an analyzer, where all information is placed and examined. The analysis of these data will determine when the adjustment is required and what to be adjusted to adapt the application to the changes in environmental state. The network quality is measured by the analyzer by pinging each host.

Each object is required to be able to adapt to the changes in its environment, which is to determine which proper action(s) should be performed when a significant change on the environmental state occurs and support the application of those actions in a safe and consistent way. For example, if one object fails, the objects that depend on it can be notified in order to promote a graceful exit. The monitor triggers the necessary actions to ensure a safe and consistent dynamic configuration of the application.

In this reflective model, the system is separated into 2 levels: one is the base level, where the application algorithm is applied and the second is the meta-level which monitors the execution of the system and provides dynamic adaptation. A meta object is aware of the behavior of the base object, but it will not change the behavior of the base object. The reflective approach benefits the constructing a distributed system in that it separates the algorithm of the application from the different underneath operating systems thus execution management becomes dynamical and easy.

7 Related Work

This chapter reviews two tasks: GLU and IE4 that have been performed in the domain of multidimensional intensional programming languages.

7.1 GLU

GLU [JDA1996] is regarded as a significant step in the evolution of Lucid programming languages, which is featured as: processing a hybrid language (mixed with Lucid and C) and being efficient.

GLU is a high-level system for developing scientific and engineering data-parallel applications. It is regarded as a high-level system because it requires an application to specify only the manner in which different parts of the application are related in terms of dependencies or constraints instead of demanding the application to explicitly specify all inter-process communication and synchronizations using provided message-passing calls.

GLU is based on a hybrid language comprising of Lucid for specifying the parallel structure of an application and the imperative language C for specifying the various functions in the application. Therefore, each C code segment runs sequentially, but many of them may work concurrently with their cooperation being controlled by the declarative Lucid part. GLU extends Lucid in the way that user-defined functions in GLU can be defined procedurally and values that the functions consume or produce can be complex data structures. Consequently, it can utilize existing sequential applications (often written in C) without writing them from scratch.

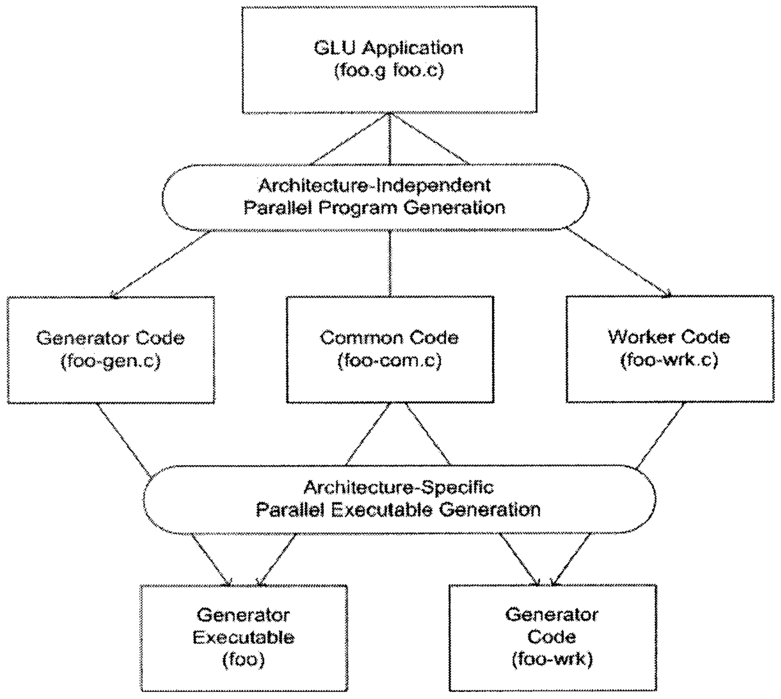


FIGURE 7.1 PARALLEL PROGRAM COMPILATION IN GLU

Figure 7.1 [JDA1997] shows how the GLU system works. A GLU application has two parts: one is written in C (foo.c in the figure) and the other is written in Lucid (foo.g). At the first stage, a parallel program is generated, which is architecture-independent that can be applied for different processors and operating systems. The C code generated from the Lucid part of the application is in three files: one is for the generator (foo-gen.c); one is for worker (foo-wrk.c) and the other is the common part of the generator and worker (foo-com.c). Then, architecture specific information and C functions are added to produce executable code. This executable code has two parts: one is for the generator (foo) and the other for the worker (foo-wrk).

