LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-529

THE DESIGN AND IMPLEMENTATION OF A PARALLEL PERSISTENT OBJECT SYSTEM

Michael L. Heytens

February 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

The Design and Implementation of a Parallel Persistent Object System

Michael L. Heytens

MIT / LCS / TR-[illegible]
February 1988

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988. The author has been supported in part by graduate fellowships from Intel and Teradyne.

The Design and Implementation of a Parallel Persistent Object System

Michael L. Heytens

Technical Report MIT / LCS / TR-529
February 1992

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

It is widely recognized that the expressive power of relational database systems is inadequate for applications that manipulate complex, non record-oriented data. Much recent research has been focused on the design of more expressive database language models that seamlessly integrate the data modeling, abstraction, and general computation of full programming languages with the features of traditional database systems such as persistence, failure recovery, and security. Such additional flexibility gives expressive power to the programmer, but complicates matters for the compiler and run-time system in their efforts to implement database programs efficiently.

In this report we describe AGNA, an experimental persistent object system that we have designed and built that utilizes parallelism in a fundamental way to enhance performance. Parallelism is incorporated into the design of the system at all levels. We begin with an implicitly parallel transaction language that includes a full higher-order programming language and the "list comprehension," a notation similar to SQL but more general. Transactions are compiled into code for a multi-threaded abstract machine called P-RISC, whose central feature is fine grain parallelism with data-driven execution. P-RISC code is emulated on each processor of a MIMD machine with multiple disks. Coarse grain parallelism is used to distribute computations of a transaction over the nodes of a parallel machine, and fine grain parallelism is used within a node to overlap useful computation with long-latency operations such as disk I/O and remote memory accesses.

A prototype of AGNA is operational, running on both a network of workstations and an Intel iPSC/2 Hypercube with thirty-two processors and thirty-two disks. Experimental results demonstrate that parallelism is exploited on both uniprocessor and multiprocessor platforms. Performance of AGNA approaches that of state of the art relational and object-oriented database systems, and relies heavily on compiler optimizations and aggressive pursuit of parallelism.

Key Words and Phrases: Persistent Objects, Functional Languages, Multi-Threaded, Object-Oriented Databases, Parallel Database Systems.

Acknowledgements

First, I would like to thank Rishiyur Nikhil, my thesis advisor, for his encouragement, guidance, and friendship throughout the development of AGNA. I am indebted to Arvind's Computation Structures Group, whose experience and previous research have influenced this work significantly. I would also like to thank David DeWitt of the University of Wisconsin, for giving me access to his Intel Hypercube computer, and for initially getting me interested in parallelism and databases. Rick Rasmussen, the local Hypercube expert at the University, was always willing to answer my questions and handle my bug reports in a timely manner. I am grateful to the MIT CAF Group, especially Duane Boning, Mike McIlrath, Paul Penfield, and Don Troxel for encouragement throughout this work.

I thank DARPA, Teradyne, and Intel for financial support, without which graduate study would not have been possible.

I am grateful to my family for their encouragement throughout graduate school. Finally, I thank my wife, Jill, for her constant love and support, especially during the last year of this work.

Contents

List of Figures

Chapter 1

Introduction

Computerized databases are vital components of a vast majority of today's information systems. Database systems are now used extensively throughout many organizations to provide a uniform and controlled interface to shared, structured information. Example applications that depend critically on database systems include automatic banking, payroll systems, and reservation systems.

1.1 Expressivity Goals

Over the past decade, general-purpose relational database management systems (DBMSs) and their associated database manipulation languages have emerged as robust, practical tools. Prior to the introduction of relational systems, so-called CODASYL and Hierarchical DBMSs were used. In these earlier systems, the programmer had to explicitly navigate through the intricate network of records in the database, while paying careful attention to the order in which records were visited, the key to achieving good performance. In relational systems, on the other hand, the programmer could pose queries in a limited, but high-level, non-procedural language; it was the responsibility of the query optimizer, a part of the DBMS, to select efficient navigation paths for query execution. Relational and other record-oriented DBMSs are now utilized in a wide variety of application systems, and on computers ranging from the largest mainframes to the smallest personal machines.

