Intensional Value Warehouse and Garbage Collection in the GIPSY

Lei Tao

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

September 2004

Your file Votre référence
ISBN: 0-612-94753-X
Our file Notre référence
ISBN: 0-612-94753-X

Abstract

Intensional Value Warehouse and Garbage Collection

in the GIPSY

Lei Tao

Intensional Programming involves the programming of expressions placed in an inherently multidimensional context space. It is an emerging and evolving domain of very general application. The General Intensional Programming System (GIPSY) aims at the implementation of a programming system that would allow very dynamic investigations on the possibilities of Intensional Programming. Intensional Value Warehouse (IVW), a part of General Eduction Engine (GEE) that is the back end of GIPSY, is built as storage of values that have already been computed. First of all, this thesis briefly introduces the GIPSY and its features and goals. Then, we describe the design methodology and principle. Afterwards, we give the detail of implementation of the IVW. In the IVW, we build a garbage collector that promotes the deleted data or writes them to file, which is different from the typical garbage collector. The use of garbage collector configured to it is of prime importance to obtain high performance. Further, we discuss the result of the IVW, which approaches the main goals of generality, adaptability and efficiency. Finally, we successfully integrate IVW to GEE in overall GIPSY system.

Acknowledgements

At first, I would like to express deep gratitude to my mentor and supervisor, Dr. Joey Paquet, for his patience and invaluable guidance. I really appreciate his dedicated mentoring and the inspiring discussions with me. I am also thankful to Dr. Peter Grogono, for giving me much valuable advice. I am forever grateful to them for their inspiration, continuous encouragement, and step-by-step guidance. Besides, I thank to all GIPSY project team-mates for their generous help and efforts, and overall collaboration. Finally, I would like to thank for my wife, Yuwei Feng, for her guidance, support, encouragement, and love. I dearly thank my parents and my parents-in-law for their continuously encouraging me to achieve this challenging work.

Table of Contents

List of Figures

List of Tables

Chapter 1 : Introduction

This chapter introduces the aim of the runtime intelligent memory value cache that we developed in this research work as part of the GIPSY project, and highlights the contribution of the thesis and describes its structure.

1.1 Context

This thesis mainly describes the development of a runtime intelligent memory value cache called IVW (Intensional Value Warehouse), which is a part of GEE (General Eduction Engine) in the GIPSY (General Intensional Programming SYstem). The GIPSY is an implementation of Intensional Programming, which involves the programming of expressions placed in an inherent multidimensional context space [1]. The GIPSY is also a very ambitious project and is directed by Dr. Joey Paquet and Dr. Peter Grogono in the Computer Science Department at Concordia University, Montreal, Canada. Several other graduate students, ant the PhD and Master level are also involved in the various parts of the project.

1.2 Thesis Statement

The GIPSY handles an intensional programming language called Lucid that is a functional language in which expressions and their valuations are allowed to vary in an arbitrary number of dimensions. To process Lucid source code, we divided the workflow into two major steps: the front-end and the backend. The front-end includes 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 is the back-end of the GIPSY system, which operates according to a demand-driven computation model in conjunction with an intelligent value cache called an IVW. Every computed value is placed in the warehouse, and every demand for an already computed value is extracted from the warehouse rather than computed anew. This model of computation has been named "eduction" by Wadge and Ashcroft, the original developers of the Lucid language. Eduction reduces the overhead induced by the procedure calls needed for the computation of demands, i.e. demand propagation does not have to be computed more than one time.

A highly modular design and complete specification of generic software interfaces enables the GEE to accept other garbage collecting algorithms with minimal programming cost and replacement overhead. A thorough analysis of the different requirements of each intended application of intensional programming enables us to identify a minimal set of garbage colleting techniques to be implemented and tested. We also investigate the possibility of using multi-level warehouses enabling faster access to values accessed regularly and allowing out-of-date computed results to be stored on the file system [3]. This thesis work focuses on building the IVW of GEE in GIPSY. One of our main goals is to design and complete a warehouse as an effective and flexible solution to approach GIPSY features: generality, adaptability and efficiency.

1.3 Contributions

This thesis work performs experience on the application of intelligent memory value cache with garbage collection technique, designs and implements the Intensional Value Warehouse (IVW), which is a part of development of GEE of the

GIPSY. It investigates the essence of building value warehouse in memory by applying Java implementation technologies. In addition, it explores the advantages of garbage collection algorithms and NetCDF dataset on IVW.

First, the thesis uses the Network Common Data Form (NetCDF) to construct IVW. The purpose of the NetCDF interface is to allow user to create, access, and share array-oriented data in a form that is self-describing and portable. Through the library provided by NetCDF, we can store and retrieve data in self-describing machine-independent datasets. Each NetCDF dataset can contain multidimensional, named variables, and each variable may be accompanied by ancillary data, such as units of measure or descriptive text. NetCDF allows direct-access storage and retrieval of data by variable name and index and thus is useful for disk-resident or memory-resident datasets. Using NetCDF interface in IVW can make GIPSY meet the requirement of generality and adaptability.

Secondly, the thesis explores three different designs of IVW module and variant garbage collecting techniques in IVW. One solution is to maintain the NetCDF memory dataset as value warehouse without a cache component. Others are to build a NetCDF memory dataset with a cache dataset, which enables faster access to values. Also, as an intelligent memory value cache, the IVW provides the independence of the garbage collection algorithm. Then, the out-of-date computed value will be swept or stored on the file system, which helps to increase the performance of the IVW. In this way, the IVW achieves the desired efficiency.

Thirdly, the thesis implements an extensible architecture that is completely object-oriented, easily manageable, and adaptable to future changes. Then, the thesis explains the integration and relationship of the generated components in the whole system. Moreover, it includes estimates of the impact of access to IVW under various garbage collection architectures and algorithms.

1.4 Structure of the Dissertation

Chapter 2 introduces the structure of the GIPSY system, and then describes its design goals, also presents the major tasks performed by the front-end components of the system. Chapter 3 analyses the methodology used to implement the IVW and gives the explanations of the advantages of method used. Chapter 4 describes the design principles, gives the details of implementation of the IVW components, moreover, it presents how to integrate IVW with GEE and RIPE. Chapter 5 shows the result of evaluation from each aspect introduced above. Chapter 6 concludes the thesis with an outline of its contributions. Chapter 7 presents the future work.

Chapter 2 : Background

This chapter presents the background of the component IVW in GIPSY. It gives an introduction of the GIPSY system, and then explains the basics of Unidata's NetCDF software package, and current techniques of garbage collection in memory data management.

2.1 Introduction to the GIPSY System

2.1.1 Intensional Programming and Lucid Language

Intensional programming is a generalization of unidimensional contextual (a.k.a. modal logic) programming such as temporal programming, but where the context is multidimensional and implicit rather than unidimensional and explicit. Intensional programming is also called *multidimensional programming* because the expressions involved are allowed to vary in an arbitrary number of dimensions; the context of evaluation is thus a *multidimensional context*. For example, in intensional programming, one can very naturally represent complex physical phenomena such as plasma physics, which are in fact a set of charged particles placed in a space-time continuum that behaves according to a limited set of laws of intensional nature. This space-time continuum becomes the different dimensions of the context of evaluation, and the laws are expressed naturally using intensional definitions [1].

Lucid is a multidimensional intensional programming language whose semantics is based on the possible world semantics of intensional logic. It is a functional language in which expressions and their valuations are allowed to vary in an arbitrary number of dimensions. Intensional programming (in the sense of Lucid) has been successfully applied to resolve problems with a new perspective that enables a more natural understanding of problems of intensional nature. Such problems include topics as diverse as reactive programming, software configuration, tensor programming, and distributed operating systems. However, these projects have all been developed in isolation. GLU is the only intensional programming tool presently available. However, experience has shown that, while being very efficient, the GLU system suffers from a lack of flexibility and adaptability. Given that Lucid is evolving continually, there is an important need for the successor to GLU to be able to stand the heat of evolution [2].

We implement a general intensional programming system (GIPSY). To cope with the fast evolution and generality of the intensional programming field, the design and implementation of all its subsystems is done towards *generality*, *flexibility* and *efficiency*.

2.1.2 Architecture of the GIPSY System

A clear and adaptive architecture is defined to reach the flexibility and adaptability goals of the system. The GIPSY is composed of three main modular subsystems: The General Intensional Programming Language Compiler (GIPC); the General Eduction Engine (GEE), and the Intensional Run-time Programming Environment (RIPE). The following sections outline the theoretical basis and architecture of the different components of the system. All these components are designed in a modular manner to permit the eventual replacement of each of its components, at compile-time or even at run-time, to improve the overall efficiency of the system

[2]. The structure of the GIPSY system is depicted in Figure 2-1.

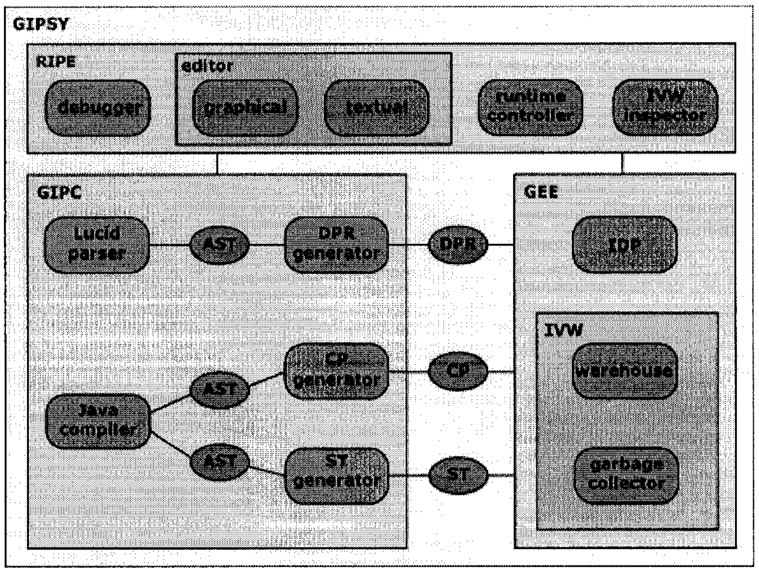


Figure 2-1: Architecture of GIPSY

RIPE: Run-time interactive programming environment

GIPC: General intensional programming language compiler

GEE: General eduction engine

AST: Abstract syntax tree

IDP: Intensional demand propagator

IVW: Intensional value warehouse

ST: Sequential thread

CP: Communication procedure

DPR: Demand propagation resources

2.1.3 General Intensional Programming Compiler (GIPC)

GIPSY programs are compiled in a two-stage process. First, the intensional (Lucid) part of the GIPSY program is translated into Java; then, the resulting Java program is compiled in the standard way.

The source code consists of two parts: the Lucid part that defines the intensional data dependencies between variables and the sequential part that defines the granular sequential computation units (usually written in Java). The Lucid part is compiled into an intensional data dependency structure (IDS) describing the dependencies between each variable involved in the Lucid part. This structure is interpreted at run-time by the GEE following the demand propagation mechanism. Data communication procedures used in a distributed evaluation of the program are also generated by the GIPC according to the data structures definitions written in the Lucid part, yielding a set of intensional communication procedures (ICP). These are generated following a given communication layer definition such as provided by IPC, CORBA or the WOS.

The sequential functions defined in the right part of the GIPSY program are translated into sequential code using the second stage (Java) compiler syntax, yielding sequential threads (ST). Intensional function definitions, including higher order functions, are flattened using a well-know efficient technique. [1]

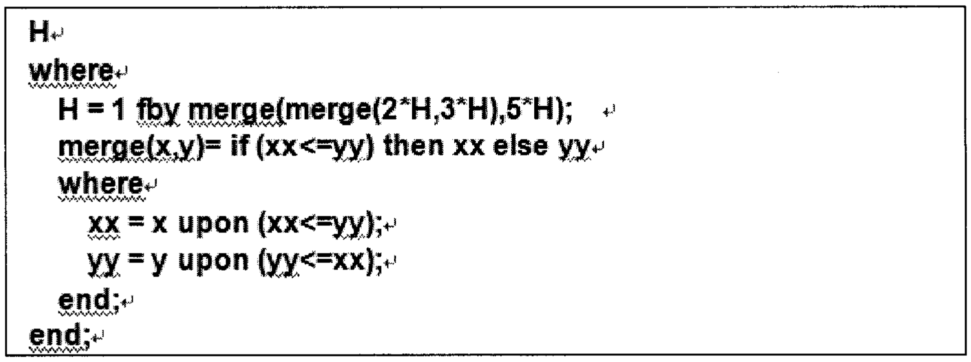
2.1.4 Run-time Interactive Programming Environment (RIPE)

The RIPE is a visual run-time programming environment enabling the visualization of a dataflow diagram corresponding to the Lucid part of GIPSY programs. The user can interact with the RIPE at run-time in the following ways. [2]

- Dynamically inspect the IVW;
- Change the input/output channels of the program;
- Recompile sequential threads;
- Change the communication protocol;
- Change parts of GIPSY itself (e.g. garbage collector).

Because of the interactive nature of the RIPE, the GIPC is modularly designed to allow the individual on-the-fly compilation of either the DPR (by changing the Lucid code) ICP (by changing the communication protocol) or ST (by changing the sequential code). Such a modular design even allows sequential threads to be programs written in different languages (for now, we are concentrating on Java sequential threads).

A graphical formalism to visually represent Lucid programs as multidimensional dataflow graphs had been devised in [1]. For example, consider the Hamming problem that consists of generating the stream of all numbers of the form $2^i3^j5^k$ in increasing order and without repetition. The following Lucid program solving this problem can be translated into a dataflow diagram.



```
H
where
  H = 1 fby merge(merge(2*H,3*H),5*H);
  merge(x,y)= if (xx<=yy) then xx else yy
  where
    xx = x upon (xx<=yy);
    yy = y upon (yy<=xx);
  end;
end;
```

Figure 2-2: Indexical Lucid program for the Hamming problem

Figure 2-3 represents the dataflow diagram corresponding to this program. Such nested definitions will be implemented in the RIPE by allowing the user to expand or reduce sub-graphs, thus allowing the visualization of large-scale Lucid definitions.