The main mode with which the GLU system is capable of running is called *centralised generator architecture* (CGA). This architecture has one generator and multiple workers. The generator is responsible for propagating demands and querying a central-managed value warehouse, and the workers are responsible for doing calculations.

A supplemental mode is called *distributed generator architecture* (DGA), which allows multiple generators, each of whom has its own warehouse. The multiple generators cooperate on propagating demands. This architecture has its advantage that a single generator is less likely to become the bottleneck of performance by being swamped by a large amount of communication — inquiring values and demanding computations. It is also potentially able to survive a whole system failure caused by a single generator because the work can be transferred to other generators. However, the management is of course much more complicated.

The generator/worker architecture presents a master and slave relationship in that a generator dispatches tasks to workers while a worker works on a task in a given context without being aware of the whole picture of the system, even the existence of the warehouse.

GLU has been used for the development of real-word applications [AFJW1995]. Being a vital extension to the Lucid language, GLU addressed the efficiency of a parallel application with a coarse-grain method. However, although being proved to be efficient, the GLU system suffers from lack of flexibility and adaptability.

7.2 IE4

The Intensional Engine on L4(IE4) [Jay1999] implements a demand-driven engine on the top of the L4 μ-kernel and is heavily influenced by GLU. The following figure [Jay1999, page 33] shows the architecture of IE4.

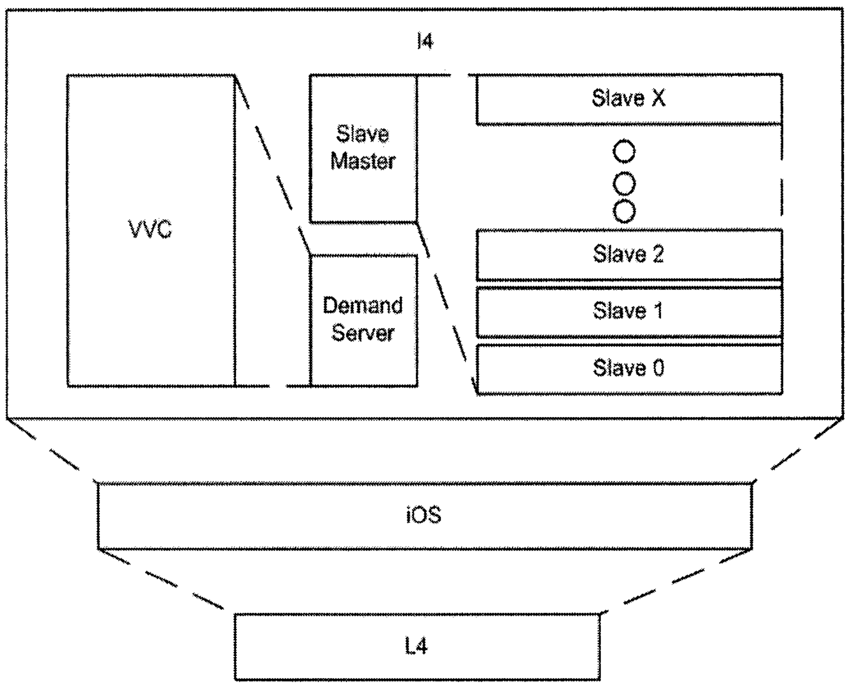


FIGURE 7.2 OVERVIEW OF L4 INTENSIONAL ENGINE

In this figure, I4 is the implemented eduction engine. iOS is an operating system sitting on top of L4. L4 is a kind of μ-kernel — μ-kernel is part of an operating system that executes in processors' privileged mode and implements only minimum functionalities. L4 has very low overhead for threads and inter process communication operations. The IE4 system improves the execution efficiency with implementing the engine at a very low level because the only thing between the IE4 engine and hardware is the high performance L4 μ-kernel.