While relational systems have met the needs of some application systems, the expressive power of the relational model is now recognized to be inadequate for applications that manipulate complex, non record-oriented data. Examples of such applications include those to support computer integrated manufacturing, software engineering, and scientific research. Complex ob-

ject structures common in these areas may be encoded in records of scalar values (the only data structures available in relational systems) just as trees may be encoded in Fortran arrays. However, this encoding obscures the high-level structure of the data, and must be managed entirely by the programmer, thus complicating programming significantly.

Even if all objects were encoded into records of scalar values, the operators available in relational systems for manipulating data provide very limited functionality. For example, one cannot express a general tree traversal using SQL, the standard relational query language. To gain the necessary computational power, one must embed SQL into a host programming language such as C or Ada. Again, this complicates applications programming significantly because the programmer must contend with two incompatible sets of data structures (*i.e.*, those in the programming language and those in the database), two error-handling mechanisms, two sets of control structures, and so on. In Chapter 2, we illustrate the complications that result from the limited expressive power of SQL by examining a graph traversal operation.

In the past few years, much research has focused on the development of more expressive database systems, which can be grouped into three broad categories:

- **Extended Relational Systems.** These systems begin with a traditional relational system, and extend it by adding user-defined procedures, objects, recursion, and other features standard in modern programming languages. Examples of this kind of system include POSTGRES [73], STARBURST [55], and LDL [25].

- **Persistent Programming Languages.** These systems start with the data modeling, abstraction, and control structures of a full programming language, and then add features of traditional database systems such as a query language, persistence, failure recovery, and security. Many of the languages used are based on object-oriented models, *e.g.*, C++ or SmallTalk, thus such systems are commonly referred to as *object-oriented databases* (OODBs). Many research prototypes have been constructed [28, 34, 49, 61, 82], and a number of OODBs have appeared on the market recently [37, 45, 53, 62, 64, 80].

- **Database System Generators.** The goal of systems of this kind is not to provide a complete general-purpose DBMS, per se, but a rich toolkit that will enable a database implementor (an expert systems programmer) to construct quickly a DBMS customized to a particular applications area. Examples of this kind of system are EXODUS [21] and GENESIS [13].

A goal of all three kinds of "third generation" systems (see Figure 1.1) is to increase the productivity of applications programmers by providing a richer, more expressive language model that allows, for example, the modeling of the structure and behavior of complex, real-world objects directly via corresponding objects and procedures in the database. This is in sharp contrast with relational database systems, as described previously, where all objects must be flattened into records of scalar values, and can then be manipulated only in pre-defined ways.

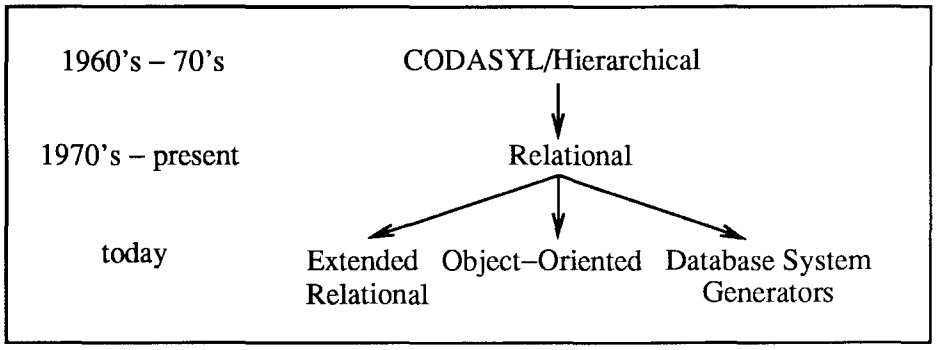


Figure 1.1: Evolution of database management systems.

1.2 Performance Goals

When relational databases were first introduced, they did not provide absolute performance greater than their historical predecessors, and even today, CODASYL and Hierarchical systems often outperform relational DBMSs. Relational databases succeeded in the marketplace because they provided competitive performance while supporting a cleaner, higher-level language model that enhanced programmer productivity. When FORTRAN was first introduced, it did not provide performance better than hand-coded assembly programs. FORTRAN was ultimately successful because, as with relational databases, it provided competitive performance and a higher-level language.