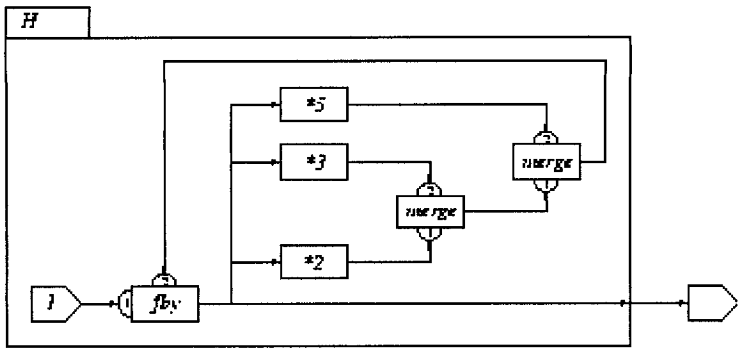


Figure 2-3: Dataflow graph for the Hamming problem

Using this visual technique, the RIPE enables the graphic development of Lucid programs, translating the graphic version of the program into a textual version that can then be compiled into an operational version. There is also the possibility to have a kernel run-time interface on top of which we can plug-in different types of interfaces adapted to different applications. [2]

2.1.5 General Eduction Engine (GEE)

GIPSY uses a demand-driven model of computation, whose principle is that a computation takes effect only if there is an explicit demand for it. GIPSY uses eduction, which is demand-driven computation in conjunction with a value cache called a warehouse. Every demand can potentially generate a procedure call, which is either computed locally or remotely, thus eventually in parallel with other procedure calls. A value should be warehoused if it is cheaper to extract it from the warehouse than to recompute it. Every demand for an already-computed value is extracted from the warehouse rather than computed anew. Eduction thus reduces the overhead induced by the procedure calls needed for the computation of demands. [1]

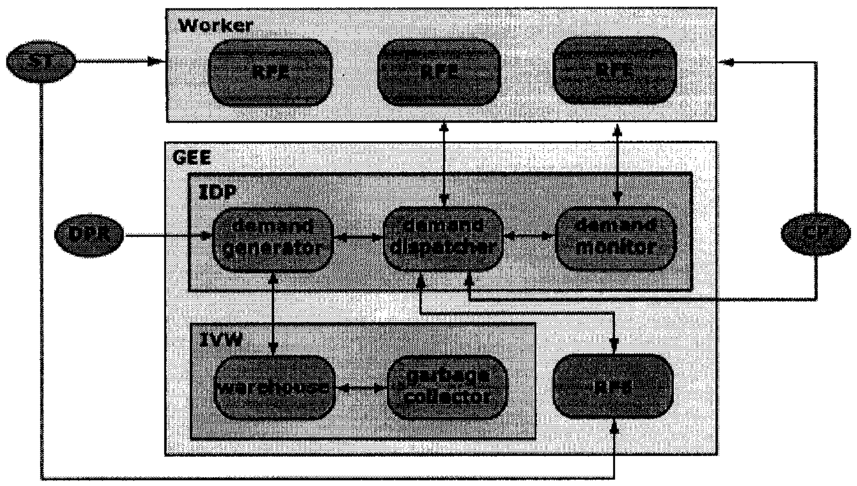


Figure 2-4: Internal Structure of the General Eduction Engine (GEE)

In Figure 2-4, we can see that the GEE is composed of two main modules: the intensional demand propagator (IDP) and the intensional value warehouse (IVW). First, the demand propagation resources (DPR) are fed to the demand generator (IDG) by the compiler (GIPC). This data structure represents the data dependencies between all the variables in the Lucid part of the GIPSY program in input. This directs in what order all demands must be generated to compute values from this program. The demand generator receives an initial demand, which in turn raises the need for other demands to be generated and computed. For all non-functional demands (i.e. demands not associated with the execution of a sequential thread (ST)), the IDG makes a request to the warehouse to see if this demand has already been computed. If so, the previously computed value is extracted from the warehouse. If not, the demand is propagated further, until the original demand is resolved and is put in the warehouse for further use.

Functional demands, (i.e. demands associated with the execution of a sequential thread), are sent to the demand dispatcher (IDD). The IDD takes care of sending the demand to one of the workers or resolves it locally (which normally means that a worker instance is running on the processor running the generator process). If the demands are sent to a remote worker, the communication procedures (ICP) generated by the compiler are used to communicate the demand to the worker. The IDD receives some information about the lifecycle and efficiency of all workers from the demand monitor (IDM), to help it make better decisions in dispatching functional demands.

The demand monitor, after some functional demands are sent to workers, gathers various information of each worker:

- Its status (is it still alive, not responding, or dead)
- Its network link performance
- Its response time statistics for all demands sent to it
- etc.

This information is accessed by the IDD to make better decisions about the load balancing of the workers, and thus achieving better overall run-time efficiency.

2.2 State of the Art of Value Warehouse

2.2.1 General introduction

The following sections present the Unidata technique and NetCDF interface, which are used to build our IVW. Also, we provide a brief overview of current garbage collection techniques in memory data management.

2.2.2 Unidata and NetCDF

Unidata is a National Science Foundation-sponsored program empowering U.S.

universities, through innovative applications of computers and networks, to make the best use of atmospheric and related data for enhancing education and research. For analyzing and displaying such data, the Unidata Program Center offers universities several supported software packages developed by other organizations, including the University of Wisconsin, Purdue University, NASA, and the US National Weather Service. Underlying these is a Unidata-developed system for acquiring and managing data in real time, making practical the Unidata principle that each university should acquire and manage its own data holdings as local requirements dictate. It is significant that the Unidata program has no data center—the management of data is a "distributed" function.

The Network Common Data Form (NetCDF) software was originally intended to provide a common data access method for the various Unidata applications. These deal with a variety of data types that encompass single-point observations, time series, regularly-spaced grids, and satellite or radar images.

NetCDF is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado.

The NetCDF is widely used nowadays. Its mailing list has over 500 addresses (some of which are aliases to more addresses) in thirty countries. Several groups have adopted NetCDF as a standard way to represent some forms of scientific data, varying from weather maps to brain imaging.

2.2.3 NetCDF Interface

The Network Common Data Form, or NetCDF, is an interface to a library of data access functions for storing and retrieving data in the form of arrays. An *array* is an n-dimensional (where n is 0, 1, 2 ...) rectangular structure containing items which all have the same *data type* (e.g., 8-bit character, 32-bit integer). A *scalar* (simple single value) is a 0-dimensional array.

NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable objects that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data. Generic utilities and application programs can access NetCDF datasets and transform, combine, analyze, or display specified fields of the data. The development of such applications may lead to improved accessibility of data and improved reusability of software for array-oriented data management, analysis, and display.

The NetCDF software implements an *abstract data type*, which means that all operations to access and manipulate data in a NetCDF dataset must use only the set of functions provided by the interface. The representation of the data is hidden from applications that use the interface, so that how the data are stored could be changed without affecting existing programs. The physical representation of NetCDF data is designed to be independent of the computer on which the data were written. Unidata supports the NetCDF interfaces for C, FORTRAN, Java, C++, and Perl and for various UNIX operating systems.

One of the goals of NetCDF is to support efficient access to small subsets of large datasets. To support this goal, NetCDF uses direct access rather than sequential access. This can be much more efficient when the order in which data is read is different from the order in which it was written, or when it must be read in different orders for different applications.

The amount of overhead for a portable external representation depends on many factors, including the data type, the type of computer, the granularity of data access, and how well the implementation has been tuned to the computer on which it is run. This overhead is typically small in comparison to the overall resources used by an application.

2.2.4 NetCDF Dataset

NetCDF Data Model

A NetCDF dataset contains *dimensions*, *variables*, and *attributes*, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset. The NetCDF library allows simultaneous access to multiple NetCDF datasets which are identified by dataset ID numbers, in addition to ordinary file names.

A NetCDF dataset contains a symbol table for variables containing their name, data type, rank (number of dimensions), dimensions, and starting disk address. Each element is stored at a disk address which is a linear function of the array indices (subscripts) by which it is identified. Hence, these indices need not be stored separately (as in a relational database). This provides a fast and compact storage method.

The Network Common Data Form Language (CDL)

We use a small NetCDF example to illustrate the concepts of the NetCDF data model. This includes dimensions, variables, and attributes. The notation used to describe this simple NetCDF object is called CDL (network Common Data form Language), which provides a convenient way of describing NetCDF datasets. The NetCDF system includes utilities for producing human-oriented CDL text files from binary NetCDF datasets and vice versa. The example can be found in Appendix A. The CDL notation for a NetCDF dataset can be generated automatically by using *ncdump*.

The NetCDF Markup Language (NcML)

NcML is an XML representation of NetCDF metadata, (roughly) the header information one gets from a NetCDF file with the "ncdump -h" command. NcML is similar to the NetCDF CDL, except, of course, it uses XML syntax. An example can be found in Appendix A.

The purpose of the Network Common Data Form (NetCDF) interface is to allow you to create, access, and share array-oriented data in a form that is self-describing and portable. "Self-describing" means that a dataset includes information defining the data it contains. "Portable" means that the data in a dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers. Using the NetCDF interface for creating new datasets makes the data portable. Using the NetCDF interface in software for data access, management, analysis, and display can make the software more generally useful.

2.2.5 Garbage Collection

In computing, **garbage collection** is a system of automatic memory management which seeks to reclaim memory used by objects which will never be referenced in the future. The part of a system which performs garbage collection is called a **garbage collector**.

When a system has a garbage collector it is usually part of the language run-time system and integrated into the language. The language is said to be garbage collected. Garbage collection was invented by John McCarthy as part of the first Lisp system. It is a bit sloppy to talk of a garbage collected language, however; being garbage collected is a property of an implementation of a language.

The basic principle of how a garbage collector works is:

- Determine what data in a program cannot be referenced in the future
- Reclaim the storage used by this data

Each data element (or "object") consumes some memory, of which there is a limited amount. When the objects are not used anymore, the memory allocated to these objects must be reclaimed. Programmers using languages such as C or C++ have to take care of reclaiming the memory themselves, but in Java the JVM reclaims the unused objects automatically through garbage collection.

An object is ready to be garbage collected when it is no longer "alive". The rules for determining if an object is alive are as follows:

- If there is a reference to the object on the stack, then it is alive.
- If there is a reference to the object in a local variable, on the stack, or in a static field, then it is alive.
- If a field of a live object contains a reference to some object, this object is also alive.

- The JVM may internally keep references to a certain objects, for example, to support native methods. These objects are alive.
- If object is not alive, it is dead and can be garbage collected.

One big difference in performance between JVM implementations depends on the garbage collector. The simplest garbage-collecting systems require all the threads in the system to wait while all the dead objects to be reclaimed. The smarter the garbage collector is, the better the performance is. There are three classes of garbage collection techniques: basic techniques, incremental tracing techniques and generational techniques.

2.2.6 Basic garbage collection algorithms

There are four main techniques for basic garbage collection: reference counting, mark-and-sweep, mark-and-compact and copying garbage collector.

In reference counting every object has a header field, where the counter for references to the object is stored. Each time a reference to the object is created, for instance when creating the object, the reference counter is incremented by one. Conversely, when a reference to the object is eliminated, the reference counter is decremented by one. When the reference counter drops to zero, the object is considered garbage and can be reclaimed. The benefit of this approach is that it can be easily made real-time, because the updates for the reference counters can be interleaved with the execution of the program. However, there is one minor and two major problems with reference counting. Firstly, the counters themselves need memory, as they have to be present in every object. Secondly, reference counting cannot handle circular references, i.e. the situation, where two objects reference each other and no other object reference to neither of the objects. Thirdly, reference counting reduces efficiency, because the amount of the work done in updating the reference counters is proportional to the amount of work done in the program.

Mark-and-sweep collection divides into two phases: distinguish the live objects from the garbage and reclaim the garbage. In the first phase the garbage collector starts with each reference in the local variable array an operand stack. It marks all these objects as alive. This applies to all the stack frames in the Java stack, not only the currently active one. Then the garbage collector checks all the fields in these objects. If the field points to an object, this object is also marked as alive. This process is recursively repeated until there are no more fields to check. Any object that was not marked as alive during the process is garbage. If the threads in the program were allowed to run during the garbage collection, a thread might move the last reference to a still-reachable object from storage location that has not been checked yet thus making a still reachable object appear unreachable and therefore garbage. That object would be reclaimed, since the garbage collector never saw a reference to it, making further references to the object return incorrect results or even crashing the system.

Mark-and-compact adds the de-fragmentation functionality to the mark-and-sweep collector. After the mark-and-sweep collection the memory can be fragmented as the garbage objects can are reclaimed anywhere from the heap. Mark-and-compact collection de-fragments the heap so that all the live objects are in continuous memory space.

Copying collection has a number of immediate attractions compared with other forms of automatic memory management. Like any non-incremental tracing collector, it places no overhead on user program writes. The cost of copying is proportional to the volume of live data rather than to the entire heap. This makes copying particularly attractive if the surviving data is a small proportion of the total heap. Copying garbage collection compacts active data structures into the bottom of the semi-space. This has three potential advantages. First, the heap is now a 'push-only' stack. Memory can be allocated linearly simply by incrementing a free space pointer by the size of the object to be allocated. Consequently space of variable-sized objects can be allocated for the same cost as other objects; complications of separate free-lists, or other fit-finding tactics, are unnecessary. Thirdly, compacting the active part of the heap onto fewer pages should reduce the size of the program's working set [7].

2.2.7 Incremental tracing collectors

For truly real-time algorithms, fine-grained incremental garbage collection appears to be necessary, because the garbage collection cannot be carried out as one atomic action while the program is halted. Instead, small units of garbage collection must be interleaved with small units of program execution. Although the reference counting seems to follow this paradigm well, its inefficiency and ineffectiveness discourages its use.

In incremental collectors the objects are traversed through as a graph. The difficulty is that because of the dynamic nature of the graph, it may change while the collector is traversing through it. Tricolor marking is one solution for this problem. Unlike in mark-and-sweep collector, where an object is either alive or garbage, in tricolor marking the object is "painted" white, grey or black. All the objects subject to garbage collection are initially painted white. As the marking process goes on, the objects are painted gray, if they have been traversed through, but their descendants have not.

Therefore, as the traversal proceeds outwards from the root, the objects are initially painted in gray. When they are scanned and the references to their descendants are traversed, they are blackened and the descendants are painted gray.

2.2.8 Generational garbage collectors

The disruptive stop-and-collect schemes are so disruptive because they have so much work to do--they must deal with the entire heap. If your heap is 600MB, then it has lots to do.

Observation: most objects live only a short time while some tend to live a long time.

The generational collector takes advantage of this observation by having a "younger" and an "older" generation. Recently allocated objects are most probably temporal, which means that they can be reclaimed soon. Therefore, objects are grouped as generations based on when they were created. The newer generations are checked more often, since they are more probably garbage. The older generations are checked less often. Objects that live a few "generations" (i.e., collection runs), are moved to the "older" generation, which reduces the amount of live objects the collector must traverse in the younger generation.

Note the similarity to the mark and copy algorithm; here, though, the "when to copy" algorithm is very different. A generational copying collector moves objects to another generation when it has survived a few generations. Mark and copy copies all live objects upon each activation, thus, not significantly reducing its workload for future generations. Also, there may be many generations, not just two spaces as in a mark and copy scheme.

In the end, even generational schemes must stop-and-collect the older generations. Hopefully this can be hidden from the user such as when the system is waiting for user input. Java does this by making the collector the lowest priority thread. When nothing else is running, the collector starts up (and hopefully finishes).

Garbage collection has come a long way in the last several years. Modern JVMs offer fast allocation and do their job fairly well on their own, with shorter garbage collection pauses than in previous JVMs. Tricks such as object pooling or explicit nulling, which were once considered sensible techniques for improving performance, are no longer necessary or helpful (and may even be harmful) as the cost of allocation and garbage collection has been reduced considerably [7].