The engine employs a Variable Value Cache (VVC) to store computed values and a *demand server* inquires the value cache each time before it attempts to generate demands. It also contains a *slave master* to manage all slaves.

In IE4, a programmer has to distinguish "heavyweight" identifiers from "lightweight" identifiers; "light" functions are executed locally while "heavy" ones are dispatched separately. The engine runs on a single machine and relies heavily on services provided by the operation system to improve the efficiency.

One feature of IE4 concerns that demands in the system include only dimensions in the rank of an identifier. This solution reduces the size of a demand but is likely to be slow because lists have to be traversed and it takes time to decode which dimensions a value is associated with.

8 Conclusion and Future Work

This chapter draws a conclusion with comparing our work with GLU and IE4. It also discusses future work.

8.1 Conclusion

The design and implementation of GEE, especially at the beginning stage, are inspired and influenced by the GLU and IE4 systems. For example, the two key steps in processing a Lucid program: replacing non-primitive intensional operators with their primitive alternatives and eliminating functions can find their roots in the IE4 implementation. The series of paper [Dod1996, JD1996, JDA1997] written by GLU creators are greatly helpful for the design of GEE. Similar to both GLU and IE4, GEE adopts a value warehouse to store computed values.

However, GEE stands out as an object-oriented system. With object technology, it addresses flexibility and adaptability issues that both GLU and IE4 lack. Moreover, it explores the distribution technology that is not well developed in GLU and is neglected altogether by IE4.

Unlike GLU, which handles a mixed language of Lucid and C, the GEE prototype processes pure Lucid programs, which both computation units and the structure control are written in Lucid. Thus, compared with GLU, the system is regarded as fine granulated, with which computation units are relatively smaller. As a consequence, higher cost is spent on exchanging information between the server and client(s).

In GLU, the demand propagation is done entirely by the “server” (more accurately, “master”). The “clients” (more accurately, “slaves”) simply do what they are told (always a sequential task) and never generate new demands. In GEE the tasks given to clients are parts of a Lucid program that may be recursively subdivided into further tasks, which are passed back to the server as new demands. In GLU, the server is a bottleneck because all demands must come from it. GEE avoids that bottleneck by distributing demand propagation. However, the server may still remain a bottleneck because of the central demand list and value warehouse.

Both GLU and IE4 focus on solving the efficiency of execution. GLU addresses the efficiency problem by increasing the workload of each piece of work with integrating C functions, while IE4 relies heavily on exploring the characteristics of a special kind of platform. GEE addresses the efficiency issue as well. It relies on a well-designed value warehouse and distributing tasks wisely. For example, the strategy of including all dimensions when storing computed values is based on the consideration of quick access. However, GEE is more emphasized on the flexibility issue. Not only does GEE consist of modules but also the technologies (object-oriented, distribution etc.) adopted by GEE are centralized around flexibility. Furthermore, GEE adopts a client-server architecture that the server holds resource and the client(s) does actual computations. With this architecture, it is relatively easy to increase the intelligence of each client and the clients can be more self-contained and process greater power.

We believe that GEE has a promising future considering the distributed computing is shifting from a narrow area to a wider scope, such as Internet, in which both the data and computing are distributed and intelligent hosts are required.

8.2 Future Work

This section describes future work that can be performed, particularly based on the consideration of possible impacts caused by including calculation functions written in another language.