Similarly, it is not necessary for third-generation database systems to provide better absolute performance than RDBs on relational queries, but only competitive performance. More expressive systems can't expect to beat RDBs on such queries, because of the static, regular nature of relational data structures, and the small set of operations that they support. This allows detailed planning of data layouts in secondary storage, and the construction of efficient indexes, all of which are exploited heavily in query optimization.

The additional flexibility of third-generation database systems gives expressive power to the programmer, but complicates matters for the compiler and run-time system in their efforts to implement database programs efficiently. This complication stems directly from the wider range of object structures and operations with which these systems must contend. For example, optimization of a transitive closure operation over a network of module objects in a software engineering system (say, to mark them "out of date") is more challenging than a query in a relational system that finds all records in a set with a particular field value.

1.2.1 Parallelism

It may be possible for an expressive database system to achieve good performance by exploiting parallelism. Broadly speaking, parallelism may be used in two orthogonal ways:

- **Inter-transaction parallelism.** Here multiple transactions are executed concurrently to increase system throughput, *e.g.*, the number of transactions executed per second. Example applications that could benefit from this kind of parallelism are those for banking, reservations systems, and securities exchange. The main research issue in exploiting inter-transaction parallelism is minimizing the resource conflicts that arise between transactions. For example, special-purpose locking schemes are used to reduce the time that locks are held, and data and computation are mapped to the nodes of a parallel machine in a way that lessens the likelihood of excessive contention at a given node.

- **Intra-transaction parallelism.** Here parallelism is used within a transaction to decrease its execution time. Example applications that could benefit from this kind of parallelism are those for scientific research, national defense, CAD/CAM, and analysis of financial data. Major research issues in exploiting intra-transaction parallelism include identifying which computations in a transaction can be safely executed in parallel, and developing compilation techniques and efficient run-time support (synchronization, communication, resource management, *etc.*) to exploit this parallelism on the target machine.

Use of inter- and intra-transaction parallelism in database systems is not new. Inter-transaction parallelism is exploited by Tandem NonStop SQL to achieve linear growth of throughput from 14 to 208 Debit Credit transactions per second as the hardware is increased from 2 to 32 processors [74]. (A Debit Credit transaction is a simple transaction that manipulates several records in a database of banking information [1].) Intra-transaction parallelism is

exploited by Gamma [33], Tandem [35], and Teradata [75] to achieve roughly linear speedup on relational queries that have sufficient parallelism. Relational languages such as SQL are declarative and highly parallel, but as described previously, they lack adequate expressive power, and thus must be embedded in a host programming language. While this gains the necessary expressive power, it limits parallelism in two important ways. First, parallel execution does not extend to the non-SQL parts of the program, *i.e.,* those written in the host language. For complex applications, this may be a significant part of the program. Second, independent SQL statements in the program, even if part of the same transaction, may not execute concurrently. This is due to the sequential nature of host languages into which one may embed SQL (Fortran, C, Cobol, *etc.*), and the lack of asynchronous query execution facilities.

In order to exploit intra-transaction parallelism, the transaction language must be able to express parallel computations, either explicitly or implicitly. The explicitly parallel approach to programming languages involves extending a sequential language with explicit constructs for parallelism, such as a threads package [16, 83] or parallel loops [40]. While this approach is popular, it is difficult to express massively parallel programs using these paradigms. For example, explicitly specifying fine grain, low-level forms of parallelism such as overlapping disk I/O and communication with computation can be very tedious and error-prone. Also, it is difficult to avoid writing programs that contain race conditions, which complicate debugging significantly.

There are two common approaches to programming using implicit parallelism. The first is an outgrowth of research on vectorizing compilers [51] and involves sophisticated *dependence analysis* of a sequential program in order to relax the original sequential semantics safely. While this approach offers an attractive model to programmers, it does not appear to be feasible to extract much parallelism from such programs, particularly in an object-oriented system [41, 42, 54].

The second approach to implicit parallel programming is to use a high-level *declarative* language such as a logic or functional language. By their very nature, declarative languages do not specify a detailed order of execution, leaving the compiler great latitude in choosing one. Studies have shown that compilers for declarative languages can effortlessly extract orders of magnitude more parallelism than is possible with traditional, sequential languages [3]. This approach is very attractive, offering the combined benefits of a high-level language and abundant parallelism.

