Object-Oriented Database Support for Scientific Data Management: A System for Experimentation

Hitomi Ohkawa
B.S. in Physics, University of Tokyo, 1981
M.S. in Physics, Massachusetts Institute of Technology, 1983

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

April 1993

The dissertation "Object-Oriented Database Support for Scientific Data Management: A System for Experimentation" by Hitomi Ohkawa has been examined and approved by the following Examination Committee:

David Maier
Professor
Thesis Research Adviser

Jonathan Walpole
Assistant Professor

Todd Leen
Assistant Professor

T. Lougenia Anderson
Sequent

Dedication

This dissertation is dedicated to my husband, A. Remy Malan.

Acknowledgements

I would like to thank my advisor, Prof. David Maier, for his rigorous review on details of the dissertation.

I would also like to thank Dr. Stephen Bryant at National Center for Biotechnolgy Information (NCBI) at National Institutes of Health and Dr. Daniel Heitjan at Center for Biostatistics and Epidemiology, Penn State University College of Medicine, for providing scientific data and applications to examine in the experiments. In particular, I was able to discover NewS, one of the software components used in the dissertation research, through Dr. Bryant's application, Protein Knowledge Base (PKB). Dr. Bryant also provided much support and encouragement throughout the dissertation research that often required me to learn about the domain of computational biophysics.

Contents

Abstract

Object-Oriented Database Support for Scientific Data Management: A System for Experimentation

Hitomi Ohkawa, Ph.D.
Oregon Graduate Institute of Science and Technology

Supervising Professor: David Maier

Scientific data management has recently become a critical research issue due to computerization and automation of scientific research procedures and the resulting explosion of electronic data. For proper management of scientific data, adequate data types must be provided for representing the semantics of scientific data directly. Issues such as large data volume and data evolution are common among scientific applications, so traditional database support, such as storage management, are also beneficial to them. Current approaches such as standardized data formats and adoption of traditional databases do not accommodate the data support requirements of existing scientific applications well. Object-oriented databases, on the other hand, seem to hold promise, by combining flexible data models with traditional database support.

In this dissertation, we constructed an experimental platform for exploring the design space for a scientific data management system. With this experimental approach, we could dynamically examine possible architectures for data support against actual instances of scientific data and typical operations executed on them. As many of the dynamic features of existing scientific applications, such as data access patterns, are yet to be discovered, the platform also provides an opportunity to explore such features.

Based on the observed potential of object-oriented databases for scientific data management, the `GemStone` object-oriented database management system was chosen for the baseline data management architecture of the platform. Among scientific applications where better data support is desired, a scientific persistent language and data analysis environment called `NewS` and applications using it were selected as a target for the study. Connecting

NewS with GemStone provided a cost-effective experimental platform where we could investigate a variety of scientific applications with single implementation. We incorporated GemStone into the NewS environment in such a manner that we could use existing NewS applications without modifications for experiments. We describe the design and implementation of our platform, **GemStone-based NewS**, and experiments performed on the platform. At the end, we describe primary contributions of the work, and assess whether or not our approach was a productive initial step toward improved scientific data management.

Chapter 1

Introduction

Scientific data management has become an important issue since computers started playing a major role as a research tool in a variety of scientific domains. Advanced lab technologies and computer simulation are producing massive datasets, and it is becoming increasingly difficult to manage them manually. Many scientific applications currently rely on a file system for data management, though there is ample evidence that most of them can benefit from some form of data support beyond the simple capabilities of a file system, such as data types, data evolution, memory management, and query languages. For example, a file system rarely provides a data model that accommodates scientific data types, so data must be mapped to files by each application. Since there are few standards for how the data are represented, the result is a variety of data formats for a given data type, with no uniformity and compatibility among them. A solution promoted by various scientific organizations (e.g., NASA, supercomputing centers) is standardized data formats. However, they still depend on the underlying file system with its basic data management capabilities, and do not provide efficient and safe access to the data. As most current approaches to scientific data management have major drawbacks, there is a need for a data management system capable of better supporting scientific data.

This dissertation describes our experimental investigation of adequate data support for

scientific applications. An experimental approach allowed us to dynamically investigate possible data support architectures against actual instances of scientific data and functions executed on them. We constructed a system called **GemStone-based NewS** as an experimental platform, and through the platform focused our attention on support provided by the **GemStone** object-oriented database for applications implemented in a scientific persistent language and data analysis environment called **NewS**.

We chose an object-oriented database for a baseline data support architecture because it combines a flexible data model with traditional database support. With GemStone, we could support scientific data types directly using object-oriented concepts, and even have a variety of alternatives in doing so, without sacrificing traditional database features such as storage management and concurrency control.

By connecting NewS with GemStone, we could examine a variety of existing scientific applications in different domains with single implementation, since NewS is widely used as a statistical and graphical data analysis environment, e.g., in mathematics, physics, biology, and economics. Had we chosen to support individual applications written in conventional languages on GemStone, we would have had to implement a connection to a database for each individually.

Current NewS lacks robust data support as it relies on the file system for storing persistent data. Traditional databases, such as relational systems, do not support NewS well, as the hierarchical data structure and functional computation model of NewS are not a good match for their data models. An object-oriented database, on the other hand, seems a better alternative to improve data support of NewS applications as its flexible data model

can capture NewS data and functions directly. NewS is a persistent language: NewS applications can access typed, persistent data items by name. A switch in the underlying data store from files to GemStone can be "hidden", so that existing NewS applications can still run on the platform without modifications. In contrast, an application in a non-persistent language such as C or FORTRAN would require extensive modification to run against a database. The mapping of data types to files is explicit in them and all the file reads and writes must be replaced by calls to GemStone. Admittedly, focusing on NewS applications limited the range of scientific applications we could examine, but NewS seemed a good compromise between generality of targeted applications and ease of experimentation.

In our investigation, we designed and implemented GemStone-based NewS, on which we subsequently performed experiments with various NewS data and applications. We then assessed major contribution of our study and whether or not our approach was a good initial step toward improved scientific data management. Design of scientific data management systems is a relatively new research topic where most of the design space is yet to be investigated. With limited resources on hand, we still wanted to cover as much of the design space as possible, to identify parts worth detailed investigation later. We examined whether the choice of GemStone and NewS provided good coverage of the design space, and assessed GemStone-based NewS in terms of such criteria as cost-effectiveness in construction, a potential for scientific data management, flexibility to make its architecture customizable to each application, generality of targets, performance level, and ease of experimentation. We also looked at the kinds of issues we could examine on GemStone-based NewS among potentially interesting design dimensions.

The rest of the dissertation is organized as follows. Chapter 2 describes various data support requirements of existing scientific applications. We discuss direct support for scientific data types as well as traditional database features such as storage management and direct data query that could potentially benefit scientific applications.

Chapter 3 describes the potential of an object-oriented database for scientific applications by comparing it to various current approaches to scientific data management, e.g., standardized data formats and traditional databases. We list advantages and disadvantages in each case with respect to the requirements described in Chapter 2.

Chapter 4 discusses our experimental approach, and proposes construction of a cost-effective platform that delivers maximal results with minimal effort. We describe criteria for the experimental platform under which the platform will be evaluated later, namely, cost-effectiveness in construction, productivity and flexibility of the architecture, generality of targeted applications, performance level, and ease of experimentation.

Chapter 5 discusses design dimensions we would be interested in investigating on an experimental platform. We refer to the data support requirements identified in Chapter 2, since our objective is to investigate physical design supporting such high-level requirements.

Chapter 6 introduces our experimental platform, GemStone-based NewS. We focus our discussion on the two main components, i.e., GemStone and NewS, and justification of their choice.

Chapter 7 describes various design and implementation issues considered in constructing GemStone-based NewS. We also discuss flexibility of the architecture as a design priority and how it affects easy execution of prospective experiments.

Chapter 8 examines flexibility of the platform's architecture by investigating possible design alternatives over its baseline architecture. We discuss how such alternatives can be examined on the platform experimentally. In some cases, we conduct simple experiments to illustrate the point being made.

Chapter 9 reports on more extensive experiments with two existing NewS applications, namely, National Health And Nutrition Examination Survey (NHANES) Data and Protein Knowledge Base (PKB). We describe those two applications and explain why we chose them. We then present the experimental results and analyze them with respect to their specific requirements and execution environment.

Chapter 10 analyzes ease of experimentation with GemStone-based NewS from our experience experimenting with the platform. Finally, Chapter 11 evaluates the platform and our approach as a whole, and describes what we consider major contributions of the study. We also discusses possible future work.

Chapter 2

Data Support Requirements For Scientific Applications

This chapter investigates data support requirements for scientific applications. We first examine data types required by scientific applications. Our analysis is not confined to a particular domain, though many examples are drawn from molecular biology and chemistry. Besides data types, we also discuss how traditional database features such as storage management can benefit scientific applications.

2.1 Scientific Data Types

Scientists view a physical system under study through a particular model. If data types that directly represent scientific models, which we will call scientific data types, are provided, scientists can deal with the relevant models directly in applications and do not have to fit the models into data structures provided by a particular language.

One of the structural properties commonly seen in scientific models is hierarchical decomposition. Hence, nested structures need to be supported by scientific data types. For example, a macromolecular structure is hierarchically represented with different levels of detail. DNA structures are typically translated into sequences of various units. Such units

include nucleotides, cleavage sites, ligand-binding sites, or "characteristic subwords" of nucleotides determined by linguistic methods, all of which have further substructure [Pon88]. The Delila (DEoxiribonucleic acid LIbrary LAnguage) system defines a schema for nucleic-acid sequences using a hierarchical model as shown in Figure 2.1. For example, `library` contains various `organisms`, each of which in turn includes one or more `chromosomes`. Each `chromosome` points to various pieces of `sequences`.

Proteins are another example of hierarchically organized macromolecules [Cre84]. The primary sequence is a linear chain of amino acids, and it folds into various secondary structures such as helices, sheets, and turns. Secondary structures are in turn grouped into super-secondary structures to form biochemically active sites. The tertiary structure describes the three-dimensional structure of the entire protein, and multiple proteins sometimes bind together to form a quaternary structure. Characteristics called `motifs`, `templates`, or `fingerprints` are associated with different levels in this hierarchy. Motifs can represent a characteristic piece of sequence, sequence patterns associated with a specific structure, or structural characteristics.

Many scientific theories offer "generic" models applicable to a variety of systems. For example, the contact potential model developed for representing protein conformation is specialization of a classical statistical mechanical model characterized by the Boltzmann statistics. Mean field theory is another example of a generic scientific model that has been applied to a variety of physical systems. A generic model provides a framework for representing a targeted system, but specifics are further provided in each case. For example, mean field theory provides a generic formula for a partition function, but a particular density

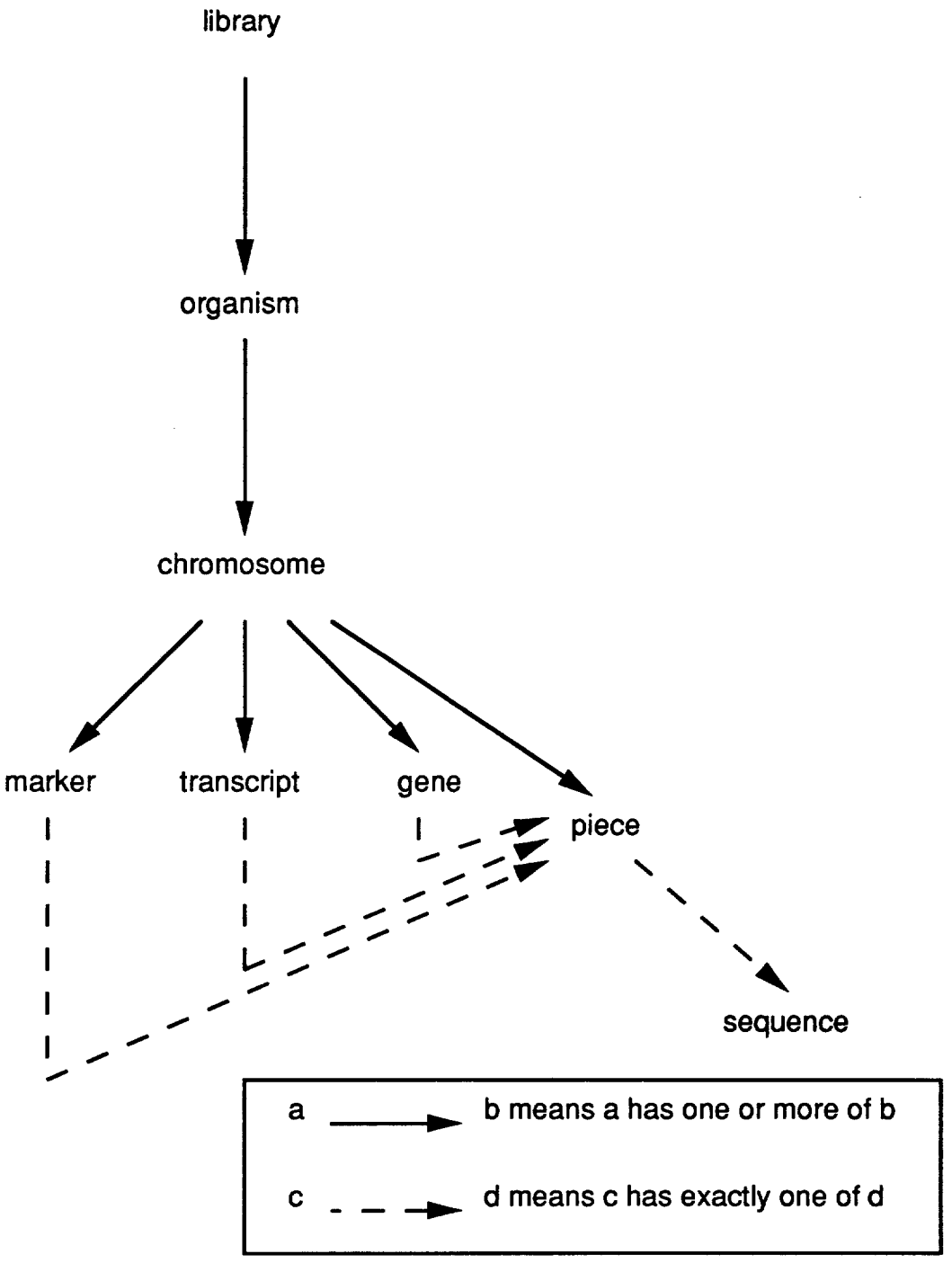


Figure 2.1 Part of Delila schema

distribution and energy function are determined for each system modeled. Hence, providing a generic model as a data type is beneficial as the type can be reused for different systems, as long as it is possible to specialize the model in each case.

As much as the same model may be used for different physical systems, it is also possible that the same physical system is viewed through different models. A solid may be modeled by its boundaries or recursive combinations of shapes. Proteins are sequences of amino acids to many mathematical biologists, but physicists examine its structure and activity at the atomic level. Among protein structural models, some represent a configuration of a main chain with positions of α carbons and torsion angles. Others, such as the contact potential model mentioned above, attempt to characterize the structure through potential energy that results from a specific conformation. In chemistry, a variety of models are available for a molecule. For example, in determining the structures of molecules by quantum-mechanical computation, various approximations are used for efficiency, yielding different models. Molecular mechanics (MM) ignores electrons, but the strength of bonds between atoms is adjusted according to experimental data. The semi-empirical molecular orbital (MO) model only considers outermost electrons in its computation. The ab initio model, on the other hand, takes into account all components of a molecule. The techniques above assume one molecule at $0K$ in vacuum. Molecular Dynamics (MD), on the other hand, assumes temperatures higher than $0K$, and considers interaction with other molecules in the neighborhood. Hence, each data type is required to store different kinds of information depending of how a corresponding model characterizes the targeted system.

The situation would be simple if each scientific application required a single model for

the targeted system, but in some cases, different models must be accommodated by the same application. For example, one of the main objectives in protein research is to deduce correlation between amino acid sequences and structural motifs, therefore both sequence and structural models must be provided for the same protein. Rarely do computational chemists use only one model, and conversion from one model to another is often required. If scientific models are represented as data types, then there must be a way to combine multiple types in a coherent manner to accommodate different models for the same system.

As mentioned at the beginning, direct representation of a scientific model as a data type serves as a layer between applications and physical data formats. For example, with the "Molecular Mechanics" type, scientists handle the MM model of a molecule directly in the applications and they do not have to deal with how the model is physically realized by specific data structures. Such a scientific data type also provides an opportunity to standardize representation of the model so that multiple applications can share data easily. For example, the Bravais lattice model that represents periodicity in the crystal structure is widely used in solid state analysis and X-ray diffraction analysis. Hence, development of a uniform data type for the Bravais model would benefit many applications in the domain and facilitate data exchange between them [Pat90].

Representing scientific models as data types, it would be beneficial to include in the types not only the actual data on the targeted physical system but also various meta-information that supplement the data, e.g., units for the values and relevant publications, to ensure accurate interpretation and better understanding of the data.

With direct representation of scientific models, it also becomes possible to define arbitrary operations on the models directly. For instance, the essence of many scientific data models is represented by a set of formulas, e.g., a linear formula that relates predictors and the response in the regression model. The application of such models to a specific system often means solving the associated formulas for that system. Hence, it would be useful if the corresponding data type provided operations to solve those formulas when requested. Those model-specific operations no longer have to be translated into equivalent functions on specific data structures, and in fact can be part of a scientific data type. Such extension of the type benefits many applications by providing a uniform, reusable definition of proper manipulation of the model.

As in abstract data types, model-specific operations in a scientific data type can provide applications with an abstract interface for the model. Applications use a model only through its abstract interface, and changes in physical representation of the model do not affect the applications as long as that interface remains the same. In the NIH Genome Database, gene sequences are represented as such abstract data types [And89]. Their physical format changes frequently due to development of new techniques that generate data in new formats. However, since applications access gene sequences only through their abstract interface, it is not necessary to change applications as long as that interface is unchanged.

Both type evolution and type extensibility are critical for scientists to merge new discoveries with existing knowledge. As scientific models evolve, it must be possible to modify the corresponding types. A related issue is type extensibility, namely, a capability to add an arbitrary type as a new model is formed in scientific research.

When defined as a data type, a scientific model is available for multiple applications to share. It would further increase reusability if variations on a type all share the common part of the definition. Such variations occur when a model evolves or a set of related models are to be represented. Object-oriented databases support such reusability through inheritance of existing definition as described in Chapter 3.

2.2 Traditional Database Support

Combined with support for scientific data types, traditional database features such as storage management, an ad-hoc query facility, and data exchangeability greatly benefit scientific applications. Storage management, handling secondary memory and data I/O to/from main memory, can provide support for large data that must be handled often in scientific research. Advances in lab technology have started producing immense amounts of data, and it is becoming increasingly difficult to manage them manually. Computer simulation also generates a large amount of data. It is difficult to manage simulation results, correctly relating results from different runs.

Traditionally information scientists acted as intermediaries for domain scientists and performed database searching for them. However, more and more domain scientists perform data search themselves these days, and there is a need for an ad-hoc query facility that domain scientists can effectively use. Traditional databases such as relational systems offer only a fixed set of data structures and queries, and domain scientists must somehow translate operations on the models to equivalent queries on the provided structures. Such

translation has proved to be a major stumbling block for domain scientists in using traditional databases, since the translation is generally non-trivial and even impossible in some cases. For example, one of the typical queries executed for the protein contact potential model is to get a list of all the contacts of a specific chemical type within a given distance for homologous proteins and compute the potential energy from the obtained list of contacts. Such a query, containing arbitrary computation, is impossible to express in current relational languages. If scientific models are supported as data types in a database, scientists can define arbitrary operations for directly querying the models. If a database supports an abstract data type, data access operations specific to the model can be defined in the type to be shared by scientists. With a model directly stored in a database, the database can also exploit the semantics of the model to optimize query processing.

In reality, scientific data are often distributed among different hardware platforms. In such a heterogeneous environment, data are typically not exchangeable since they are stored in different physical formats on different platforms. Some databases mask such differences by providing automatic conversion of the format when the data are transferred between platforms. A related issue is how to unify the data stored in different kinds of data stores, e.g., files, relational databases and object-oriented databases. As mentioned later, an object-oriented paradigm is much more flexible than the data models provided by files or relational databases, and provides a potential as a framework for such unification.

In scientific research, the same data are often analyzed from different angles, and as a result, they are accessed by multiple applications. Databases can support such a situation by providing a "view" of the data appropriate for each application. For example, Protein

Data Bank (PDB) files contain both a list of amino-acid residues and atomic coordinates for proteins. Depending on whether sequential or structural motifs are being studied, one of the datasets may be hidden as irrelevant while the other is provided to a tool as a view.

Finally, database functions such as recovery, indexing, atomicity and concurrency control make data storage safer and more secure in general. The need for concurrency control depends on the application. Some scientific data are never updated, as is the case with epidemiologic data, making concurrency control of little value. In contrast, concurrency control is a critical requirement in an application like drug design carried out by a team of medical chemists.

2.3 Summary

Through a number of examples, we recognized a need for scientific data types as well as potential benefits of traditional database support for existing scientific applications. The following is a list of the requirements identified in this chapter.

- Scientific data types as direct representation of scientific models. Scientific data types should support the following as potentially useful features for scientific applications.
    1. nested structure
    2. meta-information supplementing raw numbers
    3. operations for such functions as customized data access and model specification, especially as provided in abstract data types
    4. unification of multiple types for the same physical system

5. type evolution/extensibility

6. generalization hierarchy for generic data types

- Database support, which includes the following.

1. storage management and support for large data

2. ad-hoc query facility allowing domain scientists to query data in an intuitive way

3. inter-operability between different hardware platforms and data stores

4. multiple views of data

5. recovery

6. structures for data search such as indexes

7. concurrency control

A database that supports scientific data types can store scientific models directly, making persistent data available to scientists in a format they are accustomed to handling without sacrificing traditional database support. In Chapter 3, we will discuss the potential of object-oriented databases over current approaches to scientific data management with respect to the requirements discussed in this chapter.

Chapter 3

Potential of Object-Oriented Databases

This chapter discusses the potential of object-oriented databases to support the requirements of scientific applications described in Chapter 2. We first examine some approaches currently in use for managing scientific data and discuss advantages and disadvantages in each case. We then compare an object-oriented database to those current approaches and explore its potential advantages.

3.1 Current Approaches to Scientific Data Management

This section examines some approaches currently used for supporting scientific data. In particular, we consider customized data management systems, standardized data formats, persistent languages, and traditional database management systems.

3.1.1 Customized Data Management Systems

Customized data management systems limit attention to specific applications and implement data support customized to the needs of those targets. Though most such systems do succeed in supporting the current needs of the targeted applications, they are often not extensible to accommodate new needs or targets. Most current scientific data management systems are such "proprietary" systems, constructed as a need arises. Since traditional

databases often do not match needs of scientific applications well, needed data management functions are provided on top of a file system. Most approaches are ad-hoc individual efforts, hence there is a lack of standards and compatibility among such systems.

For example, in molecular biology, there are a variety of data banks; nucleic acid sequence data banks such as the GenBank Genetic Sequence Data Bank [B+90] and the EMBL (European Molecular Biology Laboratory) Data Library [EMB91], protein sequence data banks such as the Protein Identification Resource (PIR) at NBRF (National Biomedical Research Foundation) [PIR91], protein crystallographic structure data banks such as the Protein Data Bank (PDB) at Brookhaven National Laboratory [B+77, ABB+87], and other specialty data banks such as the PROSITE protein sequence pattern data bank [PRO90]. Most such data banks utilize a file system to manage data, though GenBank recently initiated conversion to a relational database system. Each data bank formats the data differently, and the format changes frequently to incorporate new demands from the field. In computational biology research, it is often necessary to combine the data from different data banks, e.g., DNA sequence data from GenBank and protein structural data from PDB. In unifying the data from different data banks, differences in the formats are often resolved manually to create a consistent, non-redundant set of the data.

There are also various applications developed for analyzing the data obtained from the data banks, e.g., Delila, P/FDM, PKB. Delila is a database for nucleotide sequence access and analysis [SSHG82]. P/FDM is an integration of the logic programming language Prolog with the functional data model database (FDM) used for protein structural modeling

[GPKF90]. PKB (Protein Knowledge Base) is a NewS application for managing and analyzing protein structural data that will be described in detail later [Bry89]. As with data banks, each application stores the data in a different format, so the data must be converted from one format to another if more than one application are used to analyze them.

Standardized data formats described in the next section attempt to eliminate data conversion between different systems by providing uniform data structures appropriate for scientific data.

3.1.2 Standardized Data Formats

Standardized data formats are designed to provide generic data types customized for certain classes of scientific data. Most formats offer limited type support, i.e., their types are really special "data structures" with few associated operations. Arbitrary structure is supported in most cases, e.g., nested structures similar to the UNIX directory organization are supported by the Hierarchical Data Format (HDF). The types are fixed in most cases, though some extensibility is provided with HDF. Their database support is also weak, e.g., a primitive data query facility, as described later.

Standardized data formats currently available include HDF, Data and Description Rule (D&D), and the Network Common Data Form (netCDF). HDF [Nat89], produced by the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana-Champaign, is designed for the transfer of graphical and scientific data between various computer architectures, e.g, Cray, Sillicon Graphics, and Alliant. HDF supports rectangular gridded arrays. HDF data structures for the rectangular arrays include not

only the base data but also information necessary to interpret the data, e.g., dimensions of arrays, to make HDF datasets self-describing. All the types in HDF share the same array structure, but they contain different annotation attributes that distinguish them from each other. Therefore, it is possible to add a new HDF type by defining appropriate annotation attributes as long as the data can be represented in an array form. Operations for the data are provided separately from the HDF data structures. HDF currently provides command-line utilities that operate directly on HDF files, and also calling interfaces to read and write HDF files from within FORTRAN or C programs. As mentioned later, those data I/O operations are fairly primitive. NCSA also offers a set of graphics visualization programs utilizing data in HDF. Various scientific organizations, including NASA and all of the supercomputer institutes, seem to be converging on HDF as a standard data format.

In Data and Description Rule (D&D), a dataset and its description are organized together to promote communication of data among statisticians [SST90b, SST90a]. The D&D supports relations and arrays as fundamental data structures. The data descriptions include schema information on relations and dimensions of arrays, as well as miscellaneous information such as sampling methods used to collect the data.

The Network Common Data Form (netCDF) data access library was developed to represent scientific data in a self-describing and network-transparent format in order to support the creation, access, and sharing of the data [Ful89, RD90]. Its data model supports primitive types such as integer, character, and floating point numbers. A netCDF variable represents a multidimensional array of values of the same type, and has a name, a shape described by dimensions, and various attributes. Attributes may be attached to a netCDF

variable or an entire netCDF file, and represent such miscellaneous information as units, a valid range, and quality of the data. All data are represented in a machine-independent form, and they are transferrable between different types of computers without explicit conversion. A subset of data in a file can be aggregated to form a hyperslab and accessed as a unit. Both C and FORTRAN interfaces are provided for netCDF data files.

Other scientific data formats include BUFR (Binary Universal Form for data Representation) [ECM88], Candis [Ray88], and CDF [Gou88]. BUFR is a machine-independent format for self-defining data approved by a commission of the World Meteorological Organization as a standard for interchanging and transmitting meteorological and oceanographic data. Unlike HDF, data types are not extensible in BUFR; it only supports a fixed set of predefined data types in the centralized registry. Candis is a package of C software for UNIX and accommodates self-describing scientific data. It adopts a "pipes and filters" approach to data processing, and as a consequence, data access is inherently sequential; it is inefficient to access small parts of large files. The NSSDC (National Space Science Data Center) CDF is a FORTRAN library and supports multidimensional scientific data. It has been used to archive many different kinds of data, and an extensive collection of analysis and display applications is available for data in the CDF format.

Since each data format has its own advantages and scientific data are distributed among different formats, attempts have been made to develop converters between them to promote data sharing.

As is clear from the examples above, standardized data formats attempt to accommodate structures of scientific data directly. Most formats support a multidimensional array since

it is a common structure for scientific data. In most formats, data types only define a structural template, and operations for data access and commonly used computation are provided for sharing as a library of programs separately from the formatted data. In most cases, data access operations are basic and not customizable or extensible. For example, selective data retrieval is not supported in HDF and everything in a dataset must be loaded into a program even if only part of the data is actually needed. Their database support is also weak, e.g., protection against accidental data loss, associative query, and storage management are generally lacking.

3.1.3 Persistent Languages

Persistent programming languages are the result of efforts to incorporate persistent data in a uniform, transparent fashion by providing a single language and data model for both transient and persistent data. In contrast, most programming languages currently used in scientific research are non-persistent, e.g., C or FORTRAN, and persistent data are provided through an interface to a file system or a database system. As a result, programmers are faced with two different data models: the programming language's type system and another from the file or database system. Hence, persistent data reference requires explicit translation between transient and persistent data. As scientific data typically exhibit complex structure, scientific applications can devote as much as 50% of the code to flattening and reconstructing the structures of the data between the applications and a file system. With the use of a persistent language, it is possible to eliminate the part of the applications that explicitly interacts with a persistent storage system, making the code easier to understand

and to maintain.

A disadvantage of using persistent languages is that the underlying data store is dedicated to providing persistence to data represented in a particular language. Once persistent data are created through a certain language, it is easy to access the data from that language, but more difficult to browse the data directly or to use the data from other languages.

It is also the case that database support is generally weak in existing persistent languages. As many persistent languages were developed as an extension to a programming language, their data types do not always support large collections of data items that are typically encountered in databases. Such is the case with PS-Algol, i.e., the programming language Algol extended with persistence [ABC+83]. Persistent languages such as TAXIS [MBW80] and Trellis/Owl [OBS86] provide support for database access, though their support is geared toward business data processing and data structures such as sets of records. And as with standardized data formats, the data store underneath most persistent languages does not provide a complete set of data management features.

NewS is a persistent language specifically designed for scientific data analysis applications. As a persistent language, NewS provides transparent access to persistent data as well as facilities customized for scientific applications, e.g., scientific types such as arrays and time-series and a large library of functions for statistical and graphical data analysis. As such, NewS is widely used in various scientific domains and its advantage as a persistent language is well-recognized, especially when its transparent access is compared to file access in C or FORTRAN. On the other hand, NewS is weak in its data management, e.g., a lack of robust support for large data (persistent or transient), and a need for improvement is

widely recognized in the NewS community. We chose NewS applications as the target of our experimental study, and a detailed description of NewS is provided later.

3.1.4 Traditional Database Systems

This section discusses storing scientific data in traditional database systems by fitting the data into their data models. In particular, we focus our attention on relational database systems, since they have proven to be successful for business applications and are a maturing technology.

Some scientific data repositories, e.g., the GenBank Genetic Sequence Data Bank [B+90], have started converting the data stored in ASCII files to the relational format. Some tools used for analyzing scientific data are also interfaced with commercial databases. For example, the statistical data analysis package SAS, widely used in biometry and social science, is interfaced to the relational databases DB2 and ORACLE.

A drawback in adopting a relational database for scientific data is a lack of direct support for scientific data types. For example, a relational system does not support nested structures directly, so such a structure must be mapped to a collection of flat relations. Meta-information is often very difficult to express in a rigid relational model, since not everything can be expressed as numbers or strings in relations. Storage of arbitrary operations is also not supported. SQL, a query language for relational databases, is not computationally complete, so possible data manipulation is limited to simple data access, arithmetic and aggregate operations. And such access operations must be expressed as SQL queries

on relations, not a natural format for scientific data access. Many semantic concepts potentially useful for scientific data, e.g., abstract data types and generalization, are also not available in the relational model.

Traditional database features, e.g., storage management and concurrency control, are of course provided by relational systems. However, it is not clear whether or not their particular functions, e.g., strict serializability for concurrency control, are suitable for managing scientific data.

3.2 Introduction of Object-Oriented Databases

This section introduces object-oriented databases as a potentially more viable approach to managing scientific data than those described in the previous section. We first describe what an object-oriented database is, then discuss their potential advantages with respect to the requirements in Chapter 2.

3.2.1 Object-Oriented Database Overview

As their name suggests, object-oriented database systems provide both `object-oriented` and `database` features. Basic object-oriented concepts supported by most systems include `objects`, `encapsulation`, `messages`, `methods`, `classes` or `types`, `inheritance`, and `complex objects`.

An object represents a discrete entity of an arbitrary kind. Each object is associated with a unique identifier that distinguishes it from other objects independently of its value. An object is usually encapsulated and interacts with the external world only through a

well-defined protocol. Such a protocol typically consists of a set of messages that can be sent to the associated object in order to perform some action. A method is a piece of code that implements the desired action for a message. A class or type specifies a particular set of objects as its "instances". The terms type and class are often used interchangeably, but when used together, a type typically refers to a template for representation and actions of its instances, whereas a class refers to a collection of all instances of the corresponding type. In the following discussions, we will exclusively use the term "class" unless both terms are needed to distinguish a template from a collection.

Classes are typically arranged in a hierarchy, where a subclass inherits all the properties defined by its superclasses and may define additional properties of its own. There are a variety of class hierarchies possible, depending on the kind of properties inherited from superclasses to subclasses. In a specification hierarchy, an instance of a subclass can be substituted wherever an instance of its superclasses is expected. This concept of "substitutability" puts restriction on how a method can be extended and modified from a superclass to a subclass, since a method of a subclass must behave the same way as a corresponding method of its superclasses. A subclass in an implementation hierarchy inherits and extends implementation, i.e., representation and code for operations, of its superclass. In a constraint hierarchy, a subclass inherits all the restrictions on instances specified for its superclasses and may be further constrained. In an extent hierarchy, a class is a collection of all instances of the corresponding type, and a subclass represents a specific sub-collection of a superclass. It is possible for the same collections of objects to be organized differently depending on the choice of hierarchy. Among a variety of different hierarchies described

above, implementation hierarchy offers a practical advantage by encouraging "inheritance" or reuse of existing code for easy implementation and maintenance of an application program.

There is also a variation in the form of inheritance allowed in hierarchies; some systems incorporate multiple inheritance, i.e., a class may have multiple unrelated superclasses, whereas others permit only single inheritance where there is only one direct superclass and the hierarchy forms a tree. With multiple inheritance, a mechanism is necessary to resolve conflicts in case properties (e.g., methods) of the same name are inherited from different classes.

Many object-oriented database systems support the concept of complex objects by providing references or relationships between an assembly object and its component objects. Besides direct assembly-component reference, additional properties are supported by some systems. For example, a common value shared by an assembly object and its component objects, e.g., color of paint for an entire car and its body parts, may be propagated from the assembly object to its components. Or when an assembly object is deleted, all its components may be also deleted automatically. By allowing components to be arbitrary objects, complex objects can directly represent hierarchically nested structures that are common in engineering or scientific applications.

Traditional database features include `persistence`, `storage management`, `concurrency control`, `recovery`, `indexing`, `query`, and `schema changes` or `versions`. Data persistence is fundamental to database systems, and in most object-oriented databases, objects of all

kinds can be stored permanently for repeated access from applications. Storage management typically handles management of secondary storage as well as in-core memory. Its functionalities include data transfer between secondary storage and in-core memory, clustering, and caching of data for efficient data access.

Concurrency control allows multiple users to access the same data while maintaining data consistency. In many systems, concurrency control is provided through `atomic transactions`, where a set of operations can be committed or aborted as a unit and `locks` "reserve" data for various usages.

With a recovery mechanism, a system can restore some coherent state of the data after various kinds of failures, e.g., software, disk or processor failure. And an index is a useful mechanism for efficient value-based data access.

An ad-hoc query facility could be a query language or graphical interface that allows users to specify the data they want without programming. There is currently no single object-oriented database query language that is accepted as widely as SQL in relational databases, though various extensions to SQL that incorporate object-oriented concepts have been proposed.

Schema changes or versions are supported to accommodate data update, though most systems only support very simple changes in a class hierarchy, e.g., changes to leaf classes only, or modification of methods.

3.2.2 Potential of Object-Oriented Databases

An advantage of an object-oriented database for managing scientific data is its flexible data model capable of supporting scientific data types. For example, nested structures are directly accommodated in an object-oriented data model with recursive decomposition of an object. A scientific data type can be represented as a class with both structural and operational semantics defined in a uniform framework. Unlike many standardized data formats, types are extensible with an object-oriented data model through definition of arbitrary new classes. Generalization hierarchy supports specialization of a generic model for different systems through subclassing. Multiple inheritance, though not provided in all object-oriented data models, would directly accommodate unification of multiple types for the same physical system. Generalization hierarchy and associated inheritance are often considered by domain scientists as one of the most significant benefits of an object-oriented data model, since it promotes reusable, modular design for efficient development and easy maintenance of complex applications.