8.2.1 Computation Functions

The next version of GIPSY will include calculation functions, which would be more useful for a large real scientific computation. The functions can be interpreted as computation units implemented in any languages, which could be Java, C++ or any legacy programs have been implemented for the purpose of some computations.

Because of the inherent concise nature of Lucid and based on our understanding of this programming language, it may be difficult to extract computation units from a pure Lucid program. Also, it may be hard to decide the granularity of units. So a practical approach might be to state the computation units explicitly in source code, or even use another programming language to describe the units and apply Lucid only on the control part to describe the relationships among the units. However, the eventual goal is to compile programs intelligently without assistance from the programmer.

8.2.2 Architecture

As in the present implementation, the system allows one server and multiple clients. Since it's fine granulated in that each computation does not take a long time the server would be busy generating demands and is prone to become a bottleneck. Moreover, if the server fails, the entire application would also fail.

Thus, we propose to have a peer-to-peer architecture, which consists of multiple processors, each of which acts as both a server and a client. That is, each peer is responsible for evaluating certain part(s) of a program as well as executing computations. In addition, each peer can take over responsibility of another peer that fails.

The division of an entire task across peers could be determined by grouping identifiers. At runtime, demands for values are either processed internally at a peer where they are generated or passed to the appropriate peer for further processing. When a demand is eventually solved, it is passed back to the original peer attached with the computed value.

Two advantages of the peer-to-peer architecture are scalability and fault tolerance. It is scalable in that peers can be added without creating a bottleneck at any one process; and it is fault tolerant in that no single-process failure will cause the application to cease operation. Features of Jini technology and Internet Distributed Computing can be explored for this architecture.

8.2.3 Distributed Value Warehouse

Another improvement concerns the value warehouse. There is only one central warehouse in current version, which contains all values. This central warehouse is relatively easy to manage, but very likely to become a performance bottleneck because the warehouse has to be visited by a worker before it does any computation and along with the growing of its size, the response time to enquiries tends to be longer.

A possible solution is for each worker to have its own specific warehouse that contains the values that this worker is concerned with. Thus, the cost of communication is decreased; however, the management of the warehouse is more complex. Moreover, to utilize the warehouse completely, we require the workers to be static, which means each one is fed with certain kinds of computation.

An improved practical solution is that there is one central warehouse and each worker keeps its own copy, or cache, which contains part of the content of central warehouse. If the value a worker needs is not in the cache, it turns to the central warehouse. The cache will be updated along with the center. Also, to fully utilize this cache, we hope that each machine does similar work as it did before to ensure the high hit probability of the cache.

Integrity is not an issue if the value warehouse is distributed over several processors because there is only one single value for an identifier in a certain context. Losing a value does not matter either because the value will be recomputed. Even if a node with a value warehouse goes down, all that happens is that the computation is slowed down.

Bibliography

[AFJW1995] E.A.Ashcroft, A.A..Faustini, R.Jagannathan, and W.W.Wadge. *Multidimensional Programming.* Oxford University Press, 1995.

[And2000] G.R. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addision-Wesley, 2000.

[BGW1993] D.G. Bobrow, R.P. Gabriel, and J.L. White. *CLOS in Context - The Shape of the Design Space.* In A. Paepcke, editor, *Object-Oriented Programming - The CLOS Perspective*, Chapter 2, MIT Press, 1993.

[Car1956] Rudolf Carnap. *Meaning and Necessity: a Study in Semantics and Model Logic.* University of Chicago Press. Second Edition, 1956. Pages 23-25.

[CDK1994] G. Coulouris, J. Dollimore, and T. Kinberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, Reading, MA, 1994.

[Dis] http://www.distributed.net

[Dod1996] C. Dodd. *Intensional Programming I*, chapter Rank Analysis in the GLU compiler, pages 76-82. World Scientific, Singapore, 1996.