1.3 AGNA

In this report we describe AGNA, an experimental persistent programming language that we have designed and implemented to investigate the use of parallelism in an information management system. AGNA supports a declarative transaction language that includes a full higher-order programming language, objects, a query language that is similar to SQL but more general, and single-assignment semantics for update. AGNA transactions are compiled into code for a multi-threaded abstract machine called P-RISC ("Parallel RISC"), whose central feature is fine grain parallelism with data-driven execution.

We are targeting AGNA to MIMD machines consisting of processor-memory elements (PMEs) interconnected via a high speed network (see Figure 1.2). Each PME contains a processor, some local memory, and a disk. After compilation of a transaction on the front-end machine, the user may download the compiled code into the back-end machine and execute it via issuing to the command interpreter "load" and "run" commands, respectively. P-RISC code is executed via an emulator program, a ***single*** copy of which runs continuously on each PME; all fine grain P-RISC threads, possibly from different transactions, may execute in the same emulator process.

A prototype of the AGNA system has been developed on workstations interconnected via a local area network. We have also ported our software to an Intel iPSC/2 Hypercube with thirty-two processors and thirty-two disks. More detailed descriptions of the AGNA software and the two hardware platforms are given in Chapter 5.

Our focus in the current work has been on exploiting intra-transaction rather than inter-transaction parallelism, because we feel that it is the more challenging and less understood form of parallelism. Also, as more expressive database languages become available and database systems are used in a wider range of complex applications, decreasing the response time of single, large transactions will become increasingly important. While we have focused on intra-transaction parallelism, nothing in the AGNA language, run-time system, or target architecture precludes exploitation of inter-transaction parallelism. In fact, the simulations performed by Trinder in his thesis work [77] suggest that parallelism can be used effectively in a database system and language such as ours to enhance system throughput.

The expressive power of AGNA's transaction language is comparable to the expressive power of languages used by other third-generation database systems. The transaction language is

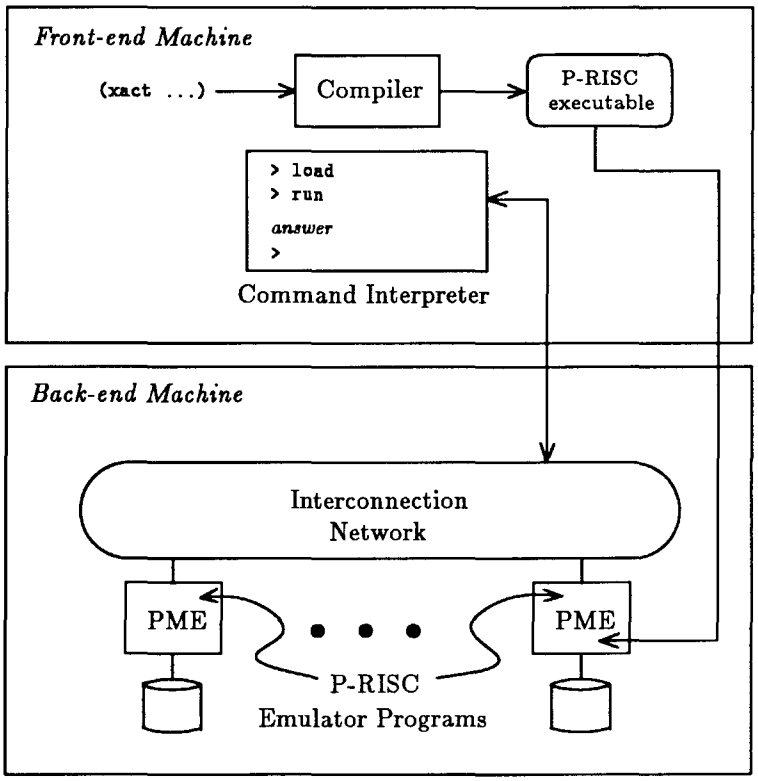


Figure 1.2: AGNA system structure.

similar to OODB languages in that it is based on a full programming language, and objects: (1) have unique identities independent of their field values; (2) may contain embedded references to other objects; and (3) may be shared. Many OODB languages support additional features such as methods, inheritance, and automatic versioning of objects. While these features provide no new, fundamental expressive power, they do make persistent programming easier. We could certainly add these to AGNA, but we have chosen to focus on language semantics and efficient parallel implementation, rather than on advanced language features.