Advantages of an object-oriented data model have been recognized in various scientific domains and the model was adopted in some applications. APEX (A Physics Expert) is a computer program built in an object-oriented programming environment (GLISP, A LISP-based programming system with data abstraction) for solving physics problems in a model-based representational framework [KN91]. Many physics problems are initially presented in terms of informal, real-world entities and relationships among them. The first step in a problem-solving process is to translate those entities and relationships into formal, abstract

models of physics. APEX assists physicists in obtaining such a formal representation of a problem by providing two kinds of models; canonical physical objects and physical models. A canonical object class represents an idealized unit in physics such as a point mass and an ideal rope. The conversion process of abstracting real-world entities as canonical physical objects is represented as a view object. A physical model is an encapsulation of a single principle or law of physics; each model class specifies canonical physical objects involved in the model and constraints imposed by the represented law of physics either as a set of equations or a specific environment for the objects. In a typical process of selecting an appropriate physical model for a problem, a generic physical model class is specialized with additional attributes for the particular case.

The Thermodynamics Workbench performs thermodynamic calculations, implemented using an object-oriented paradigm in Common LISP [MK89]. Each Workbench object corresponds to a thermodynamic subsystem or connection between subsystems. Such a connection is represented by a wall object that regulates the flow of extensive quantities between subsystems. Thermodynamic model classes are arranged in a hierarchy, with the most general class representing a basic thermodynamic model that defines such generic properties as total entropy, enthalpy, and Gibbs energy. Since subsystems are represented as encapsulated objects, it is possible to select different models for each subsystem. The Workbench is still able to find an equilibrium state among the subsystems as interaction between them is abstracted out to their external access protocol.

StrateGene represents data from cloning experiments using objects, classes of objects, and associated attributes for both as supported in the KEE frame system [CCM88]. The

objects and classes are called `units` and the attributes are called `slots`. Object or classes can have method slots that specify various operations. The subclass mechanism is used to represent taxonomic hierarchies of biological models. For example, a general class called ENZYMES is defined with components such as MOLECULAR.WEIGHT and SUBSTRATE. Subclasses of the ENZYME class such as PROTEASES and RESTRICTION.ENZYMES are then defined with the components inherited from their superclass ENZYME as well as additional components such as RECOGNITION.SEQUENCE for RESTRICTION.ENZYMES.

P/FDM utilizes certain object-oriented features to represent protein structure data [GPKF90]. In P/FDM, an object is a basic unit of data, and properties of objects and relationships among them are represented as functions over object classes. Object classes are organized in a specialization hierarchy where properties are inherited. As mentioned in Chapter 2, the NIH Genome Database also utilizes some object-oriented concepts, representing gene sequences as abstract objects independent of their internal representation [And89]. Therefore, even though internal representation is modified due to changes in technology, the external access protocol for gene objects remains invariant.

In all the examples above, scientific models in a variety of domains are directly supported as a class. An object represents an instance of a model, i.e., a "view" of a physical system through the model. Inheritance has successfully been used to represent taxonomies of models, with common properties among similar or related models defined once and effectively reused. Encapsulation provides an advantage as in the Thermodynamic Workbench and NIH Genome database, since an abstract view of a model is separated from internal representation and changes in the representation of an individual object do not require

modification to the rest of the system.

As object-oriented databases combine a flexible data model with traditional database features, they can deliver direct support for scientific data types without compromising database support needed by scientific applications. In contrast, current approaches such as standardized data formats do not provide a complete set of database capabilities, whereas traditional databases do not provide adequate type constructors for scientific data types.

3.3 Summary

This chapter examined current approaches to scientific data management, and described the potential advantages of object-oriented databases over other examined approaches. Customized data management systems often serve their purpose with respect to the targeted applications, but lack extensibility beyond that limited scope. A lack of standards among them makes analysis of the same data by different applications difficult. Standardized data formats attempt to provide uniform data structures customized for scientific applications, but generally lack data management features such as direct data queries, storage management, protection against loss of data and concurrency control over simultaneous data update. Use of persistent languages would benefit scientific applications if their data types are adequate for scientific data, but they are also limited in data management features. Traditional database systems, in contrast, have data management capabilities, but their data models are inadequate for scientific data.

Object-oriented database systems combine a flexible data model with data management capabilities to provide advantages over the current approaches to scientific data management

described above. An object-oriented data model can directly support scientific models as a class as seen in various examples. Based on this analysis, we chose an object-oriented database (GemStone) for a baseline data management architecture of the experimental platform we constructed, i.e., GemStone-based NewS, which will be introduced later in the dissertation.

Chapter 4

Criteria for an Experimental Platform

Chapter 2 discussed data support requirements of scientific applications, identifying a need for scientific data types and various traditional database features. In this study, we investigated how to support such requirements. Our investigation centered on construction of an experimental platform and experimentation on that platform.

We took an experimental approach since we wanted to examine possible data support architecture against actual instances of scientific data and functions executed on them. While it is possible to examine data structures or function definitions statically, information on dynamic characteristics such as data access patterns are hard to obtain without actual execution, especially because very little has been documented on dynamic features of existing scientific applications. We constructed an experimental platform so that we can compare possible architectural alternatives against a variety of scientific applications in the same environment. The platform also provides the possibility for exploring dynamic characteristics of existing scientific applications as many of such characteristics are yet to be discovered.

We tried to construct a cost-effective platform that would deliver maximal information with minimal effort. Hence, we put a priority on the following criteria.

1. **`Cost-effectiveness`**. Cost-effectiveness refers to the amount of work required to put

together the platform. For example, it minimizes cost to maximize reuse of existing components in constructing a platform.

2. **Productivity and Flexibility.** We expect experimentation on the platform to produce information on design and architecture leading to a production system. To make a good start toward that direction, the base architecture of a platform should be potentially a good match for the requirements of the targeted applications. Also, if the architecture is flexible, we can tune it for targeted applications.

3. **Generality.** The platform should accommodate experimentation with a wide variety of scientific applications and domains so that it is not necessary to implement a separate platform for each.

4. **Performance.** There is a minimum level of performance needed to perform a statistically sufficient number of experiments with existing applications on a platform. Ideally, a platform would evolve into a system with performance sufficiently close to a production system so that the architecture of the platform can be tested in the field. Such a field test will fine-tune the architecture, eventually yielding a final production system. In reality, such evolution of a platform would likely take multiple steps, and in each step, we should be able to learn incrementally about applications' requirements, database design, and possible performance improvement.

5. **Ease of Experimentation.** Ease of experimentation indicates a preference for a small amount of work required to set up and perform experiments. For example, a platform that allows for reuse of existing applications "as is" is preferred to a platform

that requires major modification of the applications.

Later in the dissertation, many of the design decisions for the platform will be discussed with respect to the criteria above. Through our experimentation, we will also examine if the constructed system has turned out to a good experimental platform based on those criteria.

Chapter 5

Issues for Investigation on a Platform

The last chapter proposed experimental study on scientific data management and defined a set of criteria for a cost-effective experimental platform. This chapter discusses the kind of issues one might wish to examine with such a platform. Throughout the discussion, we refer to the data support requirements described in Chapter 2, since our objective is to investigate possible design dimensions in supporting the identified requirements. Note that any single architecture is unlikely to support investigation of all possible issues, and a particular platform we constructed is no exception. Later, we will evaluate suitability of our platform by the range of issues readily examined on the platform among those discussed here.

5.1 Support for Scientific Data Types

In order to support scientific data types effectively, we must consider issues such as a type definition facility, implementation of types, and possible queries on types. We provide a separate discussion on each issue below.

5.1.1 Type Definition Facility

One would like to explore possibilities in designing a facility for defining scientific data types. Note that this dimension only concerns external specification of the types, separate from

their implementation. A definition facility should provide a set of constructs that directly supports such requirements as hierarchical structure, abstract data types, and generalization hierarchy identified earlier. Also included in this dimension is the design of syntax for type definition and a user interface aimed at maximal ease of use. Another possibility is to examine definition of data semantics apart from individual type definitions, e.g., relations over types such as specialization.

5.1.2 Implementation of Types

Having specified a collection of types, one needs to construct time- and space-efficient implementation for those types. As we identified that scientific data types should include not only data but also operations, we discuss implementation of both state and operations below.

State

To represent the state of a data type, one would like to experiment with various constructors for defining that state such as tuples, sets and arrays. Though not supported in traditional databases, arrays are one of the most common structures in scientific applications, and it is worth investigating their use for scientific data. For productive experimentation, a platform should provide a variety of constructors for representation, including arrays. Furthermore, an ability to augment a given set of constructors would be also useful.

One can also consider different implementations for a given constructor. The same conceptual constructor can be represented as an encapsulated, first-class object complete

with its unique identifier, or as a value without identity. Implementation can differ based on data size, for example, large arrays might have a different implementation from small arrays.

Another possibility is to examine different ways to structure the state for a given type. The state of a scientific data type can be organized according to the structure of a corresponding model to accommodate direct representation of the model, though variations are possible to customize the structure for individual needs. For example, in a hierarchical decomposition, frequently-accessed information, regardless of its original position in the model, can be "cached" or duplicated at the top level for efficient access.

Operations

Implementation of operations depends on a programming language and associated paradigm of a choice, e.g., procedural, functional, or equational. The essence of many scientific models is expressed by a set of formulas, for which equational representation may be appropriate. On the other hand, operations such as customized data access are better represented procedurally or "declaratively", i.e., by declaring the kind of data requested. Hence, in this dimension, one can first explore suitability of each programming language for the kind of operations executed in an application. Note that it is possible to consider mixing multiple languages as certain languages permit a call to routines written in other languages. Given a particular programming language (or its combination), one would like to consider alternative implementations of the same operation based on a variety of criteria. For example, one can explore modular design for ease of development and maintenance, or attempt to

optimize implementation for efficiency.

5.1.3 Query on Types

We mentioned in Chapter 2 a need for a query facility that allows domain scientists to access data in a way natural to them. Since scientific data types represent a conceptual framework used by scientists, it would be intuitive for them if queries can be expressed in terms of those types. For such type-based queries, one would like to investigate a variety of design options for a query-definition facility such as appropriate language syntax. A use of a graphical interface may provide domain scientists a more intuitive alternative to programming in forming a query, though it must be still possible to incorporate programming if arbitrary computation is needed for the query. It would be also useful to explore a query-processing environment for such type-based queries and consider issues such as possible optimizations and necessary auxiliary access structures. Note that if we represent queries as type-specific operations, the issues discussed on implementation and execution of operations also become applicable to queries.

5.1.4 Other Data Type Support

It would be beneficial to examine other data type support such as unification of multiple types, a potential use of generalization hierarchy, type evolution as very little has been known about such requirements on scientific applications. For example, one can explore multiple inheritance or union as a means to merge multiple types for the same physical system; as discussed in Chapter 2, it is often necessary to accommodate multiple types simultaneously in scientific applications. A variety of properties can be inherited from a

general model to a more specific version; we cited earlier an example of mean field theory where a concept and general structure of a partition function is reused in different examples. Hence, one can examine the kind of properties inherited in the taxonomy of the domain to determine an appropriate form of generalization. As for type evolution, one can ask whether specialization of existing types is sufficient, or more arbitrary changes need to be supported.

5.2 Execution of Operations

As operations are included in types, one would like to consider alternatives for their execution scheme. If an application operates on data stored in a database, actual execution can take place either in the database or in an application's memory space. A variety of policies are conceivable for deciding a location of operation execution. Another possibility is to explore dynamic optimization of an operation as it is executed on various data instances.

5.3 Data Movement

An issue related to operation execution above is cost of data movement. As operation execution spans a database and an application, data transfer is incurred between them, and an execution scheme should be mindful of that cost. The location of an operation (or its subparts) can be varied to come up with minimal data transfer cost.

5.4 Traditional Database Support

One would like to examine required database support for scientific applications, e.g., necessary indexing structures and an adequate concurrency control scheme. For example, with concurrency control, it is yet to be determined whether existing transaction models are adequate for scientific applications, or a new model needs to be developed for them. As scientific applications encompass a variety of domains, a solution may well be domain-dependent.

5.5 Dynamic Properties of Applications

At the outset of this study, we were especially interested in using a platform to run applications and examine their dynamic properties. Such "dynamic" examination would gather statistics on the actual numbers of each type of objects used, and the frequency and cost of operations executed on them. Design of database support such as storage management, query processing, and operation evaluation policy should also benefit from looking at such run-time properties as distribution of size of data instances and access patterns on data.

5.6 A Range of Applications

One of the criteria we discussed in the previous chapter was a generality of applications supported by a single platform. We would like to investigate any of the issues discussed above across a variety of applications to see how dependent it is on application specifics.

5.7 Summary

The following is a summary of design dimensions we would be interested in investigating on an experimental platform.

- Support for scientific data types
    1. type definition facility
    2. implementation of state
    3. implementation of operations
    4. queries on types
    5. other data type support such as inheritance and type evolution
- Execution of operations
- Data movement
- Traditional database support such as index and concurrency control
- Dynamic properties of applications
- Multiple applications

The rest of the dissertation describes the design of an experimental platform we constructed and investigation subsequently performed on the platform. As mentioned before, the platform does not support investigation of all the issues discussed in this chapter. We will later assess suitability of the constructed platform for our study based on the kind of issues one can easily investigate with its architecture.

Chapter 6

Introduction Of Experimental Platform: GemStone-based NewS

This chapter introduces an experimental platform we constructed, GemStone-based NewS. In designing the platform, we chose to interface a scientific persistent language NewS to the GemStone object-oriented database. Chapter 3 motivated the use of an object-oriented database for scientific data management and our choice of GemStone. NewS provided us with an entry point into a variety of scientific domains, e.g., statistical research, computational biology, epidemiology, economic analysis, where its statistical and graphical analysis capabilities are extensively used. As current NewS relies on files for persistent data store, many NewS applications suffer from a lack of robust data support. By interfacing GemStone with NewS, we could examine object-oriented database support, i.e., a potentially better alternative to files, for different NewS application areas on a single platform. NewS also provided us with an opportunity to reuse existing application programs as is.

In this chapter, we discuss why we chose two main components of the platform, i.e., an object-oriented database and a scientific persistent language, with respect to the criteria presented in Chapter 4. We also describe specific systems we chose for those components, namely, GemStone and NewS, and justify their choice.

6.1 Object-Oriented Database GemStone

The potential of object-oriented databases described in Chapter 3 was a main motivation for us to choose an object-oriented database as a baseline architecture for our experimental platform. As mentioned before, an object-oriented database combines a flexible data model that can accommodate scientific data types with traditional database support, providing a good starting point toward productive experimentation. An object-oriented database is a generic system that can be adapted to a variety of scientific applications, so it provides a cost-effective, reusable approach compared to customized data stores developed separately for each application. Flexibility of an object-oriented database architecture renders a variety of design alternatives, and it is possible to tune the architecture for targeted applications.

We chose the commercial object-oriented database GemStone for our experimental platform. As a commercial system, GemStone delivers a robust implementation of object-oriented data modeling concepts and various database features, so we did not have to implement those features from scratch or to spend much time augmenting them. Instead, we could concentrate on examining possible variations in utilizing such features. Therefore, considering criteria for the experimental platform discussed in Chapter 4, GemStone seemed a good choice in terms of cost-effectiveness, productivity, and flexibility.

Note that there exist a variety of commercial object-oriented databases besides GemStone, each with a different architecture, and we did not necessarily conclude that the GemStone architecture is the best choice for scientific applications. When this study started, there were very few object-oriented databases available, especially a commercial product,

so the choice of GemStone was mainly due to its availability for our study. In our experiments, we used GemStone features as "references", examining what features work well with scientific applications and what features are inappropriate or lacking. The following section provides a description of GemStone.

6.1.1 GemStone

GemStone is an object-oriented database system available from Servio Corporation. It provides basic object-oriented concepts such as objects, messages, methods, classes, and inheritance, essentially through Smalltalk semantics. Its data language OPAL closely follows the syntax and semantics of Smalltalk. A simple example of OPAL program for class and method definitions is shown below.

```
Vector subclass: 'Sclass'
    instVarNames: #('data' 'names')
    classVars: #('subclasses')
    poolDictionaries: #[ ]
    inDictionary: Sdef
    constraints: #[ ]
    isInvariant: false

method: Sclass
foo
    "Returns the value of an element called 'foo'."
```

```
| idx |
idx := names indexOfValue: 'foo'.
(idx > 0)
    ifTrue: [ ∧(data at: idx) ]
    ifFalse: [ ∧nil ]
```

GemStone represents many constructs in the system as objects, including class and method definitions.

GemStone is implemented as a set of two communicating processes. The `Stone` process is responsible for most database functionality, e.g., object management, authorization, concurrency control, transactions, and recovery, and communicates with the underlying file system to perform those tasks.

The other process, `Gem`, is layered on top of Stone, and provides the virtual image that consists of various kernel classes such as Numbers, Strings, Arrays, Sets, Bags, Collections, and Dictionaries. Gem also includes a compiler that translates OPAL programs into bytecodes and an interpreter to execute those bytecodes.

As a database, GemStone provides such traditional database functionalities as data persistence, storage management and efficient support for large objects, query capabilities, concurrency control and recovery, indexing, and constraints. In GemStone, all objects are persistent, and they are referenced by object-oriented pointers (OOP's). An Object Table maps OOP's to the physical locations of object state. A complex object is represented by a collection of OOP's for its subcomponents. The state of a subcomponent cannot be

stored directly within the state of its parent object, but the subcomponents can be clustered together. Indirection through OOP's allows GemStone to freely change the physical locations of objects without affecting applications. Multiple complex objects can share a common component through the OOP of the component object. Such indirection, however, contributes to the storage overhead associated with GemStone objects.

GemStone stores large arrays as a tree structure. When a particular element is accessed, GemStone only loads the portion of the tree needed to locate the element. GemStone also caches accessed data objects for efficient subsequent access.

GemStone allows queries to be specified over the hierarchical structure of an object. A sequence of instance variable names are used to specify a particular path to be queried in the object structure. This extension, though common in database queries for complex objects, conflicts with Smalltalk semantics. A Smalltalk object is accessible only through a fixed set of messages defined in its external interface, and the representation of an object, i.e., its instance variables, is not directly accessible to external programs.

GemStone provides both optimistic and pessimistic concurrency control. With the optimistic approach, each user freely accesses objects, but the changes are committed only when there is no conflict between users. With the pessimistic approach, users put locks on the data ahead of time to make sure all the work will be committed.

GemStone version 2.0, which was used in our study, provides an application programming interface to Smalltalk and also to C. An integrated, windowed Smalltalk environment is the typical way to access and to create GemStone objects and classes. Alternatively, users can use an interactive, command line interface TOPAZ for more limited forms of

object access. Later versions of GemStone also provide an interface to C++ applications.

6.2 Scientific Persistent Language NewS

For scientific applications in our study, we chose a generic persistent language and programming environment for statistical data analysis and graphical operations, namely, NewS, instead of particular scientific applications or domains. Since statistical models are widely used in different scientific domains, a variety of NewS applications from multiple domains were available for our study. The number and range of existing NewS applications are admittedly more limited than scientific applications implemented in languages like C or FORTRAN. However, unlike C or FORTRAN, NewS is a persistent language with a uniform data model for both transient and persistent data. Whether a NewS vector resides in memory or in a file, users can access it simply by giving its name. The NewS interpreter looks for the accessed vector among in-memory lists of objects called `frames` as well as in the file system data directories specified in a search list. Therefore, specifics of the underlying data store are not visible to NewS applications, and they can be made to access data in GemStone without modification. In contrast, C or FORTRAN applications contain explicit file reads and writes, and such file operations must be replaced with data access operations supported by a database to use them in experimentation. Therefore, the choice of NewS was a result of trading off generality and ease of experimentation. The following section describes NewS in detail.

6.2.1 NewS

NewS is a language and interactive programming environment from AT&T Bell Laboratories with functional computation model [BCW88]. NewS is designed for statistical and graphical data analysis, and provides many data management operations, computational techniques, and graphical operations as library functions. NewS supports a variety of scientific models as data types by providing constructors for the types as library functions. For example, the `lsfit` function is a constructor for the `linear model` type and creates an instance of the type. The collection of NewS types is also extensible, namely, it is possible to define a new constructor for an arbitrary scientific model in the NewS language. As we identified requirements of scientific applications in terms of data type support and traditional database support in Chapter 2, the following sections describe NewS in those two aspects, i.e., the NewS data model and data support for NewS.

NewS Data Model

All the data in NewS are called objects; they can be simple objects, or `vectors`, of various `modes` or instances of certain predefined `classes`. Both modes and classes define a template for objects, therefore, they are synonymous with data types. A vector consists of ordered elements or components that may optionally have associated names. (The terms "element" and "component" are used synonymously for the data values of NewS vectors.) If none of its elements contains another object, a vector is called `atomic`. Vectors that include other objects as elements are called `recursive`. Recognized modes for atomic vectors in NewS include `null` for the empty object, `logical` for a vector of boolean values, `numeric` for a

vector of numeric values, `complex` for a vector of complex numbers, and `character` for a vector of character strings. There are many recursive modes; three that can be used in a general way are `list`, `graphics`, and `expression`. As with the `expression` mode, some modes are defined for representing NewS language constructs, as many such constructs, e.g., expressions and function definitions, are treated as objects. Therefore, NewS expressions and functions can be passed as parameters to functions, or operated upon as data.

Predefined classes in NewS are `matrix`, `array`, `category`, and `time-series`. They possess not only a vector that contains data values but also attributes that describe various aspects of the objects. For example, data values of a `matrix` object are specified column by column in its `data` vector, but a `matrix` object also has an associated `dim` attribute specifying dimensions of the matrix. A `category` is an object used to represent data that are discrete-valued and the values fall into one of a set of possible ranges specified by a `levels` attribute.

Computation in NewS is based on the functional model; the basic unit of computation is a function. Most operations in NewS, including arithmetic operators, are provided as functions. Many NewS functions are "polymorphic", i.e., they accept arguments of different types. For example, various trigonometric functions in the NewS library can accept both numerical value and complex value as arguments. The functions distinguish those cases and perform an appropriate computation accordingly.

In the NewS environment, users type arbitrary expressions that are interpreted and evaluated by the NewS interpreter. Expressions are made up by combining constants, operators, functions, and named data. Expressions may have a nested, hierarchical structure.

The following are some examples of NewS expressions that illustrate NewS syntax for various operations, e.g., assignment, application of a function to a vector, implicit and explicit iteration, and data access (`[]` for subset extraction and `$` for component access).

$> small.primes \leftarrow c(2, 3, 5, 7, 11)$

assigning a vector of 2, 3, 5, 7, 11 to a name "small.primes".

"." in "small.primes" is part of vector name.

"c" is a vector constructor function in the NewS library.

$> mean(small.primes)$

the mean value of small.primes, represented as a one-element vector.

$> if\ (data.ok == TRUE)\ sum(small.primes)$

sums up all the values in "small.primes" if they are OK

$> for\ (sn\ in\ state\$name)\ income[sn]/population[sn]$

for each state, get the average income.

Vectors "income" and "population" are indexed by sn (state name).

$> small.primes[1:3]$

the first, second, and third elements of the `small.primes` vector

$> small.primes[-2]$

all but the second element

$> small.primes[extract == "yes"]$

a subset of the `small.primes` vector is specified by the positions of elements

with the "yes" value in the `extract` vector

An assignment performed at the top level of a NewS session, as in the first example above, creates a persistent object of the name on the left-hand side. The value of such a persistent object is stored in a UNIX file. Assignments within an expression, e.g., those included in a function definition, create temporary objects. To perform a permanent assignment from an arbitrary scope, a double arrow `<<-` must be used.

Most NewS functions operate on an object as unit with implicit iteration over its elements. In the second example expression above, `small.primes` represents a vector, and the mean value over all its elements is calculated by a simple expression `mean(small.primes)`. As shown in the fourth example, NewS does provide constructs for explicit iteration such as `for` and `while`, but treating an object as a unit is characteristic of all the classes provided in NewS.

Users can define new functions in the NewS language. A NewS function definition consists of a name, formal arguments, and a body. The body of a NewS function is any legal NewS expression. The following example illustrates syntax of the NewS language for a function definition with the `square` function that squares the argument.

$$> square \leftarrow function(x)\ \{x \wedge 2\}$$

Like many NewS functions, `square` is polymorphic and applies to a variety of modes and classes, e.g., numeric and complex.

Interfaces to C and FORTRAN are provided so that a user can choose an appropriate language for performing external operations. NewS is a computationally-complete language,

but some operations may be performed better in languages like C or FORTRAN, especially if the operations are computationally-intensive or for which there already exists some C or FORTRAN program.

An object-oriented flavor has been implicitly incorporated into NewS. As mentioned before, data in NewS are called objects and treated as a unit rather than on an element-by-element basis by most NewS functions. Object-oriented terminology such as classes, inheritance, and methods are used to describe NewS semantics, though unlike languages like C++ or Smalltalk, there is no explicit syntax for those features in the language. For example, a "class" is created in NewS by defining a function that creates class instances. The following `lsfit` function implicitly defines the `lsfit` class for the regression model by creating its instance with specified elements.

$$
\begin{aligned}
&> lsfit \leftarrow function(x, y)\{ \\
&xqr \leftarrow qr(x) \\
&list(coef = qr.coef(xqr, y), \\
&residuals = qr.resid(xqr, y), \\
&qr = xqr) \\
&\}
\end{aligned}
$$

After performing the QR decomposition of input `x`, i.e., a numerical technique for linear least-squares computation, the `lsfit` function creates a list of components that represent the results of the decomposition, such as the coefficients and residuals, using the NewS library function for creating a list, `list`.

NewS users can "inherit" an existing class in defining a new class. In the following example, the `extended.lsfit` class is defined as a "subclass" of `lsfit`,

$> extended.lsfit \leftarrow function(x, y)\{$

$lsfitRslt \leftarrow lsfit(x, y)$

$addEl1 \leftarrow ...$

$addEl2 \leftarrow ...$

..... }

where one of the elements contains a `lsfit` object and various other elements are also added.

The August 91 release version of NewS, later than the June 89 release used in our study, incorporates the concept of a "method" as a function with arguments of specific modes or classes. As indicated earlier, many NewS functions are generic and accept actual arguments of different modes. Such a function is now implemented by delegating an operation to an appropriate method, depending on the mode or class of actual arguments. While generic functions needed to know all possible modes of their arguments in advance and have an appropriate operation for each mode, addition of methods makes it possible to accommodate arguments of new modes simply by adding appropriate methods at any time.

Data Support in NewS

In selecting NewS applications for our study, one of the questions we had was whether or not they need traditional database support at all beyond file functions. Though a need for

improved scientific data management has been recognized in general, scientific applications encompass a variety of domains, and for some applications, with their limited requirements, files provide a satisfactory solution. So we first examined current data support in NewS and a need for improvement, to make sure that a switch from files to a database in NewS is validated effort aimed at improved scientific data management.

As mentioned before, persistent data in NewS are currently stored in the underlying operating system's files. As files provide very limited data management capabilities, NewS as an application is mainly responsible for proper representation and management of persistent data. Some data management functions have been incorporated into the current NewS interpreter; for example, evaluation of each expression is treated as a "transaction" at the implementation level, and all the assignments within the expression are committed to files as an atomic unit only on successful completion of the evaluation. However, current NewS lacks robustness in certain areas of data management. For example, it does not support very large data efficiently, nor does it cache accessed objects for later use.

Adopting a database system for storing persistent NewS data can certainly improve data support. For example, a database is traditionally strong in storage management and support for large data, and it can often avoid looking at the entire collection by utilizing mechanisms such as an index for the selection process. In contrast, current NewS processes an object as an indivisible storage unit. When a function is executed, each argument is fetched in its entirety, even if only part of it is actually needed in the computation. General-purpose query capabilities in a database would allow users to examine data directly instead of through NewS sessions. Currently, persistent NewS data are stored in files in a binary

format, and direct examination of the content is not possible. With a database, a schema can be defined and stored for the description of the data format. In current NewS, there is no facility to define a schema. The only way to check the format of an object is to display its contents. A database can also provide more sophisticated concurrency and access control for data sharing than what are usually provided with files. When multiple applications access the same data, which is a likely situation with NewS data, a database can support data access by each application by providing an appropriate "view" of the data.

The need for database features above, e.g., support for large data and concurrency control, has been recognized by NewS users. For example, time-series data are often accumulated over years, amounting to gigabytes, which the current NewS has trouble handling. A need for concurrency control on stored functions has been discussed, i.e., a need to "lock" the debugged, released functions to control their access, especially from novice users.

With a benefit of database support recognized for NewS as above, the next question we had was what kind of database would be the best choice for NewS data support.

Compared to other types of databases, an object-oriented database provides advantages for supporting NewS, since its data model is a good match for the NewS data model. NewS data are already called "objects" and operated upon functionally as an encapsulated unit. Unlike relational databases that support only relations (sets) and tuples, object-oriented databases provide a rich set of structural constructs. GemStone has direct support for arrays, which can be used to represent an ordered collection of elements in NewS vectors. The hierarchical structure of recursive NewS objects can be directly represented with GemStone objects since "instance variables" or components of GemStone objects can recursively refer

to other objects. In contrast, hierarchical structure is not supported in relational databases, and recursive objects must be decomposed into multiple relations. Some relational database interfaces for NewS have been developed and are currently in use as the NewS data are often stored in relational databases. However, retrieval of recursive NewS objects from relational databases through such interfaces has turned out to be very expensive as it requires joins of relevant relations.

With object-oriented databases, arbitrary operations can be represented as "methods". Therefore, various data access operations for NewS objects, e.g., NewS subscript functions [ ], [[ ]], $, can be directly executed on the objects in a database as a method, and transparent data access of NewS is easily maintained. This flexibility provides advantages over relational systems where queries must be issued in a specific format (i.e., SQL) and therefore either a user or the evaluator must somehow translate access to NewS objects into an equivalent query in such a format. In general, not just data access functions but any NewS functions can be stored as a method.

Though we identified benefits of database support for NewS, especially with object-oriented databases, additional data management capabilities of database systems do not come without cost. Access to a general-purpose database usually incurs performance overhead compared with file access, and trade-offs between additional data management capabilities and performance overhead must be carefully considered in order to choose data support appropriate for each user, application, or environment.

6.3 Summary

This chapter introduced our experimental platform, GemStone-based NewS. We described two main components of the platform, GemStone and NewS, and justified our choice. As we identified the potential of object-oriented database support for scientific applications compared with other possibilities such as relational databases and files, we chose a commercial object-oriented database for data support for the experimental platform. We did not construct our own data store since the objective of our study was to explore possible design alternatives in utilizing object-oriented database features for supporting scientific applications, rather than to construct a customized architecture for a specific application. With the flexible features of an object-oriented database, we anticipated productive experimentation with many design alternatives to explore.

We chose NewS for generality and ease of experimentation. As a language and environment for statistical and graphical data analysis, NewS provides features such as hierarchical decomposition and statistical models that can be used to construct data types for scientific domains. As NewS is a persistent language, we could reuse existing NewS applications without modifications in our study; the evaluator was modified to look for the objects in a database as well as in files. NewS applications indeed benefit from database support, as file-based data management in current NewS lacks robustness. To provided much needed database support, an object-oriented database seemed a good choice for NewS due to a matching data model.

The next chapter describes various design issues we investigated in constructing GemStone-based NewS.

Chapter 7

GemStone-based NewS: Design And Implementation

This chapter discusses the design and implementation of the first prototype of GemStone-based NewS. As illustrated in Figure 7.1, we incorporated a GemStone interface into the current NewS architecture as a "switch" added to the existing file interface. Such an architectural modification entailed changes in both NewS and GemStone as described in this chapter. By letting NewS access both files and GemStone objects, all the predefined NewS data and functions in files were readily available for our experiments.

The rest of this chapter is organized as follows. Section 7.1 summarizes changes made to NewS. More detailed descriptions are given elsewhere [Ken91]. Section 7.2 discusses the design of a GemStone database for NewS. Representation of NewS objects and operations are described, including possible alternatives and justification for the chosen design. Since GemStone is capable of executing operations, we also discuss what NewS operations potentially benefit from execution inside GemStone.

Section 7.3 provides physical specifics of our prototype such as versions of GemStone and NewS used and how both systems are combined.

Section 7.4 gives the priorities during the design process, i.e., flexibility of the overall design versus performance optimality. We provide some examples of experiments to be

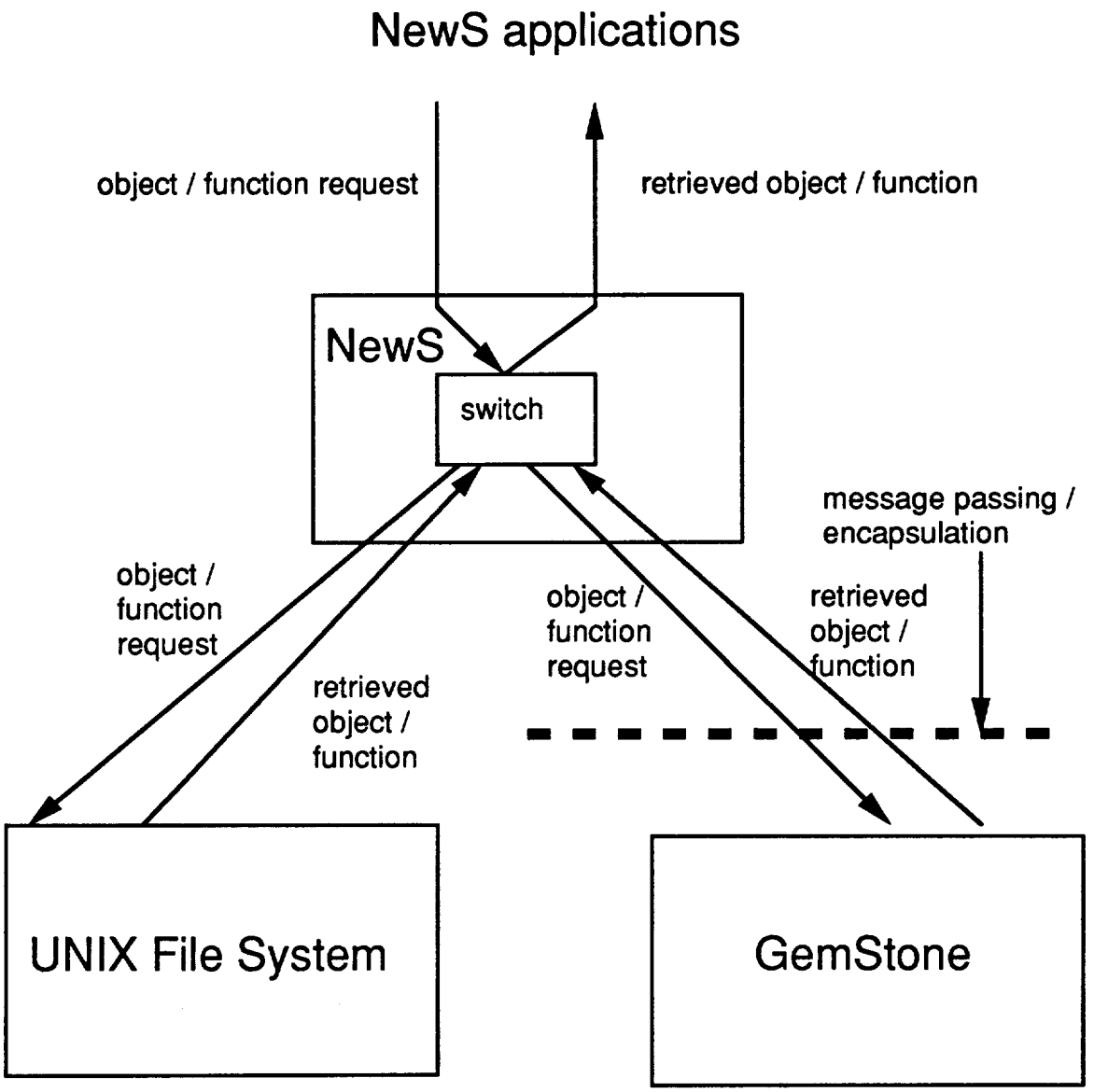


Figure 7.1 GemStone-based NewS Architecture

conducted with GemStone-based NewS, and discuss how flexible features of the system accommodate easy execution of such experiments. Finally, Section 7.5 summarizes our design decisions with respect to the criteria in Chapter 4.