2.3 Summary

In this chapter, we presented the background and the previous work done on the GIPSY. Then we take a research about the Unidata and NetCDF techniques, which are a popular data format in scientific computation. Finally, we briefly introduced the Garbage Collection Algorithms used in some languages. In the next chapter, we will explain the methods used to solve our particular problem and their theories, which is, as we will see, somewhat different from mainstream garbage collection as described in this chapter.

Chapter 3 : Methodology

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 and data dictionary generated by the front-end of the system in an efficient way. The Intensional Value Warehouse (IVW), adopted in GEE, purposes to improve the system performance.

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 executor performs generation and computation of the demands. The IVW is employed by GEE 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. The IVW, as an integrated component of the GEE in the GIPSY system, also interacts with the other component: RIPE.

The difficulty of implementing IVW is to approach the best efficiency to satisfy the performance of the whole system. This chapter will explain the techniques and tools used to construct IVW and their principles.

3.1 Motivation of Value Warehouse

One of the main goals of the GIPSY is the efficiency, then the GEE employing a warehouse is of prime importance to obtain high performance. There are two major factors affect the performance of executing Lucid program in distributed way, and also they are the reason why we need a value warehouse. First, the "weight" of each computation unit has critical influence on the communication overhead. Secondly, the recomputation of some reused interim values increases the undesired workload.

Granularity

In distributed system, the communication overhead is an obstacle for improving performance. Then, 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. One way is that we can reduce the communication overhead by decreasing the transmission. Then the value warehouse is necessary. The efficiency of that the machine retrieves the value locally from the warehouse is better than spreads demands out to the worker in the network.

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. In GLU, calculation functions were written in C considering two factors: Lucid was regarded as being more suitable for declaring the relationship between the funcitons 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. While the calculation functions, as the coarse-grain, may need much time to be worked out, then if we store the result into the warehouse, which will reduce the recomputation overhead for other computation. In sum, the GEE employing a value cache can reduce the demand for transmitting and help the system achieve the high performance.

3.2 Limitation of Previous Work

The value warehouse keeps track of the computed values along with their associated identifiers and contexts. The first draft of GEE did not include a value warehouse, and it was soon proved to be intolerably slow. Therefore, the GEE arranges the warehouse as a hashtable, which maps IC string (key) to values (value), to achieve efficiency.

The previous implementation of value warehouse was performed by declaring a variable of the java class hashtable. The class hashtable 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. However, it could not cope with the various Lucid program by tuning the performance. Also, 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 rehash method would be called frequently, which will obviously lower the performance. Moreover, it does not support remote access to values in the warehouse.

Consequently, new tools for IVW are required. The design and implementation of IVW is done towards efficiency and adaptability.

3.3 Solution Analysis

To build the IVW, we may have different choices. In this section, we discuss three possible solutions.

Solution I - a

Concretely, the warehouse should be developed at the beginning to meet the basic requirement of the GEE. According to the discussion of the NetCDF technique above, we construct the value warehouse by the data structure of NetCDF dataset, which provides direct access. The dataset stores the computed values so the IDP can extract them rather than compute anew. Since the amount of the values in the dataset may grow fast in some scientific computation program, we consider using garbage collection technique adapted to current situation. Therefore, in solution I - a, there are two main modules: NerCDF Dataset and Garbage Collector, which is shown in Figure 3-1. The solution is to reach the goal: efficiency.

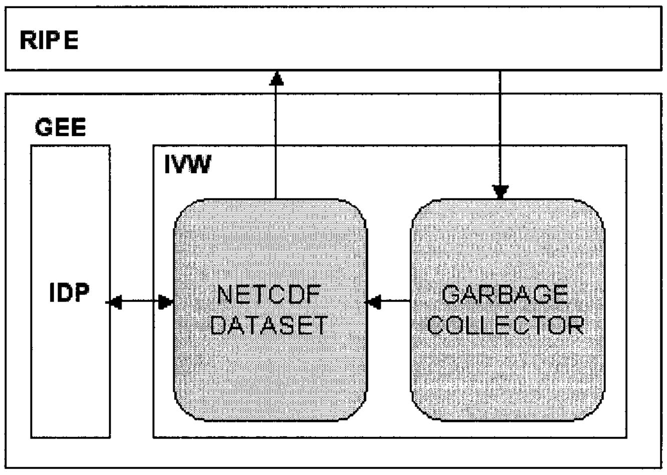


Figure 3-1: Solution I – a of the IVW

Solution I - b

The GIPSY may run on heterogeneous systems and the execution should be done in a real distributed environment for understanding parallelism in Lucid. Then, the values in the warehouse will be accessed by remote machines. Also, a program may be interrupted for some reason, and the interim values are useful to continue the computation. Thus, we need to store the values somewhere that can be accessed remotely and loaded to reuse. Since the NetCDF file works on different platform and can be accessed remotely using a suitable network file system, we build a file translator to manage those data between the file and memory and write values in the NetCDF dataset to the NetCDF file. So we get the solution I - b, shown in Figure 3-2, which improve the adaptability of the IVW.

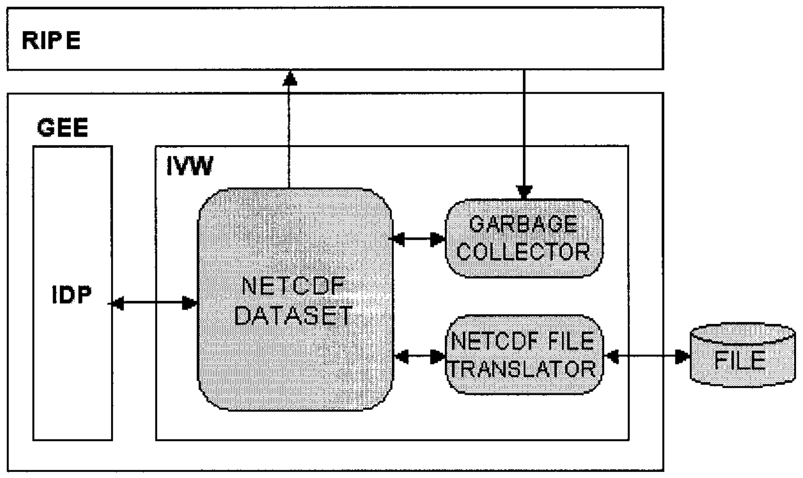


Figure 3-2: Solution I - b of the IVW

Solution II

In the evaluation process performed by the GEE, the IC string (described later) is adopted to express a demand for the value of an identifier in a context. Then, the suited data structure in the IVW will be helpful to efficiency. But the fixed data structure of NetCDF dataset brings the overhead of data conversion each time when the IDP sets or gets values from warehouse. To solve this problem and obtain high performance, we add a cache dataset module to the IVW. The result of solution II is that there are two level storages in the memory, which is shown in Figure 4-3.

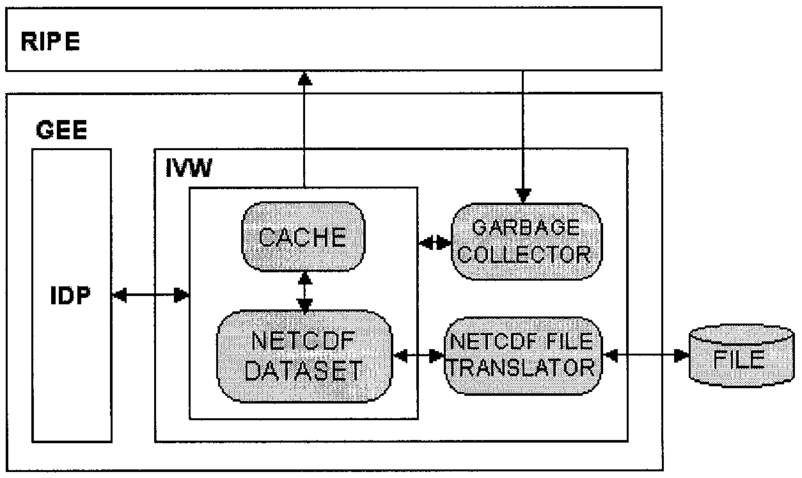


Figure 3-3: Solution II of the IVW

Solution III

Sometimes, there may be limited memory or in the GEE structure (Figure 2-4), each worker employs its own warehouse. To allow the IVW adapt different condition, we revise the Solution II. That is, the IVW is constructed by the cache dataset and there is also a file translator, which is responsible to translate the values removed from the cache dataset and write them to the netCDF file. Also, when not finding the IDP wanted value in the cache dataset, we will check the netCDF file and get the value from it. The solution III is shown in Figure 3-4.

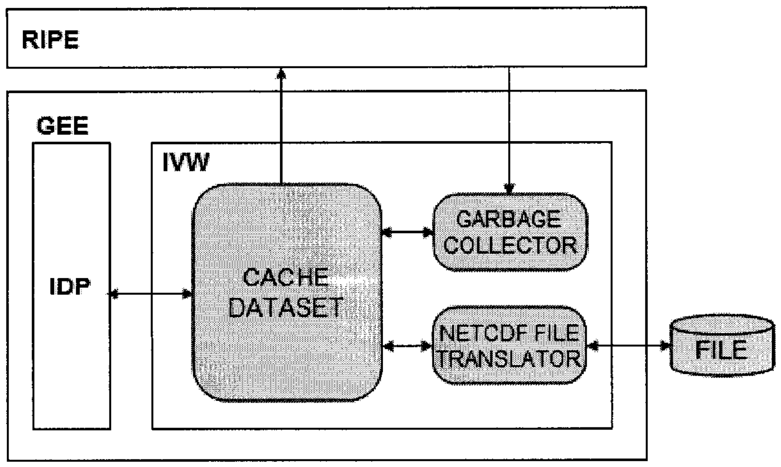


Figure 3-4: Solution III of the IVW

3.4 Multi-level Value Warehouse

In this thesis, we implement a multi-level value warehouse, which includes two level in memory: Cache dataset and NetCDF dataset. The IVW enables faster access to values accessed regularly. The Values that least accessed can be promoted to high level dataset and out-of-date computed results may be stored on the file system. Also, the cache dataset uses a fixed size that is selectable at startup, which avoids the cost of resizing the warehouse. When any level dataset is full, some garbage collector will be triggered to sweep the dataset and promote or erase values.

3.4.1 Cache Dataset

We adopt java class hashmap as the main data structure to build cache dataset. Comparing with hashtable, the hashmap, which implements map interface, has the following advantages:

- Map provides Collection-views in lieu of direct support for iteration via

Enumeration objects. Collection-views greatly enhance the expressiveness of the interface.

- Map allows you to iterate over keys, values, or key-value pairs; Hashtable did not provide the third option.
- Map provides a safe way to remove entries in the midst of iteration; Hashtable did not.

Further, Map fixes a minor deficiency in the Hashtable interface. Hashtable has a method called contains, which returns true if the Hashtable contains a given *value*. Given its name, you'd expect this method to return true if the Hashtable contained a given *key*, as the key is the primary access mechanism for a Hashtable. The *Map* interface eliminates this source of confusion by renaming the method contains Value. Also, this improves the consistency of the interface: contains Value parallels contains Key nicely.

The cache dataset here is responsible to enable us to cache in memory data that we can more slowly retrieve from an external source, such as a file, database, or network. Therefore, the IVW will approach the goal: efficiency.

3.4.2 NetCDF Dataset

NetCDF has experimental high performance for data access to its dataset. NetCDF uses direct access rather than sequential access to approach the goal that is to support efficient access. Normally, we can store and retrieve data directly by variable name and index; hence, it is valuable for the use of memory datasets or file datasets on disk. NetCDF data has following features:

- *Self-Describing.* A NetCDF file includes information about the data it contains.

- *Architecture-independent.* A NetCDF file is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- *Direct-access.* A small subset of a large dataset may be accessed efficiently, without first reading through all the preceding data.
- *Appendable.* Data can be appended to a NetCDF dataset along one dimension without copying the dataset or redefining its structure. The structure of a NetCDF dataset can be changed, though this sometimes causes the dataset to be copied.
- *Sharable.* One writer and multiple readers may simultaneously access the same NetCDF file.

Also, a measure of success has been achieved. NetCDF is now in use on computing platforms that range from CRAYs to personal computers and include most UNIX-based workstations. It can be used to create a complex dataset on one computer (say in FORTRAN) and retrieve that same self-describing dataset on another computer (say in C) without intermediate translations—NetCDF datasets can be transferred across a network, or they can be accessed remotely using a suitable network file system.

3.5 Garbage Collection

The garbage collection technique adopted in the IVW is one of the important characteristics.

Why Garbage Collector

One of the major goals of the design of the GIPSY is towards efficiency. To reach this goal, an important strategy adopted in GIPSY is "lazy evaluation", which is the eduction computation model. That is, value of variables is only computed when needed in order to avoid 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.

The GEE uses the data flow's context tags to build a store of values that already have been computed (the IVW). Since the IDP enquires the value warehouse each time before it does calculation, there would be intensive enquires demanded on the warehouse. Therefore, those stored values should be carefully arranged for being quickly retrieved.

Also, since Lucid tends to be used in large scientific computation, the number of computed values may increase dramatically, the IVW contains a Garbage Collector to discard the less "valuable" data and keep the warehouse at a reasonable size, which will definitely be good for speedy inquiries. Thus, one of the key concerns when using caches is the use of a garbage collecting algorithm adapted to the current situation. The use of a garbage collector configured or adapted to the current situation is of prime importance to obtain high performance.

How Garbage Collector works

Since we use java language to develop the IVW and the whole GIPSY system, the IVW will apply memory form virtual machine. So as the virtual machine has enough memory to fit all the interim values in the IVW, the cache dataset and the NetCDF dataset will keep them in memory. If memory becomes scarce, however, the garbage collector may be triggered and decide to clear the values in cache dataset and promote them to NetCDF dataset, furthermore, store them to file. Then the space in memory occupied by some useless data will be reclaimed. The next time the program needs to use that data, it will have to be reloaded from the external source or recomputed.

Algorithms

Normally, the Garbage Collection is a system of automatic memory management which seeks to reclaim memory used by objects that will never be referenced in the future. In the GIPSY system, it is also to manage memory automatically, but its main task is to seek the comparative useless values in the warehouse. To keep the warehouse in a reasonable size, we can define an appropriate size in the initialization of the system and do the garbage collection when the values make the warehouse full. Then the following strategies of garbage collection are taken into account:

First, those values that are least accessed could be swept or promoted to high storage level. The least accessed values mean that they might not be needed for future computation or they would be required in a long period. Thus, we remove them to reclaim the spaces for the values accessed at prime tense.

Secondly, we may consider purging the values in the warehouse that are out of date. That is, we can define an expire time for the values and let the garbage collector check the warehouse periodically to seek and sweep those values whose existing time bigger than the expire time. Consequently, this algorithm focuses on removing the earliest generated values.