Perhaps the most fundamental difference between AGNA and other third-generation database systems is AGNA is based on a language that is inherently parallel, while most of the others are based on languages that are largely sequential. In the table shown in Figure 1.3, we have categorized third-generation database systems according to the amount of parallelism present in the semantics of their underlying languages.

Sequential. Most OODB systems fall into this category, since the languages on which they are based (*e.g.*, C++, SmallTalk, Lisp) are tied inextricably to an imperative, sequential model of execution. These languages offer attractive models for programmers, but as described previ-

Sequential	*Semi-Parallel*	*Fully Parallel*
GemStone [28]	EXTRA/EXCESS [23]	AGNA [60]
O_2 [61]	FAD [10]	LDL [25]
Objectivity [62]	Iris [82]	
ObjectStore [53]	POSTGRES [73]	
Itasca [45]	Starburst [55]	
other OODBs		

Figure 1.3: Parallelism in third-generation database languages.

ously, it is difficult to extract much parallelism from them.

Semi-Parallel. Systems in the second column retain much of the declarativeness of the relational model and thus are more amenable to parallel implementation. Several of these systems [18, 38, 72] (and the parallel relational systems cited earlier) have effectively exploited three kinds of coarse-grain, intra-transaction parallelism:

- **Producer-consumer parallelism.** The producer and consumers of a stream of data may execute concurrently. For example, a join operation that consumes a stream of data may overlap execution with the operation producing the data.
- **Independent-operator parallelism.** High-level operators which do not have dependencies between them may execute in parallel. For example, two selection operations which produce the operands of a join may execute concurrently.
- **Intra-operator parallelism.** Certain high-level operators may be suitable for parallel execution. For example, many parallel algorithms for the relational join operation have been developed [12, 33, 50, 68].

These forms of parallelism, and the associated run-time model of high-level operators which communicate solely via streams of data (which do not contain inter-object references), are most effective on simple, data-intensive programs. Such programs are relatively easy to partition into high-level operators, and since the total number of operators available is generally small (*e.g.,* less than one hundred), it is feasible to hand-code their implementations for maximum parallelism.

It is difficult to see how the compilation techniques and run-time model used by current parallel database systems can be extended easily and applied effectively to more sophisticated transactions, *i.e.*, those written in full programming languages, and executed against databases containing networks of complex objects. Two difficulties are apparent. First, sophisticated transactions may utilize thousands of small operators or database procedures, thus it becomes less feasible to hand-code each procedure for maximum parallelism. Second, the run-time model must be generalized to include a persistent heap structure to support inter-object references; this is a significant, fundamental change to the storage model.

Another important issue is the amount of parallelism that can be extracted from these "semi-parallel" languages. While the three forms of parallelism mentioned above are important, we feel that many more sources, including fine grain parallelism, must be identified and exploited if a wide range of sophisticated transactions are to effectively utilize large-scale parallel machines. Parallelism in systems in the second column of the table in Figure 1.3 is constrained significantly in the following ways:

1. Updates are completely imperative, and thus for the languages to be well-defined, the sequence of evaluation must be specified for arguments to procedures, expressions in blocks, *etc.*. For example, if the relative order of execution of two expressions which read and write the same data item are not specified by the language semantics, then different results may be produced, depending on the run-time order in which the read and write are executed.

2. Much of the expressive power in these systems comes from database procedures or operators written by the user in a sequential language such as C, which makes it is difficult to exploit intra-procedure parallelism.

Thus, even if the compilation techniques and run-time model of current parallel database systems could be extended to handle complex transactions, it is not clear that significant parallelism can be extracted from their languages because of the two forms of sequentiality listed above.

Fully Parallel. Systems in the third column in the table of Figure 1.3 utilize high-level, declarative languages: AGNA is based on functional languages, while LDL is based on logic languages. Both AGNA and LDL include declarative models of update, so updates do not limit parallelism, and even update transactions may be executed with a high degree of parallelism. Both contain full programming languages, so user-defined procedures are written in a fully

parallel language. AGNA is based on functional instead of logic languages, mainly because we understand better how to incorporate features such as types, objects, higher-order functions, *etc.* into a functional language [43, 56, 57, 78], and how to compile functional languages for parallel machines [59, 76]. It is of course possible that readers more familiar with logic languages will see an application of some of the ideas presented in this report.