While the initial concept of the NewS-GemStone architecture as a vehicle for investigating object-oriented database support for scientific applications was our own, the actual construction of GemStone-based NewS was a joint research project. The work of our collaborator, Brian Kennedy, is reported elsewhere [Ken91]. Items that resulted from the work represented by this dissertation include representation of NewS objects, determination of candidate NewS functions for storage and execution in GemStone, and implementation of basic functions for data transfer between NewS and GemStone. Representation of NewS operations in GemStone is joint work with Kennedy. The changes to NewS were mostly carried out by Kennedy, though there was collaboration on the design, particularly regarding the interface to GemStone. We also investigated the base architecture of the constructed platform and its possible variations against sample NewS applications, including Protein Knowledge Base, as described later in this dissertation.

7.1 Changes Made to NewS

This section describes changes made to NewS to incorporate the GemStone interface. Since GemStone-based NewS has multiple places to look for an object, we describe the mechanism for object location. We discuss the particular mechanism chosen for NewS for deciding where an operation is executed, i.e., in NewS or in GemStone. Other issues considered include possible ways of incorporating transaction mechanisms of GemStone into NewS and partial

loading of object state from GemStone to NewS. These and other issues are described in detail elsewhere [Ken91].

In incorporating the GemStone interface into NewS, we tried to preserve the syntax and semantics of NewS so that we could reuse existing NewS applications with minimal modifications. For example, the location of operation execution was kept transparent to applications to preserve the syntax for calling functions. GemStone transactions were incorporated into NewS as a set of NewS-callable functions for committing and aborting a transaction, for setting locks on objects, and so on. No extension was made to the language itself in order to incorporate transactions.

7.1.1 Name-To-Object Binding

NewS binds names to objects by following a set of search rules, namely, first looking in local data frames and then searching the UNIX file system via a search list of directories. The NewS search list contains a user's working data directory along with other directories that contain NewS system objects. The user can add to, or change the order of, directories in the search list. GemStone has an ordered list of dictionaries for organizing objects and resolving their names.

In order to combine object access in different places (files and GemStone) smoothly, we added to NewS the capability of allowing a name of a GemStone dictionary to be used within the NewS search list, along with the UNIX directory names. The search path that NewS takes to look for an object will then be able to include any arbitrary combination of GemStone dictionaries and UNIX directories. This approach gives a user access to arbitrary

GemStone dictionaries while preserving the object search paradigm of UNIX-based NewS. (For the rest of the dissertation, we designate the original NewS as "UNIX-based NewS" when contrasted to our platform, GemStone-based NewS.)

7.1.2 Delegating Operations to GemStone

With GemStone-based NewS, objects can be stored in GemStone, and we speculate that in certain cases it is advantageous to perform some of the operations on the objects within the database. For example, a database system usually provides efficient support for access to large data including indexing and incremental loading of objects. Therefore, it is potentially more efficient to extract the desired part of a large object inside the database (if the object is stored there), and to move only the selected part to NewS, thereby reducing I/O and transfer cost.

In order for NewS to execute operations in GemStone, a mechanism is necessary for selectively replacing NewS expressions or their subparts with GemStone operations. Most NewS expressions are represented as nested function calls. Hence, we chose an individual function call as the unit of action delegated from NewS to GemStone and added the ability to replace any NewS function call with the activation of an appropriate GemStone method. Once a function call is dispatched to GemStone, all the work is done inside the database until the result is obtained and returned to NewS. GemStone does not initiate communication with NewS.

With this approach, it is possible to have the same functionality both in NewS and in GemStone, so that the corresponding operation can be executed in either place depending

on where the data are. The following mechanism was added to NewS to determine where to execute a function.

GemStone-based NewS decides the location of function execution by examining the types (NewS object or GemStone object) and sizes of the actual arguments before they are evaluated. Arguments to function calls can be literal constants, variables, or arbitrary expressions. In case of expressions, GemStone-based NewS estimates the type and size of the result before evaluation. When an argument is a function call with only a NewS implementation, the result is assumed to be a NewS object. Similarly, a function call with only a GemStone implementation represents one GemStone object. A call to a function with both NewS and GemStone implementations is treated recursively and its actual arguments are examined in the same way. The size of the object returned from a function call is unknown before evaluation, so a fixed value is used as an estimate. When arguments to a function call are of mixed types, it is necessary to convert some arguments from the original type to equivalent representation of the other type to execute a function in one place. The sum of the sizes for each type is considered as the estimated cost of such argument conversion, e.g., the total sizes of file argument objects is an estimated cost for converting them to equivalent GemStone objects. The version of a function that requires the least amount of data conversion is then called. This particular mechanism is applied to various examples and compared to other possible algorithms elsewhere [Ken91].

7.2 Design of a GemStone Database for NewS

This section discusses the design of a GemStone database for NewS. We describe representation of both NewS objects and operations. We also discuss the kinds of operations that could benefit from being executed in GemStone, and list potential candidates for database execution.

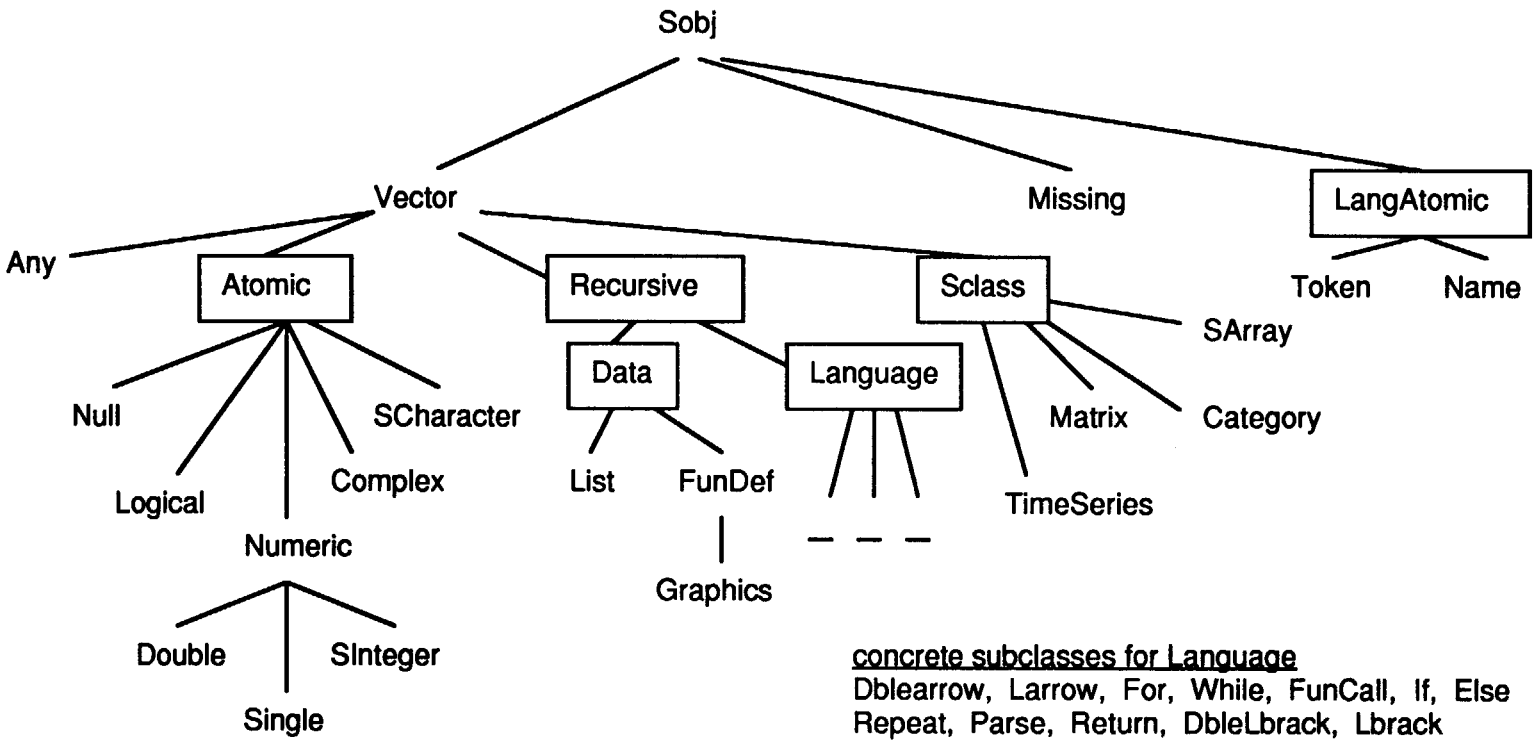
7.2.1 Representation of NewS Objects

NewS currently uses a single C structure called `vector` to represent a vector of any mode (note that NewS simple objects and a C structure that implements them have the same name, "vector") or an object of any class. For representing NewS objects in GemStone, we simply created a mirror image of `vector` as a GemStone class. This approach is possible since GemStone is able to represent recursive C-like structures directly. All the fields of the C structure `vector` that store persistent values are incorporated in the `Vector` class in GemStone either as methods for computing the values or instance variables with access and update methods. NewS objects of most modes and classes are stored in GemStone as instances of the `Vector` class or its subclasses.

This baseline approach was chosen to minimize differences between the C structure for NewS objects and its GemStone representation so as to decrease conversion cost between them. NewS is an interactive language, and an object must be brought into memory from persistent data storage as a user references it. No preloading of objects can be incorporated into such interactive data access, and it is crucial to minimize data transfer cost, including conversion cost between C and GemStone representations.

Figure 7.2 shows the GemStone class hierarchy for NewS objects. In general, a subclass of the `Vector` class was created for each distinctive NewS mode or class so that the NewS evaluator can distinguish the NewS mode or class of an object simply by looking at its GemStone class. With this design, it is also possible to have different implementations for different NewS modes or classes. Subclasses of the `Atomic` abstract class represent atomic vectors of various modes, i.e., `null`, `logical`, `double`, `single`, `integer`, `complex`, and `character`. The `Data` class under the `Recursive` class represents objects of recursive mode, including `list`. The `Sclass` class and its subclasses are for various predefined classes in NewS such as `time-series` and `array`. Other classes illustrated there, including subclasses of the `Language` class and the `LangAtomic` class, represent system objects and their modes, e.g., `Token` for an atomic language token and `For` for a recursive language token representing iteration. Those classes are used for storing language constructs such as expressions as data. Note that function definitions are stored in GemStone as lexical structures. They can be retrieved to NewS and parsed and executed by the NewS evaluator, but they cannot be executed in GemStone. In contrast, NewS operations delegated to GemStone for execution are implemented as OPAL methods as described in the next section.

Note that various GemStone abstract classes shown in Figure 7.2 do not have instances; they are used to represent common concepts or characteristics among different NewS modes or classes. Note also that the particular hierarchy shown in the figure was designed after close examination of NewS source code to determine how objects of each mode and class are actually implemented. For example, vectors of the mode `any` or objects of various predefined NewS classes are actually represented by the same C structure as atomic vectors. Therefore,



concrete subclasses for Language
Dblearrow, Larrow, For, While, FunCall, If, Else
Repeat, Parse, Return, DbleLbrack, Lbrack
Lbrace, Argument, Comment, CommentExpr
FlexCall, Internal, Compiled, Frame, Lpar

LEGEND

<class name>	*: abstract class*
<class name>	*: concrete class*

Figure 7.2

the `Vector` class was defined first in GemStone, and other GemStone classes such as `Atomic` (for atomic vectors), `Any` and `Sclass` were derived as its subclasses, inheriting the structure and behavior of `Vector`.

All the NewS objects stored in GemStone, namely, all the instances of the classes shown in Figure 7.2, are currently stored in one GemStone dictionary called `S`.

7.2.2 Representation of Operations

NewS and GemStone adopt different computation models. NewS is a functional language. The basic unit of computation is a function that may take objects of different modes and classes as arguments. In contrast, GemStone models behaviors as methods of particular classes that the receiving object belongs to. We considered the following two alternative representations of NewS functions in GemStone.

The first approach is to preserve the GemStone model, and implement NewS generic functions as a collection of OPAL methods for all possible arguments. For many functions, one of their arguments can be identified as the primary data object, and whenever such a function is called, a message is sent to its primary data object along with other argument values. For example, a function call,

```
mean(x=sample, trim=1.5),
```

computes mean value of `sample`, which is a vector of numerical values. The argument `trim` represents the fraction of values to be trimmed off each end of `sample`. In this case, `sample` can be chosen as the primary data object. For the function call above, the message below is sent to `sample` in order to obtain an equivalent result.

```
sample meanWithTrim:1.5
```

For some NewS functions, however, it is not clear which argument should be chosen as primary. An example is the `min` function, which takes any number of numeric arguments and returns the single minimum value among all the arguments. In this case, all arguments are equal and there is no obvious reason to choose any one of them as primary.

The second approach to representing a NewS operation in GemStone adopts the NewS functional model and creates a separate GemStone object for each function. Such an object is responsible for implementing the behaviors of the function, either by executing its own methods or sending messages to argument objects. This approach, compared to the first one, adds an extra level of indirection from the NewS point of view. In order to execute a function in GemStone, all NewS must do is to send a message to the corresponding function object. It does not have to know how such a function is actually represented in GemStone, and the implementation can freely change there. For example, the example function call above:

```
mean(x=sample, trim=1.5),
```

can be translated into a message sent to the `mean` function object in GemStone where each keyword specifies an actual argument:

```
mean x:sample trim:1.5.
```

This approach also allows the NewS interpreter to determine whether a certain function has a GemStone implementation simply by querying for a corresponding function object in

the database. In contrast, in the first approach, functions are implemented as a collection of methods in various GemStone classes that correspond to the modes and classes of arguments. Therefore, information on whether or not a certain function is supported in the database is not available in a single place, but spread among several GemStone class-defining objects. Also, with a function object independent of any argument, it is possible to represent NewS functions with no arguments. The first approach cannot support such functions.

Based on the above analysis, we preserved the NewS functional model and adopted the second approach in the design. Figure 7.3 shows the GemStone class hierarchy for function objects. For each NewS function, a subclass of the `Sfun` class is created and the behavior of the function is implemented as its sole method. Such a subclass has only one instance, which performs the operations of the corresponding function. As mentioned before, a GemStone function object can either execute the operations itself or send a message to the argument objects to accomplish the task. The latter preserves the modular approach to implementation promoted by an object-oriented paradigm and will probably make maintenance of the code more manageable than the former.

There are various differences between NewS functions and GemStone methods, besides what is described above, that require careful mapping. For example, some arguments of NewS functions have default values, and a function may be called without explicitly giving a value to such arguments. The function above, `mean`, can be called with or without a value for `trim`, namely, both the forms `mean(x=sample)` and `mean(x=sample, trim=1.5)` are possible. In the former case, `trim` assumes the default value of 0. Therefore, the class of the `mean` function object, i.e., the `Mean` class, must provide messages for both argument

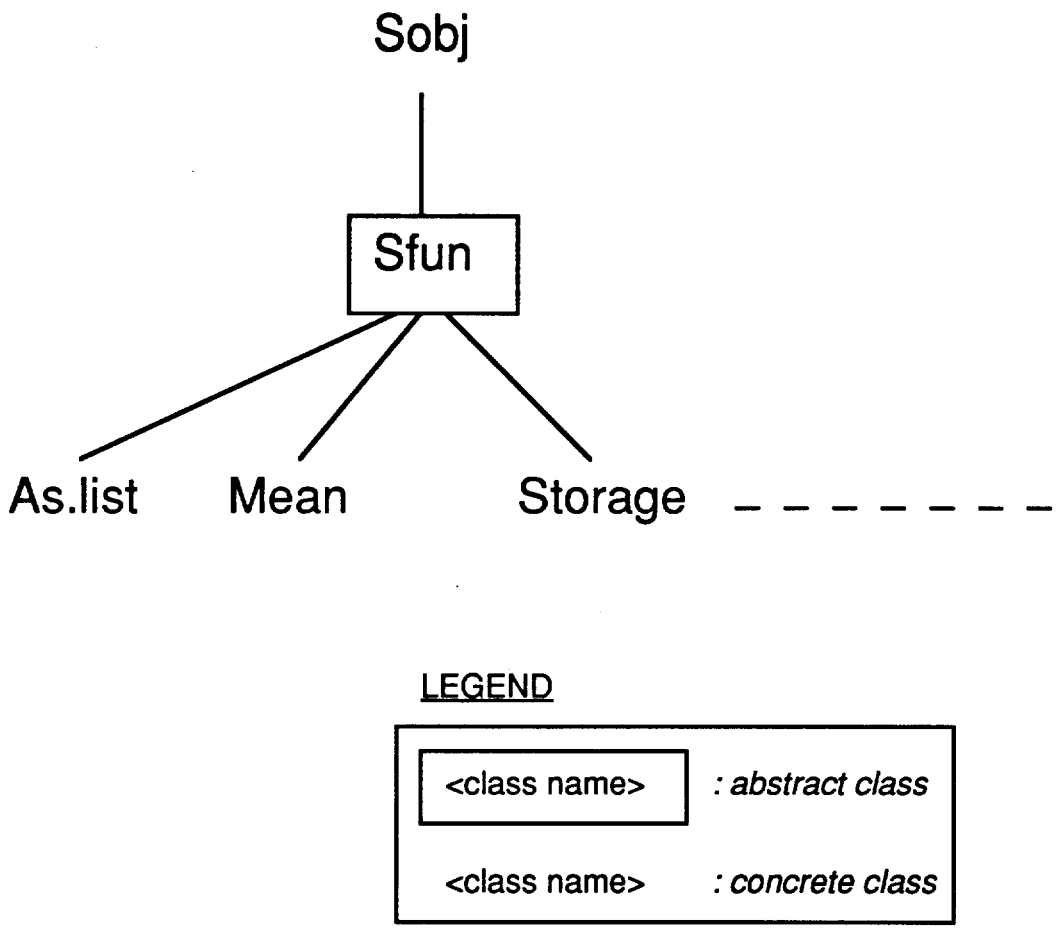

Sobj
Sfun
As.list
Mean
Storage
LEGEND
<class name>
: abstract class
<class name>
: concrete class

Figure 7.3

patterns, i.e., `mean x:sample` and `mean x:sample trim:1.5`. Another solution is to define only one method `x:trim:` for the `Mean` class, and to provide a special object denoting a "missing" argument that the `x:trim:` method replaces with the appropriate default value. Certain arguments in NewS function definition are denoted with the token "...", and may be given a variable number of of argument values. One way to handle such a collection of arguments is to group them into one array object and to pass it from NewS to GemStone as a unit in all cases. Various features of NewS functions and how they affect the GemStone representation are discussed in detail elsewhere [Ken91, BCW88].

7.2.3 Choice of NewS Functions Stored and Executed In the Database

With functions stored in GemStone, the NewS evaluator can delegate function execution to the database. In choosing the functions for the database in our first prototype, we looked at each function in isolation and examined whether the function is suitable for execution in GemStone or NewS. This section first discusses what kind of operations potentially benefit from execution in NewS and GemStone, respectively, and selects candidate functions from NewS to be stored and executed in GemStone. Since each NewS function is represented by a corresponding GemStone function object, it is easy to add or to delete functions in GemStone to experiment with different criteria for choosing functions stored in the database. Note also that even if a certain function is available in GemStone, it is not always the version executed. The NewS evaluator still has a choice of the original NewS version.

Functions in NewS

Below are our observations on implementing and executing functions in NewS.

- Operations that deal with I/O functions for the user interface should be performed in applications. GemStone v.2.0 does not provide user interface capability (though it is scheduled to be added in later releases).

- NewS specializes in statistical and graphical operations, giving them a better support than OPAL. Many predefined functions are readily available in the NewS library for such operations. Using a NewS version of those functions would save time for re-implementing the function in GemStone, and likely yield more efficient performance than executing a GemStone version.

- When a data item is stored in GemStone as an object, there is a certain storage overhead (e.g., entries in the Object Table), since a GemStone object is internally represented by a logical identifier, separate from its state. If a function creates many temporary objects during execution, such storage overhead may cancel benefits of GemStone storage management. In such a case, it could be better to execute a function in NewS even if the input data are originally stored in the database.

Functions in GemStone

Similarly, the following are observations on storing and executing functions in GemStone.

- When functions are stored in GemStone, it is possible for multiple tools to share them. This capability is a nice feature if we extend the architecture of GemStone-based NewS, having NewS communicate and share data and functions with other tools

that manipulate similar kinds of data. Among different kinds of functions, functions such as generic data access would be shared by more tools than application-specific data analysis functions.

- It is possible to perform operations on data in GemStone in order to reduce overhead of data transfer from GemStone to NewS. For example, when a part of a dataset stored in GemStone is needed, selection can be performed in GemStone, and only a desired subset is brought into NewS. In contrast, with subset selection implemented as a function in the current NewS, the whole data item must be brought into memory first. This observation applies not only to subset selection but also to any NewS function that produces an output expected to be smaller than its input, e.g., `min` and `sum`.

- A database has a better strategy for space management than most applications, so it is usually better to manipulate very large data items in GemStone. The previous chapter described a tree representation of large arrays in GemStone that allows incremental loading of the elements. In contrast, current NewS does not deal with large datasets gracefully, and it often crashes when users try to read them in. Yet, NewS users often want to perform operations on large data items. Therefore, it makes computation on large data more manageable to handle large data as much as possible inside GemStone. Those operations that only require part of the data at a time especially can benefit from incremental loading in GemStone.

- Certain predefined classes in GemStone, e.g., Array, are useful for representing NewS

data directly. Associated with such classes are basic access and update methods as well as certain arithmetic and logical operations. Therefore, GemStone versions of those NewS functions that handle data access and update, simple arithmetic, aggregate, logical operations are relatively easy to implement in OPAL using predefined methods.

Candidate NewS Functions to be Stored and Executed in GemStone

Based on the discussions presented above, we considered the following NewS functions as possible candidates for GemStone implementation.

- Arithmetic, aggregate, and logical functions. Arithmetic functions are one of

  $+$, $-$, $*$, $/$, Λ (exponent), %/% (integer divide), or %% (mod).

  Aggregate functions include

  `min`, `max`, `mean`, `median`, `range`, `sum`, and `prod`.

  NewS logical functions are such operations as

  & (binary AND), | (binary OR), ! (not), `xor`, `ifelse` (conditional data selection), `all` (AND of all the elements of all the arguments), `any` (OR of all the elements of all the arguments).

The functions above are easily implementable using existing OPAL methods with little or no additional programming. They are also efficient to execute in GemStone. Since those functions operate on vectors and only need part of the data at a time, GemStone's localized access in large vectors would be potentially advantageous.

- General data access and update functions. They include functions that access attributes of NewS classes such as

  - `attr`, `attributes`, `length`, `mode`, `names`
  - `col`, `row`, `ncol`, `nrow`, `dim`, `dimnames` (matrices or arrays)
  - `levels` (category data)
  - `tsp`, `start`, `end`, `frequency` (time-series data)

  These functions extract specific attributes actually needed in computation, so it reduces the amount of data transferred to NewS to execute them in GemStone. The `subscript` function, `[ ]`, which selects a subset of data elements, is also considered for the same reason. These functions are generic and widely used for different kinds of analysis. It would be useful to include them in GemStone, allowing multiple tools to access them.

Among the candidates discussed above, we actually stored and executed in GemStone those functions used in the sample applications selected for the experiments. They are described later in the dissertation.

7.3 Specifications of the First Prototype

This section describes various specifics of the first prototype of GemStone-based NewS constructed for the study. Both NewS and GemStone are commercial products; NewS is provided by the statistical department of AT&T Bell Laboratories, and GemStone is provided by Servio Corporation. The June 89 Release of NewS and GemStone version 2.0

were used for the first prototype. Since both NewS and GemStone are running on the same workstation, a DECstation 2100, NewS was linked directly into the Gem process that manages a GemStone session for efficient communication between the two systems. Because the Gem and NewS processes are linked, though, the physical size of the GemStone-based NewS executable is significantly larger than UNIX-based NewS: 3.9 Mbytes versus 2.3 Mbytes. Effects of the size of an executable on its performance is not well-documented except for the large overhead in initial loading, but they certainly make direct performance comparisons between UNIX-based NewS and GemStone-based NewS a little difficult. This point is revisited when the performance characteristics of GemStone-based NewS are discussed in Chapter 9. Note that both NewS and GemStone have more recent versions than the ones available in our study. Later, we discuss the possible effects of using the new versions of GemStone and NewS in GemStone-based NewS, in particular performance changes.

7.4 Flexibility As Design Priority

For the first prototype of GemStone-based NewS, we put priority on flexibility of the overall design rather than performance optimality so that the architecture can be easily modified to examine a large number of design alternatives. Design alternatives we considered include different GemStone representations of NewS objects, optimization of NewS functions, loading partial objects, such as certain attributes or components only, from GemStone to NewS, different mechanisms for deciding where an operation is executed, and different choices of NewS objects stored in GemStone.

We adopted an "object-oriented approach" in the design of GemStone-based NewS to

encapsulate details of NewS and GemStone from each other and to facilitate easy, independent modifications of each system. The NewS evaluator does not make any assumption about what is in the database, and it asks GemStone for all the necessary information through a well-defined interface. On the GemStone side, objects respond to the request from NewS and deliver an appropriate answer through the same interface. GemStone does not know anything about the NewS evaluator except for the information communicated through the interface.

Since NewS is implemented mostly in C, in particular the part for persistent data I/O, the GemStone-C application interface (GCI) was used for communication between NewS and GemStone. GCI has two options for database access. One option allows NewS to access and to update the internal representation of GemStone objects directly, somewhat violating encapsulation at the object level. The other option preserves encapsulation, only allowing access to GemStone objects by sending messages to them. Due to the emphasis on a flexible design, the latter approach was taken, that is, GemStone object encapsulation was respected.

With this design, the representation of each system can be modified without affecting the other as long as the same interface between NewS and GemStone is maintained. This feature facilitates easy execution of many of the experiments discussed in later chapters, since they require modifications either in GemStone or in NewS, but such modifications do not propagate beyond the boundary of each system. Among examples of experiments listed above, different GemStone representations of NewS objects and optimization of NewS functions can be tested with modifications inside GemStone only. Different mechanisms for

delegating operations from NewS to GemStone can be tested by modifying NewS without affecting representation on the GemStone side.

One piece of information NewS needs to know is the kind of objects and operations stored in GemStone. NewS does not hard-code such information, and simply inquires of GemStone whether a certain data object or function is stored in the database as needed. Therefore, to add a GemStone version of a certain NewS function, a corresponding object for the function needs to be created in GemStone, but no modification is necessary on the NewS evaluator. For example, if only desired attributes of objects are to be brought into NewS from GemStone, one only needs an implementation of the `attr` function in GemStone. NewS notices that both the object and `attr` are in GemStone, and extracts the desired attributes inside GemStone and brings only those attributes into NewS. If, on the other hand, no attribute access function is stored in GemStone, then the object is brought into NewS and attribute extraction is executed there. In order to see if it is beneficial to execute attribute access in GemStone, the appropriate functions can be simply added or deleted (or renamed), observing the differences.

Neither NewS nor GemStone "remembers" information obtained from the other through the GCI, so it is possible to reorganize one system even during the same execution without triggering changes in the other.

We also minimized the amount of information exchanged between GemStone and NewS. For example, when an operation is executed inside GemStone, an OOP of the result object is returned to NewS. Only when the NewS evaluator decides that the state of that object is needed in NewS memory does the evaluator load the actual state of the object into NewS.

7.5 Summary

This chapter described various issues considered in designing and implementing GemStone-based NewS. Many decisions were made according to the criteria in Chapter 4; a priority was placed on acceptable performance and productivity of resulting architecture as an experimental platform. In some situations, however, those two criteria were in conflict. For productive experimentation, GemStone-based NewS should provide flexibility and a wide coverage of the design space. Modular design and encapsulation together facilitate easy modification of various parts of architecture; modifications can often be made locally without affecting the whole architecture. On the other hand, such a "modular" architecture and the information hiding that goes with it sometimes imposes a performance penalty due to communication overhead between modules.

When in conflict, flexibility was favored over performance. Had performance been given absolute priority, the resulting architecture would likely be very rigid and difficult to modify. As a result, even if excellent performance were obtained from GemStone-based NewS, it would be very difficult to experiment with multiple design alternatives on the platform, defeating much of its purpose. On the other hand, if GemStone-based NewS is left flexible, there are still possibilities for improving its performance, e.g., by changing hardware and software components used. The modular architecture facilitates an easy switch to new, improved versions of GemStone or NewS in the future.

As for cost-effectiveness, we tried to reuse existing capabilities of both GemStone and NewS as much as possible in designing and constructing the platform, keeping additions,

modifications minimal.

For the rest of the dissertation, we describe possible design variations we investigated and the experiments we performed to evaluate the design against sample NewS applications. We also evaluate GemStone-based NewS as an experimental platform through such investigation.

Chapter 8

Design Alternatives in the GemStone-NewS Interface

Earlier we discussed requirements for scientific data management and advantages offered by object-oriented databases over traditional approaches such as files or relational databases. Compared to traditional data models, an object-oriented data model simplifies schema design by allowing direct representation of scientific models. However, the flexibility of the object-oriented model does create many alternatives to compare in designing the schema. There are also different ways applications can make use of the support provided by object-oriented databases.

This chapter explores possible alternatives in designing GemStone support for NewS applications. We describe possible design dimensions in creating the GemStone schema and in modifying the current NewS source. We discuss various alternatives within each dimension, and how such alternatives can be examined on GemStone-based NewS experimentally. For running NewS application on the platform, some optimizations are possible in evaluating NewS expressions. We discuss such optimization strategies and how we can experiment with them with changes in GemStone only. It is also possible to change the kind of objects and functions stored in GemStone to adjust the platform to individual cases.

In some cases, we actually conducted simple experiments to illustrate the point being

made, and the results from such experiments are presented as numbers or graphs. As this is the first chapter to present our experimental results, we specify the computing environment and the way we conducted the experiments below. In all figures presenting the experimental results, lines were drawn using the smoothing technique for scatter plots based on robust locally weighted regression, which is available in the NewS library.

8.1 Computing Environment for the Experiments

All the experiments described in this dissertation were conducted on a DECstation 2100 running the Ultrix operating system version 4.1 with 16.78 Mbytes of real memory and one each of a 104 Mbyte DEC RZ23 disk and a 600 Mbyte CDC Wren V disk. To make the environment relatively uniform, we ran all the test cases on the system when no other user processes were running and the system was not being backed up. A system can exhibit slow performance initially since all the necessary data are to be fetched from disk into memory. The performance can also vary due to changes in system load. To eliminate such short-time fluctuations of performance, we executed the test cases several times before we started measurement, and repeated the tests by 10 times to obtain their average. (When the test cases took longer than 15 minutes, we took an average of 5 repetitions.)

8.2 GemStone Schema

As mentioned before, a flexible object-oriented data model yields many design alternatives to consider in schema design. This section explores possibilities in representing NewS objects in GemStone, i.e., different representations for their states and methods.

8.2.1 Different State Representations

As described in Chapter 7, we designed the initial GemStone representation of NewS objects as a "mirror" image of the NewS internal representation for efficient data transfer between NewS and GemStone. However, it is possible to customize the representation to accommodate needs of a particular NewS mode or class. For example, if NewS frequently requests specific information from GemStone, it would be advantageous to have it available by a single call, with a method that delivers that information directly. The following experiment shows an example case for such customization of the representation and the resulting benefits.

We created a database of lists; the number of components in the lists ranges from 25 to 1000. In the current architecture of GemStone-based NewS, NewS lists are stored in GemStone as parallel arrays of values and names, held in instance variables **data** and **names**. To look up a component by name, the value of the **name** instance variable is linearly searched for the position of the name, which is used as an index into the **data** array. (Component access by name is implemented in this manner in current UNIX-based NewS.) As an alternative, frequently accessed components can be held directly in the top-most part of the object under an instance variable of the same name, as shown in Figure 8.1. In the experiment, pairs of lists were created with identical component values. One of the pair uses the current implementation. The other, in addition to **data** and **names**, has the value of the last component replicated under the instance variable of its name. Figure 8.2 compares the performance of GemStone-based NewS in accessing the last component of the lists using

List without caching

data : (1, 2, 3, ...)

names : ("a", "b", "c", ...)

List with caching

data : (1, 2, 3, ...)

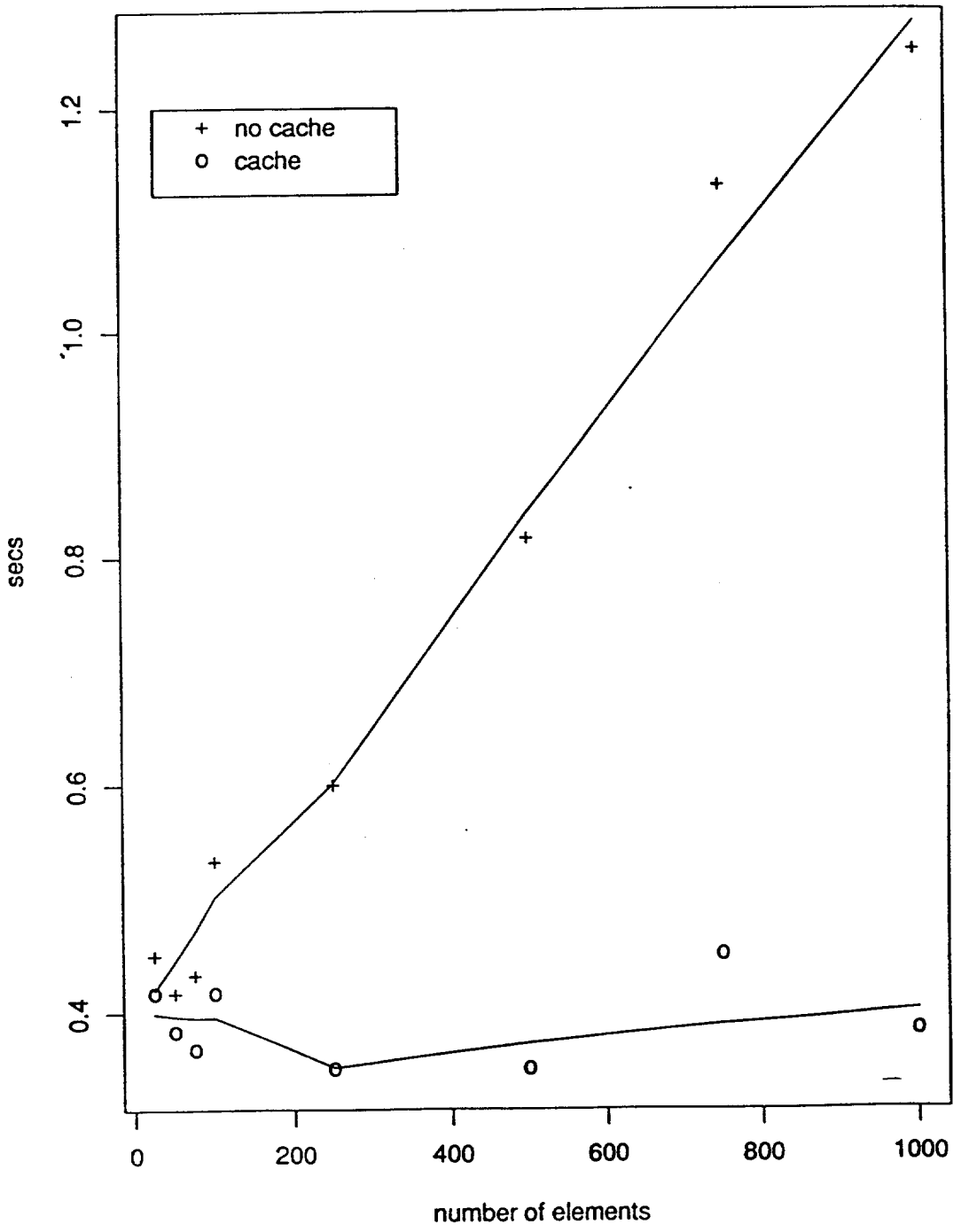
names : ("a", "b", "c", ...)

a : 1

c : 3

Figure 8.1

Figure 8.2 Caching Element in State



the two representations. As the number of components increases, the execution time for the lists without caching increases. This behavior is to be expected, since the entire array for component names must be searched before the position of the last name is found, and the number of component names increases with the size of the lists. In contrast, the execution time of those list with caching remains about the same, i.e., around 0.4 second, since the accessed component is readily available as a value of a top-level instance variable.

Another possible modification for efficient search of a particular component is to sort component names in `names` to make a binary search possible, more efficient than a linear search in the original design. Note, however, that if sorting and binary search are to be used, we actually need to store (name-position) pairs instead of name strings in `names` to retain the original position of component names. In the original design, components of `names` and `data` assume the same order, and the position of the name is used to access the corresponding value in `data`.