At last, there is another very important fact that should be considered, which the time for a value to be work out. Those values that take the shortest time to be produced would be discarded first; on the contrary, the values that needs a awfully long calculation time should be kept in the warehouse even they are not accessed for a long time. These additional information, such as computation time of a value, can be supplied by the GEE or the IVW makes decision through the time that such value is enquired and set.

In the implementation, these strategies could also be combined, which is called hybrid algorithm. Also, the design of the IVW should be easy to add more algorithms in the future or accept external garbage collection algorithm. We may provide the function to the users that they can choose one of the algorithms at the beginning of the computation or in the runtime. All of them above will help the warehouse achieve the design of the goal: adaptability.

3.6 Summary

In conclusion, we implement the components in multi-level warehouse with garbage collection technique. The design and the implementation of the IVW reflect the goals: efficiency and adaptability. In the next chapter, we will discuss the concrete implementation.

Chapter 4 : Implementation

In this chapter, we describe different implementations of the IVW, starting with a design rationale, to the implementation of various architectures and algorithms for caching and garbage collection.

4.1 Design Rationale

Design for change is the most important principle concerned in the architecture design. A successful software system has to be maintainable to survive the evolution of its changing requirements and environment specifications. Thus, a successful design must be easy to change. Here, we will introduce the nature of change and how to fulfill the goal of design for change.

When building software, one has to make the rightful assumption that everything can be required to change over time. It can be change of algorithm, change of data representation, change of underlying abstract machine, change of peripheral devices, change of social environment, and so on. Any change may lead software updating, from component to detailed algorithm.

The main strategies to solve the problem of design for change are information hiding and modularization. Modularization requires building a loosely coupled system of cohesive modules. A cohesive module provides a small number of closely related services. A loosely coupled system ensures that a change to a module will not impact other modules at all, or minimal-impact changes.

Information hiding encapsulates a module by only allowing access to module services via a minimal yet complete interface. The implementation details are hidden, therefore limiting the propagation of changes when a change is required.

Finally, design should keep things as simple as possible so that a design can be understood with minimal overhead before it can be changed.

4.1.1 Design Patterns

Software architects generally think in terms of high-level abstractions rather than low-level programming details. Representing a system in terms of high-level abstractions promotes understanding of the system and reduces its perceived complexity. One such abstractions mechanism is software design patterns. Design patterns are successfully applied in almost all software to simplify and solve recurring problems in software design.

Façade pattern

The IVW model is a subsystem in the GEE of the GIPSY with many classes. Each class has its own interface, which provides services to other classes in the whole system and hides the implementation details from others. While a class wants to use the services provided by another subsystem, it does not care in which classes of this subsystem the service is implemented, or how it is implemented.

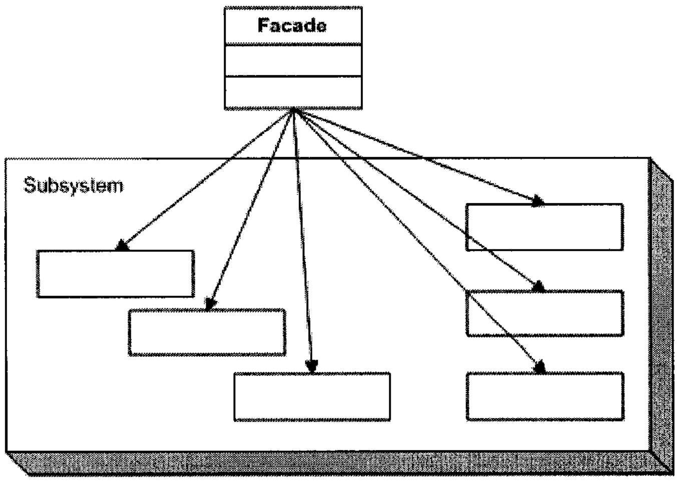


Figure 4-1: Façade design pattern

In order to make the subsystem easy to use, a high level interface for the IVW model subsystem is created. Here, we use façade pattern (represented in Figure 4-1) to decouple the IVW subsystem from the other subsystems in the GEE. We create an interface and façade class for the IVW subsystem. The public IVW interface as well as its implementation subclass constructs the façade of value warehouse. They are declared as public and exposed to the outside of the IVW subsystem. If the other subsystems depend on the IVW subsystem, the dependency is simplified by making communication between those and the IVW façade.

Factory pattern

The factory design pattern (represented in Figure 4-2) is used to create objects as a Template Method to implement different service "instances" that have different implementations. A superclass specifies all standard and generic behavior, and then delegates the creation details to subclasses that are supplied by the client.

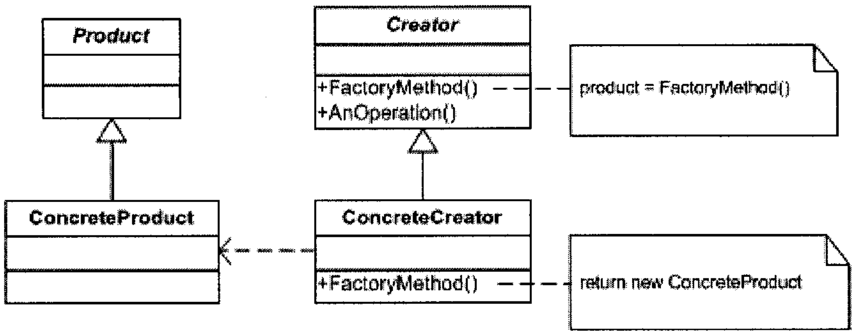


Figure 4-2: Factory design pattern

The Factory pattern makes a design more customizable. We call this a Factory pattern since it is responsible for "Manufacturing" an object. It helps instantiate the appropriate subclass by creating the right object from a group of related classes. The Factory pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code.

Factory methods are sometimes a more flexible way to instantiate a class than directly calling the constructor of the class, for the following reasons:

- Unlike constructors, factory methods are not required to create a new object each time they are invoked. Factory methods can encapsulate the logic required to return a singleton instance of the class requested, or they can return an instance of the requested class from a pool of instances.
- Factory methods can return any subtype of the type requested, and can require that clients refer to the returned object by its interface, rather than the implementation class. This enables an API to return objects without making their classes public.
- The class for the object returned by a factory method need not even exist at the time the factory method is written. This is one of the classic benefits of polymorphism: "old code uses new code," which means that new classes

can be added, and their instances returned by the factory method, without changing any of the existing code.

In our design of the garbage collector, we implement it using Factory methods design pattern because the IVW provides several garbage collection algorithms to users. Through Factory method pattern, users can select one of the algorithms they prefer at the beginning of the system or even at the run time. Also, some new algorithm can be added easily without changing the existing code, which makes the IVW more adaptable.

4.1.2 Implementation Language

As with all the other parts of the GIPSY, 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 networking technologies. Generally speaking, Java shows the following characteristics:

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

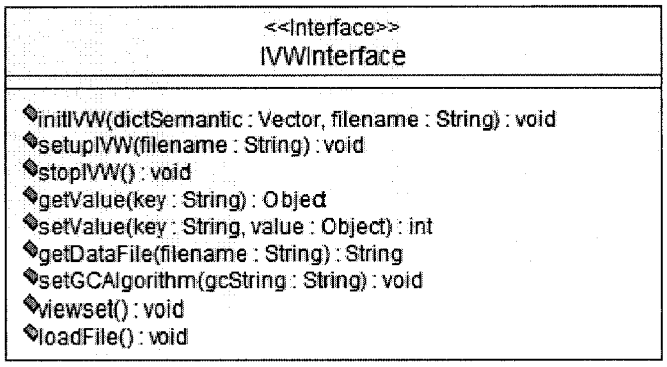
Java is portable because its code is translated into "bytecode" that can be interpreted using a Java Virtual machine (JVM) that has been compiled in the operating system version in which we want to run the program. Consequently, the Java bytecode can be understood by a computer installed with any architecture or operating system so long as it supports JVM natively, or a JVM can be compiled in it. 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 because of the interpreted mode of execution, which is the basis for most of the criticism against Java.

4.2 External Design

This section presents the design of the IVW from an external point of view, i.e. the façade interface that is provided by the IVW implementation to the other components of the GIPSY.

4.2.1 External Software Interfaces

Since we use façade pattern to design the IVW, which hides the implementation details from other components in the GIPSY system, then other parts can only interact with the IVW by the interface provided by the IVW. Figure 4-3 shows this interface.



```
<<Interface>>
IVWInterface

initIVW(dictSemantic : Vector, filename : String) : void
setupIVW(filename : String) : void
stopIVW() : void
getValue(key : String) : Object
setValue(key : String, value : Object) : int
getDataFile(filename : String) : String
setGCAlgorithm(gcString : String) : void
viewset() : void
loadFile() : void
```

Figure 4-3: External interface of the IVW

void ***initIVW***(Vector dictSemantic, String filename);

Initializes the Value Warehouse NetCDF file, Gets the name of lucid program and reads the data from data dictionary.

param dictSemantic Vector - which contain the dimensions and variables

param filename String - the name of the lucid program file

void ***setupIVW***(String filename);

Sets up the Value Warehouse. Loads the NetCDF file to memory, then initialize cache and NetCDF data set.

param filename String - NetCDF file that contain the definition

void ***stopIVW***();

Stop the Value Warehouse. Write the data in cache to NetCDF dataset, then write whole NetCDF data set to NetCDF file.

Object ***getValue***(String key);

Get the value from Warehouse associated the given IC String.

param key String - the key whose associated value is to be returned.

return Object - return the value for the IC String, or null if Warehouse does not contain this key.

int ***setValue***(String key, Object value);

Put the value with the specified key in Warehouse, If Warehouse previously contained a value for this key, the old value is replaced. If the rate of cache is more than the limit, the Garbage Collector will be triggered.

param key String - the key whose associated value is to be set.

param value Object - the value for the key

String ***getDataFile***(String filename);

Get the NetCDF Data File.

param filename String – name of a NetCDF fil

return String

void ***setGCAlgorithm***(String gcString);

Set the garbage collection algorithm to decide which algorithm will be used on next garbage collecting. Default is "MarkSweep".

param gcString String - description of GC algorithm

void ***viewset***();

View data in the cache.

4.3 Internal Design

As the most critical part in developing an object-oriented system, the architectural design determines the top-level system structure of the IVW. A good system architecture will let us keep an easy way to design a system.

Structuring a system into subsystems helps to reduce complexity. Our design model divides the IVW system based on the function module. Therefore, there are four main components in the IVW (as depicted in Figure 4-4):

1. cache dataset manager
2. NetCDF dataset manager
3. NetCDF file manager
4. garbage collector

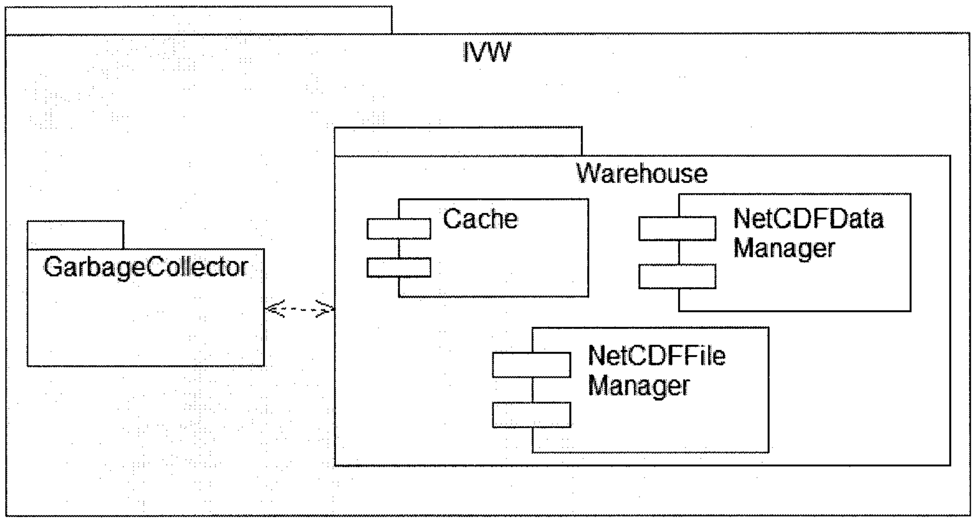


Figure 4-4: Components of the IVW

4.3.1 Overview of the Components

The *Warehouse* subsystem is the controller whose role is to control the execution flow of the whole IVW system. It also takes a role of the façade of the system. It is visible to the other subsystems of the GEE. It takes the responsibility to response the GEE's requests for values of its generated demands and to offer the storage and retrieve services.

The task of the *Cache* component is to act as a cache of the warehouse. It provides fast access for the values in the warehouse upon request.

The *NetCDF Data Manager* component is a higher-level layer of storage of the values in the warehouse. It stores and manages the values less accessed (i.e. those that have already been garbage-collected), as well as exchanges some values with the cache dataset and the NetCDF file manager.

The *NetCDF File Manager* component is to initialize, write, and load files stored on the file system in the NetCDF file format. Also, it provides the data file to the clients if needed.

We will further discuss each component in the following sections.

4.3.2 Dependency Relationships among Components

In the architectural design, the idea of building the design model for the potential solution is to capture the relationships between the components and define the top-level classes for each component. The relationships between the components reflect the flexibility and extensibility features of the warehouse. We aim for reducing the coupling among the components.

To reduce the coupling, the major components of the IVW do not have mutual knowledge of each other. The class *IVWControl* coordinates the execution of the system and the message passing between these components. It also plays a role of the creator and the owner of the instances of the top-level classes in these components. Thus, the relationships among the cache, *NetCDF Data Manager* and

NetCDF File Manager component can be designed as the association relationships between each of the system and the controller via the callback implementations (see Figure 4-5).