AGNA is, to our knowledge, the first parallel implementation of an expressive persistent object system. As we shall see in Chapters 4 and 5, the compiler and run-time system pursue parallelism very aggressively. Unlike previous parallel database systems, our compilation and implementation techniques handle transactions written in a full programming language. AGNA includes a novel, distributed heap divided into volatile and persistent parts, so networks of complex objects may be manipulated and stored in the database.

Experimental results given in Chapter 6 demonstrate that the performance of AGNA on simple queries from the Wisconsin Database Benchmark [17], a standard set of relational queries, approaches that of state of the art uniprocessor and multiprocessor relational systems. Parallelism is exploited in AGNA on the multiprocessor platform to achieve near-linear speedup and scaleup (*i.e.*, increasing the number of processors and database size proportionally while maintaining a constant response time) of performance on transactions that have sufficient parallelism. Additional results given in Chapter 6 demonstrate that the uniprocessor performance of AGNA approaches that of state of the art object-oriented systems on more complex queries (*e.g.*, a transitive closure operation) taken from the Engineering Database Benchmark [24], and that both AGNA and OODBs provide significantly better performance than RDBs on such queries. Even on a uniprocessor platform, the results show that parallelism is exploited in AGNA by overlapping useful computation with disk I/O, thus mitigating the effects of long-latency disk transfers.

1.4 Outline of Thesis

In Chapter 2, we give an informal description of our declarative transaction language and discuss its expressive power and suitability for parallel implementation. In Chapter 3, we give a formal operational semantics of the language by describing, via a number of rewrite rules, how a transaction is reduced to a value. The rewrite rules also describe how an update transaction forms the new state of a database, based on the old. Additionally, the rules demonstrate

very clearly the abundance of fine grain parallelism present in the language. In Chapter 4, we describe the compilation of AGNA transactions into P-RISC code. We also describe a number of significant optimizations performed on database queries, some of which are borrowed from relational databases, and some of which are specific to our parallel implementation. In Chapter 5, we describe how the P-RISC abstract machine is implemented on a network of workstations and an Intel iPSC/2 Hypercube with thirty-two processors and thirty-two disks. In Chapter 6, we describe the results of a preliminary performance evaluation of both uniprocessor and multiprocessor versions of AGNA. In Chapter 7, we conclude with a summary of the present work, a comparison with related work, and directions for future research.

Readers interested in quickly skimming the thesis may wish to skip directly to Chapters 6 and 7, which contain analysis and conclusions, respectively. All of the details of the language, its compilation, and implementation can be found in Chapters 2 through 5.

Chapter 2

The Agna Transaction Language

In order to exploit intra-transaction parallelism, the language must be able to express parallel computations, either explicitly or implicitly.

The explicitly parallel approach to programming languages is a popular one, and involves extending a sequential language with explicit constructs for parallelism, such as a threads package [16, 83] or parallel loops [40]. We decided not to pursue this approach for two reasons. First, it is difficult to express massively parallel programs using these paradigms. Some of the kinds of parallelism we wish to exploit are very fine grain and low-level, such as overlapping disk I/O and communication with computation, and dealing with these details explicitly can be very tedious and error-prone. Second, it is difficult to avoid writing programs that contain race conditions, which complicate debugging significantly.

There are two common approaches to programming using implicit parallelism. The first is an outgrowth of research on vectorizing compilers [51] and involves sophisticated *dependence analysis* of a sequential program in order to relax the original sequential semantics safely. We decided not to pursue this approach because, while it is certainly an attractive model for programmers, it does not appear to be feasible to extract much parallelism from such programs, particularly in an object-oriented system [41, 42, 54].

The second approach to implicit parallel programming is to use a high-level *declarative* language. By their very nature, declarative languages do not specify a detailed order of execution, leaving the compiler great latitude in choosing one. This is the approach that we chose to pursue, as the combined benefits of a high-level language and massive parallelism are very appealing.