[Du1994] W. Du. *Object-oriented Implementation of Intensional Languages.* In Proceedings of The 7th International Symposium on Lucid and Intensional Programming, SRI International, Menlo Park, California, U.S.A., pp. 37-45, 1994.

[Due1991] E.Duesterwald. *A Demand-driven Approach for Efficient Interprocedural Data Flow Analysis.* Ph.D. Thesis. University of Pittersburgh, 1991.

[Emm2000] W.Emmerich. *Engineering Distributed Objects.* London University, John Wiley & Sons, Ltd, pp. 141-155, 2000.

[FJ1991] A.A. Faustini and R. Jagannathan. *Indexical Lucid.* In Proceedings of the Fourth International Symposium on Languages for Intensional Programming, Menlo Par, California, 1991.

[Fox1997] G. Fox. *Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modelling.* Concurrency: Practice and Experience, Vol. 9(6), June 1997.

[FSDF1993] C. Flanagan, A. Sabry, B.F. Duba, M. Felleisen. *The Essence of Compiling with Continuations.* Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, pp. 23—25, Albuquerque, NM, USA, June 1993

[Gho2002] D. Ghose. *A Feedback Strategy for Load Allocation in Workstation Clusters with Unknown Network Resource Capabilities Using the DLT Paradigm.* Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02), Vol.1, pp.425-428, Las Vegas, USA, June 2002,.

[GK1998] D. Ghose and H.J. Kim. *Load Partitioning and Trade-off Study for Large Matrix Vector Computations in Multicast Bus Networks with Communication Delays.* Journal of Parallel and Distributed Computing, Vol.54, 1998.

[GKHF2001] P. Graunke, S. Krishnamurthi, V. Hoeven and M. Felleisen. *Programming the Web with High-Level Programming Languages*. Proceedings of ESOP 2001.

[GMO2001] R.M. Gebala, C.M. McNamee and R.A. Olsson. *Compiler to Interpreter: Experiences with a Distributed Programming Language*. Software -- Practice & Experience, Vol. 31, pp. 893-909, July 2001.

[GR1999] M. Gergatsoulis and P. Rondogiannis. *Intensional Programming II*. NCSR Demokritos, Greece, June 1999.

[Hask] http://www.haskell.org/

[Hen1980] P. Henderson. *Functional Programming Application and Implementation.* Prentice Hall, June 1980.

[Jag1995] R.Jagannathan. *Coarse-Grain Dataflow Programming of Conventional Parallel Computers*. In L. Bic, J-L. Gaudiot, and G. Gao, editors, Advanced Topics in Dataflow Computign and Multithreading. IEEE Computer Society Press, 1995.

[Jay1999] V. Jayawardene. *An Intensional Engine on L4*. Master Thesis, The School of Computer Science and Engineering of the University of New South Wales, 1999.

[JD1996] R. Jagannathan and C. Dodd. *GLU Programmer's Guide, version 1.0.* Technical report, SRI International, Menlo Park, California, 1996.

[JDA1997] R. Jagannathan, C. Dodd, and I. Agi. *GLU: A high-level system for granular data-parallel programming*. Concurrency: Practice and Experience, (1):63-83, 1997.

[Jen1996] Bob Jenkins. http://burtleburtle.net/bob/hash/evahash.html

[Kie1998] R.B. Kieburtz. *Reactive Functional Programming.* Proceeding PROCOMET, Chapman and Hall, 1998.

[KM1966] R.M. Karp and R.E. Miller. *Properties of a Model for Parallel Computation: Determinacy, Termination, Queuing.* SIAM Journal of App. Math, pp.1390-1411, November 1966.

[Lea1997] D. Lea. *Concurrent Programming in Java^{TM}: Design Principles and Patterns.* Addison-Wesley, 1997.

[LGP2003] Bo Lu, Peter Grogono, and Joey Paquet. Distributed Execution of Multidimensional Programming Languages. Proceedings 15th IASTED International Conference on Parallel and Distributed Computing and Systesm (PDCS 2003), International Association of Science and Technology for Development. November 2003., pages 284-289.