Besides structural variations of object state such as above, we can also try different encodings of values. Some encodings render more compact representation than others, resulting in fewer pages to read for data retrieval and therefore more efficient performance.

For example, in GemStone, floating-point numbers are stored as objects, and a mapping between their object identifiers (OOP's) and locations of values are provided through an Object Table. In contrast, integers are stored directly as sequences of bytes, and no dereferencing is necessary to obtain their values. Therefore, floating-point numbers not only require more space, e.g., entries in an Object Table, but also extra indirection through their OOP's to get their values. The following simple experiment tried to assess the effect of the

size differences of representation.

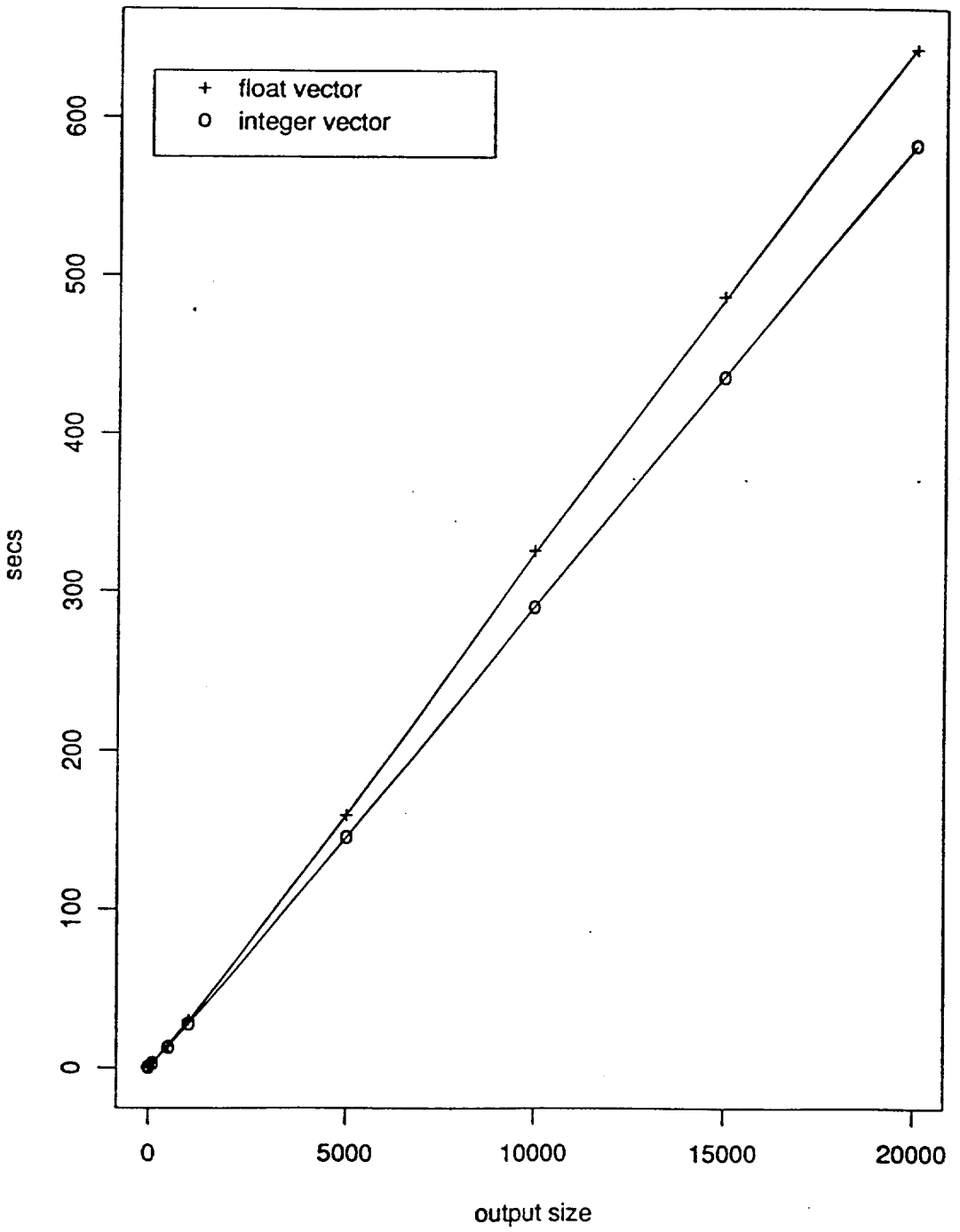
One of the vectors from National Health and Nutrition Examination Surveys (NHANES) data, an example NewS data we experimented with and described in detail in the next chapter, is `sbp`, a collection of blood pressure values. The `sbp` is stored as a vector of floating-point numbers even though the data values are actually integers, since the elements are expected to be converted to floating-point numbers in regression analysis. We created a vector `intsbp` identical to `sbp` except that the data values are integers. Subsets of `sbp` and `intsbp` of various sizes were accessed. As shown in Figure 8.3, somewhat surprisingly, no large gain is observed with `intsbp` over `sbp`. Only about 9% of execution time is saved for the largest extracted subset of 20000 elements.

Since GemStone supports a byte object whose value is encoded in bytes, it is possible to encode floating-point values directly in an array instead of using a GemStone floating-point object. Though not supported in GemStone, the concept of structured value without identity would also be useful in representing hierarchical structure of NewS objects without indirection through OOP's at each level.

Defining appropriate indices for the representation is another way to make object access more efficient. Current GemStone only provides indices for non-sequenceable collections (sets and bags). As a NewS object is essentially an ordered collection that allows for value-based access, indices for sequenceable collections would be a useful addition for them.

In trying out different representations for a particular NewS mode or class, we can evolve GemStone schema to examine one representation at a time. It is also possible to define them in GemStone as a set of subclasses of a common superclass. In that case,

Figure 8.3 Float Vector vs. Integer Vector



the superclass defines properties common to that NewS mode or class. For example, if we want to try caching various components of an object of the mode `list`, we can inherit the corresponding GemStone class `List` whose state does not cache any components, and create subclasses with additional instance variables for cached components. Each subclass incorporates methods specific to cached components, e.g., access and update operations, but general operations for the `List` class are still available. If different representations actually assume very different states or operations, their superclass would be mostly "empty" as the states or operations must be defined separately for each subclass. Even so, it is still advantageous to have a common GemStone superclass for all the variations rather than to represent them as completely disjoint classes as the NewS evaluator can check their common mode or class simply by looking up their superclass.

With different GemStone classes for different representations, we need to specify a particular class for an appropriate representation when an object is created. We would make all the representations for the mode or class visible through a NewS session. That design would change NewS semantics, as NewS users would now be faced with a set of representation variations for a mode or class. We might encode what representation to choose as "rules" in the NewS evaluator, but it would be difficult to define a complete set of such rules that covers all possible cases. In our experiments, we created objects of different representations (subclasses) directly in GemStone.

Once objects are created, their mode or class can be identified simply by checking their membership of certain GemStone classes. Objects of all the different representations for the same mode or class can be treated uniformly by the NewS evaluator as they belong

to a common GemStone superclass, and we can compare them in the same operational environment.

8.2.2 Alternative OPAL Methods

With encapsulation at the object level, the code for a method can be varied for the same message. Therefore, we can try various strategies at the method level to make execution of GemStone-based NewS suitable for individual cases. For some NewS functions, certain optimizations are possible. For example, in processing NewS logical functions such as `all` and `or`, it is not always necessary to evaluate all the arguments. For `all`, which performs "logical-and" of all the elements of all the arguments, if one of the elements turns out to be FALSE, the result is also FALSE and other elements can be left unevaluated.

Some strategies are designed to accommodate large datasets. For example, when a large array is created, we can preallocate a certain amount of space instead of allocating space element-by-element. Savings can also result from elimination of unnecessary copying of objects as performed by current NewS. Instead of eagerly copying objects when update may be possible, we can adopt the "copy-on-write" strategy, i.e., only copy those parts of the objects modified.

As with different representations of NewS objects, the most obvious way to compare alternative programs for a method is to test one program at a time, altering the code after each test. While it is not possible to compare different possibilities side by side with this approach, we can reuse the same test expressions for all the different programs as they respond to the same message. Alternatively, for each different program we want to

examine, we can inherit a class (and the name of the method) and define its subclass with a different definition for the method. The NewS mode of the objects is still recognized by their common superclass, but the actual code executed for the sent message depends on what subclass they actually belong to.

8.3 Changes to NewS Source

With GemStone as persistent data storage in addition to files, NewS as an application must be modified to incorporate communication with GemStone. There are various design alternatives we must consider in such modifications. This section explores some of such alternatives. Below we discuss different mechanisms for processing function calls, different GCI calls made to accomplish the same communication, caching of various information on GemStone objects in NewS, and different ways of incorporating GemStone capabilities into NewS.

8.3.1 Different Mechanisms For Delegation of Operations to GemStone

In the previous chapter, we described a mechanism we incorporated into NewS for delegating operations to GemStone. In choosing the delegation mechanism, we actually considered two alternatives. Both mechanisms examine unevaluated actual arguments to a function call. In one scheme, the number of arguments of each location type (GemStone or NewS) is counted; the other scheme estimates the total size of arguments of each location type. The quantities are used to estimate the cost when arguments are moved to the other location, and the version of the operation with the smaller expected conversion cost is chosen. Our

investigation showed that the scheme that considers the size of the arguments works better in general, so we incorporated that scheme into the platform.

A variety of other mechanisms are also possible. Current GemStone-based NewS examines unevaluated arguments. Therefore, when arguments are expressions, the result size must be estimated. Though a fixed value is currently assigned to the size of the returned object in such a case, more sophisticated strategies for size estimation are possible. It is also possible to consider not just the cost for moving objects but the total execution cost. We can also take into account not only the arguments but also where a result of a function call will be used in subsequent computation.

As with any changes to the NewS evaluator, it requires modification of the source code and recompilation of NewS to incorporate such schemes as above into the evaluator. Though current GemStone-based NewS makes a decision at the beginning of an evaluation cycle by looking at unevaluated arguments, different schemes require a decision-making process to take place in different places. For example, if we are to examine the size of evaluated arguments, then we need to identify the point in an evaluation cycle where all the arguments have been evaluated to modify the evaluator appropriately. Since expressions are typically nested, recursion is used to implement expression evaluation, so for those schemes requiring the evaluator to make decisions in the middle of an evaluation cycle, we must make sure that a decision is made at the right moment in the recursive iterations.

8.3.2 Different GCI (GemStone C Interface) Calls

There are a variety of GCI functions available in GemStone v.2.0 for communication between NewS and GemStone. Using different functions, we can experiment with different mechanisms for communication. For example, we compared two different granularities for communicating the same information between NewS and GemStone as follows. In one case, multiple OPAL statements were executed by a single GCI call, given to a GCI function `GciExecuteStr` as a single string. In the other case, each statement was executed by a separate call, resulting in as many GCI function calls as the number of the statements.

We investigated the difference between the two cases above by executing the following OPAL program from a C program through GCI.

```
a := Array New.
a add: 1.
a add: 2.
a add: 3.
...
a add: 100.
```

In the first case, an array of 100 integer elements was created using one GCI call, putting all the above OPAL statements into one large string. In the second case, the same operation was executed using multiple GCI function calls: one call for array creation plus a call to add each element, a total of 101 calls. The execution time of the first case was 14.96 seconds, whereas the second case 16.96 seconds, so there was about a 12 % savings for the first case

over the second case.

Another example of different communication mechanisms possible with GCI is pre-compilation of OPAL statements into a method in advance. If, as above, an OPAL statement is passed as a string to `GciExecuteStr`, it is first parsed and compiled by the OPAL interpreter before execution. A compiled method is represented as an object in GemStone, so such a compilation process actually invokes object creation. Alternatively, the operation can be pre-compiled as a method in GemStone to avoid an overhead of parsing and compiling. Therefore, by executing the same message using these alternative mechanisms, it is possible to estimate overhead of parsing and compiling. The array-building example above was again used for experimentation. As mentioned before, when the collection of statements was given as one string to `GciExecuteStr`, the execution took 14.96 seconds. With the same statements already compiled as a method and executed by calling `GciPerform`, the execution time was 12.56 seconds. Hence about 16 % of the time was saved for not having to parse and compile the string first. In general, it is beneficial to define a method for the operation frequently performed or the information often asked in advance so that a desired result is obtained by a single GCI function call.

We said earlier that communication between NewS and GemStone is handled only through message passing in the first prototype, in order to maintain encapsulation between the two systems. However, structural access to GemStone objects is also possible through the GCI. The structural interface tends to deliver efficient performance compared to message passing, but the state of objects is directly accessed and updated by NewS and encapsulation at the object level is broken in GemStone. Hence, switching from message

passing to structural access would make it harder to experiment with different representations of GemStone objects as such differences would be visible to NewS and the evaluator must be modified to access each representation.

In the experiments above, instead of the NewS evaluator, we used a simple C program whose only task was to make specific GCI calls to examine each design dimension, e.g., a granularity of communication, in isolation. The NewS evaluator makes a series of GCI function calls over an evaluation cycle. In current GemStone-based NewS, the evaluator issues a separate GCI function call whenever a specific piece of information is needed from GemStone. The obtained information is immediately used and discarded. As the kind of information needed for evaluation is fixed, e.g., classes of arguments, the GCI call strategy can be varied by changing when and how each piece of information is obtained from GemStone during an evaluation cycle. It is relatively easy to change the communication mechanism while maintaining the same positions of calls, as it simply requires replacing one set of GCI calls with another in those positions. It is more tricky to change the positions of the calls, as we must make sure all the information is available before it is needed. For example, if we want to consolidate multiple GCI calls made in different places into a single call to obtain all the necessary information at once, it is likely some pieces of information are obtained in a different place from where they are actually used. Therefore, the evaluator must "remember" such information until it is actually used, i.e., a diversion from the current GCI call strategy.

To see how different communication schemes would affect the behavior of the platform, we would also need to examine dynamic patterns of communication between GemStone and

NewS. Those sequences of GCI function calls made frequently by the evaluator are a good candidate for testing different communication mechanisms for potential optimization, as improving them would have a large effect on the overall performance of the platform. We gave GemStone-based NewS the capability to monitor the GCI function calls made by the evaluator. We used an array-creating example in the experiments above because monitoring GCI calls issued by the evaluator revealed that such operations are performed often.

In most cases, the change in the GCI call strategy only requires modification of the NewS evaluator and does affect GemStone. However, some of the communication mechanisms discussed above require that certain methods exist on the GemStone side in advance, e.g., encoding of frequently-asked information as a method. Also, as mentioned above, structural access breaks encapsulation of GemStone objects, and changes in object representation would require modification of the evaluator.

8.3.3 Caching GemStone Information in NewS

GemStone-based NewS needs to know about various properties of GemStone objects, e.g., their classes, in order to process them properly. As mentioned before, currently, once such information is obtained from GemStone by the evaluator, it is used and immediately discarded. Consequently, the NewS evaluator may request the same information over and over again during a single session. For example, when a data item named `foo` is accessed, the NewS evaluator first checks its location. When `foo` is located in GemStone, its OOP is returned to the evaluator. The evaluator then asks for `foo`'s class by sending the message `class` to the obtained OOP. However, the NewS evaluator does not remember `foo`'s class,

and every time `foo` is referenced, the evaluator repeats the same communication process. It is therefore possible for GemStone-based NewS to cache certain information in order to reduce the communication cost. As with any caching strategy, the effectiveness of this scheme depends on identification of information that is repeatedly used, therefore worth being cached. We can examine profiles of interaction between GemStone and NewS to identify candidates for caching.

To incorporate the caching strategy as described above, the evaluator must be modified to look for the needed information in its cache first before issuing a GCI call. It is also necessary to maintain consistency between cached information and actual state of GemStone. However, most information sought by the NewS evaluator remains stable during a session, e.g., classes of objects, OID's of class definition objects and existence of GemStone implementation for NewS functions, which would simplify consistency maintenance.

8.3.4 Interface to Existing GemStone Facilities

GemStone provides many built-in database facilities such as explicit locking, indexing, and transaction management. Just as current NewS provides a mechanism to execute commands in the underlying operating system from within a session, it is possible to provide an interface in NewS to access built-in capabilities of GemStone to see if such features benefit existing applications. Incorporating the transaction and concurrency control mechanism of GemStone into NewS would extend features of NewS as a persistent language. Some possible mechanisms have been considered on how a transaction management may be introduced into NewS, as summarized below.

Currently, UNIX-based NewS provides a limited form of atomicity. It will not write the results of any assignments within a top-level expression to a file unless the whole expression evaluates without error. Similarly, when processing a file of NewS commands, none of the assignments are permanent unless all the expressions in the file are evaluated successfully. When NewS writes to a file, it actually writes to a temporary file, and when the enclosing expression completes successfully, it renames the file to the actual object name. However, no concurrency control is provided along with such "transactions". Multiple NewS users can overwrite each other's updates without knowledge, and the last writer wins.

One approach for incorporating the GemStone transaction mechanism into NewS is to preserve the atomic behavior of UNIX-based NewS as much as possible and to execute GemStone commits (ending one transaction, and starting another) at the same places that NewS would normally write to the file system. With this scheme, when multiple users access the same GemStone objects concurrently, certain expressions may not execute due to a conflict, but the users will know of a conflict due to a failure of commit.

Another approach is to let users control transactions explicitly. Since GemStone provides the transaction and concurrency control capabilities as predefined methods, this scheme can be easily implemented by providing a set of NewS callable functions that execute those methods through the GCI. This way, a user can control the length of each transaction explicitly. This approach of user-controlled transactions appears to be useful to many NewS applications and users, and is scheduled for the next version (after August 91 Release) of NewS.

Note that GemStone facilities are directly available only on NewS objects stored in GemStone. It is of course possible to transfer any NewS objects to GemStone if the GemStone facilities prove to be useful for them.

As with transactions and concurrency control, all GemStone features are accessible through the GCI, and they can be incorporated into NewS by making appropriate GCI function calls to initiate desired actions in GemStone. We can have the NewS evaluator make the calls, or create separate NewS functions for them.

8.4 Optimized Evaluation of NewS Expressions

In considering possible optimizations in evaluating NewS expressions, we turned to those strategies for relational databases as there are certain similarities between relational queries and NewS expressions. For example, NewS objects are actually collections of elements, but many NewS functions operate on an object as a single entity rather than on an element-by-element basis, just as relational queries treat a relation as a whole. Examples of optimizations used in relational databases are "pushing down" or applying selection or projection as soon as possible to minimize the size of data for further processing and reordering arguments to groups of binary operators such as joins to reduce the size of intermediate results.

One way to apply "pushing down selection" to NewS is as follows. NewS has the `subscript` function for extracting a subset. As shown in Figure 8.4, when `subscript` is combined with iteration functions such as `lapply` that apply a function `f` to each element of input data `D`, it is possible to apply `subscript` to `D` first to reduce the number of elements in `D` to which `f` is applied. More generally, if a function and `subscript` commute as shown

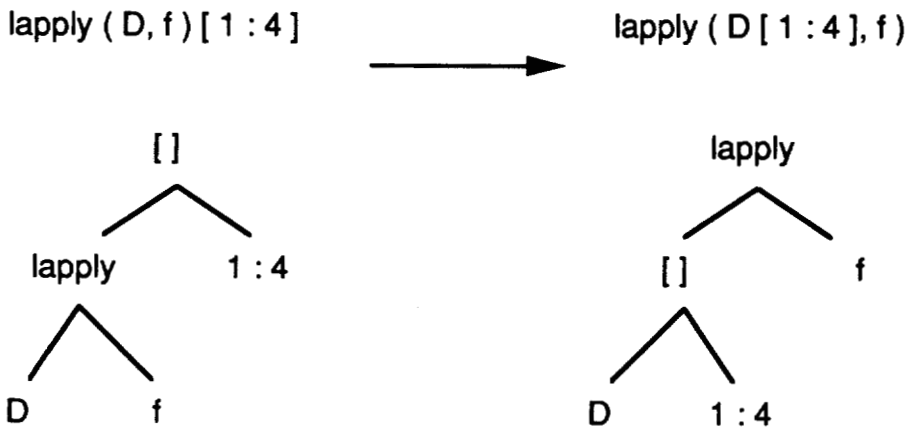


Figure 8.4

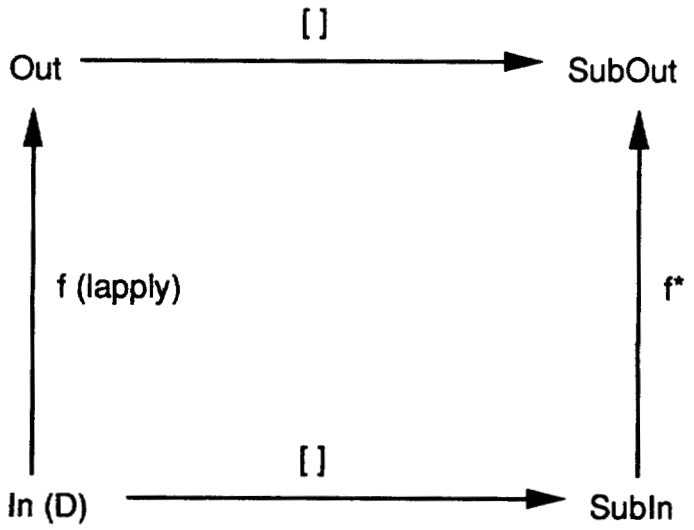


Figure 8.5

in Figure 8.5, it is possible to "push down" and apply **subscript** before the function.

Another example of a possible NewS optimization, also involving selection, is to merge successive selections into a single operation to eliminate costly creation of intermediate results. A series of calls to the **subscript** function, such as

$(a[1000:2000])[200:500]$

\# Select the 1000th–2000th elements of "a" first followed by

\# further extraction of the 200th–500th elements from the intermediate result.

can be combined into a single call:

$a[1200:1500]$

One example of where such successive subset extractions could occur is with the following function *foo* that contains a subset extraction, i.e.,

$> foo \leftarrow function(x, ...)\{$

$+$ .....

$+ y \leftarrow x[2:500]$

$+$ .....

$\}$

and when *foo* is called with an actual argument that is itself a subset extraction, e.g., $foo(a[400:1400], ...)$. In such a case, the expression in the function body, $y \leftarrow x[2:500]$, is instantiated to $y \leftarrow (a[400:1400])[2:500]$, and subset extraction is to be applied to a

twice successively. (We used this example of successive subset extractions in experimenting with optimization of expression evaluation on the platform as described below.)

There are also certain differences between NewS and relational databases that make some relational optimizations inapplicable in NewS. For example, with NewS objects being ordered collections, that order is utilized in certain operations. Hence, for such operations, optimizations that involve reordering of the data are not applicable. However, not all NewS functions require preservation of the order, e.g., for the `mean` function, the order of the elements is irrelevant and therefore the order can be freely changed in an optimization process. In any case, we decided to investigate database-like optimizations applicable to NewS on the platform.

In general, database-style optimizations require examination and modification of expressions as a whole. In the first prototype of GemStone-based NewS, the NewS evaluator interprets expressions and delegates operations to GemStone one function at a time. Hence, it might seem that fundamental changes in the NewS evaluator would be required, to pass whole expressions to GemStone, in order to experiment with optimizations in GemStone. However, without changing the basic architecture of the platform, we were actually able to examine the effect of optimization performed by GemStone via modifications to GemStone objects and classes as described below.

When individual function calls are delegated to GemStone, in some cases the successive function calls are accumulated to reconstruct the expressions on the GemStone side instead of immediately being evaluated. This "lazy" evaluation scheme makes whole expressions

available for GemStone to optimize. Note that GemStone dynamically reconstructs expressions as actually evaluated, so it can sometimes find optimizable cases that are not obvious from static inspection of the source code. For example, in the case of successive subscripts in a function call above, i.e., $foo(a[400 : 1400], ...)$, an optimizable case is hard to detect from static inspection since we cannot tell what an actual argument x would be to foo. We must also trace actual execution of a function to identify that $x[2 : 500]$ in the body of foo actually yields optimizable successive subscripts $(a[400 : 1400])[2 : 500]$ as those two subscripts are temporally apart with intervening expressions possibly operating on other datasets.

We conducted the following optimization experiment to illustrate the expression reconstruction technique above using the example of successive `subscript`'s described previously. Two functions, `subrange` and `materialize`, were defined and given GemStone implementations. The `subrange` function is much like a simplified version of the `subscript` function, except that it does not immediately compute the result. It instead creates a GemStone object that represents the requested subset extraction and stores information necessary for carrying out the operation later. We will call such an object a `delayed operation object`. An argument to `subrange` can be a `delayed operation object` created by another `subrange`, in which case a new `delayed operation object` is created that represents multiple subset extractions. The `materialize` function demands an actual result, thereby initiating computation. With a `delayed operation object` as its argument, the function tries to perform optimization over the represented operation before evaluation. Therefore,

the expression *materialize*(*subrange*(...)) actually evaluates and returns an extracted subset, whereas *subrange*(...) just creates a **delayed operation object**, returning its OOP to the NewS evaluator.

In the experiment, we examined the execution time of the following two expressions,

$$materialize(subrange(subrange(< vector >, < begin1 >, < end1 >),$$
$$< begin2 >, < end2 >))$$

and

$$materialize(subrange(materialize(subrange(< vector >,$$
$$< begin1 >, < end1 >)), < begin2 >, < end2 >))$$

In the first expression, two **subrange** operations result in a **delayed operation object** for the whole operation. An intermediate result from the first **subrange** is not created as execution of subset extraction is delayed with a **delayed operation object** passed from the first **subrange** to the second **subrange**. The **materialize** function recognizes two successive applications of subset extraction and optimizes them into one subset extraction before evaluating the result.

In the second expression, each subset extraction is evaluated separately, actually creating an intermediate result.

The **begin1** and **end1** determine the size of extracted subset from the first **subrange**; the values are given in the experiments so that 90% of an input vector is accessed here. The **begin2** and **end2** are given to form a final subset from the second **subrange** with 10

elements in the middle of an input vector. The size of input is varied from 200 to 17500. As shown in Figure 8.6, savings from this optimization increase as the size of an input vector increases. In the optimized case, the intermediate result from the first **subrange** is not computed, saving much time, especially for large input vectors.

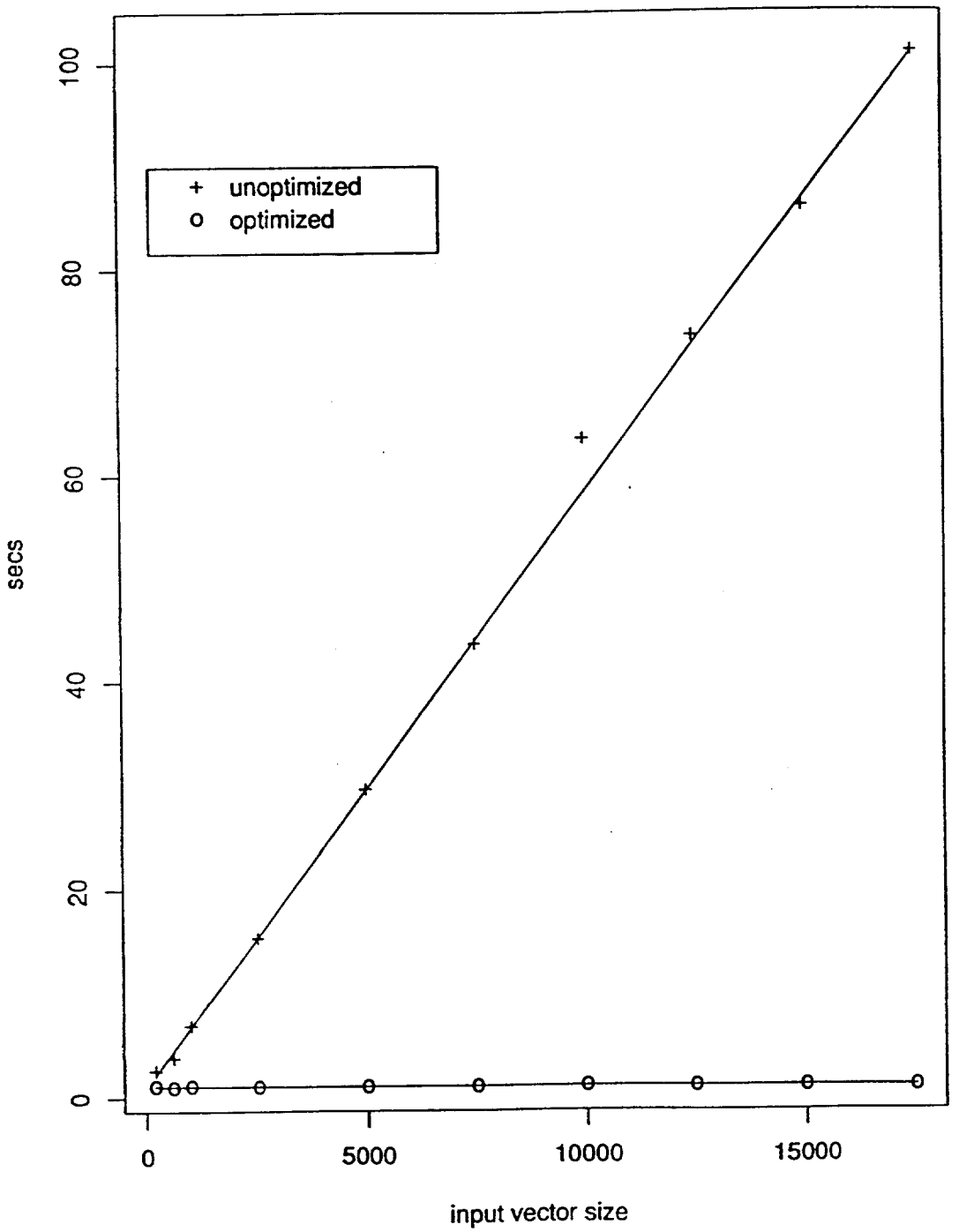
In order to examine possible optimizations on GemStone-based NewS, we can incorporate lazy evaluation into those functions that are possibly included in optimizable expressions such as `subrange` above. Each such function produces its own `delayed operation object` that represents a corresponding operation. The `materialize` function above only performed specific optimization for successive subset extractions, but it can be generalized to handle different kinds of `delayed operation objects` and to perform optimizations appropriate for each case. Therefore, we can examine different expressions for possible optimization by adjusting what functions are lazily evaluated. Static optimization, on the other hand, would require inspection and modification of the source code.

8.5 Different Collections of Objects and Functions Stored in GemStone

Chapter 7 discussed candidate functions to execute in GemStone. However, we expect the optimal selection of functions for GemStone execution to vary from application to application. For the experiments we performed, those expressions actually tested determined which functions to store in GemStone.

The collection of objects in the database can also be changed according to various criteria. Generally, large objects shared among different users and applications are good

Figure 8.6 Optimized Successive Subrange's



candidates for GemStone storage. On the other hand, file storage may be better for objects that must be accessed efficiently during a session but do not survive between sessions. Also, as with functions, the selection of objects will likely depend on the application.

Since we preserved the object access paradigm of current NewS, objects can be created or destroyed in GemStone the same way as in files. We just have to adjust the entries of the NewS search list to incorporate GemStone dictionaries where object creation or removal should occur. GemStone classes for NewS objects were created for the platform as described in Chapter 7, and the evaluator creates a GemStone representation of an object by instantiating those classes through the GCI and inserts it into a specified GemStone dictionary. When an object is removed with the `rm` function, the evaluator looks up its name in the dictionary to see if the object exists in GemStone. (Any discrepancy in naming convention between NewS and GemStone is resolved by the evaluator automatically.) If the object is found in the database, it is removed from the dictionary.

As for functions, we represented each function with a separate GemStone class that has a single instance for the function. Therefore, to store a function in GemStone, we first create a GemStone class for it by inheriting the `Sfun` class. We described in Chapter 7 various issues that need to be considered in representing NewS functions in GemStone. For example, most NewS functions are generic that accept arguments of different modes and classes. For such generic functions, we had corresponding GemStone function classes identify the NewS class or mode of the argument and send an appropriate message to the argument. Therefore, actual operations are carried out by the methods defined in the GemStone classes of the arguments. Such "modular" implementation, made possible by the

object-oriented paradigm, facilitated efficient implementation of GemStone representation for NewS functions. For missing arguments that are to be replaced by default values, we created a special object `Missing` to denote the missing value, which a GemStone function object recognizes and replaces with an appropriate value. When a NewS function accepts a variable number of arguments, such arguments are put into one array object so that a GemStone function object can expect a single object for that set of arguments in all cases.

Once a GemStone class for a particular function was created, we instantiated it directly in GemStone to create a GemStone implementation of the function. Such an instance was inserted into the GemStone dictionary specified by the search list. As mentioned before, functions are represented as NewS objects and handled the same way as data. Extending the uniform paradigm to GemStone, a function object is looked up by the evaluator in the dictionary just as a data object. The removal of the function object from GemStone is also handled the same way as data objects.

One of the experiments we can perform by changing objects and functions in GemStone is to vary the amount of data transfer from GemStone to NewS for possible performance improvement. For example, NewS functions such as `subscript` for subset extraction, `$` for component access, `mean` for computation of the mean value all produce the result smaller than the input. Therefore, when the data are in GemStone, it would be potentially advantageous to have GemStone implementation of such functions and to execute them inside GemStone to reduce the size of the result transferred to NewS. In the next chapter, we will examine the effect of such reduction of data transfer with sample NewS data and applications.

8.6 Summary

This chapter examined various design dimensions in providing GemStone support for NewS applications. Dimensions examined include the following.

- classes and methods created for storage of NewS objects and functions
- different mechanisms for delegation of operations to GemStone
- different GCI calls
- caching of GemStone information in NewS
- interface to GemStone facilities
- optimized evaluation of NewS operations
- which objects and functions are stored in GemStone

We explored alternatives in each dimension, and explained how such alternatives can be examined on GemStone-based NewS experimentally.

Alternative GemStone schemas for NewS objects and functions can be examined using inheritance. Inheritance is especially effective when the alternatives are variations of a common base design since their base design is defined once in the common superclass and inherited by all the subclasses. We experimented with alternative structures of state by caching an element of a list at a top level and comparing different representations of integers and floating-point numbers, effectively using inheritance in both cases.

Changes to NewS source included modifying the current NewS source code, i.e, the NewS evaluator and existing functions, to find better ways to communicate with GemStone as well as adding new functions to introduce GemStone capabilities into NewS. We examined some of the alternatives provided by the GCI, e.g., different granularities of statements communicated by a single call and precompilation vs. postcompilation of statements. Experiments that investigated different mechanisms for delegating operations to GemStone and introduction of GemStone concurrency control into NewS are reported in detail elsewhere [Ken91].

It is possible to examine optimization strategies for evaluating NewS expressions with changes to GemStone only. In our experiment using an example of successive subscript's, GemStone function objects, potentially part of optimizable expressions, accumulated function calls to reconstruct expressions to be optimized.

Since we preserved the object access paradigm of current NewS as much as possible, data objects can be created or removed in GemStone simply by inserting an appropriate GemStone dictionary into the search list. Storing a function in GemStone involves more efforts since a GemStone class for the function must be created and instantiated in GemStone first. Once created, though, a function can be looked up and removed from the GemStone dictionary transparently, just as data objects. The next chapter describes our experiments with various scientific data, with associated access functions stored and executed in a database.

As this study is an initial attempt at investigating scientific data management by means of an experimental platform, the architectural variations and experimental means for investigating them discussed in this chapter would pave a way to the next stage of the investigation,

i.e., more extensive experimentation.

The next chapter describes the experiments we conducted with sample NewS applications. We store and execute various data and functions in GemStone in each case, and examine the effect. We also discuss further our experience in exploring design alternatives on GemStone-based NewS when we evaluate ease of experimentation in Chapter 10.

Chapter 9

Experiments with Health Surveys Data and Protein Knowledge Base

This chapter describes the experiments we conducted with example NewS data and operations, i.e., National Health And Nutrition Examination Surveys (NHANES) data and the Protein Knowledge Base (PKB). The NHANES data are various data items obtained from national health surveys on blood pressure, e.g., blood pressure values and age of sampled subjects. Each data item is represented as a large vector of numerical values. The data are to be analyzed to identify the correlation between blood pressure and such factors as age and income.

PKB is an application for protein structural analysis. Each protein is represented as an object, with operations such as component access, statistical analysis and structural display implemented as library functions.