Two varieties of declarative languages are common in the literature: logic languages and

functional languages. AGNA is based on functional languages, mainly because we understand better how to incorporate features such as types, objects, higher-order functions, *etc.* into a functional language [43, 56, 57, 78], and how to compile functional languages for parallel machines [59, 76]. It is of course possible that readers more familiar with logic languages will see an application of some of the ideas presented here.

We begin this chapter with a high-level view of our database system as an object repository that responds to incoming transactions. Next we describe the base language, which is a full, higher-order functional programming language. We then describe our object model and how the declarative framework is extended to include updates using a single-assignment semantics, permitting even update transactions to be executed with a high degree of parallelism. Finally, we describe a high-level query notation called "list comprehensions," and compare the AGNA transaction language to SQL, the standard query language for relational database systems.

2.1 Databases and Database Systems in AGNA

To the functional language core on which the transaction language is based we add two fundamental features: a *persistent storage class* and *transactions*. The persistent storage class contains objects whose *lifetimes* exceed those of the transactions that created them. In other words, when a transaction that creates a *persistent object* completes, the object remains in the persistent store, where it may be accessed by other transactions. Other common object lifetimes include the duration of a procedure and the duration of a transaction. Objects with lifetimes of the first kind are often stored in a stack-based activation record in conventional language implementations, while objects of the second kind are generally stored in a volatile *heap*. Such objects are termed *ephemeral* because of their relatively short lifetimes and their inaccessibility outside of the transactions that created them.

For convenience and modularity, the persistent store is logically structured into *databases*. In AGNA, a database is a persistent environment of bindings that associates names with types and objects. The objects reside in a persistent heap and may be of any type— scalars, complex objects, lists, procedures, *etc.* Objects in the database include all and only those that are "reachable" from the persistent environment, *i.e.,* those bound directly to names in the environment, and those that are accessible through inter-object references (as we shall see in Section 2.3, objects may contain embedded references to other objects.) Persistence in AGNA

is completely transparent to the programmer—he does not have to explicitly write an object to the persistent heap, nor does he have to read it into main memory from a persistent storage medium. Such tasks are performed implicitly by the system.

A transaction `(xact ...)` is a construct executed in a database environment, which may contain definitions of new types, definitions of new bindings, declarative update specifications, and *queries* (expressions to be evaluated). For example, a database for a university, shown in Figure 2.1, might be queried with the following transaction to find the address of John Smith, a student:[1]

```
> (xact
    (student-address (student-with-name "John Smith")))
```

Procedure `student-with-name` is used to locate the desired student object, and procedure `student-address` is used to access the address. Top-level identifiers `student-address` and `student-with-name` are looked up in the database environment.

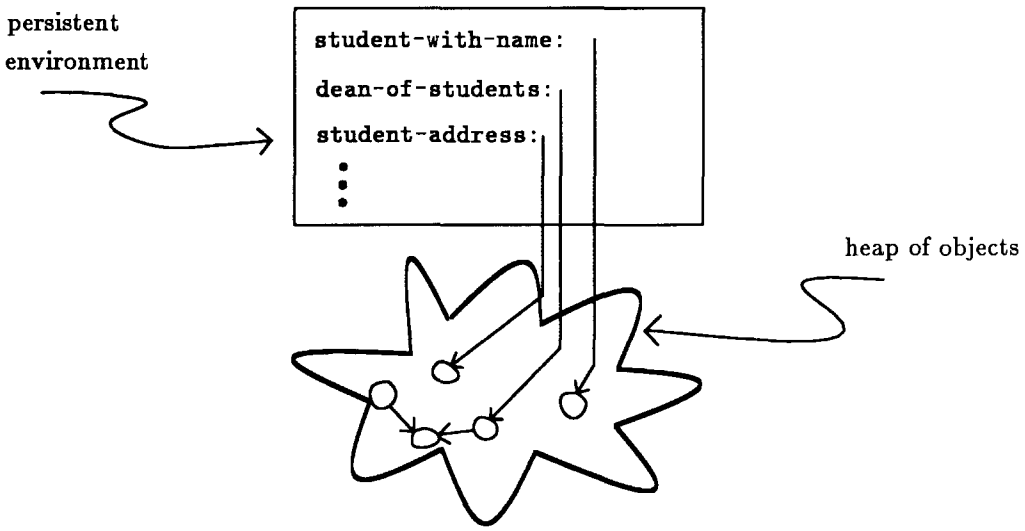


Figure 2.1: Database for a university.