[Mae1987] Pattie Maes. *Concepts and Experiments in Computational Reflection.* In Proceedings of the ACM Conferenct on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 147-155, 1987.

[Mae2002] J. Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell.* Ph.D. thesis. Massachusetts Institute of Technology, June 2002.

[MRKB2003] M. Milenkovic, S.H. Robinson, R.C. Knauerhase, D. Barkai, S. Garg, V. Tewari, T.A. Anderson, M. Bowman. *Toward Internet Distriubted Computing.* Computer, May 2003.

[Paq1999] J.Paquet. *Intensional Scientific Programming.* Ph.D thesis, Department of Computer Science, Laval University, Sainte-Foy, Canada, 1999.

[PiH] http://www.cecm.sfu.ca/projects/pihex/pihex.html

[PK2000] J.Paquet and P.Kropf. *The GIPSY Architecture.* In Distributed Computing on the Web, Proceedings of the Third International Workshop, DCW2000, Lecture Notes in Computer Science, Vol. 1830, Springer, 2000.

[Pla2000] J. Plaice. *Multidimensional Lucid: Design, Semantics and Implementation.* In Distributed Computing on the Web, Proceedings of the Third International Workshop, DCW2000, Lecture Notes in Computer Science, Vol. 1830, Springer, 2000.

[PW1993] J. Plaice and W.W. Wadge. *A New Approach to Version Control.* IEEE Transactions on Software Engineering, 3(19):268–276, 1993

[Raj1995] S.P. Rajan. *Transformations on Dependency Graphs: Formal Specification and Efficient Mechanical Verification.* Ph.D. Thesis, Department of Computer Science, University of British Columbia, Vancouver, Canada, 1995.

[Ren2001] C. Ren. *Parsing and Abstract Syntax Tree Generation in the GIPSY System.* Master Thesis, the Department of Computer Science, Concordia University, Montreal, Canada, 2002.

[SET] http://setiathome.ssl.berkeley.edu

[SR1995] R.Sekar. I.V. Ramakrishnan. *Fast Strictness Analysis Based on Demand Propagation.* ACM Transactions on Programming Languages and Systems, Vol. 17, No 6, November 1995

[SRM1997] R. Sekar, I. V. Ramakrishnan, and P. Mishra. *Efficient Strictness Analysis of Haskell*. Journal of the ACM, Vol. 44, No 3, May 1997.

[Sun] *Java Remote Method Invocation-distributed computing for Java*. Java White Paper. http://java.sun.com/marketing/collateral/javarmi.html

[Sun1997] *Java Object Serialization Specification*. http://java.sun.com/products/jdk/1.2/docs/guide/serialization/spec/serialTOC.doc.html

[Sun2001] *Jini Network Technology, An Executive View*. Sun microSystem, Inc, copyright 2001. http://wwws.sun.com/software/jini/whitepapers/jini-execoverview.pdf

[TB2001] Z. Tari and O. Bukhres. *Fundamentals of Distributed Object systems: The CORBA Perspective*. A Wiley-Interscience Publication, John Wiley & Sons, INC, 2001.

[VGMR1996] B. Veeravalli, D. Ghose, V. Mani, and T.G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Almitos, California, 1996.

[VGR2003] B. Veeravalli, D. Ghose, and T.G. Robertazzi. *Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems*. Special Issue on Divisible Load Scheduling in Cluster Computing, Kluwer Academic Publishers, Vol. 6, No. 1, January 2003.

[VR2002] B. Veeravalli and S. Ranganath. *Theoretical and Experimental Study on Large Size Image Processing Applications Using Divisible Load Paradigm on Distributed Bus Networks*. Image and Vision Computing, Elsevier Publishers, USA, Vol. 20, pp. 917-936, Issues 13-14, December 2002.