Working with real NewS data presented us an opportunity to examine representation of scientific data as objects, e.g., their size and performance for data access, as well as to try some of the design alternatives discussed in the previous chapter on the real data. We were interested in NHANES data since the data exposed weakness of the storage management in current NewS. Current NewS tends to copy data whenever there is a possibility of update. The NHANES data vectors have often crashed the system due to such a copying scheme,

and seemed a good example to test support for large data in GemStone against. With NHANES, we could also examine one of the most common data structures in scientific applications, i.e., an ordered collection of numerical values. As with many other NewS data, multiple applications are applied to NHANES data, e.g., NewS and SAS. Therefore, it would be advantageous to store NHANES data in a database to accommodate access from different applications in a single place, eliminating explicit data conversion between different formats by users.

PKB provided us with an opportunity to examine protein data and operations within an object-oriented framework. Proteins with their hierarchical structure cannot be directly represented in traditional systems such as files and relational databases, resulting in awkward data manipulation. In contrast, an object-oriented database is capable of representing protein models directly. As protein research is statistical analysis of common characteristics among a large volume of data, design of an adequate database is critical for its success. And recent development such as the human genome project has started producing the large amount of protein data at a rapid rate, making a need for a database even more urgent.

NHANES data and PKB data together also presented very different data types to examine in our experiments, i.e., a large flat vector of numbers and deeply-nested tree structure, both common in scientific applications. Unlike NHANES data, PKB is a complete application that includes not only data but also a set of library functions. As PKB has been actively used in state-of-the-art research on protein structure, we were able to examine the dynamic characteristics of protein structural analysis such as typical data access patterns and the time frame of the analysis, and to interpret the performance of GemStone-based

NewS within a context of typical PKB analysis.

The rest of this chapter describes the two applications, a set of operations selected for the experiments, and the results of the experiments and their analysis.

9.1 NHANES Data

The NHANES data are combined results from three national health surveys conducted by the National Center for Health Statistics in order to examine trends in blood pressure. Those surveys are Health Examination Survey I (HES I) conducted in 1960, National Health And Nutrition Examination Survey I (NHANES I) conducted around 1970, and NHANES II conducted around 1974. The following subsections describe data vectors, selected operations for the experiments, and analysis of the experimental results.

9.1.1 Data Vectors

The NHANES data are a collection of 20 items including sex, age, blood pressure and treatment status from 22975 sampled subjects. Values of all the items are encoded in numeric values and stored in a table of 22975 rows by 20 columns. Each row is a sampled subject and a column represents a specific item from the subjects. An atomic vector of 22975 numeric values is created from each column so that correlations between columns may be studied in later analysis. Therefore, the database consists of 20 atomic vectors of 22975 numeric values, each vector named after the item it represents. Brief description on each vector is given below.

1. `survey`: 1 indicates subjects from HES I, 2 is NHANES I, 3 is NHANES II.

2. **sex**: 0 indicates female subjects, 1 is male subjects.

3. **sbp**: systolic blood pressure.

4. **dbp**: diastolic blood pressure.

5. **sbprx**: **sbp** value if treated; otherwise missing.

6. **dbprx**: **dbp** value if treated; otherwise missing.

7. **sbpnorx**: **sbp** value if untreated; otherwise missing.

8. **dbpnorx**: **dbp** value if untreated; otherwise missing.

9. **age1**: (actual age) - 50.

10. **age2**: (**age1**)**2.

11. **bmi**: BMI (Body Mass Index) - 26.

12. **educ**: 0 indicates subjects who didn't finish high school, 1 is those who finished high school.

13. **income**: 0 indicates subjects at the middle or low income level, 1 indicates high income level.

14. **race**: 0 indicates white subjects, 1 is black subjects.

15. **rx**: 0 indicates subjects whose blood pressure problem has not been treated, 1 is those who have been treated, otherwise missing.

16. **stratum**: sampling stratum.

17. **ppsu**: pseudo-primary sampling unit.

18. **subset1**: the same as **survey**.

19. **subset2**: 2 * **survey** + **sex** - 1. Computed from other vectors to represent a subset to be used in analysis.

20. **id**: serial values (starting with 1) for identifying each subject.

BMI (Body Mass Index) used to compute the 11th column is weight (in kg) divided by height squared (in meters**2). The 17th column (**ppsu**), a pseudo-primary sampling unit, is a technical indicator of how the data should be analyzed if one wants to account properly for the sampling design. The 18th and 19th columns, **subset1** and **subset2**, specify **sex** and **survey** subgroups to be used in further analyses. (Even if **subset1** is identical to **survey**, a separate vector is provided to represent a concept of a specific subgroup separately from the information on surveys.) Missing data are coded as '-99' for reasons related to the planned analyses. Vectors such as **age2** and **subset2** are precomputed from other vectors since they are expected to be used frequently in various computations. Note that such vectors can be stored as an OPAL method on original vectors, if these vectors are stored in GemStone.

9.1.2 Selected Operations for the Experiments

The objective of NHANES data analysis is to find trends in blood pressure, especially with respect to other factors such as age and income. Therefore, in the analysis, we expect frequent access to blood pressure values as well as examination of correlation between blood

pressure and other data items. Hence, we chose access to **sbp**, the data vector for systolic blood pressure, based on index as well as values of other vectors, as a "representative" example for our experiments. Individual test cases are described below. The size of output is listed when a subset of **sbp** is extracted based on element value.

1. $sbp[11487 - i : 11487 + i]$ where $0 <= i <= 500$

2. $sbp[sbp == -99]$: output size 2246

3. $sbp[survey == i]$

   $i = 1$: output size 4181

   $i = 2$: output size 10762

   $i = 3$: output size 8032

4. $mean(sbp[survey == i])$

   $i = 1, 2, 3$: output size 1

5. $sbp[sex == i]$

   $i = 0$: output size 12625

   $i = 1$: output size 10350

6. $mean(sbp[sex == i])$

   $i = 0, 1$: output size 1

7. $sbp[race == i]$

$i = 0$: output size 19909

$i = 1$: output size 3066

8. $mean(sbp[race == i])$

$i = 0, 1$: output size 1

9. $sbp[rx = i]$

$i = 0$: output size 17258

$i = 1$: output size 3471

$i = -99$: output size 2246

10. $mean(sbp[rx == i])$

$i = 0, 1, -99$: output size 1

11. $sbp[subset2 == i]$

$i = 1$: output size 2229

$i = 2$: output size 1952

$i = 3$: output size 6142

$i = 4$: output size 4620

$i = 5$: output size 4254

$i = 6$: output size 3778

12. $mean(sbp[subset2 == i])$

$i = 1, 2, 3, 4, 5, 6$: output size 1

The first case represents direct examination of elements of `sbp` specified by their index. The value of `i` is adjusted between 0 and 500 so that the size of the index range varies from 1 to 1000 around the middle position. The second operation identifies cases of a missing value in `sbp` before the number of such cases is counted in the analysis. The rest of the operations represent preliminary examination of correlation between `sbp` and other data vectors, and each extracts a subset of `sbp` specified by values of others. For example, in evaluating $sbp[survey == 1]$, $survey == 1$ is evaluated first and a logical vector is computed based on the predicate. The logical vector is then used to extract the subset of `sbp` where the logical vector contains TRUE. As indicated in the size of output, extracted subsets vary in size. The `mean` function is also applied to the extracted subsets of `sbp`. Note that in all cases, applying the `mean` function reduces the output size to 1.

We stored all the vectors in the selected expressions, i.e., `sbp`, `survey`, `sex`, `race`, `rx`, `subset2`, in GemStone as objects. All the functions in the expressions above (`[ ]`, `==`, `mean`) were also provided with GemStone implementations. For the vectors in GemStone, the whole expressions were executed in the database, and only the results were brought into NewS. On the other hand, with the vectors in files, the NewS implementation was selected for all the functions.

9.1.3 Analysis of Experimental Results

Figure 9.1, 9.2, and 9.3 summarize the results obtained from running the selected operations on the NHANES data described in the previous section. Figure 9.1 shows the performance

Figure 9.1 GemStone-based NewS vs. UNIX-based NewS

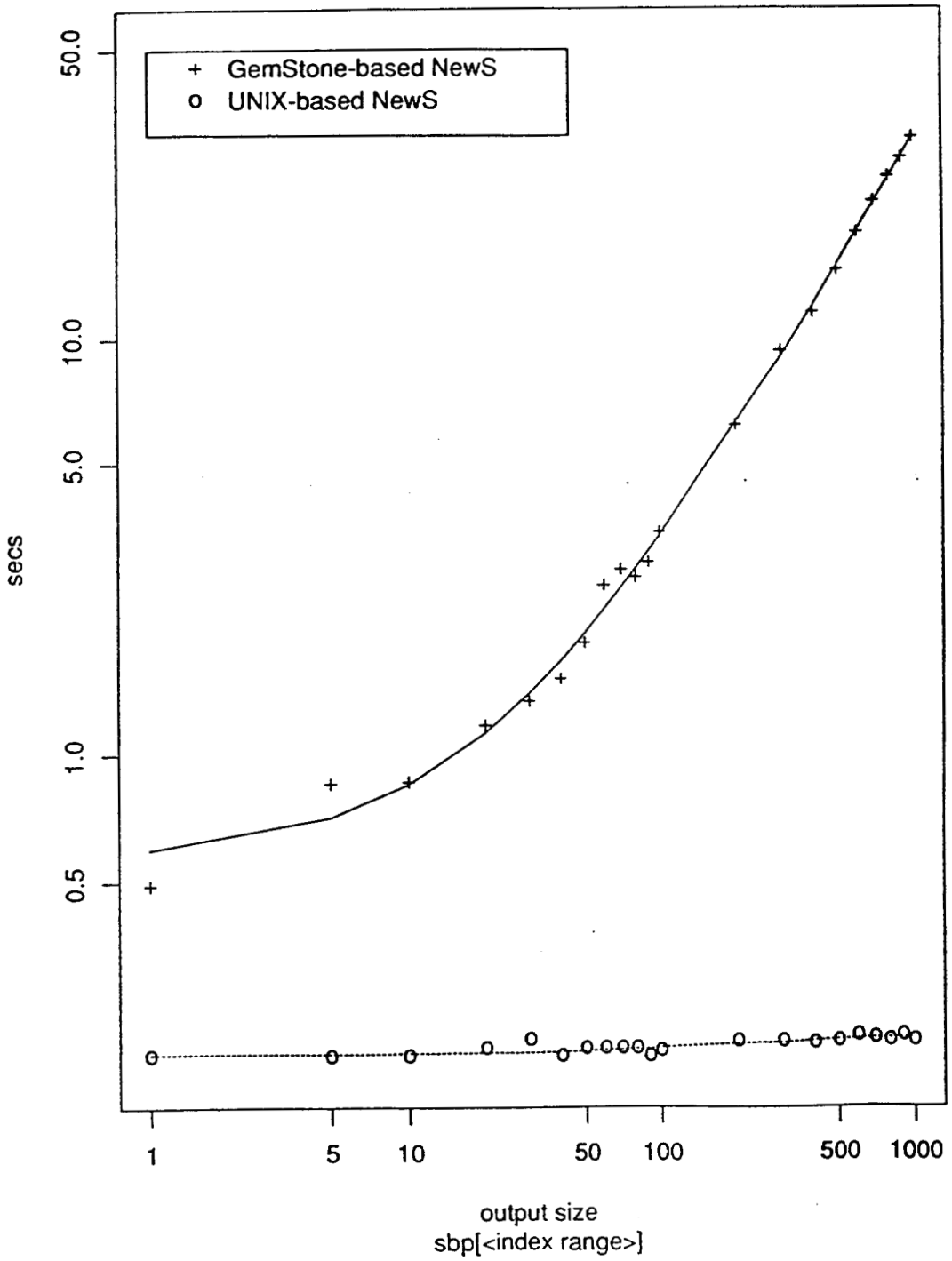


Figure 9.2 GemStone-based NewS

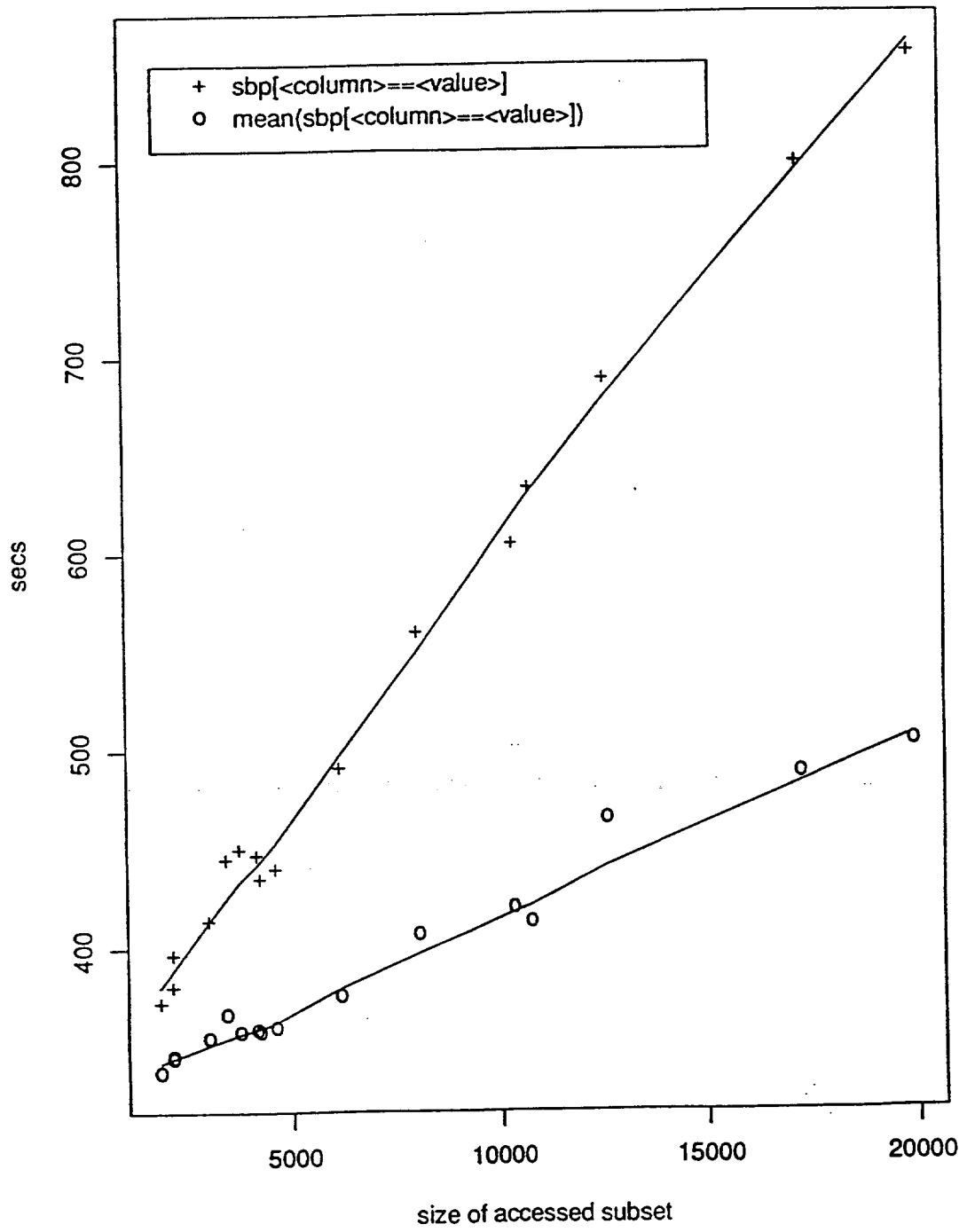
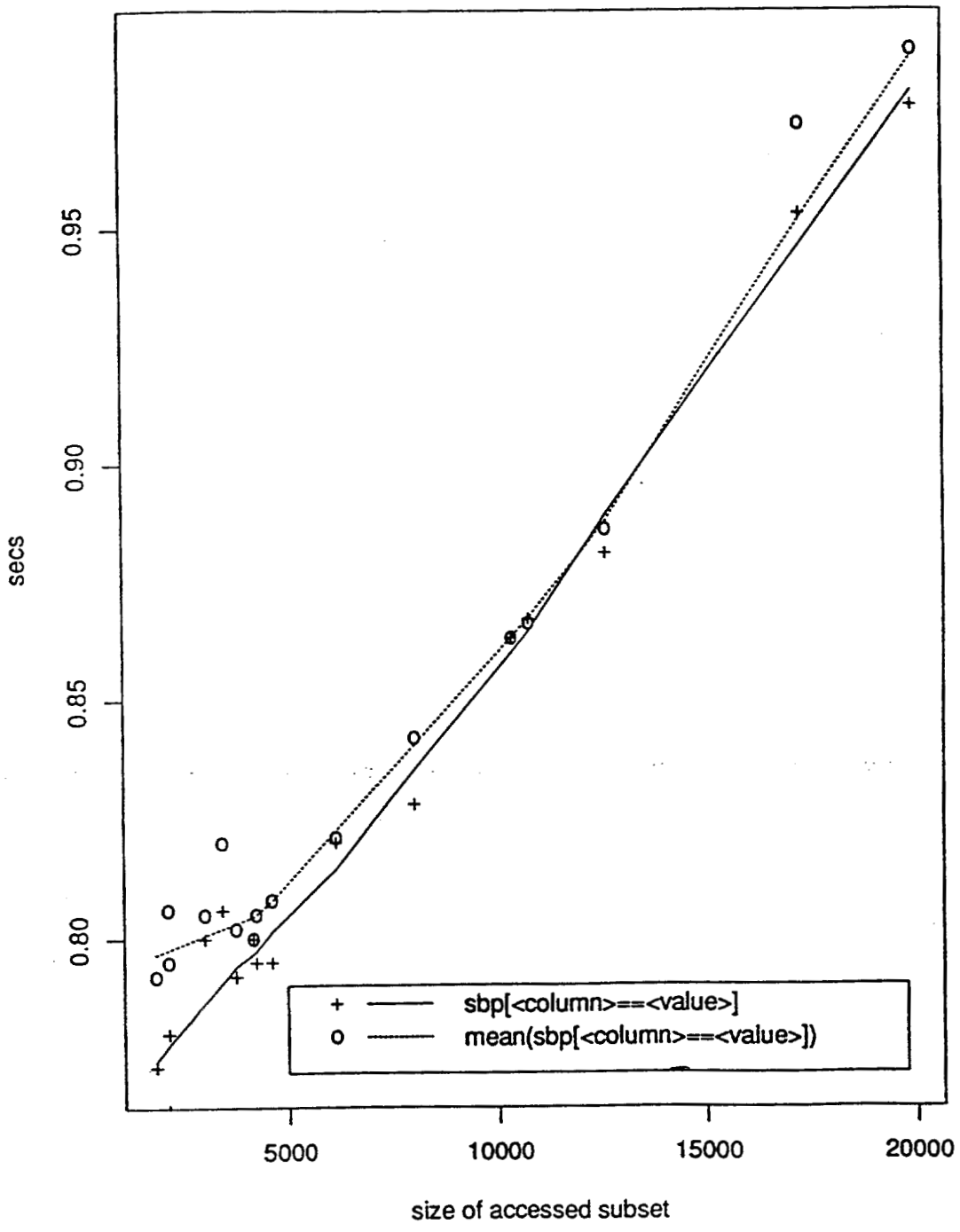


Figure 9.3 UNIX-based NewS



of GemStone-based NewS when a subset of the `sbp` vector is accessed using an index range of various sizes, that is,

$$sbp[11487 - i : 11487 + i] \text{ where } 0 <= i <= 500.$$

The size of an accessed subset varies from 1 to 1000. As the size of the accessed subset increases, the execution time of GemStone-based NewS also increases. In GemStone-based NewS, the `subscript` function has a GemStone implementation and in this case is executed inside GemStone. The OID of the result is then returned from `subscript` to the NewS evaluator. At this point, an implicit `print` function is executed for writing out the content of the result on the screen, which prompts retrieval of all the values in the accessed subset into memory. Therefore, the cost of data transfer increases with the size of the accessed subset, resulting in larger execution time.

As a reference, the performance of UNIX-based NewS for the same operation is also shown in Figure 9.1. There is a large difference between performance of UNIX-based NewS (around 0.2 second on average) and that of GemStone-based NewS (around 8 seconds on average), indicating a large overhead for accessing GemStone objects over file access. UNIX-based NewS also exhibits different patterns of performance from GemStone-based NewS with respect to the size of the accessed subset. Namely, the execution time remains about the same regardless of the size of the accessed subset. In UNIX-based NewS, the entire `sbp` vector is loaded into memory as an actual argument to the `subscript` function. That cost of transferring the argument vector into memory is uniform in all the test cases, dominating the execution time and yielding approximately the same performance throughout.

Figure 9.2 and 9.3 show the results from test cases 2 to 12 described in Section 9.1.2, i.e., extraction of subsets of **sbp** with respect to values of other vectors. Figure 9.2 shows that the performance of GemStone-based NewS is somewhat influenced by output size in computing *sbp*[< *column* >==< *value* >], more than doubling from 373 seconds to 855 seconds as output size grows from 1952 to 19909. Again retrieval of all the values in the accessed subset for an implicit **print** function seems to dominate the execution time.

In contrast, as illustrated in Figure 9.3, the execution time of UNIX-based NewS does not vary much as output size increases, ranging from 0.77 to 0.99 second. In executing *sbp*[< *column* >==< *value* >], two large numeric vectors, **sbp** and another vector for value reference, are loaded into memory in their entirety as arguments to the == function. As with the previous case, transfer of large input vectors executed universally in all the test cases yields approximately the same execution time.

Figure 9.2 and 9.3 also shows the results from calculating the mean values of such extracted subsets. With GemStone-based NewS, in evaluating *mean*(*sbp*[< *column* >== < *value* >]), the OID of the result from **sbp** is returned to the NewS evaluator, which finds that the **mean** function has a GemStone implementation and hands back the returned OID to execute **mean** on the result inside GemStone. The size of the result of the whole expression remains 1 in all cases, and the implicit **print** has to fetch only one value. As shown in Figure 9.2, it reduces execution time to apply the **mean** function to an extracted subset of **sbp** inside GemStone as it reduces the size of the transferred result. In contrast, with UNIX-based NewS, adding calculation of mean values to subset extraction does not change the performance drastically, as shown in Figure 9.3, since transfer of input vectors

still dominates the total execution time.

Storing and executing functions in GemStone was discussed as one of the design alternatives in the previous chapter. Having functions such as `subscript` and `mean` in GemStone was advantageous in the experiments, especially if the size of the result transferred to NewS was reduced. However, compared to file access, we generally observed a large overhead in accessing GemStone objects, i.e., about 1 to 3 orders of magnitude.

We noted before that the size of NHANES data has caused problems with UNIX-based NewS due to over-copying. UNIX-based NewS tends to use up available memory very quickly when large data are handled. With the NHANES vectors stored in GemStone and accessed there in the experiments, we observed much fewer crashes and more stable system activities in general.

9.2 PKB

PKB is an NewS application that combines a database of three-dimensional protein structures with a series of algorithms for pattern recognition, data analysis, and graphics [Bry89]. The following sections describe the data schema and implemented algorithms in the PKB library. We also discuss typical PKB analysis, selected operations for the experiments as likely operations in such analysis, and the results of the experiments.

9.2.1 PKB Data

The PKB database contains one NewS list for each protein registered in the Brookhaven Protein Data Bank. These lists are named according to the identification code of the entry

in the Data Bank, and various information on each protein structure is encoded as a named component of the list. Figure 9.4 shows the deeply-nested hierarchical structure of the PKB protein structure lists. The depth of the list ranges from 3 to 5.

Among various components, PEPTIDE represents a protein's structural data. Proteins consist of amino-acid residues each of which in turn includes a number of atoms. PKB models protein structure somewhat differently, and PEPTIDE has both the ATOM and RESIDUE components; ATOM has a subcomponent pointing to an associated RESIDUE. This organization permits efficient access to the ATOM component. Note that if both ATOM and RESIDUE were represented as GemStone objects with associated OOP's, it would be very easy to represent a logical reference from ATOM to RESIDUE using the OOP of RESIDUE. The HETEROGEN component contains information on groups of atoms outside amino-acid residues. The CRYSTAL component represents various crystallographic data. As X-ray crystallography is used to obtain the structural data in the PEPTIDE component, CRYSTAL is a sort of "meta" data that supplement data values in PEPTIDE.

9.2.2 PKB Functions

PKB extends the NewS library with its own functions for maintaining the database, for querying the database based on sequential and conformational motifs, and for analyzing the data based on a number of different structural parameters. Figure 9.5 shows a complete list of PKB functions and brief description of what each function accomplishes. The PKB functions can be classified into the following seven categories based on the pattern of data access and manipulation that take place. Note that biological semantics of functions are

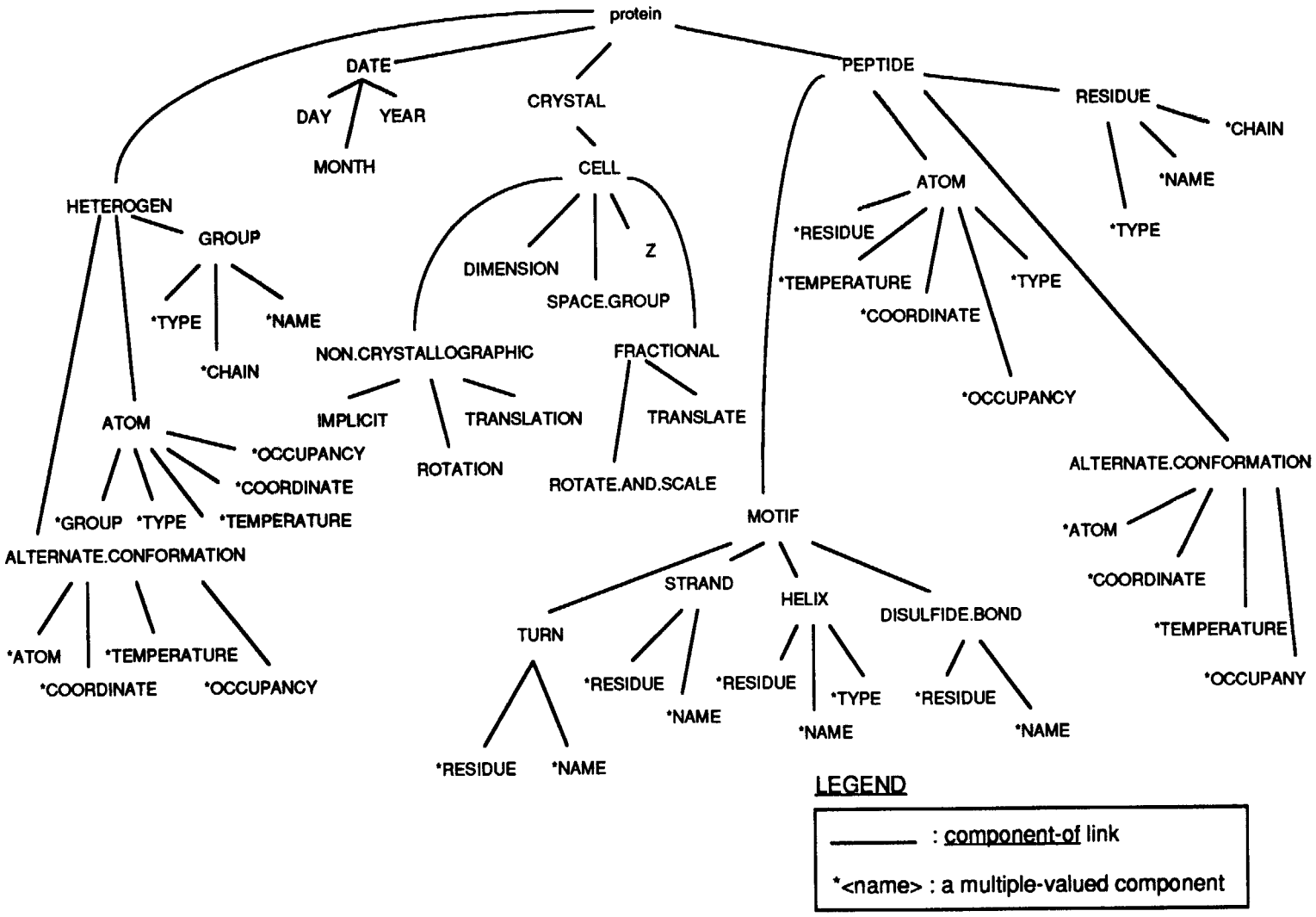


Figure 9.4 PKB Schema

function name	operation description	number of uses in the PKB library
ACXP	Atom contact pairs from coordinates	3
ADDM	Dimer atom distance matrix from coordinates	1
ADXP	Dimer atom contact pairs from coordinates	2
ALPHA.CARBONS	Carbon-alpha coordinates from PKB protein object	4
ANGLE.PLOT	Axes for scatterplot of angular data	1
ATOM.COORDINATE	Polypeptide coordinates from PKB protein object	33
ATOM.RESIDUE	Residue pointers by atom from PKB protein object	28
ATOM.TYPE	Atom type labels from PKB protein object	15
ATOMS	Selected atom coordinates by residue	14
ATOMS.PLOT	Stick drawing of atomic coordinates	1
BDAN	Bond angles from selected atom triplets	2
BDLN	Bond lengths from selected atom pairs	2
BETA.CARBONS	Beta-carbon coordinate matrix	1
BOND.ANGLES	Standard bond angles for PKB protein object	1
BONDS	Standard covalent bonds for PKB protein object	1
BONDS.PLOT	Stick drawing from list of bonds	1
CHAINS	Chain numbers by residue from PKB protein object	13
CONTOUR.MATRIX	Contour plot of distance or contact matrix	1
CTSN	Convert spherical polar to Cartesian coordinates	1

Figure 9.5 (1) PKB Functions

function name	operation description	number of uses in the PKB library
D2F	Convert distance object to symmetric matrix	2
DIAN	Dihedral angles from selected atom quartets	3
DIHEDRAL.ANGLES	Standard torsion angles for PKB protein object	2
DIMER.RESIDUE.CONTACT.MATRIX	Dimer atom contacts by residue pair	1
DISULFIDE.BONDS	Disulfide bonds from PKB protein object	2
FOR.PROTEINS	Iterate, calling a function once per protein	1
GPAN	Bonded atom triplets using a bond angle dictionary	2
GPBD	Bonded atom pairs using a bond dictionary	2
GPTO	Bonded atom quartets using a dihedral dictionary	3
HETEROGEN.ATOM.COORDINATE	Heterogen coordinates from a PKB protein object	2
HETEROGEN.ATOM.GROUP	Heterogen group pointers by heterogen atom	2
HETEROGEN.ATOM.TYPE	Heterogen type labels from a PKB protein object	2
HETEROGEN.BONDS	Bond list for heterogens by distance calculation	1
HETEROGEN.GROUP.CHAIN	Chain labels by heterogen group	2
HETEROGEN.GROUP.NAME	Heterogen group numbers by group	2
HETEROGEN.GROUP.TYPE	Heterogen group types by group	3
HYDROGEN.BONDS	Main-chain hydrogen bonds by distance calculation	1
LDP	Linear distance transformation from coordinates	5
LKAN	Bonded atom triplets using a bond angle dictionary	2

Figure 9.5 (2) PKB Functions

function name	operation description	number of uses in the PKB library
LKBD	Bonded atom pairs using a bond dictionary	1
LKTO	Bonded atom quartets using a dihedral dictionary	1
NACX	Atomic contact count by atom	2
NRCX	Atomic contact count by residue	7
PDB.FILE	Write PDB file from PKB protein object	10
PDBF	Write PDB file from coordinate matrix	3
PDMQ	Partitioned distance matrix query	2
PEPTIDE.BONDS	Main-chain peptide bonds by distance calculation	2
RAMACHANDRAN.PLOT	Ramachandran diagram from PKB protein object	3
RCXM	Atomic contacts by residue pair	1
RDXM	Dimer atomic contacts by residue pair	2
RESIDUE.CHAIN	Polypeptide chain number by residue	1
RESIDUE.CONTACT.MATRIX	Residue contact matrix from PKB protein object	3
RESIDUE.CONTACT.NUMBER	Residue contact number from PKB protein object	1
RESIDUE.NAME	Conventional residue numbers by residue	1
RESIDUE.TYPE	Polypeptide residue type by residue	1
RESIDUES	Polypeptide residue type in standard codes	2
RTAT	Rotate atomic coordinates about x, y, or z	2
SCCN	Side chain centroid from atomic coordinates	3

Figure 9.5 (3) PKB Functions

function name	operation description	number of uses in the PKB library
SECONDARY.STRUCTURE	secondary structure flags by residue	1
SEQUENCE.MOTIF	Instances of a subsequence in PKB protein object	1
SIDE.CHAIN.CENTROID	Side chain centroid from PKB protein object	1
SPLR	Convert Cartesian to spherical polar coordinates	1
SSQ	Instances of subsequence from residue type vector	2
SUPR	Optimal superposition of two coordinate matrices	1
VABS	Lengths of 3-D vectors stored as n by 3 matrix	1
VCRS	Vector products of 3-D vectors as n by 3 matrices	3
VDOT	Scalar products of 3-D vectors as n by 3 matrices	1
VIRTUAL.BETA.CARBONS	Beta-carbon coordinates computed from main chain	1
VIRTUAL.BONDS	Alpha-carbon to alpha-carbon virtual bonds	1
VNRM	Normalize 3-D vectors stored as n by 3 matrix	5

Figure 9.5 (4) PKB Functions

not considered, i.e., functions in each category do not necessarily perform biologically similar computations.

1. Functions that retrieve the data from the database, derive quantities based on them, and call FORTRAN subroutines using derived data as actual parameters. There are 21 such functions:

ACXP	ADDM	ADXP	BDAN	BDLN	CTSN
D2F	DIAN	LDP	NACX	NRCX	PDMQ
RCXM	RDXM	SCCN	SPLR	SUPR	VABS
VCRS	VDOT	VNRM			

   For example, ACXP retrieves coordinates of atoms and computes atom contact pairs from them using a FORTRAN routine.

2. Functions similar to those in 1, but calling C subroutines. There are 8 such functions:

GPAN	GPBD	GPTO	LKAN	LKBD	LKTO
PDBF	SSQ				

   For example, GPAN computes bonded atom triplets by matching protein data to a bond angle dictionary.

3. Functions that retrieve components of protein structural data. There are 7 such functions, and they are frequently called in other PKB functions:

ATOM.COORDINATE	ATOM.RESIDUE
CHAINS	HETEROGEN.ATOM.COORDINATE
HETEROGEN.ATOM.GROUP	HETEROGEN.GROUP.NAME
RESIDUE.NAME	

4. Functions that access protein structural data and perform simple manipulations: There are 6 such functions:

ATOM.TYPE	HETEROGEN.ATOM.TYPE
HETEROGEN.GROUP.CHAIN	HETEROGEN.GROUP.TYPE
RESIDUE.CHAIN	RESIDUE.TYPE

5. PKB plotting functions that utilize the graphical capabilities of NewS. There are 5 such functions:

ANGLE.PLOT	ATOMS.PLOT
BONDS.PLOT	COUNTOUR.MATRIX
RAMACHANDRAN.PLOT	

6. Functions that build the PKB database. They read ASCII files from the Brookhaven Protein Data Bank and build NewS lists containing various data in their components. There are 13 such functions:

LOAD	LOADHEAD	LOADCRYS

LOADSCAL	LOADMTRI	LOADATOM
LOADSSBO	LOADHELI	LOADSHEE
LOADTURN	LOADHETA	LOADDOCS
LOADRESO		

7. All other PKB functions that consist of NewS expressions. There are 22 of them:

ALPHA.CARBONS	ATOMS
BETA.CARBONS	BOND.ANGLES
BONDS	DIHEDRAL.ANGLES
DIMER.RESIDUE.CONTACT.MATRIX	DISULFIDE.BONDS
FOR.PROTEINS	HETEROGEN.BONDS
HYDROGEN.BONDS	PDB.FILE
PEPTIDE.BONDS	RESIDUE.CONTACT.MATRIX
RESIDUE.CONTACT.NUMBER	RESIDUES
RTAT	SECONDARY.STRUCTURE
SEQUENCE.MOTIF	SIDE.CHAIN.CENTROID
VIRTUAL.BETA.CARBONS	VIRTUAL.BONDS

As evident from the above, many operations are actually performed in FORTRAN or C, especially such computationally intensive operations as pattern recognition. NewS is mainly used for accessing and managing protein objects as well as graphically displaying analysis results. In the PKB library, there are 82 functions implemented in NewS and 53

subroutines implemented in FORTRAN, RATFOR, or C.