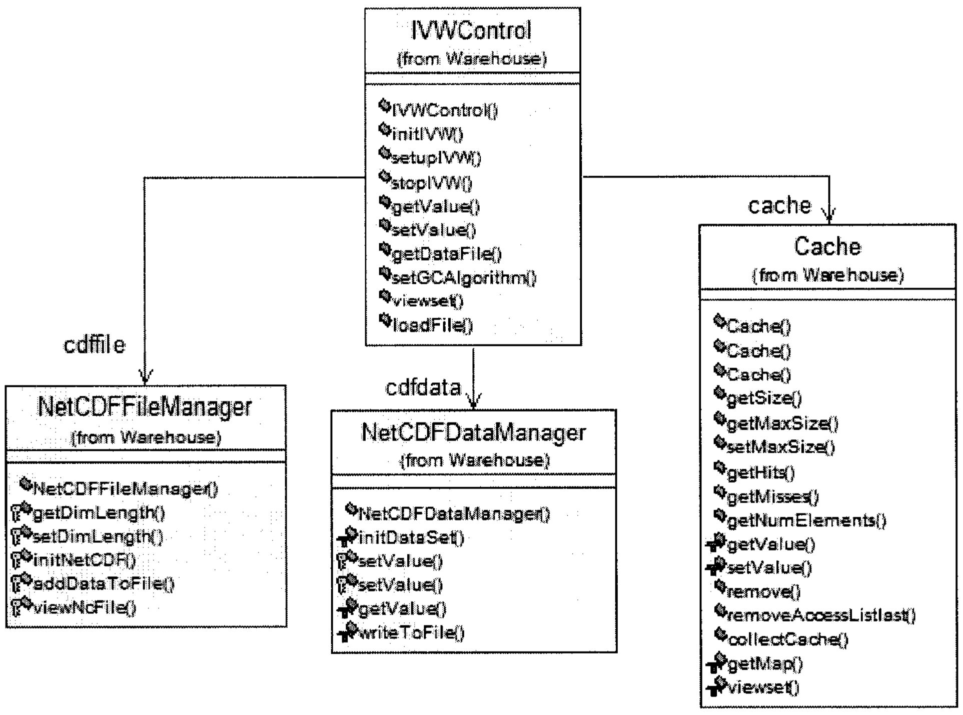


Figure 4-5: Dependency relationships between the IVW components

4.4 Detailed Design

The GEE employs the *eduction* (i.e. demand-driven) computation model and uses a value cache to avoid redundant computations. It takes the abstract syntax tree and dictionary produced by the GIPC (see Chapter 2), and propagates demands required for working out the desired value. The computed values are placed in the value warehouse, and the GEE looks up values in the warehouse each time before it propagates demands or calls a computation method to avoid redundant computations.

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 in many cases (see Chapter 5 : for results using various cases).

4.4.1 Cache

Our first draft of the IVW just included one dataset layer, the NetCDF dataset. To improve the efficiency of access to the IVW, we implement two dataset layers. The cache dataset and the NetCDF dataset adopt different data structures to store values for variables in a specified context, which is more appropriate to cooperate with other components in the GEE.

IC String

As designed an implemented by Bo Lu in her PhD thesis [4], the GEE represent a demand for a program identifier in a context as an instance of the associated class possessing a specified context, which is called an *IC* object (*I* stands for "identifier" and *C* for "context"). It can be cited as: (1) exchanged information between a client and server, (2) a demand, and (3) a key of value warehouse. It is demanded to design a suitable data structure, preferably consisting of a primitive data type, to replace the object. Thus, a *String (IC string)* is employed in the GEE, which can be speedily transmitted over the network, quickly generated, easily stored, and rapidly matched to an *IC* instance and for inquiries performed in the value warehouse.

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@[x:2, y:1] is "*2:2:1*". In this example, the program has only two dimensions. The *IC string* as the key and together with the value constructs a pair to be stored in the value warehouse. Similarly, the demand propagator retrieve the value from the warehouse associated the key (*IC string*).

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 cache dataset of the warehouse as a *hashmap*, which maps *IC strings* (key) to *values* (value). Large numbers of elements may be stored and retrieved efficiently. The cache class diagram is shown in Figure 4-6.

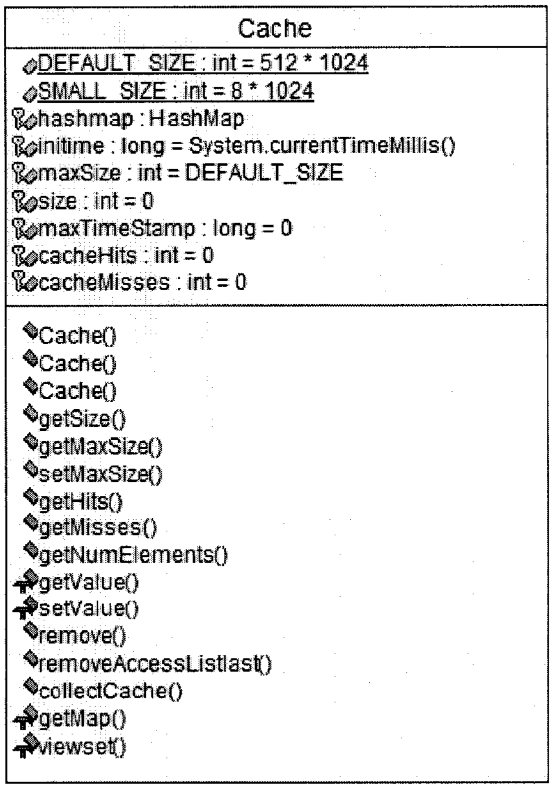


Figure 4-6: *Cache* class diagram

The cache dataset stores values associated with unique keys in memory for fast access. We provide a class *cachesizes* to calculate the size of the values in the cache, which allows the cache to determine object size much more quickly. This allows a cache to never grow larger than a specified number of bytes.

4.4.2 NetCDF Data Manager

The NetCDF dataset is the main storage that is used to store and manage values as the second level in memory. We implement it using the Java package of NetCDF interface: ucar.ma2, known as "MultiArray version 2" package. The ucar.ma2 package is independent of the ucar.nc2 package, and is intended for general multidimensional array use. Its design is motivated by the needs for NetCDF data to be handled in a general, arbitrary rank, type independent way.

It is often critically important for performance that the movement of data between memory and disk is carefully managed. The data in a NetCDF file, for example, is stored out of memory on a local or network file, and the NetCDF library allows efficient extraction of data subsets. At the same time, it is sometimes a useful abstraction to handle data in a general way independent of storage location. A ucar.ma2.Array object has its data in the computer's main memory, and so is said to be *memory-resident*.

4.4.3 NetCDF File Manager

The *NetCDF File Manager* is responsible of dealing with the NetCDF files. It creates a NetCDF file when the IVW is initialized. To fulfill the initialization, it adds dimensions, variables and attributes to the NetCDF file. Thus, when a file is first opened, it is in "define mode" where these may be added. During execution, all of them will be loaded to the memory set when the IVW is set up. Once the *create()* function is called, the dataset structure is immutable. After *create()* has been called we can then write the data values. After executing the whole program, it will write all the data values in memory set to the NetCDF file.

4.4.4 Garbage Collector

The IVW consists of two components: *Warehouse* and *Garbage Collector*. The Garbage Collector is independent of the warehouse in order to reduce the coupling. Note that it is possible to have an IVW that does not have any garbage collecting feature. We implement the garbage collector using the Factory design pattern, which helps to meet the requirement of adaptability and extensibility. Figure 4-7 shows the relationships among the classes of the *Garbage Collector* component.

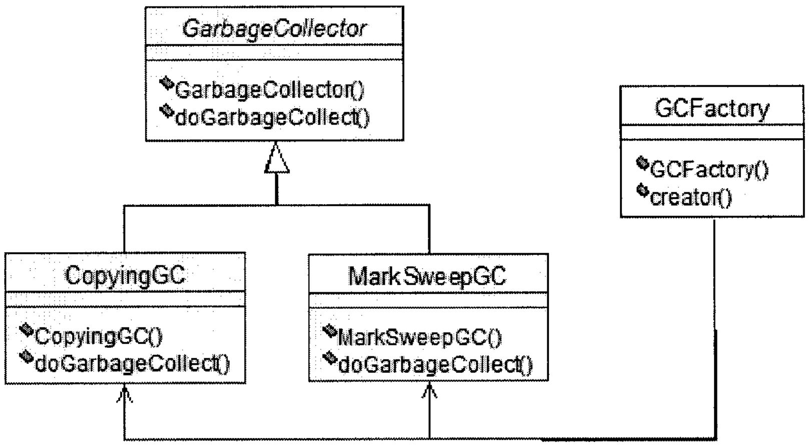


Figure 4-7: *Garbage Collector* classes

4.5 Use of the IVW

4.5.1 Initialization of IVW

In the front-end of the GIPSY, the GIPC generates an Abstract Syntactic Tree (AST) and a dictionary. We use this dictionary to initialize the value warehouse that includes dimensions and variables with their attributes, which will be used at run time. Since we implement the IVW using NetCDF dataset, here we initialize the components of the NetCDF dataset.

A NetCDF dataset contains *dimensions*, *variables*, and *attributes*, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset.

We use an example of Lucid program (Figure 4-1) to illustrate the initialization process.

```
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 4-8: A Lucid program

The GIPC generates an Abstract Syntax Tree (AST) (Figure 4.2) and a Dictionary (Table 4-1) for the program. We just present part of the attributes of the dictionary in terms of the definition of the class Dictionaryitem(), which are necessary to the value warehouse.

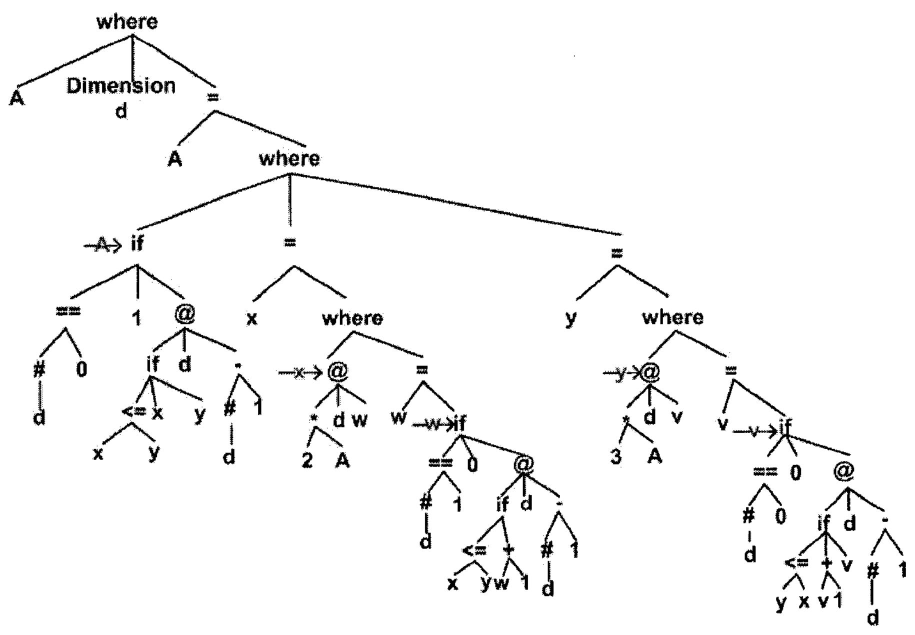


Figure 4-9: Abstract Syntax Tree of the program in Figure 4-8

ID	Name	Kind	type	rank
0	Start		2	
1	d	dimension	2	
2	A	identifier	0	1,
3	x	identifier	0	1,
4	w	identifier	0	1,
5	y	identifier	0	1,
6	v	identifier	0	1,

Table 4-1: Example of data dictionary

ID:	the unique code of each identifier
Name:	the name of each identifier
Kind:	dimension or identifier
Type:	0-int, 1-float, 2-string, 3-boolean
Rank:	the rank of the identifier

We can make a few observations about this dictionary:

- There is only one dimension: d, which type is string.
- It defines 5 other identifiers: A, x, w, y, v.
- The type of all the identifiers is integer.
- The rank of all the identifiers is dimension d.

Network Common Data form Language (CDL)

According to the concepts of the NetCDF data model, it includes dimensions, variables, and attributes. The notation used to describe this simple NetCDF object is called CDL (network Common Data form Language), which provides a convenient way of describing NetCDF datasets. The NetCDF system includes utilities for producing human-oriented CDL text files from binary NetCDF datasets and vice versa. The CDL notation for a NetCDF dataset can be generated automatically by using *ncdump*, a utility program described later. Another NetCDF utility, *ncgen*, generates a NetCDF dataset from CDL input.

The CDL notation is simple and largely self-explanatory. It will be explained more fully as we describe the components of a NetCDF dataset. CDL statements are terminated by a semicolon. Spaces, tabs, and new lines can be used freely for readability. Comments in CDL follow the characters '//' on any line. A CDL description of a NetCDF dataset takes the form

```
NetCDF name {
   dimensions: …
   variables: …
   data: …
}
```

where the *name* is used only as a default in constructing file names by the *ncgen* utility. The CDL description consists of three optional parts, introduced by the keywords dimensions, variables, and data. NetCDF dimension declarations appear after the dimensions keyword, NetCDF variables and attributes are defined after the variables keyword, and variable data assignments appear after the data keyword. Also see example in Appendix A.

Algorithm

One local call is needed to create a NetCDF dataset, at which point we will be in the first of two NetCDF *modes*. When accessing an open NetCDF dataset, it is either in *define mode* or *data mode*. In define mode, we can create dimensions, variables, and new attributes, but we cannot read or write variable data. In data mode, we can access data and change existing attributes, but we are not permitted to create new dimensions, variables, or attributes.

The main algorithm of this part is to traverse the dictionary, which adopts vector to store the dimensions and identifiers. We can distinguish the dimension or identifier by checking the type of each item in the dictionary. We still use the dictionary mentioned above (Table 4-1) to describe creating NetCDF Dataset.

Begin from the "start" item in the vector of dictionary and ignore it because its kind is neither "dimension" nor "identifier". Then get to the second item "d", and we create a dimension named "d" in the NetCDF dataset cause its kind is "dimension". After that, read the next item "A". Since its kind is "identifier" and type is "0", we create a variable with type integer and make it an attribute "id" and assigned a value "2". We record the "id" because 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. Here all the identifiers in the dictionary will be "variable" in the NetCDF dataset. In the same way, we can create other four variables "x, w, y, v". Thus, we initialize the NetCDF dataset when we invoke function *create()*, which will be used in execution process. For the example dictionary we can initialize the following NetCDF dataset:

```
ncfile = NetCDF
G:\GIPSY\gipsy\src\gipsy\tests\lucid\t2.ipl.nc {
   dimensions:
       d = 8;
   variables:
         int A(d);
         :id = "2";
         int x(d);
         :id = "3";
         int W(d);
           :id = "4";
       int y(d);
           :id = "5";
       int V(d);
           :id = "6";
}
```

4.5.2 Setup of the IVW

In the GEE multithreading and distributed implementation, they convert each identifier in a Lucid program to Java class, *IC* Class, rather than operating on the abstract syntax tree directly. The conversion is done by a component called *code generator*. After that, the GEE will perform the whole execution for the program by *code executor*. That is the reason why we separate the initialization and setup of the IVW.

Together with the code generation, we fulfill the initialization of the IVW, known as a NetCDF file without data. Then, the GEE will go to evaluation process that is started by a demand for the value of an identifier in a context until the desired result has been found. It can be observed that, in a complex computation, some values may be needed more than once. To avoid redundant calculation, the computed values are stored in value warehouse. Therefore, we set up the value warehouse to store data values when the evaluation process starts. Setting up the IVW includes setup of cache dataset and NetCDF dataset.

The main data structure of the cache dataset is *hashmap.* Also, we construct linked list to record the access sequence of the values and the time to work them out.

Cache size

It is known that increasing cache size reduces miss rates of instruction caches, data caches and unified caches for conventional programs [Hennessy and Patterson, 1996]. For example, Table 4-2 shows the miss rates for a direct-mapped data cache on DECStation 5000 for an average of SPEC92 benchmarks [Gee et al., 1993].

Zorn confirms these findings for a range of Lisp programs supported by generational mark-sweep garbage collection: caches larger than 512 kilobytes performed substantially better than smaller ones [Zorn, 1991].

Cache size (kilobytes)	Miss rate (percent)
1	24.61
2	20.57
4	15.94
8	10.19
16	6.47
32	4.82
64	3.77
128	2.88

Table 4-2: Data cache miss rates for the SPEC92 benchmark suite on a DECStation 5000 [Gee et al., 1993]

Consequently, we define a default size of the cache, which are 512K bytes. Also, we provide other two choices to users, small size and customer size. The small size is for the program that requires a small quantity of values. While by defining customer size, users can have arbitrary size that they want under the condition of physical memory.

After that, to set up the NetCDF dataset, we will load data from the NetCDF file that was created on initialization. NetCDF dataset Is implemented by the package "ucar.ma2.Array". Array is the abstraction for multidimensional arrays of primitives with data stored in memory. Arrays can have arbitrary rank, and there are concrete implementations for arrays of rank 0-7 for efficiency. The underlying storage can use any of the Java primitive types (double, float, long, int, short, byte, char, boolean). The data can be accessed in a type independent way, for example *getDouble()* can be called on an Array of any numeric type. The implementing class casts the data to the requested type (and throws a runtime *ForbiddenConversionException* if the cast is illegal), or uses a direct assign when the requested type is the same as the data type.

All of the complicated manipulation of data happens on the memory-resident Array object obtained from the *read()*. The data type, rank, and shape of an array are immutable, while the data values themselves are mutable. Generally this makes Arrays thread-safe, and no synchronization is done in the Array package.

ArrayAbstract is the superclass for the implementations of *Array*, which use Java 1-D arrays for the data storage. There is a concrete class that extends *ArrayAbstract* for each data type, e.g., *ArrayDouble.* For each data type, there is a concrete subclass for each rank 0-7, for example *ArrayDouble.D3* is a concrete class specialized for double arrays of rank 3. These rank-specific classes are static inner classes of their superclass. This design allows handling arrays completely generally (through the Array interface), in a rank-independent way (though the

Array<Type> classes), or in a rank and type specific way for ranks 0-7 (through the *Array<Type>.D<rank>* classes).

In our case, we retrieve the variables from the NetCDF file and their rank (number of the dimensions) and shape (size in each dimension), then, build an Array for each variable.

For example: we can directly instantiate the type-specific subclasses with an arbitrary rank. These also add type-specific get/set accessors.

```
public class ArrayDouble extends ArrayAbstract {
    // constructor
    public ArrayDouble(int [] dimensions);
   // type-specific accessors
    public double get(Index i);
    public void set(Index i, double value);
        ...
  }
```

4.5.3 Data Access

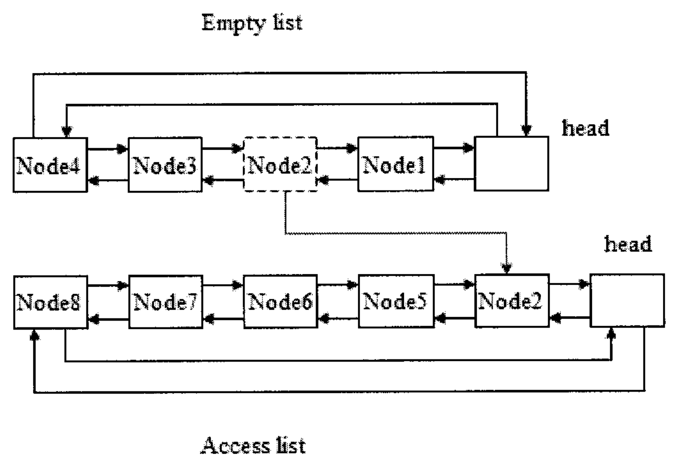
In the process of execution, the IVW works as a data cache of the GEE and the IDP will access value warehouse frequently. Each time, when a new demand is generated, the IDP will check value warehouse first to see if there is the value that it wants in the warehouse. If the warehouse returns no value of the requirement, the demand will be dispatched to workers to work it out.

The GEE implementation presents a demand for an identifier in a context as IC string, such as "2:2:1", which describes the demand A @ (x: 2, y: 1). If the *IC string* as the input inquiry matches the key exactly, the warehouse return the matched value. Otherwise, the warehouse will return null if the requested value is not found. The cache dataset of the IVW is optimized for fast access. The algorithm for cache dataset is as following: a *Hashmap* is maintained for fast value lookup. Besides the *Hashmap*, we maintain three linked lists in the cache dataset: access list, empty list and age list. The access list keeps objects in the order they are accessed from warehouse; the empty list contains the accessed objects without values; and the age list keeps objects in the order they were originally added to warehouse.

Here to implement the linked lists, the *LinkedList* in Java Collections package is not adopted, while we implement one own *LinkedList,* which is a double linked list. The main feature of this *LinkedList* is that list nodes are public, which allows very fast delete operations when one has a reference to the node that is to be deleted. This feature will reduce the waste of the garbage collection, which is described later.

When an inquiry is coming, the IVW will check if there is the value matching the given IC string. If not, we create an element with empty value of the hashmap in the cache dataset, then make a node for it and add the node to the first of empty list. Thus, the nodes in the empty list are all associated with the elements that have empty values. Each node maintains an attribute: *timestamp,* which records the calculation time of a demand. To get the time, we write the current time to timestamp here named creation time. It is stored as long value and represents the number of milliseconds passed since January 1, 1970 00:00:00.000 GMT.

After a demand is worked out by some worker, the IDP will store the value into the warehouse, which may be reused for other computations. At this time, the pair of the IC string and the value is given to the warehouse. The cache dataset accepts them and replace the value of the matched element that has empty value, and then removes the node associated with this element from empty list, add it to the first of the access list, as well as put its age node to the first of the age list, which is shown in Figure 4-10. Also, we use current time minus the creation time and get the calculation time. The value with a big calculation time will be reserved in warehouse for a long time to reduce the calculation overhead.



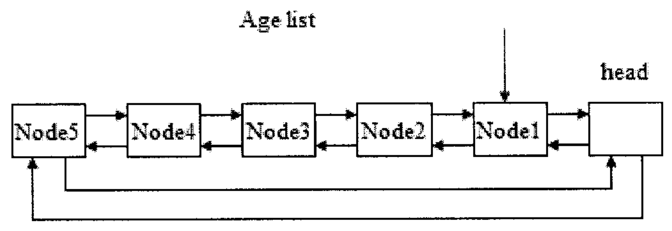


Figure 4-10: Node from Empty list to Access list

Add a node to the first of Age list

Each time, when some value is accessed by the IDP, it will be moved from its position to the first of the Access list. As a result, in the Access list, those values that are accessed recently will be always in the front, which is shown in Figure 4-11.

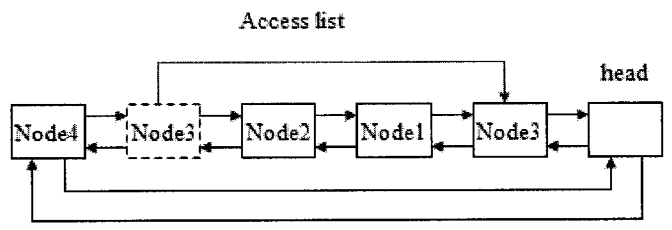


Figure 4-11: Node is moved to the head of Access list

4.5.4 Garbage Collecting

We use *hashmap* as the main data structure in the cache dataset, which maps *IC string* (key) to *values* (value). When each value is added to cache dataset, it is first wrapped by a CacheElement, which maintains the size of the value (in bytes) and the reference to the *listNode* in some linked list in the cache dataset. Then, we store the pair (IC string, CacheElement) in the hashmap. The relationships of them are shown in Figure 4-12. To get a value from the cache dataset, a hash lookup is performed to get a reference to the CacheElement that wraps the real value we are looking for. The element is subsequently moved to the front of the accessed linked list.

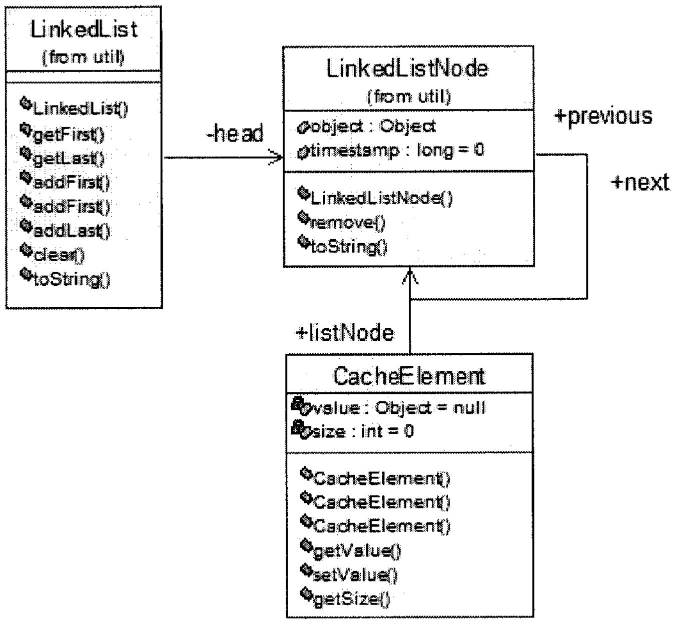


Figure 4-12: Relationships of LinkedList, LinkedListNode and CacheElement

The *LinkedList* used in the IVW is a double linked list that has a head node. It only provides functions to add a node to the first or to the end. Therefore, we will never insert a node to the middle of the linked list for our use. In addition, the *LinkedListNode* has references of previous node and next node, as well as, a remove method, which allows user easily remove the node from the linked list that it is located in. That is, having the reference of the node, we can remove the node from some linked list quickly and avoid linear scans of the linked list. In a word, the implementation of the *LinkedList* and *LinkedListNode* gives us fast delete operations and special insert operations.

Comparing with the *LinkedList* in java package, the feature of fast delete operation of our implemented *LinkedList* is helpful for speedy garbage collecting. The class *LinkedList* in java package provides two *remove*() functions: remove(int index) and remove(object o) [9]. For the first one, users must know the position of the object in the list, which will make the code more complicated, and searching the index also needs time. The second one removes the first occurrence of the specified element in the list, which definitely needs more time than our implementation. From the above reasons, we use our own *LinkedList* rather than the one in java package.

According to the above feature of the linked list, we can quickly perform the garbage collecting when necessary. We configure a maximum size for the cache dataset when the warehouse is started up. Also, there is a current size that equals to the sum of the objects in the cache dataset. Each time, when some value is put in the cache dataset, the current size will be added by the value size. Normally, the garbage collector will be triggered when the cache dataset is too full, for example, we may define "too full" as within 5% of the maximum cache size and define a desired rate of the cache dataset, such as 20%, which indicates the cache dataset is at least 20% empty.

We use an example to illustrate a garbage collecting process. In this example, the least frequently accessed values will be promoted from the cache dataset to the NetCDF dataset. When the current size is more than the 95% of the maximum size, the *doGarbageCollect()* method will be invoked. Then, the last node in the access list will be chosen for removing first. The first step is to delete that element from the *hashmap* and at the same time subtract the value size from the current size. Next, remove the node from the access list. At last, make the reference to the node in the element point to null. Thus, one value is completely removed from the cache dataset. Then we may sequentially delete the current last node from the access list until the current size if lower than the desired size. All of the deleted values will be stored in a temporary hashmap and then be promoted to the NetCDF dataset.

4.6 Integration with other Components of the GIPSY

GIPSY is a large system, which consists of several subsystems. Each part is responsible for a different task. So, it is also very important to integrate the different parts developed by different people. The IVW is a part of the GEE in the GIPSY. We can see the architecture of the whole GIPSY system in Figure 2-1 (page 7). In this section, we illustrate the specific relationship among the other parts of the GEE and with the RIPE.

4.6.1 Integration within the GEE

The IVW is employed by the GEE. The GEE accepts its computing resources, i.e. AST, data dictionary, sequential threads and communication procedures from the GIPC. To integrate the IVW with the GEE is just to instantiate the façade class *IVWControl* in the *main()* method of the *GEE* class. Also, it initializes a warehouse for a specific Lucid program, then sets up the warehouse before execution process and stop the warehouse at the end. We present a segment of code as the example:

```
   public static void main(String argv[])
   {
      ......
      // Interpretation Stage and Warehousing
      Vector
dictSemantic=oSemanticAnalyzer.getDictionary();

      IVWInterface vh = new IVWControl();
      XLucidInterpreter xi = new XLucidInterpreter(vh);

      vh.initIVW(dictSemantic, strFilename);
      vh.setupIVW(strFilename);
      ......

      vh.stopIVW();
      ......
   }
```

4.6.2 Integration with the RIPE

The 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, the user can interact with the RIPE at the runtime. The IVW provides the interface to the RIPE, which includes some methods. The RIPE can invoke these methods to operate the IVW at the runtime. Those include the following functions:

- *setCacheSize()* can configure the size of the cache dataset, which let user define or change the size of cache dataset as needed.
- *setGCAlgorithm()* can set the garbage collection algorithm, which let user select GC algorithm even at runtime. When this method is invoked, the given algorithm will be adopted in the next garbage collecting.
- *viewCacheSet()* can return all the entries in the cache dataset, which let user dynamically view the data at the runtime.
- *writeToFile()* can write all the values in the warehouse to NetCDF file, which give the user choice to store data to NetCDF file when the IVW is stopped.
- *getNetCDFFile()* can view the items and data in the NetCDF file, which let user inspect what data are stored in the NetCDF file. Now we use **ncdump** to output the data in the NetCDF file as two style: CDL and NcML.

There are many tools that can be used for manipulating or displaying NetCDF data. Here, we presents available utilities in the current NetCDF distribution from Unidata, **ncdump**, for converting NetCDF files to an ASCII human-readable form, and **ncgen** for converting from the ASCII human-readable form back to a binary NetCDF file.

The **ncdump** tool generates the CDL text representation of a NetCDF dataset on standard output, optionally excluding some or all of the variable data in the output. See the CDL example in Appendix A. The output from **ncdump** is intended to be acceptable as input to **ncgen**. Thus **ncdump** and **ncgen** can be used as inverses to transform data representation between binary and text representations. **ncdump** may also be used as a simple browser for NetCDF datasets, to display the dimension names and lengths; variable names, types, and shapes; attribute names and values; and optionally, the values of data for all variables or selected variables in a NetCDF dataset.

4.7 Summary

We develop a value warehouse to store the values computed by the GEE. It has multi-level architecture and employs the garbage collector. The warehouse works as a data cache of the GEE. Consequently, the GEE reduces the overhead of recomputing values. In the next chapter, we demonstrate testing to make sure the design and implementation.

Chapter 5 : Experimental Results

This chapter reports the testing results. It reviews the two approaches adopted in GEE implementation, analyze their results and addresses two important issues: performance and adaptability.

5.1 Testing Approaches and Results

We used a bar temperature application written in Lucid (see [4]) to test the implemented prototype on a single machine. At first, we tested this program to compare the performance between with warehouse and no warehouse in GEE interpreter solution, which will prove the importance of the warehouse. Then, we tested different implementations of the warehouse in GEE distributed solution. Testing results are reported in this section.

5.1.1 Tested Application

The experimented program finds an approximate temperature distribution for a bar that is heated at one end using relaxation. Its two-dimensional version is shown in Figure 5-1.