[W3C2002] Web Services Description Requirements, W3C Working Draft 28 October 2002. http://www.w3.org/TR/ws-desc-reqs/#notations.

[WA1985] W.W. Wadge, E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[Wu2002] A. Wu. *Semantic Analysis and SIPC AST Translator Generation in the GIPSY.* Master Thesis, the Department of Computer Science, Concordia University, Montreal, Canada, 2002.

Appendix: Hash Function

Original one was written by Bob Jenkins in 1996 (http://burtleburtle.net/bob/hash/evahash.html) and the modified parts are as marked with "//-------//".

```
typedef  unsigned long  int  ub4;   /* unsigned 4-byte
quantities */
typedef  unsigned       char ub1;   /* unsigned 1-byte
quantities */

#define hashsize(n) ((ub4)1<<(n))
#define hashmask(n) (hashsize(n)-1)

/*--------------------------------------------------
    mix -- mix 3 32-bit values reversibly.
------------------------------------------------------------
*/

#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12);  \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}

/*--------------------------------------------------
hash() -- hash a variable-length key into a 32-bit value
  k       : the key (the unaligned variable-length array of
bytes)
  len     : the length of the key, counting by bytes
  initval : can be any 4-byte value
Returns a 32-bit value.  Every bit of the key affects every
bit of the return value.  Every 1-bit and 2-bit delta
achieves avalanche. About 6*len+35 instructions.
```

```
The best hash table sizes are powers of 2.  There is no
need to do mod a prime (mod is sooo slow!).  If you need
less than 32 bits, use a bitmask.  For example, if you need
only 10 bits, do
h = (h & hashmask(10));In which case, the hash table should
have hashsize(10) elements.

If you are hashing n strings (ub1 **)k, do it like this:
  for (i=0, h=0; i<n; ++i) h = hash( k[i], len[i], h);

By Bob Jenkins, 1996.                                 You
may use this code any way you wish, private, educational,
or commercial.  It's free.

See                                               Use for
hash table lookup, or anything where one collision in 2^^32
is acceptable.  Do NOT use for cryptographic purposes.
-----------------------------------------------------------
*/

ub4 hash( k, length, initval)
register ub1 *k;        /* the key */

//-----The length of a key is fixed in GEE represented by a
constant integer KEY_LENGTH --------------------------//
register ub4  length;   /* the length of the key */

register ub4  initval;  /* the previous hash, or an
arbitrary value */
{
   register ub4 a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an arbitrary
value */
   c = initval;         /* the previous hash value */

   /* handle most of the key */
   while (len >= 12)
   {
      a += (k[0] +((ub4)k[1]<<8) +((ub4)k[2]<<16)
+((ub4)k[3]<<24));
      b += (k[4] +((ub4)k[5]<<8) +((ub4)k[6]<<16)
+((ub4)k[7]<<24));
      c += (k[8] +((ub4)k[9]<<8)
+((ub4)k[10]<<16)+((ub4)k[11]<<24));
```

```
      mix(a,b,c);
      k += 12; len -= 12;
   }

   /* handle the last 11 bytes */
   c += length;
   switch(len)    /* all the case statements fall through
*/
   {
   case 11: c+=((ub4)k[10]<<24);
   case 10: c+=((ub4)k[9]<<16);
   case 9 : c+=((ub4)k[8]<<8);
      /* the first byte of c is reserved for the length */
   case 8 : b+=((ub4)k[7]<<24);
   case 7 : b+=((ub4)k[6]<<16);
   case 6 : b+=((ub4)k[5]<<8);
   case 5 : b+=k[4];
   case 4 : a+=((ub4)k[3]<<24);
   case 3 : a+=((ub4)k[2]<<16);
   case 2 : a+=((ub4)k[1]<<8);
   case 1 : a+=k[0];
     /* case 0: nothing left to add */
   }
   mix(a,b,c);
   /* report the result */
   return c;
}
```