9.2.3 PKB Analysis

Without looking into dynamic examples, static examination of the PKB library reveals patterns of function execution and data access in PKB analysis to some extent. For example, Figure 9.6 shows the 20 NewS library functions that are most frequently used in the PKB library. The figure shows the heavy usage of the $ function in the PKB library for accessing protein components. (The complete list of frequencies of NewS library functions used in the PKB library is found in Appendix A.)

Figure 9.7 shows all the expressions containing `get(prt)`, i.e., a function call for accessing a protein object in a database, that appear in the PKB library. All such expressions involve access to a particular subcomponent of a protein structure list using a series of applications of the component access function ($). We also indicate what function each such expression is contained in, and how frequently the function appears in the PKB library. The figure indicates that access to atomic coordinates and amino-acid residues are frequently performed in the PKB library.

As PKB is actually being used in state-of-the-art protein structural analysis, we could also examine real examples of the analysis to deduce typical scenarios and data access patterns. The objective of protein structural analysis is to find correlation between amino-acid sequences and structural motifs. Among published examples of PKB analysis are study of conformations of Arg-Gly-Asp sequences and sequences of $\beta\alpha\beta$ substructures [Bry89]. In most PKB analyses, a database is initially searched to extract a subset of proteins that

NewS function	operation description	number of uses in the PKB library
$	Component access	3284
[	Subset extraction	283
len	Number of elements in vector	239
paste	Glues arguments to create string	209
as.integer	Conversion to integer vector	203
nrow	Number of rows in matrix	158
stop	Terminates execution	145
||	Binary OR for expressions	143
:	Create numeric vector of specified range	142
print	Print data	124
as.character	Conversion to character vector	119
list	Create list	115
!=	Element-wise not equal for numeric or complex vector	105
!	Negation of elements in logical vector	104
c	Construct composite vector from elements in arguments	104
unix	Execute unix command	92
match	Look up elements in table	84
vector	Vector constructor	83
as.double	Conversion to double-precision number vector	79
as.single	Conversion to single-precision number vector	74

Figure 9.6 20 Most Frequently Used NewS Library Functions in the PKB Library

protein component access	appears in <function name> (frequency in the PKB library)
get(prt)$PEPTIDE$ATOM$COORDINATE	ATOM.COORDINATE (33)
get(prt)$PEPTIDE$ATOM$RESIDUE	ATOM.RESIDUE (28)
levels(get(prt)$PEPTIDE$ATOM$TYPE)[get(prt)$PEPTIDE$ATOM$TYPE]	ATOM.TYPE (15)
ac_match(atm, levels(get(prt)$PEPTIDE$ATOM$TYPE))	ATOMS (14)
an_get(prt)$PEPTIDE$ATOM$TYPE~~ac	ATOMS (14)
sl_get(prt)$PEPTIDE$MOTIF$DISULFIDE.BOND$RESIDUE	BOND.ANGLES (1), BONDS (1), DIHEDRAL.ANGLES (2)
get(prt)$PEPTIDE$RESIDUE$CHAIN	CHAINS (13)
drs_get(prt)$PEPTIDE$MOTIF$DISULFIDE.BOND$RESIDUE	DISULFIDE.BONDS (2)
get(prt)$HETEROGEN$ATOM$COORDINATE	HETEROGEN.ATOM.COORDINATE (2)
get(prt)$HETEROGEN$ATOM$GROUP	HETEROGEN.ATOM.GROUP (2)
levels(get(prt)$HETEROGEN$ATOM$TYPE)[get(prt)$HETEROGEN$ATOM$TYPE]	HETEROGEN.ATOM.TYPE (2)
x_get(prt)$HETEROGEN$ATOM$COORDINATE	HETEROGEN.BONDS (1)
g_get(prt)$HETEROGEN$ATOM$GROUP	HETEROGEN.BONDS (1)
levels(get(prt)$HETEROGEN$GROUP$CHAIN)[get(prt)$HETEROGEN$GROUP$CHAIN]	HETEROGEN.GROUP.CHAIN (2)
get(prt)$HETEROGEN$GROUP$NAME	HETEROGEN.GROUP.NAME (2)
levels(get(prt)$HETEROGEN$GROUP$TYPE)[get(prt)$HETEROGEN$GROUP$TYPE]	HETEROGEN.GROUP.TYPE (3)
levels(get(prt)$PEPTIDE$RESIDUE$CHAIN)[get(prt)$PEPTIDE$RESIDUE$CHAIN]	RESIDUE.CHAIN (3)
get(prt)$PEPTIDE$RESIDUE$NAME	RESIDUE.NAME (2)
levels(get(prt)$PEPTIDE$RESIDUE$TYPE)[get(prt)$PEPTIDE$RESIDUE$TYPE]	RESIDUE.TYPE (7)
(if (lis.null(get(prt)$PEPTIDE$MOTIF$HELIX)) s[get(prt)$PEPTIDE$MOTIF$HELIX$RESIDUE]_2)	SECONDARY.STRUCTURE (1)
(if (lis.null(get(prt)$PEPTIDE$MOTIF$STRAND)) s[get(prt)$PEPTIDE$MOTIF$STRAND$RESIDUE]_3)	SECONDARY.STRUCTURE (1)
(if (lis.null(get(prt)$PEPTIDE$MOTIF$TURN)) s[get(prt)$PEPTIDE$MOTIF$TURN$RESIDUE]_4)	SECONDARY.STRUCTURE (1)

Figure 9.7

share certain characteristics, e.g., a common amino-acid sequence or conformational motif. Such a subset is then statistically analyzed to identify the trends among its members, e.g., whether all the proteins with a common sequence share the same structural features. The result of the analysis is then graphically displayed at the end. Note that even if NewS generally promotes interactive data access, most PKB analyses load all the necessary data into memory in batches at the beginning. This is because the statistical analysis is very computationally intensive and time-consuming, spanning days, weeks or even months. In order to facilitate initial batch loading of the data for efficient analysis, PKB typically runs on a workstation with large memory space. For example, at National Center For Biotechnology Information (NCBI) at National Institutes of Health, PKB is running on the SGI 4D-35 with 128 Mbytes of memory all dedicated to PKB analysis.

9.2.4 Selected Operations for the Experiments

As Figure 9.7 shows, access to the ATOM and RESIDUE component of proteins appears frequently in the PKB library. Therefore, we selected the following expressions that access properties of those components for the experiments.

1. $get(prt)\$PEPTIDE\$ATOM\$COORDINATE$

   in the ATOM.COORDINATE function

   (number of appearance in the PKB library for ATOM.COORDINATE: 33)

2. $levels(get(prt)\$PEPTIDE\$ATOM\$TYPE)[get(prt)\$PEPTIDE\$ATOM\$TYPE]$

in the ATOM.TYPE function

(number of appearance in the PKB library for ATOM.TYPE: 15)

3. $get(prt)\$PEPTIDE\$RESIDUE\$CHAIN$

in the CHAINS function

(number of appearance in the PKB library for CHAINS: 13)

4. $levels(get(prt)\$PEPTIDE\$RESIDUE\$TYPE)-$

$[get(prt)\$PEPTIDE\$RESIDUE\$TYPE]$

in the RESIDUE.TYPE function

(number of appearance in the PKB library for RESIDUE.TYPE: 7)

Since the **$** function has GemStone implementation, all component access in cascaded $'s is executed inside GemStone when protein objects are stored there, and only the accessed component is retrieved into NewS.

The second and fourth expressions include some manipulation of accessed components. In those expressions, the accessed component, TYPE, is a category object; its **levels** attribute specifies possible residue or atom type names, and its data is a collection of indices into the possible names in the **level** attribute. When the expression

$$levels(get(prt)\$PEPTIDE\$ATOM\$TYPE)[get(prt)\$PEPTIDE\$ATOM\$TYPE]$$

is evaluated,

$$levels(get(prt)\$PEPTIDE\$ATOM\$TYPE)$$

is evaluated first, returning the `levels` attribute that consists of all possible atom type names. The expression

$$get(prt)\$PEPTIDE\$ATOM\$TYPE$$

evaluates to the collection of indices into the value of the `levels` attribute, i.e., all possible atom type names, so as a whole, the expression above delivers a collection of actual atom type names contained in a protein.

In contrast to various component access, entire protein structures are also accessed at once for display in the experiments.

Due to the limited disk space, we did not create all the entries in the Brookhaven Protein Data Bank (over 600) as GemStone objects. Figure 9.8 lists the 11 proteins we stored in GemStone for the experiments. Those proteins were chosen based on their sizes in terms of the number of amino-acid residues, and they contain a range of numbers of residues. Therefore, the properties of their ATOM and RESIDUE components accessed in the experiments exhibited a range of sizes as well. As there is no mechanism in GemStone to report the physical size (in terms of bytes) of objects, it was impossible to determine what collection of proteins could be safely stored as GemStone objects in a given amount disk space. Therefore, after we successfully created those 11 proteins, we did not attempt to increase the size of the database.

9.2.5 Analysis of Experimental Results

The series of diagrams Figure 9.9 to 9.20 shows the results from running the selected PKB expressions described in the previous section.

PKB

protein names	number of amino-acid residues	number of atoms	size in bytes when in file
0FX3	0	0	2,673
6HIR	65	664	66,329
4FD1	106	841	83,025
3FXN	138	1073	106,677
8DFR	189	1504	151,794
2GCH	245	1738	168,075
1HCO	287	2192	201,690
2CYP	294	2299	225,180
4CPA	346	2727	235,062
2AAT	396	3065	259,524
2YHX	457	3290	284,958

The above protein names are abbreviations provided by the Bookhaven Protein Data Bank. For example, 0FX3 represents oxidized flavodoxin from anacystis.