```
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 5-1: Temperature distribution for a bar (two dimensions)

The application can also be easily made bigger to a square of three dimensions, shown in Figure 5-2.

```
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 5-2 : Temperature Distribution for a Bar (three dimensions)

5.1.2 Testing Configurations

To run the application, we used one PC that has the following configuration:

- A single processor: AMD Athlon XP 2600+
- Memory: 512 MB RAM
- Operation System: Microsoft Windows XP Professional

We have respectively tested the program in different architectures mentioned in chapter 3.3 to reach various experimental results. In addition, we tested different cases in particular parameters of unlike cache size and garbage collection algorithms. In each case, we try to compute the temps in various contexts, for example, temp@ (T=1, X=1), temp@ (T=3, X=3), etc.

Test A:

We test two cases: the first is that our system executes the tested program without warehouse, and the second one is that GEE employs our warehouse.

Test B:

In the following test cases, we will approach various experimental results in different value warehouse implementations.

Case 1:

We use the NetCDF file as the warehouse and access NetCDF file directly when the values are put into the warehouse or get from the warehouse. This test case is to simulate the values located in the file or network.

Case 2:

The warehouse is only constructed by NetCDF dataset. There is no cache dataset. Other components of the system will access the values in the NetCDF dataset directly, which is our solution I - b (chapter 3.3).

Case 3:

The warehouse is built in two levels: cache dataset and NetCDF dataset. The values input by other components of the system will be stored in the cache dataset first, and we will promote some of them into the NetCDF dataset when the cache dataset is full. Here we initialize the cache size as default size: 512K Bytes and default GC algorithm: Remove Least Access Values (RLA). It is the solution II (chapter 3.3)

Case 4:

The warehouse has the same architecture as the case 3. We initialize the cache size as small size: 8k Bytes and set the garbage collection algorithm by Remove Least Access Values (RLA).

Case 5:

The warehouse has the same architecture as the case 3. We initialize the cache size as small size: 8k Bytes and set the garbage collection algorithm by Remove Oldest Age Values (ROA).

Case 6:

The warehouse has the same condition as the case 4. The cache size is initialized as 8k Bytes and the garbage collection algorithm is RLA. But this time, the values removed from the cache dataset will be written to the NetCDF file, which is solution III (chapter 3.3).

Case 7:

The warehouse has the same condition as the case 5. The cache size is initialized as 8k Bytes and the garbage collection algorithm is ROA. But this time, the values removed from the cache dataset will be written to the NetCDF file, which is also solution III (chapter 3.3).

5.1.3 Results

In this section, we present the testing results of all the cases.

First, we got the results of the **Test A**.

Comparing with warehouse and no warehouse

Figure 5-3 shows the testing results of with warehouse and no warehouse in the test A. The numbers on X-axis represent the integer values of T, X in the program; and the Y axis represents the time spent on computing the value under the corresponding context, which is with a unit of millisecond.

From the results, we can definitely get the fact that the performance is obviously higher with warehouse than no warehouse, which indicates the importance of the warehouse in our system.

After that, we will show the results of all the cases in the **Test B**.

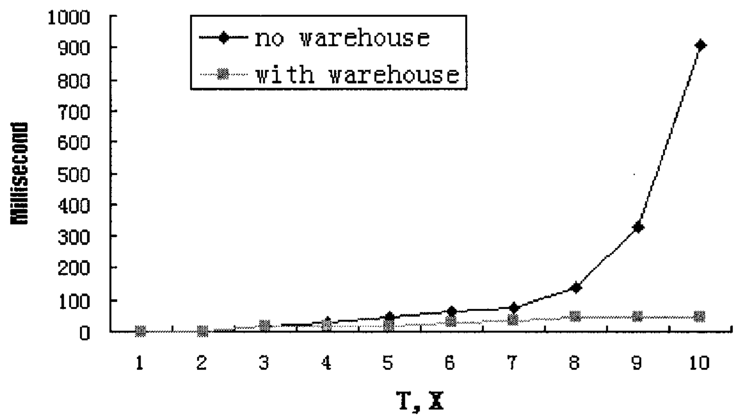


Figure 5-3: Testing results with warehouse and no warehouse

Table 5-1 shows the testing results of case 1, 2 and case 3. In the table, the first column gives the values of T, X in every test. The second and the third column show the values of access hits and misses to the warehouse.

Case 1, 2 and 3			Case 1	Case 2	Case 3
T, X	Hits	Misses	Timing (millisecond)	Timing (millisecond)	Timing (millisecond)
1	7	6	204	63	55
2	60	93	724	344	328
3	205	384	2046	1125	1093
4	382	1021	3516	1812	1750
5	613	2405	6016	2922	2828
6	1277	5144	11141	5047	4922
7	1261	6649	13281	5938	5766
8	2032	12523	22609	9719	9375
9	2239	15222	26875	11515	11125
10	3757	31824	51000	20750	20531
11	5989	49331	77625	31109	30921
12	7818	77661	118375	47172	46609
13	7679	98600	147125	57797	57593
14	8882	114388	170906	65937	67578
15	10074	130611	195375	74578	78234
16	8945	124922	187875	73000	74188
17	12049	184065	271969	105625	107344
18	18825	283982	420843	161360	169593
19	21083	372074	547797	212500	216078

Table 5-1: Results of case1, 2 and 3

Comparing different IVW architectures

Figure 5-4 shows the testing results of different architectures, case 1, 2 and 3. The numbers on X-axis represent the integer values of T, X in the program; and the Y axis represents the time spent on computing the value under the corresponding context, which is with a unit of millisecond.

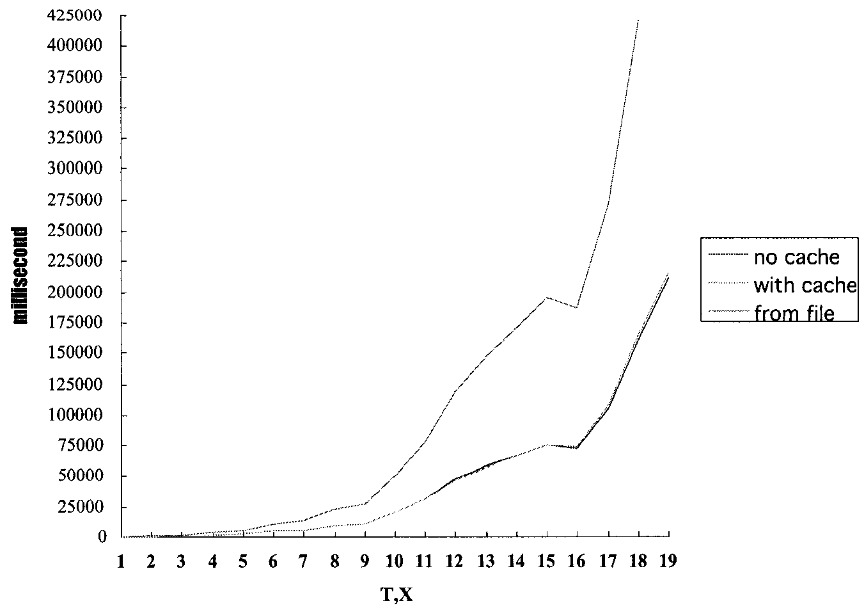


Figure 5-4: Testing results with different architectures

The "no cache" curve presents the testing result with only NetCDF dataset without cache dataset, case 1, and the "with cache" curve shows the result in case 2, which has a default maximum size cache (512 Kbytes). In our testing cases, the cache has enough size to contain the values from other components so that there is no garbage collecting being triggered.

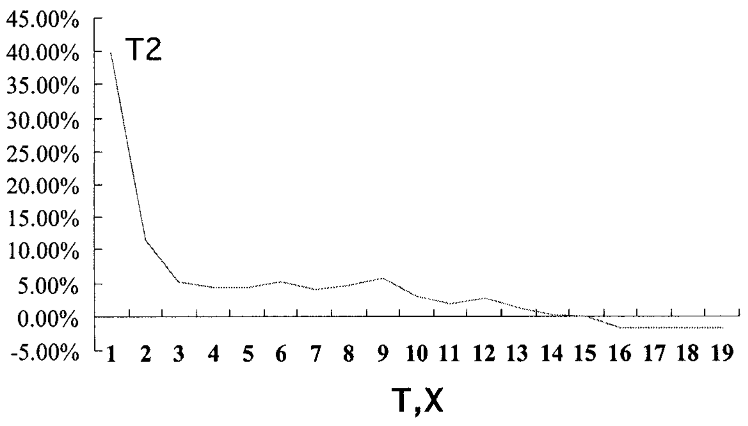


Figure 5-5: Relative performance with different architectures

The same results used in Figure 5-4 are expressed in Figure 5-5 but in a different way. From this figure we may inspect the comparison more clearly. The integer numbers on X-axis are the T, X ranges and the Y-axis represents the "relative performance", measured as *(T1-T2)/T1/100,* where *T2* is the time taken by with a cache dataset and *T1* is the time taken by NetCDF dataset without cache dataset. The curve is tagged with T2 at the beginning place of a curve, which shows the degree of the difference between case 3 and case 2.

Comparing different garbage collection algorithms

Then we test the same application of case 4 and case 5 with small maximum size cache (8 Kbytes). Also, we implement two garbage collection algorithms. One is Removing Least Access values (RLA), and another one is Removing Oldest Age values (ROA). We define the condition to trigger the garbage collection that current size is more than 95 per cent of the maximum cache size, and then clean out the cache until it has 20 per cent free. Here, we have tested the program with T, X value at 1 to 12. Since the results of all cases are same when T, X value at 1 to 7, we did not add them to the Figure 5-6.

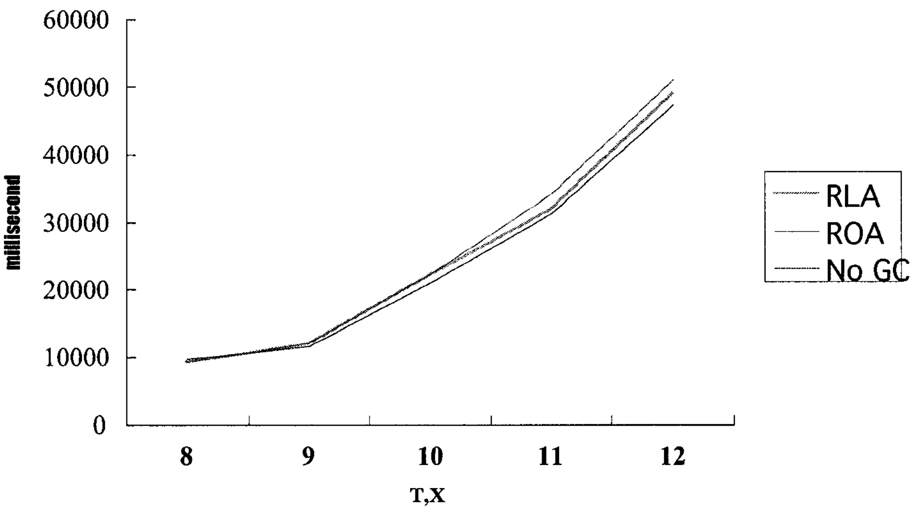


Figure 5-6: Testing results with different garbage collection algorithms

Comparing promotion and storing in file on RLA

Now we test the same application of case 6 with small maximum size cache (8 Kbytes) same as case 4. But this time, we write the values that are removed from cache dataset to the netCDF file and read them from the netCDF file when they are needed. Also, we use the RLA garbage collection algorithm. Since the results of two cases are same when T, X value at 1 to 7, we just compare the results between case 4 and case 6 when T, X at 8 to 12, which is shown in the Figure 5-7.

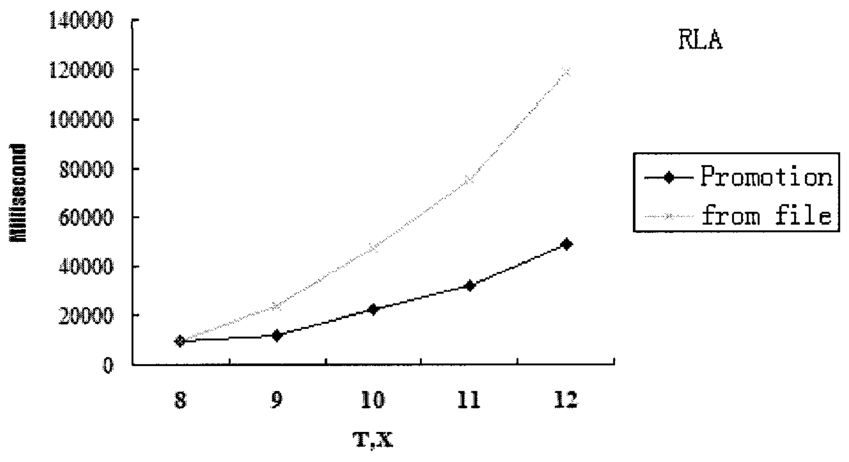


Figure 5-7: Testing results RLA between promotion and file

Comparing promotion and storing in file on ROA

At last we test the same application of case 7 with small maximum size cache (8 Kbytes) same as case 5. But this time, we write the values that are removed from cache dataset to the netCDF file and read them from the netCDF file when they are needed. Also, we use the ROA garbage collection algorithm. Since the results of two cases are same when T, X value at 1 to 7, we just compare the results between case 4 and case 6 when T, X at 8 to 12, which is shown in the Figure 5-8.

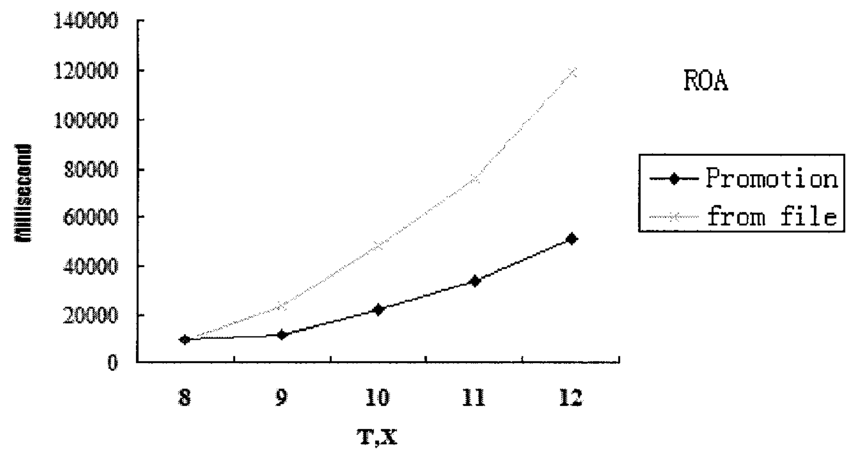


Figure 5-8: Testing results ROA between promotion and file

Comparing each list varies in cache dataset

The testing results are separately shown when the value of T, X is 10 in the following figures (Figure 5-9, and 5-10) because their element number varies seriously under different garbage collection algorithms. The numbers on X-axis represent the running time with a unit of second, and Y-axis represents the element number of the list in cache dataset.

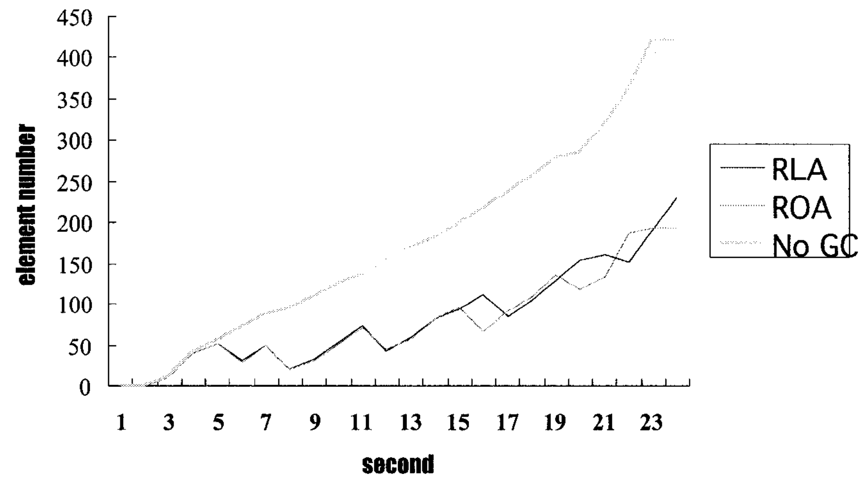


Figure 5-9: *AccessList* and *AgeList* variance

From above figure, we found that the element number of *AccessList* and *AgeList* grows up in the runtime when there is no garbage collecting happening. While other two lines have some curves turning down, which means that a garbage collecting occurs at those points.

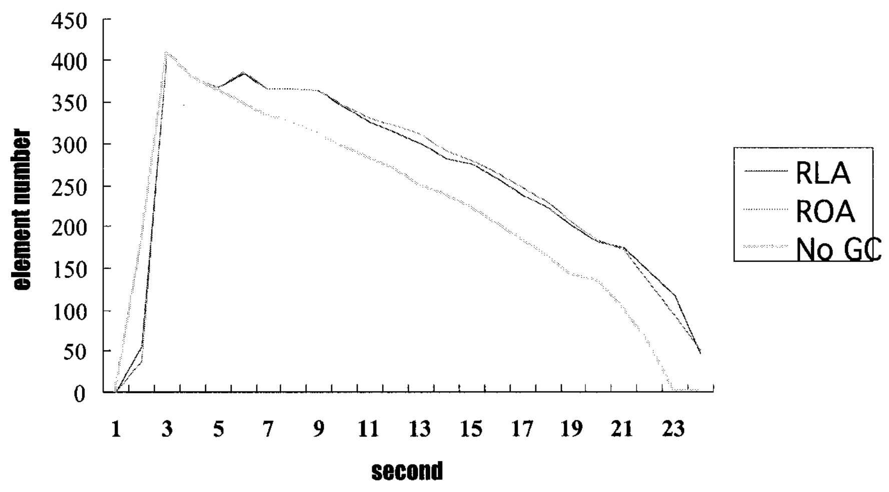


Figure 5-10: *EmptyList* variance

In Figure 5-10, we got the information of *Emptylist* at the runtime. It shows that the element number of the *Emptylist* grows up fast at the beginning of the execution, then decreases gradually.

Comparing hits of the cache in different garbage collection algorithms

The following two tables show the hits probability of the cache under different garbage collection algorithms. They show that the hits probability is same at the value of T, X from 1 to 8. Actually, there are 3, 6, 58 and 163 times garbage collectings happen at T, X value of 9, 10, 11 and 12 when we initialize cache as small maximum size. The result is that when some garbage collecting happen, the hits probability of the cache decreases in this case.

	Default max cache size	Small max cache size	
T, X	No GC	RLA	ROA
1	7	7	7
2	60	60	60

3	205	205	205
4	382	382	382
5	613	613	613
6	1277	1277	1277
7	1261	1261	1261
8	2032	2032	2032
9	2239	2220	2191
10	3757	3302	3182
11	5989	3284	2945
12	7818	2737	2575

Table 5-2: Cache hits in different GC algorithms

	Default max cache size	Small max cache size	
T, X	No GC	RLA	ROA
1	6	6	6
2	93	93	93
3	384	384	384
4	1021	1021	1021
5	2405	2405	2405
6	5144	5144	5144
7	6649	6649	6649
8	12523	12523	12523
9	15222	15241	15270
10	31824	32279	32399
11	49331	52036	52375
12	77661	82742	82904

Table 5-3: Cache misses in different GC algorithms

5.2 Interpretation of Results

The Value Warehouse has been implemented with two different approaches: warehouse constructed by NetCDF data structure with cache dataset or without cache dataset. Also, different garbage collection algorithms are adopted on the dataset. In this section, we will give the interpretation of the above results.

With a closer look to the two approaches in Figure 5-4, we found that those two curves show the similar values in each test of computing the program. It seems that the performance was not obviously optimized when we implement the warehouse with a cache dataset. We analyze the reason that both the cache dataset and the NetCDF dataset are located in the memory at the runtime. While the speed of access to memory decides they have no big diversity though they store data as different format. Since the GEE describes a variable in a context as IC string, then we can easily construct the pair data with the IC string and its value to store in the cache dataset. While for NetCDF dataset, we have to interpret the IC string and translate it into an acceptable format to NetCDF dataset. Moreover, the values in the warehouse will be written to NetCDF file on the disk or even network if the user needs. Another condition is that we have to write the values to the file or network when there is limited memory. Then we can slowly retrieve values from these sources. In that case, the cache dataset will present better performance.

Also, in Figure 5-5, we got that with cache solution has better performance than without solution when T,X values are less than 15, while it has inverse results going with T,X values rising. We distinguish this result from the *hashmap* performance. A *hashmap* 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 *hashmap* also results in a low incidence of collision. Under light to medium load the hash table performs very well.

In our case, we define a fixed size of cache, which means it has a fixed size *hashmap,* then, the more elements it has, the more probability collision occurs. While the NetCDF dataset stores data in a kind of multidimensional arrays, then it has no collision problem. Hence, the performance of the solution with cache is lower than the one without cache when the values of T, X grow up in the Figure 5-5.

In the Figure 5-6, without garbage collection performs better than other two cases. That means when the garbage collecting happens, the computed time is longer than the case no garbage collecting happening. This outcome seems in conflict with the design of warehouse with garbage collector we did before, which indicates the garbage collector will enhance the performance of the system. We analyze that the reason is as the following. The lucid program used in testing has the nature of fine granularity and high data-dependency. Fine granularity means that each demand needs a little computed time. Moreover, the values are rarely accessed after being stored in the warehouse. So the garbage collector does not operate well. From the hits and misses tables, we found that the times of the missed access are further larger than those of the access hits. It results in high data-dependency of the tested application, in which values of variables px, x and nx all depend on the values of the variable temp. Thus, before a demand being computed, the IDP will ask warehouse many times for the values it depends till reaching the initialized value. Therefore, for this application and with the current implementation, we received the above results.

Through the Figure 5-7 and 5-8, we can obviously found that when the values are promoted to the high level from the cache dataset, it performs better than they are written to the netCDF file because the latter there is some overhead of access to the netCDF file on the disk. But they will adapt different conditions, for example, the netCDF file is located at remote side, then we can make cache dataset local side.

Also, we may explain the Figure 5-9 and 5-10 as the nature of the Lucid program for testing. Because of the "followed by" operator, an initial demand for a big value of T and X (e.g. 19) will generate demands for 18,17,16…3,2,1,0, and up to then, all demands are empty, and they start to have their values computed one after the other in reverse order. This will happen in any program largely defined by using the "followed by" operator, which creates such a "linear chain of dependences" because of its inherent recursion.

5.3 Limitations

We did not test various lucid programs on our system. In this case, some results are encountered for the limitation of the nature of the testing application. For example, the external functions are accepted by GIPSY, which may have high "weight" of computation and frequent access. Also, some applications will have low-dependency among variables. In these cases, we may approach other results.

5.4 Summary

In this chapter, we have provided some test cases, which can ensure the quality of our project. Also, we explained the experimental results and presented the limitations of our implementations.

Chapter 6 : Conclusion

In this chapter, we will review the thesis and draw conclusions. At first, we introduced the Intensional language and the GIPSY system, which consists of three subsystems: GIPC, RIPE and GEE. The IVW is employed by the GEE, and it works as a data cache to store the interim values generated by the GEE. Then we explored the NetCDF of Unidata and general garbage collection technique and designed the IVW applying those techniques. Next, we presented the design and implementation of the IVW according to some reasonable rational. Moreover, we did various test on different Lucid programs to prove that our implementation is effective. Also, we discussed the experimental results to interpret the feature and limitation of our implementation.

Through the IVW, we demonstrate how to build a value warehouse for GIPSY. The IVW provides portable data storage and efficient data access and supports various data type of the values. Furthermore, the IVW adopts garbage collection technique of various algorithms and has arbitrary given size of cache dataset so that it serves as a flexible tool with convenient performance tuning. Consequently, the IVW plays an important role in the GIPSY system.

Chapter 7 :   Future Work

Up to now, we have built the IVW for the GEE in the GIPSY. However, in a complex system, this is only a part of achievement. There is still some further work to be done. This section describes future work that can be performed.

First, more efficient garbage collecting algorithms will be added to the system to deal with various programs. The next version of GIPSY will include calculation function, which would be more useful for a large real scientific computation. Because of the inherent concise nature, it may be difficult to extract computation units and decide the granularity of units. So a practical approach might be to state the computation units explicitly in source code, then the warehouse would have more information to cull the useless values. That will make the system more efficient to perform well on different intensional languages.

Secondly, in current version, there is only one central warehouse that contains all the calculated values. It is relatively easy to manage, but likely to become a performance bottleneck in a distributed system because the warehouse will be accessed by other nodes in the system before a demand is computed. That will bring much communication overhead. Then a possible solution is considered that each node in the distributed system has a warehouse or a cache, which contains part of content of the central warehouse.

Finally, a warehouse monitor will be added to the graphic interface of GIPSY. When the system is performing, the user might care the real-time values in the warehouse and even modify its configuration. Also, after the computation, the user may review the values in the file system we dumped from the warehouse. Then, another tool like monitor is needed.

References

[1] Joey Paquet. *Intensional Scientific Programming*. Ph.D. Thesis, Departement d'Informatique, Universite Laval, Quebec, Canada, 1999.

[2] 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.

[3] Ai Hua Wu, Joey Paquet, Peter Grogono, *The Design of a Framework for the General Intensional Programming Compiler in the GIPSY,* 15th IASTED International Conference PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS (PDCS 2003) November 3-5, 2003, Marina del Rey, CA, USA

[4] Bo Lu. *Framework for the General Eduction Engine (GEE) in the GIPSY.* PhD, Thesis, Computer Science Department, Concordia University, Quebec, Canada, March 2004.

[5] Ai Hua Wu. *Semantic Analysis and SIPL AST Translator Generation in the GIPSY* M.Sc, Thesis, Computer Science Department, Concordia University, Quebec, Canada, December 2002.

[6] Russ Rew, Glenn Davis, Steve Emmerson, and Harvey Davies, *NetCDF User's Guide, version 3,* Unidata Program Center 1997

[7] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* 1996

[8] Erich Gamma. *Design Patterns: elements of reusable object-oriented software*, 1995

[9] Sun Java API Specifications. http://java.sun.com/j2se/1.4.2/docs/api/index.html

Appendix A: [NetCDF example]

A.1 [example of CDL notation for a NetCDF dataset]