Figure 9.8 NewS Objects from PKB in GemStone

Figure 9.9 GemStone-based NewS

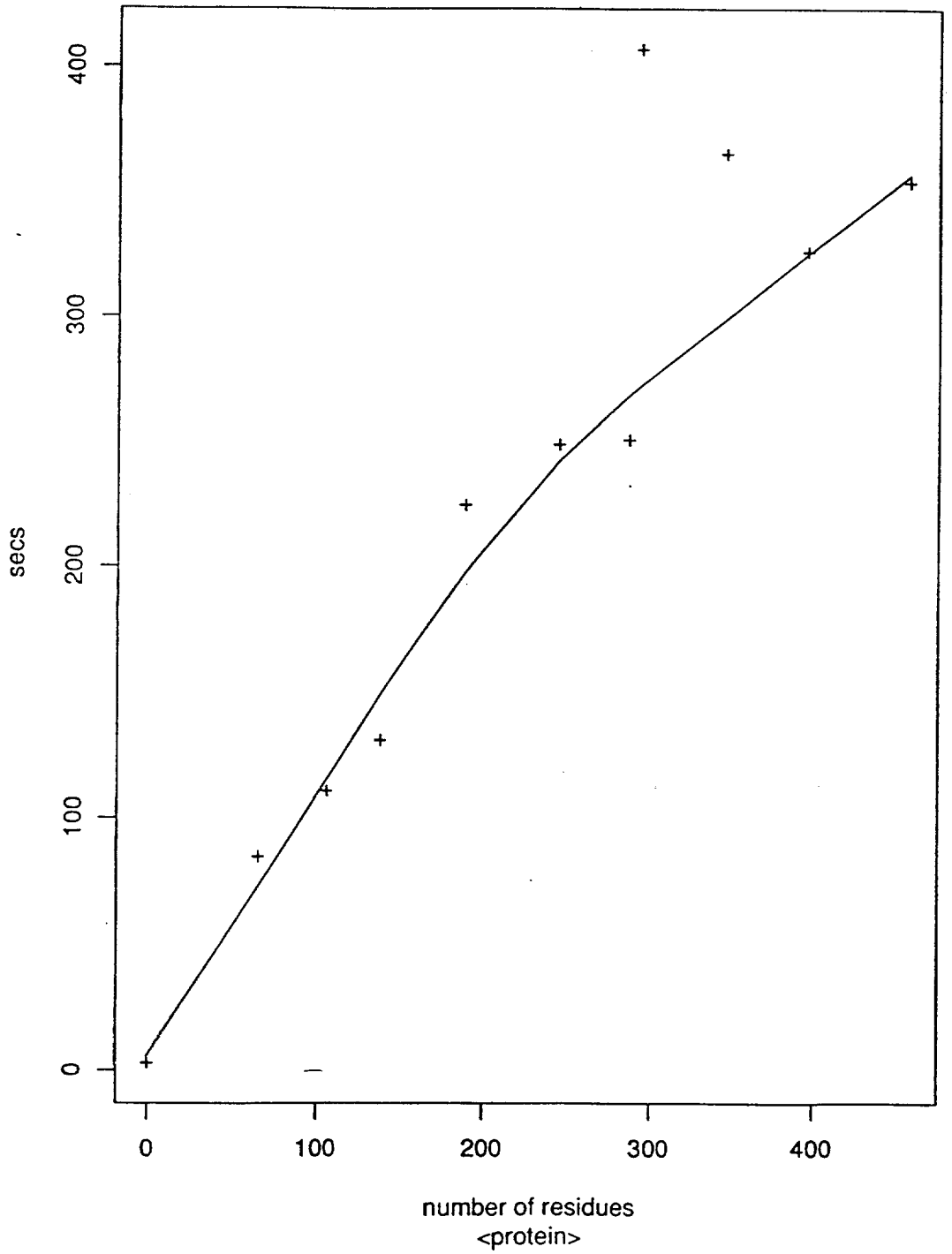


Figure 9.10 UNIX-based NewS

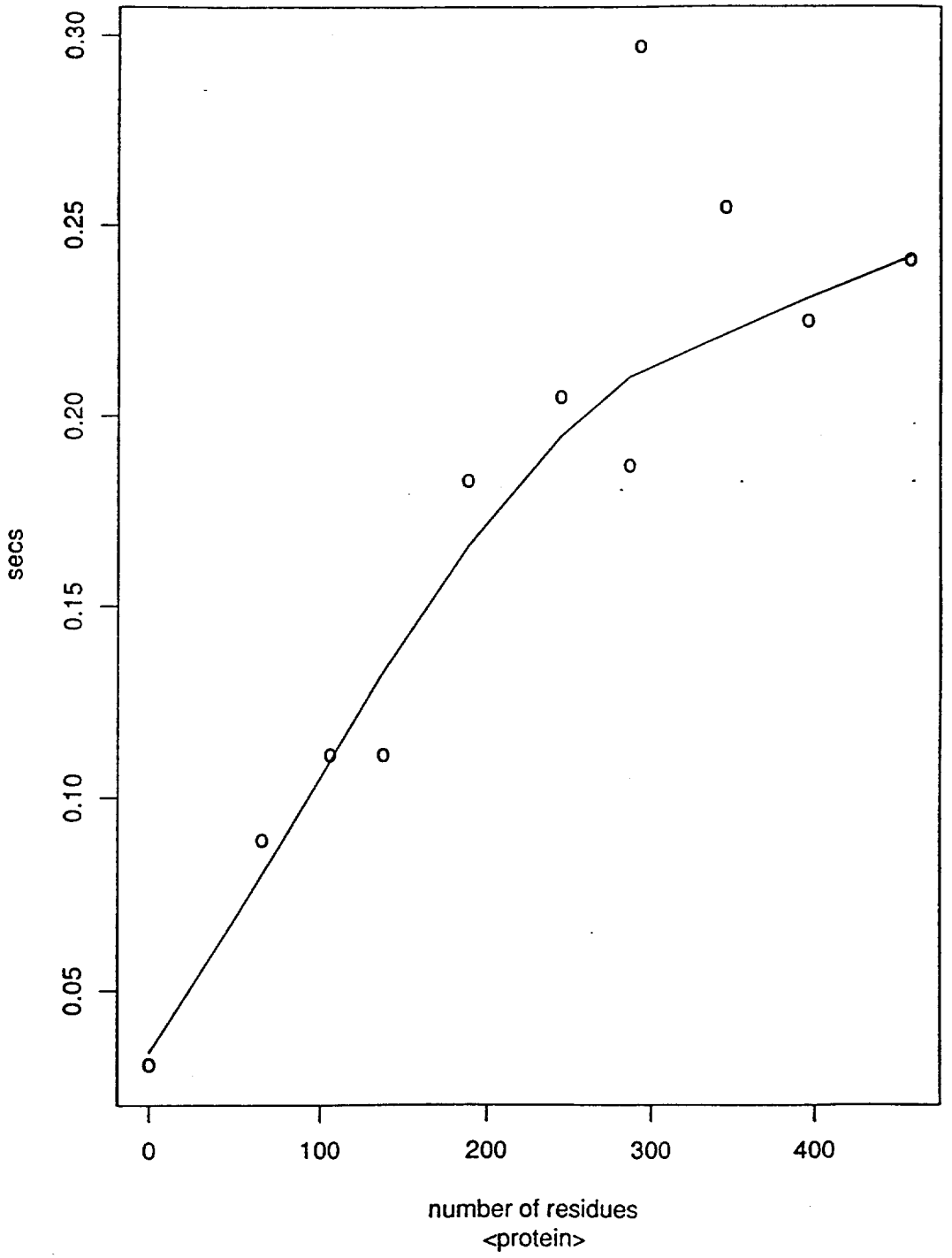


Figure 9.11 GemStone-based NewS

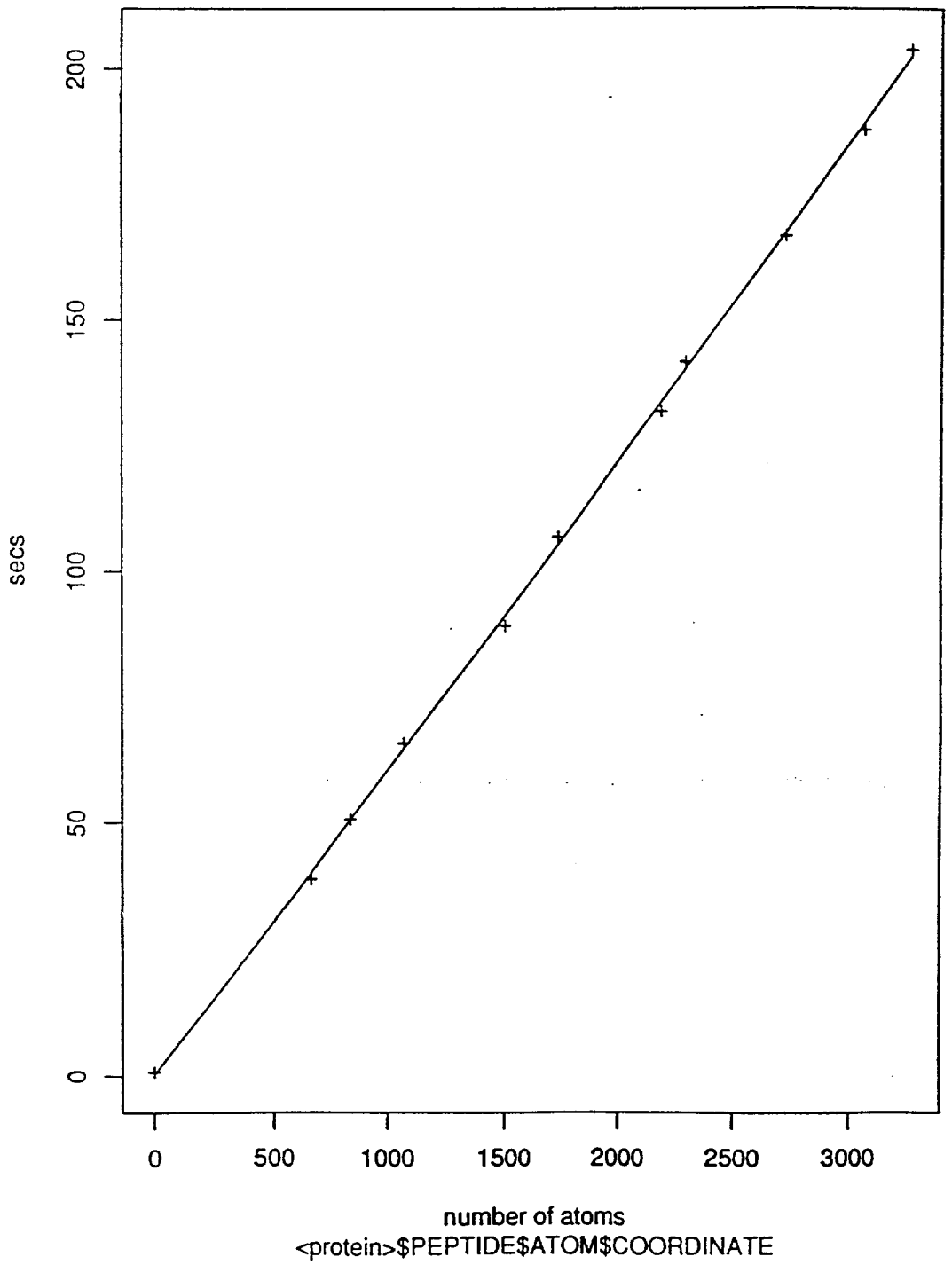


Figure 9.12 UNIX-based NewS

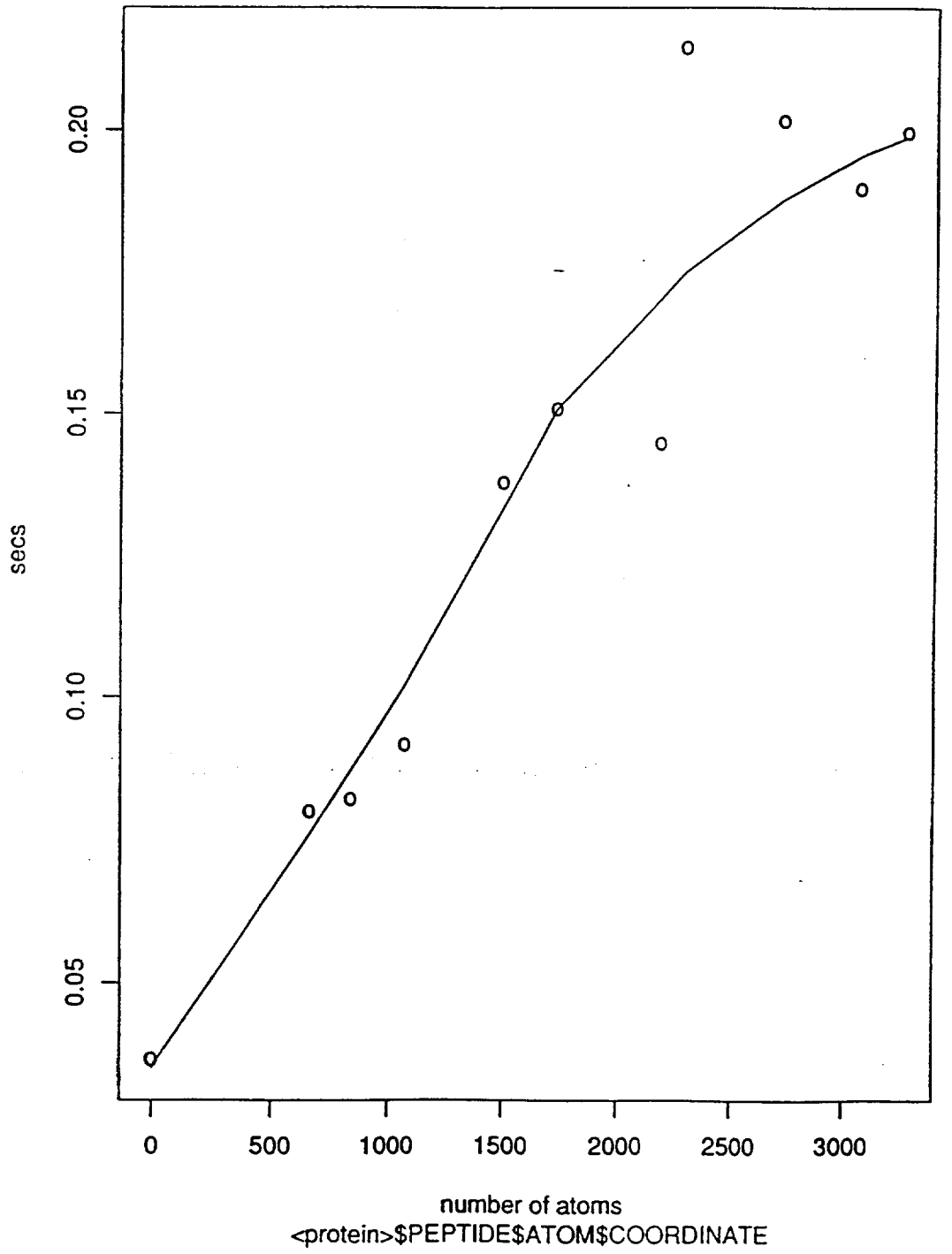


Figure 9.13 GemStone-based NewS

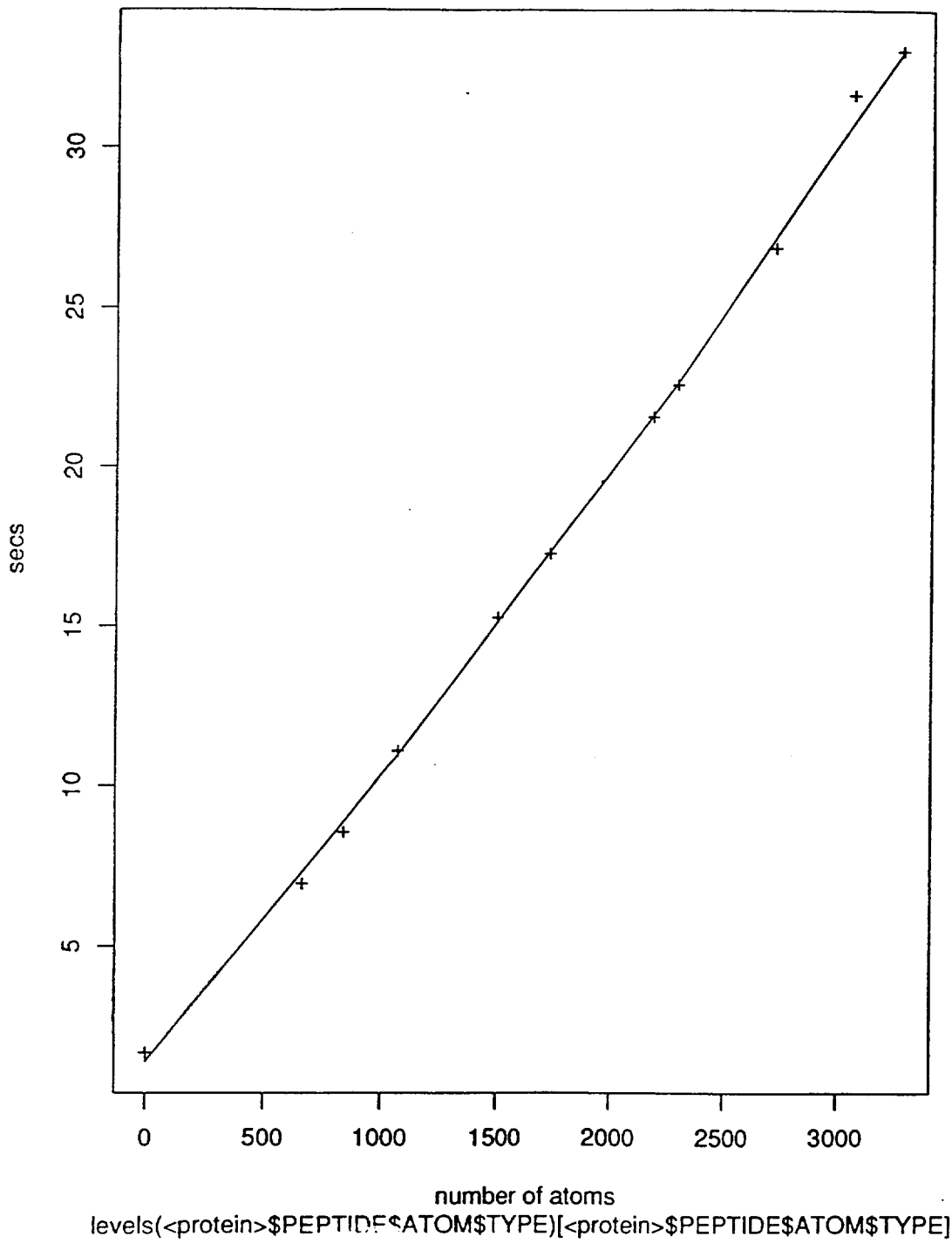


Figure 9.14 UNIX-based NewS

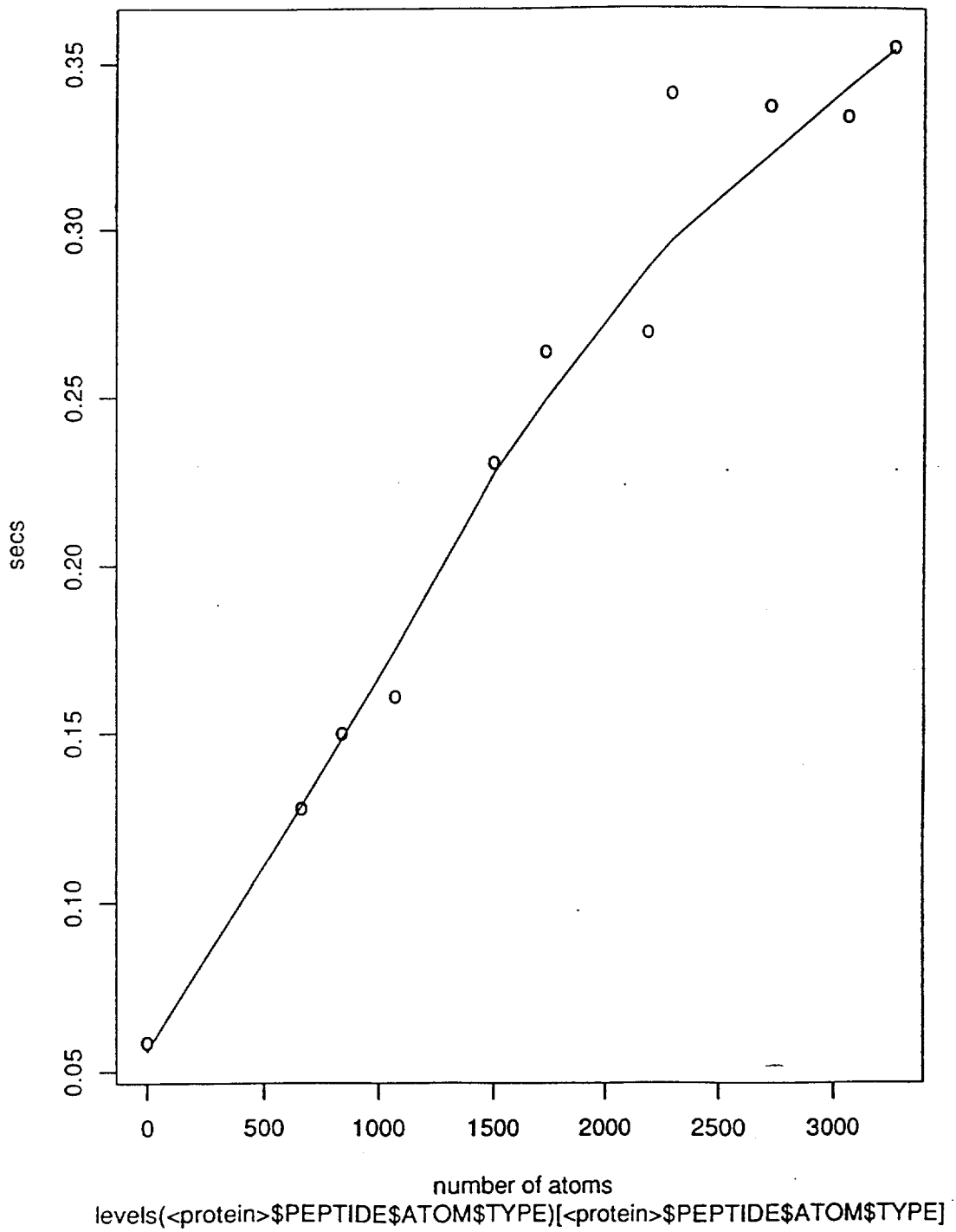


Figure 9.15 GemStone-based NewS

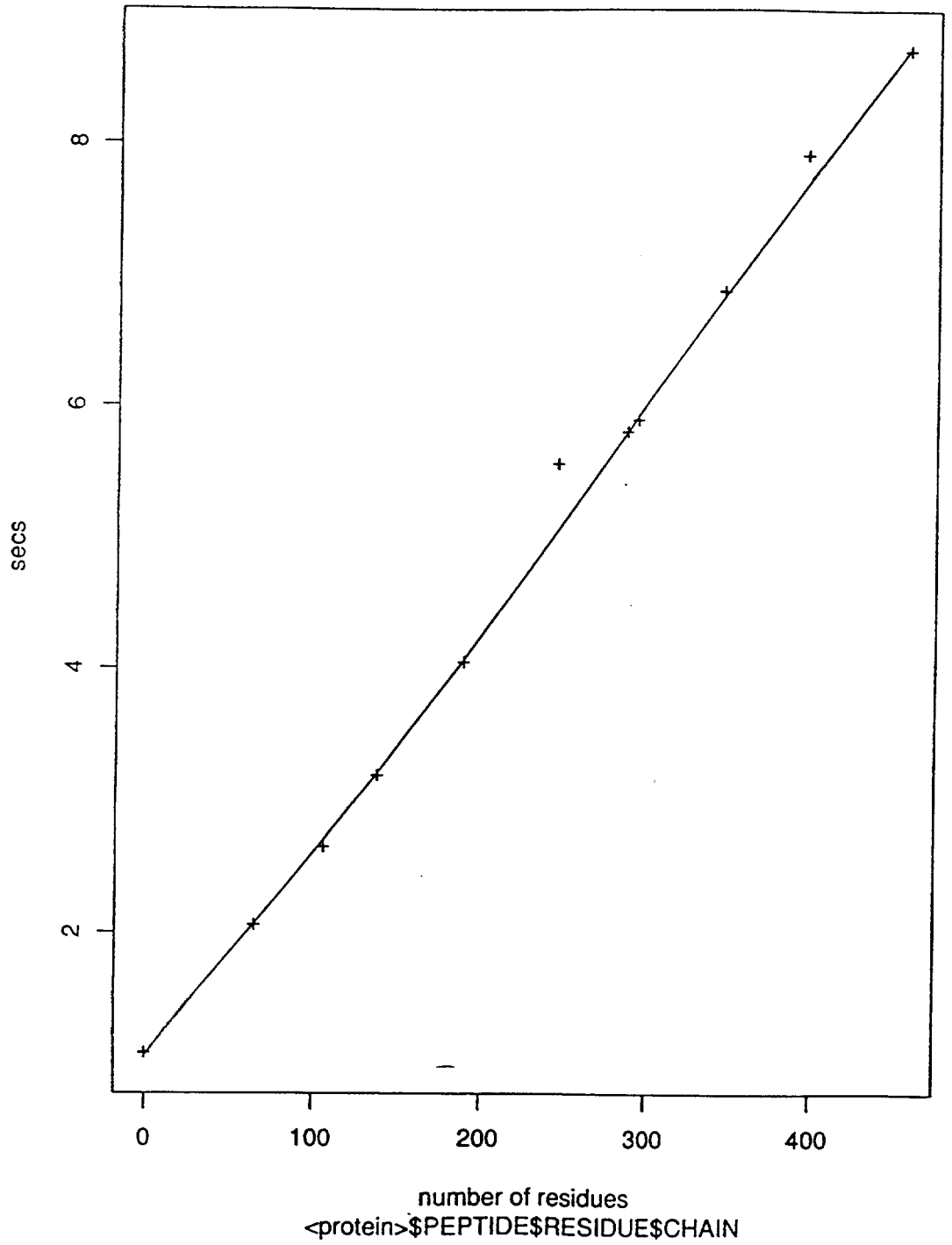


Figure 9.16 UNIX-based NewS

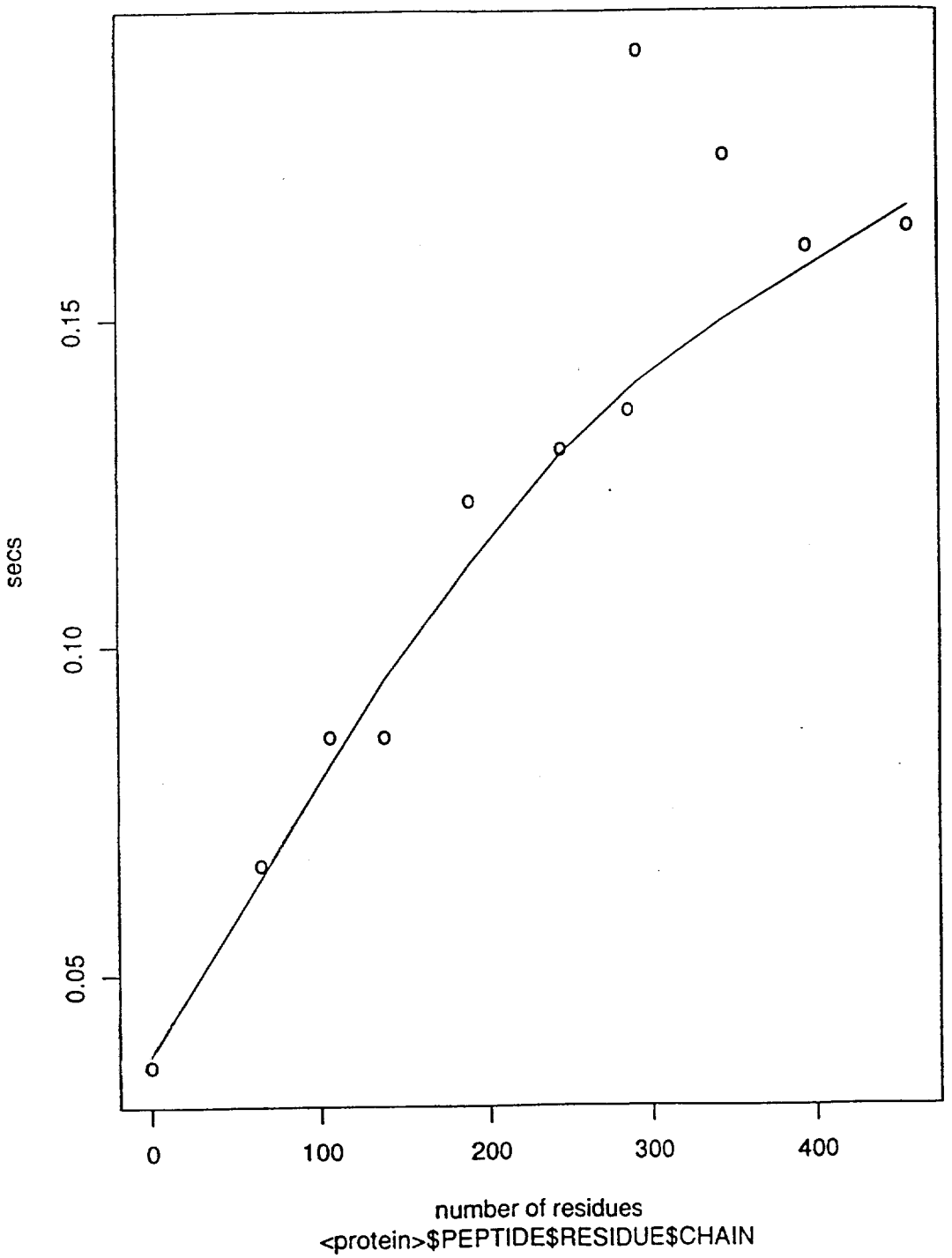


Figure 9.17 GemStone-based NewS

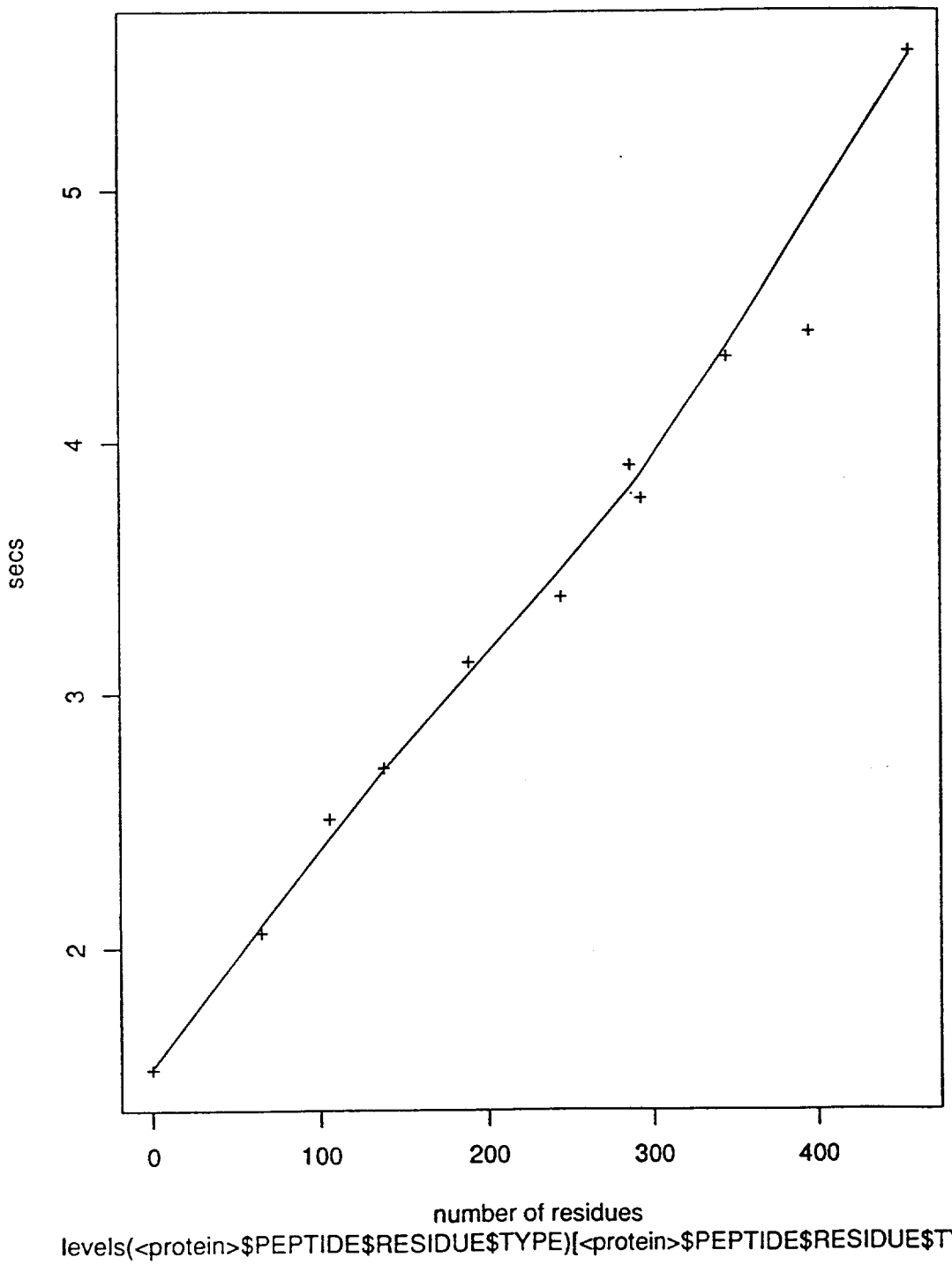


Figure 9.18 UNIX-based NewS

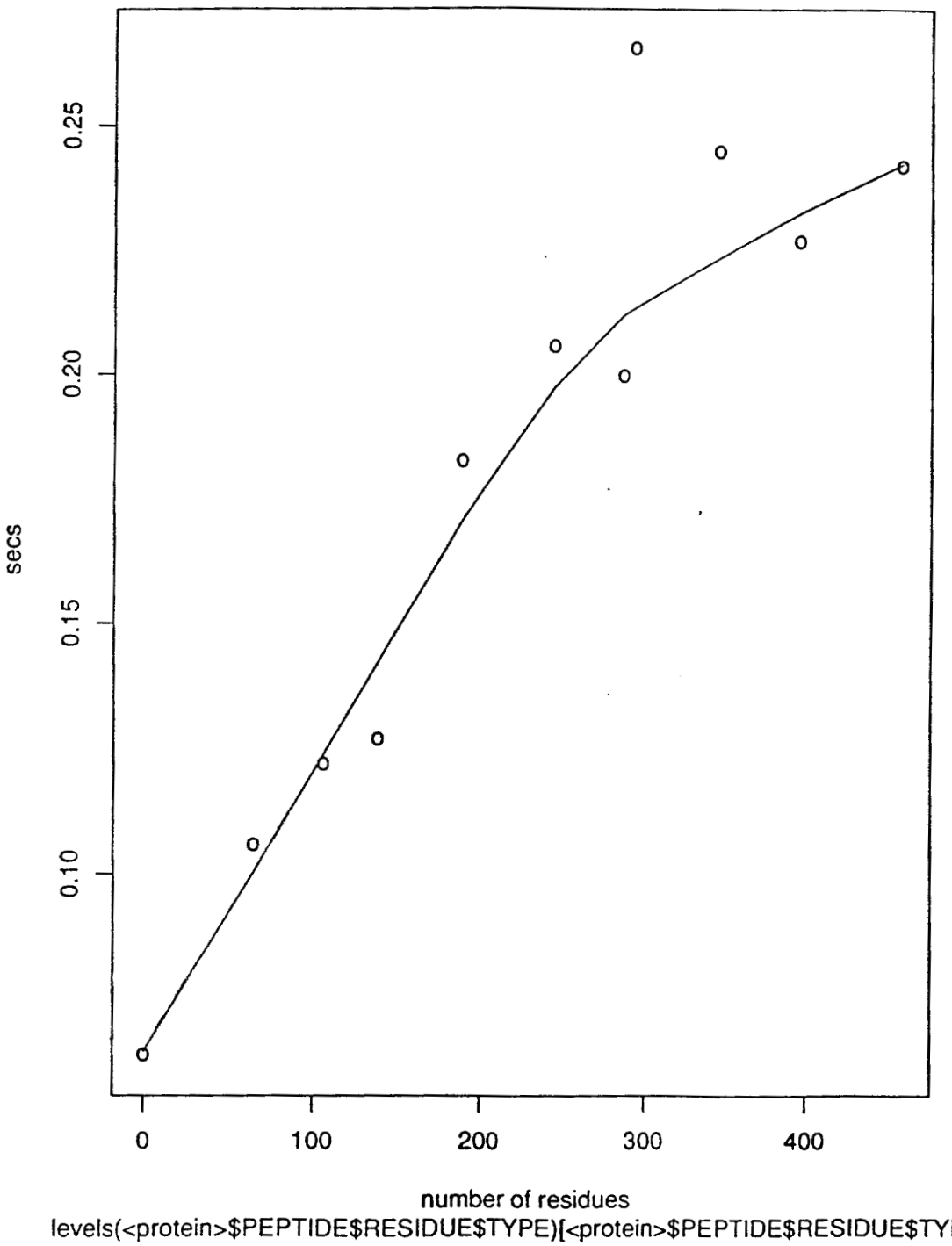


Figure 9.19 GemStone-based NewS vs. UNIX-based NewS

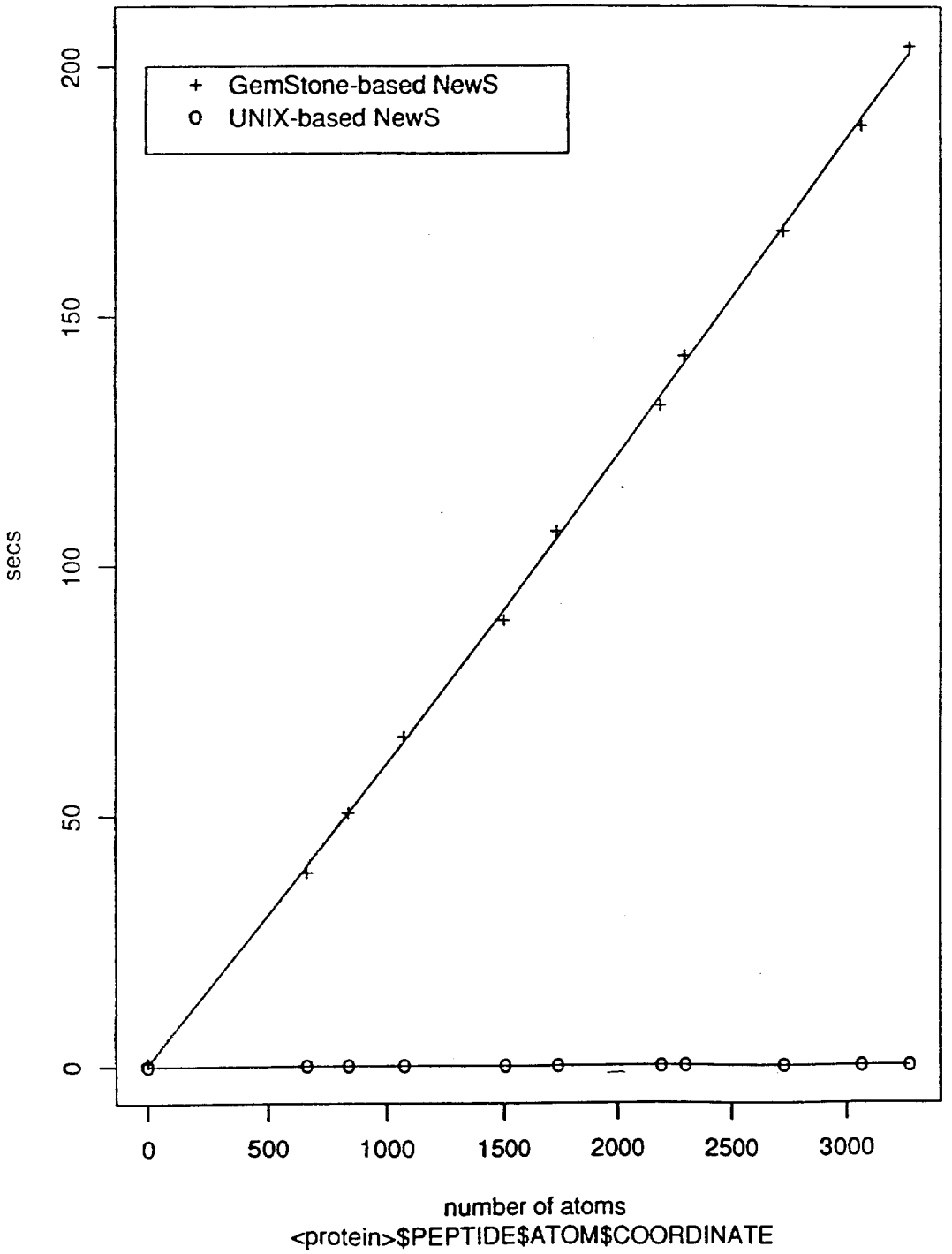


Figure 9.20 GemStone-based NewS vs. UNIX-based NewS

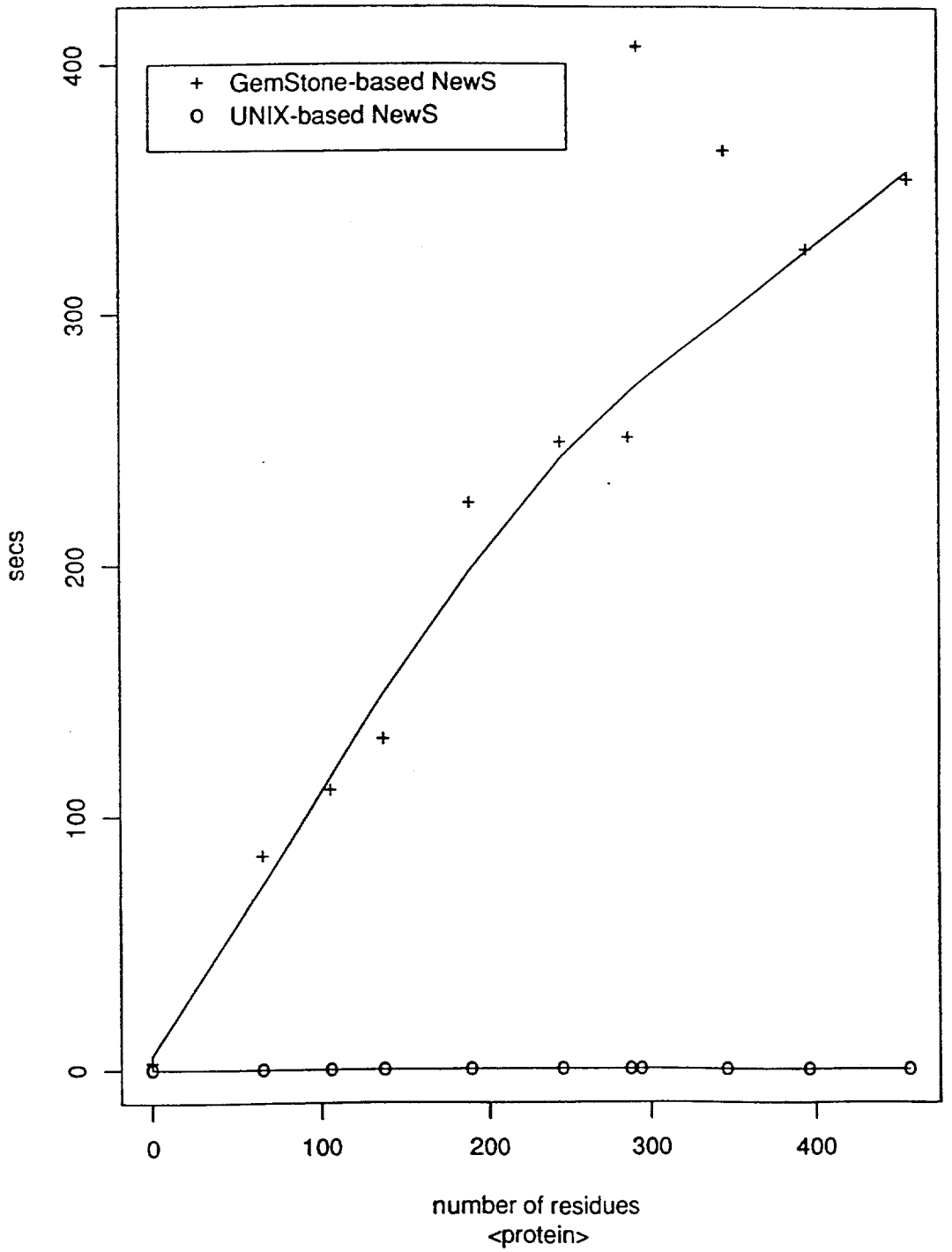


Figure 9.9 shows the performance of GemStone-based NewS when the whole protein is retrieved into memory. For a reference, the performance of UNIX-based NewS for the same operation is shown in Figure 9.10. As with NHANES data, there is a large overhead for accessing GemStone objects over file access, especially when the size of the retrieved objects is large. Note that in both figures, the access time is proportional not to the number of residues but to the size of the whole data. As some Brookhaven entries contain much miscellaneous information besides the residue data, the number of residues does not necessarily indicate the size of the whole data file.

Figure 9.11, 9.13, 9.15, and 9.17 show the results from accessing components of proteins for GemStone-based NewS. In evaluating an expression

$$get(prt)\$PEPTIDE\$ATOM\$COORDINATE$$

with proteins in GemStone, each call to **$** in GemStone returns to NewS an OID of the corresponding component. For example, GemStone returns to NewS the OID of the `PEPTIDE` component selected from the whole protein, which NewS subsequently hands back to GemStone for further processing of **$**'s in a database. After the OID of

$$get(prt)\$PEPTIDE\$ATOM\$COORDINATE$$

is obtained in GemStone, its data values, i.e., x, y, and z coordinates of all atoms, are actually transferred to NewS for displaying the results. (The NewS evaluator makes one GCI function call per each value transferred.) The number of transferred values in this case ranges from 0 to 3290, depending on the accessed protein. The performance in all cases

is approximately proportional to the size of the accessed components, which indicates that the transfer of actual data values at the end dominates the execution time.

Figure 9.11 shows the case where coordinates of atoms in a protein are accessed; the returned result therefore contains three numerical values per atom for x, y, and z coordinates. In contrast, the case illustrated in Figure 9.13 accesses the type names of atoms, and the returned result includes one string for a name of each atom. As each number and string are both retrieved by a single call in GemStone-based NewS, the performance of GemStone-based NewS in Figure 9.13 is about an order of magnitude faster than that shown in Figure 9.11 due to a smaller number of calls required for the data transfer. Figures 9.15 and 9.17 show the cases where various properties of residues in a protein are accessed. Each residue consists of a number of atoms and consequently, the number of residues in a protein is smaller than the number of atoms. Therefore, the performance of GemStone-based NewS in these cases is faster than Figure 9.11 or 9.13.

Figure 9.12, 9.14, 9.16, and 9.18 show the results of accessing protein components for UNIX-based NewS. All the test cases shown there exhibit very similar execution time, an average being somewhere between 0.1 second and 0.2 second, and very similar patterns of performance across different proteins. Whenever any protein component is accessed in UNIX-based NewS, the whole protein is brought into memory as an actual argument to $, which explains the similar value and pattern in the execution time. In fact, the same pattern of execution time is observed in Figure 9.10, i.e., the retrieval of the whole protein for GemStone-based NewS.

The obtained results in general illustrate advantages of processing component access

inside the database; the smaller the accessed component is with respect to the whole protein, the more time is saved by eliminating unnecessary data transfer. Comparison of Figures 9.11, 9.13, 9.15, 9.17 to Figure 9.9 illustrates a range of the time saved from accessing protein components in GemStone, from about 50% reduction to difference by a couple of orders of magnitude. As large data benefit from local, segment-based access in GemStone, execution of component access in GemStone eliminates unnecessary access to those pages that do not contain accessed component.

However, as with NHANES data, experiments with PKB in general revealed a large overhead for accessing GemStone objects when compared to file access, especially when the accessed objects are large. Figure 9.19 and 9.20 compare the performance of UNIX-based NewS and GemStone-based NewS to indicate the overhead explicitly. It was slower by 1 to 3 orders of magnitudes to access protein objects in GemStone than in files, as was the case with the NHANES data.

We compared the performance of GemStone-based NewS to UNIX-based NewS since the comparison gave us concrete numbers to discuss. However, the performance would be best interpreted when discussed with respect to each application's requirements. As PKB is being used actively in protein structural analysis, we could examine its execution environment and assess the performance in that context as follows.

Note that data access are performed initially in batches in PKB analysis, so performance requirements are less stringent than other NewS applications that require interactive access to a database. A typical time frame of PKB analysis spans days, weeks, or even months, so in terms of numbers, the performance of GemStone-based NewS reported in this chapter is

not far off from that range. According to Figures 9.15 and 9.17, time to access a property of the `residue` component averages around 5 seconds for GemStone-based NewS. Therefore, it would take the platform 50 minutes to iterate over 600 proteins. More complex PKB queries would likely take several hours to days with the data in GemStone.

9.3 Summary of Analysis

This chapter described the experiments we performed with two sample NewS applications, NHANES data and PKB. The experiments gave us an opportunity to examine one of the design alternatives discussed in the previous chapter, i.e., storing and executing functions in GemStone to control the amount of data transfer from GemStone to NewS. With both applications, having a GemStone implementation of such functions as subscript (`[]`), `mean`, and component access (`$`) resulted in reduced execution time, especially when it eliminated a large amount of unnecessary data transfer.

The performance of GemStone-based NewS, when compared to UNIX-based NewS, showed a large overhead for accessing GemStone objects over file access. Throughout the experiments, GemStone-based NewS was slower than UNIX-based NewS by 1 to 3 orders of magnitude. One of the possible causes for the slow performance is the space overhead for object representation. Namely, when NewS objects are stored in GemStone, they seem to take up much more storage space than the equivalent file objects in UNIX-based NewS. Since there is no facility for monitoring physical space occupied by objects in GemStone, it was difficult for us to find out such storage overhead exactly. However, we once observed from a difference in used disk space before and after object creation that an NHANES data

vector, 0.046 Mbytes in file, was expanded to 1.4 Mbytes when converted to a GemStone object. Therefore, more data pages must be accessed by GemStone-based NewS to read an object, clearly a disadvantage in terms of performance.

We were also concerned with the difference in the size of the executables. Since NewS was directly linked into the Gem process that manages a GemStone session for efficient communication between the two systems, the physical size of GemStone-based NewS was significantly larger than UNIX-based NewS, i.e., 3.9 Mbytes (GemStone-based NewS) vs. 2.3 Mbytes (UNIX-based NewS). However, we observed no apparent effect of the size difference. During the experiments, we did not observe swapping activities particular to GemStone-based NewS. More page-ins and page-outs were initiated by GemStone-based NewS than UNIX-based NewS, though it is not clear how much of it was due to large object size in GemStone and how much to different sizes of executables.

PKB presented us with an example where storing the data in a database rather than as NewS file objects is clearly advantageous, even with the performance overhead. At NCBI, there is currently an attempt to unify PKB data and genetic sequence data, which are currently stored separately in different formats, in a common database. Such an effort would facilitate easy access to all kinds of biological data by multiple applications and expand the scope of individual analyses, e.g., PKB can now incorporate DNA sequence data in the analysis or even combine its functions with operations available in other applications through a database. As a relational database (SYBASE) has not served well as such a common data repository, object-oriented databases are being considered as a potentially better alternative.

Chapter 10

Ease of Experimentation with GemStone-based NewS

In the previous two chapters, we described the experiments we performed on GemStone-based NewS. This chapter discusses our experience experimenting with GemStone-based NewS and evaluates ease of experimentation with it. As mentioned before, the purpose of GemStone-based NewS was to explore possibilities in designing object-oriented database support for scientific applications, and the platform was designed to make such exploration as easy as possible.

Section 10.1 describes the experimentation process of creating objects in GemStone and performing operations on them. Section 10.2 then discusses modifying the baseline architecture to examine architectural alternatives as described in Chapter 8. Finally, Section 10.3 assesses ease of experimentation based on our experience. Though ease of experimentation is a somewhat subjective criterion, we attempt to identify those architectural features of GemStone-based NewS that contributed to a reduced amount of work in comparison to what we would expect with other approaches.

10.1 Experimentation Process

As mentioned before, object creation in GemStone for the experiments was simplified by preserving the object access paradigm of current NewS. We just had to include an appropriate GemStone dictionary as the first entry in the search list, and an object was created in GemStone by a permanent assignment, just as in files. (In order to distinguish UNIX directory names from GemStone dictionary names in the search list, we attached a prefix "/db" to the latter.) For both applications tested, i.e., NHANES Data Analysis and PKB, input data were initially contained in ASCII files. The applications included functions for reading data from ASCII files and creating NewS objects from them using permanent assignments. We reused such functions for creating objects in GemStone for those applications. Actual work for object creation, e.g., conversion of NewS objects to GemStone representation, mapping of NewS names to equivalent GemStone names, was automatically performed by the evaluator.

Creating functions in GemStone involved more effort as it required creation of a GemStone class for the function with the definition of its operation, and instantiation of the class. We described various issues in mapping a NewS function to an equivalent GemStone function object in Chapter 7.

In many cases, we compared file access and GemStone access for the same operation, and objects were duplicated in files and GemStone. The first entry in the search list was adjusted to either a file directory or a GemStone dictionary to duplicate objects in both places. In order to distinguish a pair of objects representing the same data, we named the

GemStone version "db<object name>" where <object name> was a name of a file object. GemStone implementations of NewS library functions such as `subscript`, `$` were created so that different versions were executed depending on where the data were stored.

Once objects and functions were created for the experiments, we had a uniform, transparent access to them regardless of their locations. All the entries in the search list, including file directories and GemStone dictionaries, were searched to locate objects. In the case objects resided in GemStone, conversion of object name and structure from GemStone to NewS was automated in the evaluator. As a result, we could reuse on GemStone-based NewS not only those functions for creating objects but also any existing NewS functions that access NewS objects without modifications, and all the test cases in the experiments were essentially NewS expressions.

With the test cases being NewS expressions, we could apply the `unix.time` function provided in the NewS library to them to time their evaluation. In case of GemStone access, the result from `unix.time` included time required for execution of NewS and GemStone as well as communication between them. Since NewS is implemented mostly in C, we could also use the UNIX command `getrusage` to monitor resource utilization during computation of arbitrary subparts of an expression by inserting the command in appropriate places of the source code.

To expedite the experimentation process with repeated runs, we wrote a shell script that automated a testing process for multiple expressions. The script automatically started up a NewS session, logged into GemStone, executed a collection of test expressions, and recorded the results from `unix.time` and `getrusage`.

10.2 Examining Design Alternatives

Chapter 8 described possible architectural alternatives in designing GemStone support for NewS applications. We investigated how to examine such alternatives on GemStone-based NewS, i.e., how to alter architecture of the platform to realize a set of alternatives so that we can examine how the same function behaves differently against them. Figure 10.1 summarizes a set of design alternatives we examined and the required modifications for them. As we retained encapsulation of GemStone and NewS in constructing the platform, most modifications were local to one system or the other, making it simple to set up the alternatives.

Setting up and comparing the alternatives required direct interaction with GemStone. For example, as mentioned before, an alternative GemStone schema was represented as a set of subclasses, and rather than introducing such variations into the NewS environment, e.g., "cached" lists for the `list` mode, we directly instantiated those subclasses in GemStone. Unlike the file system that is hidden under NewS, GemStone provided an interface for direct object access, which served well for our experiments. In some cases, we monitored GemStone operations directly for preliminary assessment. For such monitoring, GemStone provided several methods for keeping track of the Gem process such as **`gemStatistics`** and **`millisecondsToRun:<aBlock>`**. However, information on the Stone process that handles most physical data access was only available through a special method **`stoneStatistics`** and only to a privileged user, making it somewhat difficult to obtain an overall picture of

Design Alternatives	Required Modifications	System(s) Affected by Modifications
different GemStone schema (different representations of objects, alternative OPAL methods)	modifications to class structure (creation of subclasses for alternatives)	mainly GemStone changes NewS could be affected if the change in class structure is accompanied by the modified external interface
different mechanisms for delegating operations	different decision-making algorithms incorporated into NewS interpreter (and subsequent recompilation)	NewS changes only
different communication mechanisms between GemStone and NewS (different granularities, message passing vs. structural access, etc.)	different GCI calls incorporated into NewS evaluator	mainly NewS changes may require modifications on the GemStone side, e.g., may encode frequently-asked information as an OPAL method so that such information is available by a single GCI call

Figure 10.1(1) Required Modifications for Design Alternatives

Design Alternatives	Required Modifications	System(s) Affected by Modifications
caching GemStone information in NewS	modification to NewS evaluator to look up cache first before making a GCI call also a mechanism to maintain consistency between cached information and actual state of a database	NewS changes to set up caching
interface to GemStone functionalities	modification to NewS interpreter or addition of NewS functions that execute functionalities through GCI	NewS changes only
optimized evaluation of NewS expressions	optimizations can be examined by having GemStone function objects accumulate NewS expressions for subsequent optimization	GemStone changes only (modification of OPAL methods) Must also find or write potentially optimizable NewS expressions to experiment with
different collections of objects and functions stored	insertion/deletion of objects and functions in a GemStone dictionary (functions also require definition of GemStone function classes)	GemStone changes only

Figure 10.1(2) Required Modifications for Design Alternatives

GemStone activities.

10.3 Ease of Experimentation

In executing the experiments, our decision to preserve transparent object access of NewS contributed to ease of experimentation on GemStone-based NewS by increasing reuse of existing NewS functions. Changes to the NewS environment were kept minimal, and GemStone was presented to users as a set of "dictionaries", i.e., "containers" of persistent data synonymous to file directories. Objects were created in GemStone the same way as in files by including such a GemStone dictionary in the search list. Most effort was actually spent on initial creation of functions in GemStone as it required implementation of an equivalent operation as GemStone methods. Once a database was prepared, we could run NewS expressions against objects stored in GemStone, and reuse existing functions for the experiments. Had we chosen C or FORTRAN applications instead, we would have had to modify the applications extensively due to a lack of transparent access to a database; all the file reads and writes would have had to be replaced by GCI calls.

To examine design alternatives of the platform, its modular, encapsulated architecture made most of the required changes for investigating such alternatives local to either NewS or GemStone. As one of the most difficult aspects of making changes is their complete propagation to ensure consistency of the code, the modular architecture and resulting localized changes had a large effect on overall ease of experimentation.

Chapter 11

Conclusion and Future Work

As mentioned at the beginning, an objective of our work was to explore how to improve scientific data management through experimentation on a platform. At this point, we assess whether or not our approach was effective for the purpose, and describe what we consider primary contributions of this study. We also discuss possible future work.

11.1 Evaluation of Our Approach

In assessing our approach, we first examine how our platform, GemStone-based News, measures up to the criteria discussed in Chapter 4. We also look at suitability of the platform for investigating a range of issues described in Chapter 5. Finally, we evaluate GemStone features for supporting NewS applications. As we adopted GemStone mainly due to its availability, our intention was to use GemStone features as a "reference" and examine their suitability for NewS.

11.1.1 Evaluation of GemStone-based NewS

In Chapter 4, we discussed criteria for an experimental platform, namely, cost-effectiveness in construction, productivity and flexibility of the architecture, generality of the targeted applications, performance level, and ease of experimentation.

A choice of NewS as a target gave generality in scientific applications examined, where

better data support is desired. Data support requirements of NewS are a union of those found in many scientific applications. We examined two sample NewS applications, NHANES Data Analysis and PKB, that use very different data types.

A choice of an object-oriented database contributed to productivity and flexibility of the platform's architecture. Unlike files or relational databases, an object-oriented database has both a flexible data model and traditional database support. Hence, we could directly represent NewS data and functions while examining the potential of storage management for large NewS datasets or contemplating how GemStone concurrency control could benefit certain NewS execution environments. Chapter 8 illustrated flexibility of the platform's architecture with possible variations in GemStone schema, communication with GemStone, introduction of GemStone capabilities, optimization of NewS expression evaluation, and NewS objects and functions stored in GemStone.

Adoption of NewS and GemStone also amounted to a cost-effective platform. By using a commercial object-oriented database GemStone, all the object-oriented database features were readily available, so we could concentrate on how to best adapt such features. In adopting NewS, we minimized changes to the NewS semantics and environment, and reused existing functionality as much as possible.

Chapter 10 discussed ease of experimentation in detail. In executing the experiments, the transparent data access of NewS that we preserved contributed to easy experimentation, especially after a database was prepared, since we could reuse existing NewS applications as is. For modifying the architecture to examine possible design alternatives, a modular, encapsulated architecture of the platform keeps required modifications local and easy to

manage.

As for performance level, we said earlier that there is a minimum level of performance necessary for repeated experimentation. Our first prototype of GemStone-based NewS provided performance fast enough to execute those experiments reported in this dissertation. Many existing NewS applications would actually benefit from the GemStone-NewS architecture, even with the kind of performance reported in this dissertation, since their primary requirement is transparent access to a database from an application for ease of use; such transparency is readily supported by GemStone-based NewS. For such applications, performance is not as critical as transparency because batch-oriented data access is often sufficient. However, for the platform to evolve into a "field-testable" system with a performance close to a production system, performance improvement is definitely needed.

In summary, GemStone-based NewS was evaluated favorably in all of the criteria except for performance level. There is a lot to be desired on its performance, though we contend that our first prototype provided reasonable performance for an initial attempt; with PKB data, the access time was not far off from a time frame of typical analysis. We will discuss possible performance improvements later as part of future work.

11.1.2 Suitability for Various Investigations

Chapter 5 identified design dimensions we would be interested in investigating on an experimental platform. This section analyzes how suitable GemStone-based NewS is for investigations in each of these areas.

Type Definition Facility

With GemStone-based NewS, the type definition facility is essentially that of the current NewS due to our decision to preserve the NewS environment and existing NewS applications. A type-definition facility of an underlying object-oriented database is not accessible from users. That design decision was based on our initial interests in characteristics of existing scientific applications rather than a definition facility for scientific data types. In NewS, a fixed set of types is provided as a constructor, e.g., a vector, an array, and one can use them to construct arbitrary application types. It is conceivable to change the NewS type definition facility without disrupting existing applications, e.g., adding new constructor types. However, existing generic NewS functions, e.g., `print`, which consist of a list of case statements for all possible argument types, must be modified to work on newly added types. (Note that with the later version of NewS, addition of a new constructor type is much easier since one only needs to add a new method for it.) Hence, the platform is fairly limited in a way a type-definition facility is varied. In order to explore an adequate type definition facility, one needs a platform constructed with a different design principle. For example, one can construct a type definition facility from a scratch, experimenting with different constructor types. Another, more manageable approach is to introduce object-oriented features provided in an underlying database to the type definition facility of the language, e.g., an explicit syntax for class definition and inheritance.

Implementation of State

As mentioned before, application data types are defined with a fixed set of "constructor" types provided in NewS. (One consequence of this design is that there is no need to modify existing NewS applications to run on the platform.) We designed GemStone classes that represent the NewS constructor types. GemStone schema for NewS includes a class for a generic vector, but each mode is represented by a separate subclass of the generic vector class, making it possible to customize representation for each mode. Hence, NewS data types are represented in GemStone according to how they are defined with the constructor types. It is relatively easy to vary GemStone representation for the NewS constructors. Instead of modifying GemStone classes, we can use inheritance to represent such different representations as subclasses of a common superclass, an approach particularly useful when different representations are variations of a common basic design. It is more difficult to differentiate a representation for NewS data types that are constructed the same way. For example, it would be difficult to represent type T1, which is an array of arrays of integers, with one GemStone class C1 and represent type T2 of the same structure with another GemStone class C2 simultaneously.

Possible implementation for each NewS constructor is constrained by a set of representation structures provided in GemStone. GemStone represents most constructs as an object. It is conceivable to encode arbitrary values in bytes as byte objects are provided in GemStone, though with such a user-defined representation of the values, one must also provide

all the linguistic support expected with those values explicitly. Note that other object-oriented databases provide more flexibility in terms of representation structures, e.g., an arbitrary-structured "value" without an identity separate from an object in O2.

Structural variations in organizing state, e.g., different choices of top-level instance variables, are easy to investigate with GemStone-based NewS, as GemStone does not dictate how object state should be structured, leaving the decision up to each designer.

Implementation of Operations

A programming paradigm used to represent S functions is the OPAL semantics provided by GemStone. Since OPAL is a computationally-complete language, it is possible to implement the same behavior in different ways. With one-to-one mapping between a NewS function and a GemStone function class, variations in implementation of a NewS function are directly represented through differentiation of a corresponding GemStone function class. As with different representations of state, inheritance can be effectively used to represent such variations as a set of GemStone subclasses. Each implementation of a function requires creation of a new GemStone class, with understanding of GemStone representation for NewS objects and operational semantics of the function. However, it is not necessary to modify applications or to re-compile the NewS interpreter to examine different database implementations of the same function, since difference in implementation is effectively encapsulated in GemStone objects.

Queries on Types

With GemStone-based NewS, a query definition facility is provided by NewS, and queries are essentially NewS expressions. Though all the NewS applications, regardless of their domains, share the same query syntax, it is possible to semantically tailor the queries by providing domain-specific functions. Provision of domain-specific query syntax would require modification or extension of the NewS language. NewS also encourages object-at-a-time computation as opposed to explicit iteration through its elements. Hence, queries are expressed at a large granularity in NewS and easy to translate into equivalent queries on underlying database structures.

The current architecture of GemStone-based NewS allows for investigation on query processing to some extent. As shown earlier, dynamic query optimizations can be explored by having GemStone methods accumulate function calls and perform optimization on the accumulated calls. Auxiliary access structures such as index can be also explored, though with some effort. GemStone does not provide index for ordered collections, so the use of indexes requires implementation of index as a user-defined class, or encoding of arrays into sets, for which index is currently available. Note that the general architecture of the platform, i.e., a combination of a persistent programming language and an object-oriented database with a well-defined interface between them, offers a potential as a vehicle for investigation of query processing, by having whole query expressions transferred from the language to be processed in a database.

Other Data Type Support

Investigation of other data type support, such as a use of generalization hierarchy and type evolution, is limited on GemStone-based NewS due to indirection between scientific data types and database support. Inheritance is not directly available in the version of NewS used in our study for representing a taxonomy of scientific data types, and we did not introduce GemStone-like generalization explicitly to investigate its potential for NewS data types. (The generalization hierarchy of GemStone was used instead to come up with modular, reusable designs of GemStone classes for NewS data types.) As NewS is in a process of incorporating more object-oriented features, there is a potential for explicit inheritance in the future version of NewS.

GemStone-based NewS supports type evolution as much as permitted in NewS, i.e., through modification and extension of NewS objects and functions. With GemStone representation for NewS type constructors, any modification and extension of application data types possible in the language are automatically supported on the database side without modifying or extending the GemStone classes for NewS. However, schema changes possible in GemStone cannot be easily used to support such type evolution.

As for possible unification of multiple types, merging multiple types on the platform is essentially programming effort that attempts to combine behaviors of multiple functions consistently as NewS types are implicitly represented by constructor functions of their instances. Hence, without modifying NewS semantics, the platform does not allow for investigation of more formal options for unifying types such as union or multiple inheritance.

Note also that multiple inheritance is not directly supported in GemStone. Hence, other object-oriented databases that provide multiple inheritance explicitly are a better choice for its investigation.

Execution of Operations

It is relatively straightforward to adjust the location of operation execution within GemStone-based NewS. When both data and a function definition are stored in GemStone, the function can be executed on the data in a database by sending a message to the appropriate GemStone function object from NewS through the GCI. Alternatively, the data can be transferred to NewS and converted into an in-memory representation, and any NewS function can subsequently operate on them there. Arbitrary policies for choosing the location of operation execution can be encoded into the NewS interpreter as a set of rules, based on which the interpreter decides whether to call a GemStone function object or trigger data transfer to NewS for in-memory computation.

Note also that the discussion provided earlier on query optimization also applies to arbitrary operation execution, i.e., possible dynamic optimizations can be explored not just with queries but in a more general context on the platform.

Data Movement

Various schemes are possible to adjust data loading with GemStone-based NewS. For example, by storing and executing NewS data access functions in GemStone, it is possible to reduce the amount of data transfer from GemStone to NewS. Loading only attributes

of NewS data requires minor, localized changes to the NewS interpreter to create a special mode for "attribute-only" objects [Ken91]. GCI also allows certain variations in implementing data transfer, e.g., different granularities of information communicated by a single function call.

Traditional Database Support

With GemStone-based NewS, available database support we can examine is constrained by capabilities provided by GemStone. Though such support as storage management cannot be varied without modifying GemStone internals, some capabilities, e.g., transactions and locks, are provided as predefined GemStone methods, which NewS can trigger through the GCI. Hence, in some cases, it is possible to customize database support by varying the way provided methods are called from NewS or modifying the methods in GemStone. For example, one can customize a transaction scheme for an application by varying positions of commits and the way locks are provided, by choosing between implicit and explicit transactions, and so on. Earlier, we presented alternative designs in introducing GemStone concurrency control into NewS, i.e., definition of NewS functions for locking and modification of the NewS interpreter for automatic execution of transactions.

As for providing data views to support multiple applications, GemStone-based NewS provides much more potential than current, file-based NewS where no functionalities are provided for other applications to access NewS objects. Our schema stores basic values in generic arrays, having GemStone classes for NewS refer to them. The design can be easily extended to include GemStone classes for other applications that refer to the same arrays

for actual values. Operations on the types can be incorporated in GemStone classes for each application, as done for NewS in our study, or if multiple applications share certain operations, those operations can be incorporated as methods of the array class for the values.

Dynamic Properties of Applications

GemStone-based NewS provides much potential for examining dynamic properties of applications. Modular, structured design of the NewS source isolates persistent data access to a few procedures, and it is easy to profile the communication between NewS and GemStone by monitoring GCI function calls made by the NewS interpreter and examine database operations separately from in-memory computation. However, for extensive investigation, e.g., repeated runs of entire interactive sessions, the performance of the platform needs to be improved. As mentioned later, performance improvement is possible without major architectural modifications, i.e., through adopting new versions of software and a more powerful hardware platform. Another possibility is to tune the architecture to gain performance at the cost of less modularity.

A Range of Applications

GemStone-based NewS allows for examination of a wide range of existing NewS applications due to a diverse use of NewS in a variety of scientific domains. Besides epidemiology and computational biology examined in this dissertation, NewS is also used in domains such as financial analysis, statistics research, cartography, geology, physics, and social science.

In summary, GemStone-based NewS provides much potential for investigating a wide range of scientific applications and their dynamic properties. The combination of a persistent language and an object-oriented database with a well-defined interface between them makes it easy to observe communication between an application and a database, and operations in a database can be investigated separately from in-memory computation for looking at query processing and potential optimizations in a database. Operation execution schemes with varying locations of execution, possible data movement patterns between an application and a database are also easy to investigate on the platform for the same reason.

The limitations of the platform are largely due to the priority initially put on access to a wide range of existing scientific applications and a use of existing systems such as NewS and GemStone for cost-effectiveness. For example, a desire to use an existing application as is has led to preservation of NewS data types for the platform, so very little can be varied in terms of application type definition, query definition syntax, etc. Available database support is also constrained to some extent by the capabilities provided by GemStone. Representation structures provided in GemStone restrict database implementation of state of NewS type constructors, and only structural variations can be extensively explored. GemStone data language, OPAL, fixes the programming paradigm for GemStone implementation of a NewS function. However, its computational completeness allows for different implementations of the same behavior to be explored. GemStone database support such as storage management cannot be modified, though some variations are possible with features such as transactions and views. Our decision to have GemStone support NewS type constructors constrains GemStone representation of a NewS data type according to how the type is defined with

NewS constructors, and prevents such data type issues as a use of generalization and type evolution from being examined directly.

We justify our platform since its advantages meet our priority, i.e., an access to a wide range of scientific applications and domains. For a fixed data model, NewS seems to offer a reasonable choice with its flexible support for scientific data types; a wide usage and acceptance of NewS in scientific communities partially validate our choice. GemStone was used mainly as a "reference" object-oriented database to judge adequacy of its capabilities for scientific data, and served its purpose in that sense. In order to explore such dimensions as type-definition facility and traditional database support more extensively, one needs different kinds of platform, with a different design principle and priority. For example, one can conceivably construct a new language and a database from a scratch for the platform. Another possibility is to modify the language and database internals extensively within the platform. Both approaches are more costly than our platform, and given a fixed amount of resources, a range of applications may be limited for which design variations are explored.

11.1.3 Evaluation of GemStone Features

Earlier, we discussed in detail advantages of the GemStone data model for flexibility and encapsulation at the object level and resulting easy experimentation. We also noted a lack of desired performance in GemStone access. Such features as structured values for representing recursive NewS objects without indirection and indexing for arrays were also recognized as desirable additions. Below, we describe other GemStone features that influenced our investigation.

Though GemStone schema changes were not used to support type evolution with GemStone-based NewS, we used them extensively as the design of the platform evolved. And we have found that with GemStone, schema change is a little awkward. When a class is modified, there is no way to explicitly initiate recompilation of its subclasses. More flexible schema changes, with recompilation of classes and migration of the instances when possible, would be a useful addition.

GemStone also lacks facilities for providing information on size of physical space occupied by an object. Scientific applications are often associated with very large data, and it is important for users as well as database administrators to be able to know how much space is physically occupied by particular datasets.

In comparing the performance of GemStone-based NewS and UNIX-based NewS, we noted large difference in the size of executables and questioned its impact on the performance. Current GCI functions are provided to GemStone users as object modules, i.e., *.o files. With an object module, the entire library is linked into NewS even when only a few functions are actually used. It would help to provide GemStone-C Interface as library modules, i.e., *.a files. When a library module is linked, only the functions actually called by the application are bound, resulting in an executable of much smaller size.

11.2 Contributions of the Work

The main contributions of this work lie in a collection of various findings described in this dissertation. Among those findings, we consider the following as most significant.

11.2.1 Scientific Data Types

Through an extensive survey of existing applications in various scientific domains, we identified a list of data support requirements for scientific applications. In particular, we claimed that data types should directly represent scientific models, and examined necessary support for such scientific data types, e.g., representation of static and dynamic properties of each type, inter-type relations such as specialization, type-based queries, and evolution of types over time.

11.2.2 Combination of Persistent Language and Object-Oriented Database

Through an extensive use of NewS in a variety of scientific domains, we found that a concept of a persistent language is well-accepted in scientific communities as a conceptual framework for defining and manipulating persistent data, as long as data type support adequate for scientific applications is provided in the language. However, file-based implementation that most current persistent languages are based on showed a lack of robustness and flexibility in manipulation of scientific data.

By replacing files with an object-oriented database, we strengthened data support provided by a persistent language with compatible data model, robust storage management, sophisticated concurrency control, recovery, and so on. Our decision to have an object-oriented database support a language instead of individual applications proved to be cost-effective as a single design and implementation of database classes provided support for a variety of applications, as contrasted to the need for new database classes for each application. This approach also incurred very little disruption at the conceptual level for domain

scientists as it preserved the data model of the language and existing applications, and benefits of object-oriented database were provided indirectly through a conceptual framework familiar to the scientists. Storing data in a database instead of files allowed for flexible, customized design of data representation, direct data query, and a potential for supporting multiple applications. And compared to traditional databases such as relational systems, an object-oriented database offered an advantage due to a flexible data model that provided a direct support for scientific data types through their constructors.

Our study indicated the state of the commercial object-oriented database technology and its use for scientific applications. If the technology is adequate, domain scientists would rather simply adopt an existing database for managing data so that their effort can be focused on algorithmic and computational issues. An object-oriented database showed much promise for scientific applications, in particular its flexible data model, though the technology is not mature yet and there exists a need for improvement, e.g., its performance. Note that most commercial object-oriented databases currently support transparent manipulation of persistent data through its application programming interface (API). However, unlike our architecture, their API's are based on non-persistent languages, e.g., C++. Each system extends the languages differently to come up with their "persistent" version, e.g., the added C++ Collection classes in ObjectStore. Such a lack of standardization results in disparate API's, even if they are based on the same language, and increased effort in porting applications between systems.

11.2.3 Findings in Design and Implementation

In our study, we not only conceived the architecture of a persistent language supported by an object-oriented database but also validated the conceived architecture through construction of a workable prototype. The following are various findings from our experience in designing and implementing the first prototype of GemStone-based NewS.

Direct Support for Different Functional Semantics

Our study illustrated that an object-oriented paradigm is flexible enough to provide relatively direct support for a language with a different computation model. In our design, a polymorphic function was represented as an independent "function" object. Such one-to-one mapping simplified the design, while it was still possible to maintain modularity at the cost of indirection, i.e., actual operations were implemented as methods of argument objects and function objects delegated operations to appropriate methods.

Representation of Data Type Constructors

Our design provided object-oriented database representation for data type constructors of a persistent programming language. Hence, any data type defined in the language automatically had a database representation structured according to how it was defined with provided constructors. This approach was cost-effective with single schema supporting a variety of scientific data types, though it was still possible to customize representation through further subclassing. For example, we defined a class for a generic vector but actually represented different modes as separate subclasses for the generic vector class. This approach allowed

us to customize representation for each mode, e.g., inclusion of those functions specific to the mode as methods.

Modularity of the Design

As we intended to experiment with architectural variations of the platform, its baseline architecture was left very modular, making most subsequent changes local to either the object-oriented database or persistent programming language. We performed a variety of experiments to demonstrate that indeed many kinds of architectural changes can be made locally without affecting other parts of the platform. Such an extreme modularity makes it easy to tune the architecture, though it also impedes efficient performance somewhat. We identified various opportunities where we can trade in a little modularity for improved performance, e.g., caching of certain database information on the language side.

11.2.4 Evaluation of Our Approach

In order to assess whether our study was a productive initial step toward improved scientific data management, we evaluated our platform-based approach at the end. We examined a process from design, implementation of the platform to investigation on the constructed prototype to see whether the platform satisfied desired criteria such as cost-effectiveness and ease of experimentation. We also assessed the suitability of the platform according to how easy it is to investigate a variety of data support issues with its architecture. Such assessment helped us identify the constributions of the work and validated our approach.

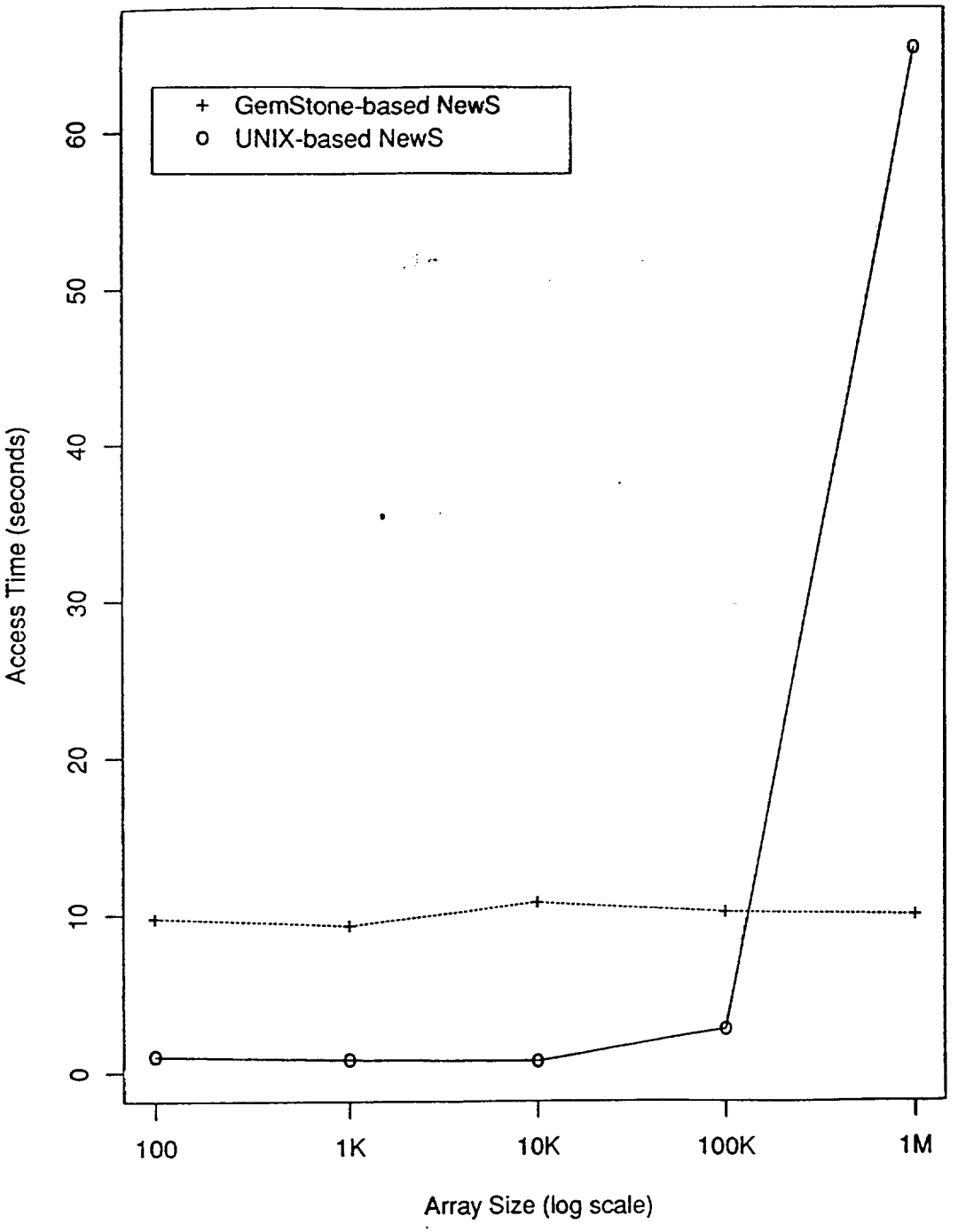
11.3 Future Work

The work presented in this dissertation serves as an initial attempt at improving scientific data management rather than a complete solution. Hence, the work is open-ended with a number of possible future directions. For example, we can continue experimentation on GemStone-based NewS, e.g., with different design alternatives and different NewS applications. For extensive experimentation, though, the platform needs to be improved, especially its performance. It is also conceivable to come up with different platforms that cover different parts of the design space.

As the platform illustrated with NHANES data, GemStone seemed to provide better support for large datasets than current NewS. Hence, one of the possible further experiments is to focus on support for very large data. To get an idea of advantages provided by GemStone storage management, we compared the performance of GemStone-based NewS and UNIX-based NewS when the middle element of an array of integers was accessed [Ken91]. The result from that experiment is shown in Figure 11.1. As shown in the figure, the access time for GemStone-based NewS remains about the same regardless of the size of an array, i.e. about 10 seconds, and when the size of the array exceeds about 10^5, data access becomes more efficient than in UNIX-based NewS.

To improve the performance of the platform, one of the simplest options is to adopt better hardware and software. For example, GemStone version 2.5, a later version than version 2.0 we used, provides improved performance and GCI. Since Servio Corporation discontinued support for DECstations after version 2.0, adoption of version 2.5 also creates

Figure 11.1 Localized Access to Integer Array



an opportunity to port the platform to more powerful hardware, e.g., a Sun SPARCstation. We ran the following OPAL program for array creation and access to get a rough idea of difference in performance between GemStone version 2.0 running on a DECstation 2100 and version 2.5 running on a Sun SPARCstation2.

```
a := Array new: 100.
a at: 1 put: 1.
a at: 2 put: 2.
...
a at: 100 put: 100.
a at: 1.
a at: 2.
...
a at: 100.
```

The program creates an array of 100 elements and initializes value of each element in turn. After the initialization, each element is then accessed in turn. It took GemStone version 2.0 on a DECstation 2100 291 milliseconds to execute the program, whereas it took version 2.5 on a Sun SPARCstation2 62 milliseconds, a little less than 5 times as fast. The Sun SPARCstation2 was running SunOS 4.1.1b with 48 Mbytes of real memory, and it had 2 CDC Wren V disks with total of 1110 Mbytes of available space. The configuration for a DECstation was described in Section 8.1.

GemStone version 2.5 also provides a new and expanded set of GCI functions. For example, current GemStone-based NewS makes one GCI function call for accessing each component of a NewS object. We chose this design because in version 2.0, the facility for simultaneous multiple-component access was not usable due to poor interface and documentation. Version 2.5 provides better facilities for accessing multiple components at a time. Even if each GCI function call is essentially a machine-instruction procedure call with the overhead on the order of milliseconds (since we linked NewS directly into the Gem process), it could still save time to reduce the number of calls by accessing multiple components simultaneously. There is always a possibility of swapping pages between calls, and the number of calls could influence the amount of communication between two heavy-weight processes, i.e., Gem and Stone. GemStone version 2.5 also provides better functions for creation of objects with multiple components.

A new version of NewS, i.e., an August 91 Release, is also available on a Sun SPARCstation. The new version incorporates better memory management and more object-oriented features such as methods in the language, making its data model even more compatible to the GemStone data model.

It should be relatively painless to change to a new version of either GemStone or NewS in upgrading GemStone-based NewS. The GCI of GemStone version 2.5 is a superset of version 2.0, so most of the implementation of the first prototype can be reused. The NewS data access paradigm for persistent data remains the same in the August 91 Release, so we can also reuse the changes made to UNIX-based NewS.

So far, our work has motivated interests in object-oriented database support in the NewS

community, particularly in the environments where a need for better data support is recognized. We described earlier the environment at NCBI where an object-oriented database is being considered for consolidating all kinds of biological data currently distributed among different systems. In particular, ObjectStore is being used on a "trial" basis to explore a potential of object-oriented databases, not unlike our experimental approach with GemStone. Even if it turns out to be impractical to migrate all the data to a single system, an object-oriented database still provides a potential framework for unifying different systems, since its data model is more general than those provided in files or relational databases. An object-oriented database can also customize a view of the data for individual applications with its methods. This case presents an example where advantages of database support, especially from object-oriented databases, likely outweighs performance overhead generally incurred by database access, especially since data can be batch-loaded initially from a database for most applications.

Financial analysis is another example where batch loading of the data is sufficient. A critical data support requirement there is that a database be able to store all the relevant information, from time-series data to mathematical models for the analysis, and to support their efficient access, for which an object-oriented database provides a potential. Much information is corporate data currently distributed among different systems, which must be somehow consolidated. At Celeritas Corporation, a prototype is being built for a financial analysis application that incorporates object-oriented databases and NewS.

Other example cases include a research group at the AT&T Bell Laboratories that is looking into GemStone for storing their Computer-Aided-Manufacturing data to be accessed

from NewS.

In general, we have found that the approach we took of adapting existing components has a strong appeal to domain scientists. If at all possible, they would rather use off-the-shelf components for such tasks as data management so that they can concentrate on their main research interests such as designing alignment algorithms for homologous proteins and building mathematical models for financial forecast. We expect interests in object-oriented databases and their actual use for scientific data management to continue to grow, especially with the improving and evolving technology.

Bibliography

[ABB+87] E. F. Abola, F. C. Bernstein, S. H. Bryant, T. F. Koetzle, and J. Weng. Protein Data Bank. In F. H. Allen, G. Bergerhoff, and R. Sievers, editors, *Crystallographic Databases - Information Content, Software Systems, Scientific Applications*, pages 107–132. Data Commission of the International Union of Crystallography, Bonn/Cambridge/Chester, 1987.

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983.

[ABD+89] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, December 1989.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[AH87] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. *SIGPLAN Notices*, 222(12):430–440, December 1987.

[ALR91] Malcolm Atkinson, Christophe Lecluse, and Philippe Richard. Bulk Types for Data Programming Languages: A Proposal. Technical Report 67-91, Altair, February 1991.

[And89] G. Christopher Anderson. NIH Genome Database Breaks New Ground. *THE SCIENTIST*, 3, September 1989.

[AWSL91] Shamim Ahmed, Albert Wong, Duvvuru Sriram, and Robert Logcher. A Comparison of Object-Oriented Database Management Systems for Engineering Applications. Technical Report R91-12, MIT, May 1991.

[B+77] F. C. Bernstein et al. The Protein Data Bank: A Computer-based Archival File for Macromolecular Structures. *Journal of Molecular Biology*, 112:535–542, 1977.

[B+90] C. Burks et al. GenBank: Current Status and Future Directions. In *Methods in Enzymology vol.183*, pages 3–22. Academic Press, San Diego, 1990.

[Ban87] J. Banerjee. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, 1987.

[BCW88] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The NEW S Language.* Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, California, 1988.

[Bel88] Jean L. Bell. A Specialized Data Management System For Parallel Execution of Particle Physics Codes. In *Proceedings of ACM International Conference on the Management of Data*, pages 277–283, June 1988.

[Bor89] Stu Borman. NLM Efforts Facilitate Access to Biotechnology Information. *American Society for Microbiology News*, 55(10):540–543, 1989.

[Bry89] Stephen H. Bryant. PKB: A Program System and Data Base for Analysis of Protein Structure. *PROTEINS: Structure, Function, and Genetics*, 5:233–247, 1989.

[BSST87] T. L. Blundell, B. L. Sibanda, M. J. E. Sternberg, and J. M. Thornton. Knowledge-based Prediction of Protein Structures and the Design of Novel Molecules. *Nature*, 326(26):347–352, March 1987.

[BW90] Alan J. Bleasby and John C. Wootton. Construction of Validated, Non-redundant Composite Protein Sequence Databases. *Protein Engineering*, 3(3):153–159, 1990.

[CCM88] Raymond E. Carhart, Howard D. Cash, and John F. Moore. StrateGene: Object-Oriented Programming in Molecular Biology. *CABIOS*, 4(1):3–9, 1988.

[CDG+87] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: an Overview. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 474–499. MIT Press, Cambridge, MA, 1987.

[Cre84] Thomas E. Creighton. *PROTEINS.* W. H. Freeman and Company, New York, 1984.

[CS90] Michael Caruso and Edward Sciore. The VISION Object-Oriented Database Management System. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 147–163. Addison-Wesley, 1990.

[ECM88] Binary universal form for data representation. Included in FM 94 BUFR Collected Papers and Specification by European Centre for Medium-Range Weather Forecasts, February 1988.

[EMB91] Information Provided by The EMBL Network File Server, January 1991.

[F+87] D. H. Fishman et al. IRIS: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, 1987.

[FJP90] James C. French, Anita K. Jones, and John L. Pfaltz. Scientific Database Management. Technical Report 90-21, University of Virginia, August 1990.

[Ful89] David W. Fulker. Seminal Software to Analyze and Manage Geoscientific Information (Unidata Building Blocks for the Scientist-Programmer). *Conference on Interactive and Information Processing Systems for Meteorology, Oceanography and Hydrology*, pages 44–49, 1989.

[Ful91] D.W. Fulker. Unidata Strawman for Storing Earth-Referencing Data. *Proceedings of the Seventh International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, pages 210–217, 1991.

[Gou88] Michael L. Gough. *NSSDC CDF Implementor's Guide (DEC VAX/VMS)*. National Space Science Data Center, 88-17, NASA/Goddard Space Flight Center, 1.1 edition, 1988.

[GPKF90] Peter M. D. Gray, Norman W. Paton, Graham J. L. Kemp, and John E. Fothergill. An Object-Oriented Database For Protein Structure Analysis. *Protein Engineering*, 3(4):235–243, 1990.

[GR87] R C Glen and V S Rose. Computer Program Suite for the Calculation, Storage and Manipulation of Molecular Property and Activity Descriptors. *Journal of Molecular Graphics*, 5(2):79–86, June 1987.

[HZ87] M. F. Hornick and S. B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1):70–95, 1987.

[Ike90] Mitsuru Ikei. A Summary of Computer Procedures for Chemistry. An unpublished research note, August 1990.

[Ken91] Brian Kennedy. Architectural alternatives for connecting a persistent programming language and an object-oriented database. Master's thesis, Oregon Graduate Institute of Science and Technology, 1991.

[KN91] Hyung Joon Kook and Gordon S. Novak, Jr. Representation of Models for Expert Problem Solving in Physics. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):48–54, March 1991.

[LMB] John R. Lawton, Frances A. Martinez, and Christian Burks. The LiMB Database. a draft.

[Lot90] Jerry Lotto. An email posting to the news group sci.chem, October 1990.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O2, an Object-Oriented Data Model. In *Proceedings of ACM International Conference on the Management of Data*, pages 424–433, June 1988.

[LWS87] Richard H. Lathrop, Teresa A. Webster, and Temple F. Smith. ARIADNE: Pattern-directed Inference and Hierarchical Abstraction in Protein Structure Recognition. *Communications of the ACM*, 30(11):909–921, November 1987.

[MBW80] J. Mylopoulos, P. Bernstein, and H. K. T. Wong. A Language Facility for Designing Interactive Database-Intensive Systems. *ACM Transactions on Database Systems*, 5(2):185–207, 1980.

[MK89] K. J. Meltsner and G. Kalonji. The Thermodynamics Workbench: A Simulation Support System for Research and Education. In W. T. Thompson, F. Ajersch, and G. Eriksson, editors, *Computer Software in Chemical and Extractive Metallurgy*, pages 45–58. Pergamon Press, New York, 1989.

[MS87] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. MIT Press, Cambridge, MA, 1987.

[Nat89] National Center for Supercomputing Applications. *NCSA HDF Calling Interfaces and Utilities*, November 1989.

[Nim85] G. Nimtz. A New Protein Sequence Data Bank. *Nature*, 318(7):19, November 1985.

[Obj90] ObjectStore Technical Overview. Object Design, Inc., Burlington, MA, 1990.

[OBS86] Peter O'Brien, Bruce Bullis, and Craig Shaffert. Persistent and Shared Objects in Trellis/Owl. In Klaus Dittrich and Umeshwar Dayal, editors, *International Workshop on Object-Oriented Database Systems*, pages 113–123, September 1986.

[Pab87] Carl O. Pabo. New Generation Databases for Molecular Biology. *Nature*, 327(11):467, June 1987.

[Pat90] Mitch Patenaude. An email posting to the news group sci.bio.technology, August 1990.

[PIR91] Protein Identification Resource Newsletter, March 1991.

[PLBO88] Annalisa Pastore, Arthur M. Lesk, Martino Bolognesi, and Silvia Onesti. Structural Alignment and Analysis of Two Distantly Related Proteins: Aplysia Limacina Myoglobin and Sea Lamprey Globin. *PROTEINS: Structure, Function, and Genetics*, 4:240–250, 1988.

[Pon88] Sandor Pongor. Novel Databases for Molecular Biology. *Nature*, 332(3):24, March 1988.

[Pre89] Leonard Presta. Protein Structure Analysis and Development of Databases. *Protein Engineering*, 2(6):395–397, 1989.

[PRO90] Prosite Electronic News Bulletin, March 1990.

[QS88] Ning Qian and Terrence J. Sejnowski. Predicting the Secondary Structure of Globular Proteins Using Neural Network Models. *Journal of Molecular Biology*, 202:865–884, 1988.

[Ray88] David J. Raymond. A C language-based modular system for analyzing and displaying gridded numerical data. *J. Atmos. Oceanic. Tech.*, 5:501–511, 1988.

[Ray91] David J. Raymond. Proposal for NetCDF Algebra. Technical report, Physics Department and Geophysical Research Center, New Mexico Institute of Mining and Technology, Socorro, NM 87801, January 1991.

[RD90] Russel K. Rew and Gleen P. Davis. The Unidata netCDF: Software for Scientific Data Access. *Sixth International Conference on Interactive Information and Processing Systems*, February 1990.

[RR89] Jane S. Richardson and David C. Richardson. The *de novo* Design of Protein Structures. *Trends in Biochemical Sciences*, 14:304–309, July 1989.

[RW88] Marianne J. Rooman and Shoshana J. Wodak. Identification of Predictive Sequence Motifs Limited by Protein Structure Data Base Size. *Nature*, 335(1):45–49, September 1988.

[SF91] Lynn A. Sherretz and D.W. Fulker. Unidata: Enabling Universities to Acquire and Analyze Scientific Data. *Bulletin of the American Meteorological Society*, 69(4):373–376, April 1991.

[SSHG82] Thomas D. Schneider, Gary D. Stormo, Jeffrey S. Haemer, and Larry Gold. A Design for Computer Nucleic-Acid-Sequence Storage, Retrieval, and Manipulation. *Nucleic Acids Research*, 10(9):3013–3024, 1982.

[SST90a] R. Shibata, M. Shibuya, and M. Takagiwa. D & D Examples. Technical Report KSTS/RR-90/002, Keio University, February 1990.

[SST90b] R. Shibata, M. Shibuya, and M. Takagiwa. Data and Description Rule. Technical Report KSTS/RR-90/001, Keio University, February 1990.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[TG89] Janet M. Thornton and Stephen P. Gardner. Protein Motifs and Data-base Searching. *Trends in Biochemical Sciences*, 14:300–304, July 1989.

[Uni91] Unidata Program Center, University Corporation for Atmospheric Research, Boulder, CO. *NetCDF User's Guide*, 1.11 edition, March 1991.

[Ver] Product Profile. VERSANT Object Technology, Menlo Park, CA.

[VER91] VERSANT Object Technology. *VERSANT System Reference Manual*, September 1991.

[Whi91] Mike Whitbeck. A posting to the news group sci.chem, January 1991.

[Wip91] W. Todd Wipke. An email posting to the news group sci.chem, January 1991.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann Publishers, Inc., California, 1990.

Appendix A

Frequency of NewS Library Functions Used in PKB

NewS function	frequency	NewS function	frequency
!	147	ifelse	1
!=	108	invisible	22
$	627	is.atomic	19
&	9	is.character	19
&&	3	is.list	3
*	21	is.matrix	45
+	11	is.na	62
-	18	is.null	27
:	14	is.numeric	8
<	41	len	218
<=	7	levels	20
==	59	lines	7
>	36	list	51
[	326	match	49
^	1	matrix	31
abs	1	max	6
all	6	menu	2
any	83	min	2
apply	14	missing	16
array	3	names	7
as.character	67	ncol	49
as.double	55	nrow	129
as.integer	178	order	12
as.null	3	par	22
as.single	54	paste	114
assign	11	plot	7
attr	15	points	3
c	91	print	69
cat	11	range	1
category	13	rbind	9
cbind	17	rep	42
contour	1	row	12
cos	1	scan	25
cumsum	1	segments	4
encode	3	seq	21
get	45	sin	1

NewS function	frequency
source	1
sqrt	1
stop	145
storage.mode	17
sum	1
sweep	5
switch	10
text	2
unique	18
unix	49
vector	61
|	13
||	144

Biographical Note

The author was born and grew up in Japan. She studied physics at University of Tokyo and earned B.S. in physics. She then came to the U.S.A. to further study physics at M.I.T. During two-year stay at M.I.T. working on first-order phase transitions in polymers toward M.S. in physics, she became interested in computer science, eventually enrolling in the Ph.D. program in the Department of Computer Science and Engineering at OGI. At OGI, she specialized in research on object-oriented systems, databases, and scientific applications. After completion of her dissertation, she is planning to pursue research on computational biophysics, taking full advantage of her inter-disciplinary background.