```
NetCDF example {  // example of CDL notation for a NetCDF dataset

dimensions:        // dimension names and lengths are declared first
    lat = 3, lon = 4, time = unlimited;

variables:         // variable types, names, shapes, attributes
    float  T(time,lat,lon);
           :long_name   = "surface temperature";
           :units       = "degC";
    float  rh(time,lat,lon);
           :long_name   = "relative humidity";
           :units       = "percent";
    int    lat(lat);
           :units      = "degrees_north";
        Int        lon(lon);
           :units      = "degrees_east";
    short  time(time);
           :units      = "hours";
    // global attributes
           :source = "Fictional Model Output";

data:              // optional data assignments
    lat    = 41.0, 40.0, 39.0;
    lon    = -109.0, -107.0, -105.0, -103.0;
    time   = 6, 18;
    rh     =.5,.2,.4,.2,.3,.2,.4,.5,.6,.7,
            .1,.3,.1,.1,.1,.1,.5,.7,.8,.8,
            .1,.2,.2,.2,.2,.5,.7,.8,.9,.9,
            .1,.2,.3,.3,.3,.3,.7,.8,.9,.9,
             0,.1,.2,.4,.4,.4,.4,.7,.9,.9;
}
```

A.2 [example of NcML notation for a NetCDF dataset]

```
<?xml version="1.0" encoding="UTF-8"?>
<nc:NetCDF xmlns:nc="http://www.ucar.edu/schemas/NetCDF"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
 xsi:schemaLocation="http://www.ucar.edu/schemas/NetCDF
http://www.unidata.ucar.edu/schemas/NetCDFDataset2.xsd"
 uri="file:///E:/dev/NCdataset/example.nc">

 <nc:dimension name="time" length="2" isUnlimited="true"/>
 <nc:dimension name="lat" length="3"/>
 <nc:dimension name="lon" length="4"/>

 <nc:variable name="rh" type="int" shape="time lat lon">
  <nc:attribute name="long_name" type="string" value="relative humidity"/>
  <nc:attribute name="units" type="string" value="percent"/>
 </nc:variable>

 <nc:variable name="T" type="double" shape="time lat lon">
  <nc:attribute name="long_name" type="string" value="surface temperature"/>
  <nc:attribute name="units" type="string" value="degC"/>
 </nc:variable>

 <nc:variable name="lat" type="float" shape="lat">
  <nc:values separator="">41.0 40.0 39.0</nc:values>
  <nc:attribute name="units" type="string" value="degrees_north"/>
 </nc:variable>

 <nc:variable name="lon" type="float" shape="lon">
  <nc:values separator="">-109.0 -107.0 -105.0 -103.0</nc:values>
  <nc:attribute name="units" type="string" value="degrees_east"/>
 </nc:variable>

 <nc:variable name="time" type="int" shape="time">
  <nc:values separator="">6 18</nc:values>
  <nc:attribute name="units" type="string" value="hours"/>
 </nc:variable>

 <nc:attribute name="source" type="string" value="Fictional Model Output"/>
</nc:NetCDF>
```