Transactions are *serializable* and *total*. Serializability means that multiple transactions may execute concurrently in the same database environment, but the effect on the database is as if they were executed in some serial order. All programs in AGNA are structured into transactions, so concurrent execution is well-defined. Totality means that transactions either execute entirely

[1] We use a simple Lisp-like notation to avoid detailed syntax design. While some readers will undoubtedly find this unattractive, our focus is on the underlying language semantics and expressive power, not on designing elegant syntax. If necessary, our language can be viewed as a back-end language for one's favorite user-interface.

or not at all.

The execution of transactions in AGNA is coordinated and managed by a *database system*, which contains a database and accepts a list of incoming transactions. Again, transactions in the list are executed in some serializable order. Conceptually, the behavior of such a system can be modeled by the following function:[2]

```
(define dbsystem
  (lambda (db transactions)
    (letrec ((result (dbeval (hd transactions) db)))
      (cons (reply result)
            (dbsystem (new-db result) (tl transactions))))))
```

In words: the database system is a function that takes a database and a list of transactions as input, and produces a list of replies as output. Function `dbeval` evaluates a transaction relative to a database, and produces a composite result consisting of a (possibly) new database and a reply (*i.e.*, the value of the transaction). The new database is used in the evaluation of the next transaction.

If transactions come from multiple users, then somehow they must be merged into a single input list, and replies in the output list must somehow be routed back to the appropriate source. The exact manner in which this is accomplished, and the possible user interfaces to a database system such as ours, are outside the scope of this dissertation.

2.2 Base Language

In this section we present the base functional language informally through a series of examples. Readers familiar with functional languages may wish to omit this section. A grammar of the complete transaction language is given in Appendix A.

Functions

The most important type of object in the base language is a function, which is created via a `lambda` expression. For example, the following expression evaluates to a function that takes two numbers as arguments, and returns their sum:

```
(lambda (x y) (+ x y))
```

[2]This is not an essential characterization, though; see, for example, [11] for other possibilities.

The formal parameters of the function are `x` and `y`, which are added together by the body expression `(+ x y)` to form the result of the function. A function may be applied by enclosing it in parentheses along with its arguments. For example, the preceding function may be applied to arguments `3` and `4` as follows:

```
((lambda (x y) (+ x y)) 3 4)
```

Functions may be named, for convenience, in a variety of ways. For example, the function above may be named and added to the database as follows:

```
(xact
  (define plus (lambda (x y) (+ x y))))
```

Here `define` introduces into the database environment a new top-level binding of the name `plus` to the procedure object returned by `lambda`. After this binding is defined, the name `plus` may be used in expressions as follows:

```
(plus 3 4)
```

Free identifier `plus` is looked up in the database environment, and the procedure to which it is bound is applied to arguments `3` and `4`.

We may wish to establish a binding of a name to an object only during execution of a single transaction. This is accomplished via `define-local`. For example, the following transaction binds local name `fact` to a procedure which computes factorials:

```
(xact
  (define-local fact (lambda (n)
                       (if (<= n 1)
                           1
                           (* n (fact (- n 1))))))
  ...)
```

The name `fact` is only available in the body of the transaction, *i.e.*, inside the `lambda` expression and in "...", the remainder of the transaction body. It is important to note that the binding is *not* added to the database. `Fact` utilizes the conditional expression `if`, which is a special form that first evaluates the predicate (*i.e.*, `(<= n 1)`) to a boolean value, and then evaluates and

returns the value of the "then" (constant `1`) or "else" (the multiplication) branch.

We may wish to restrict the scope and lifetime of a binding even further. This is accomplished via a `letrec` block. For example, an alternative definition of `fact` is:

```
(define-local fact (lambda (n)
                     (letrec ((f (lambda (counter product)
                                   (if (<= counter 1)
                                       product
                                       (f (- counter 1) (* counter product))))))
                       (f n 1))))
```

Here `letrec` is used to introduce a local binding of `f` to a function that computes factorials in a manner different from the previous method. Function `f` computes a factorial by iterating `counter` times, multiplying the counter by a running product during each iteration. The scope of `f` includes only the inner `lambda` expression and the body of the `letrec`, *i.e.*, `(f n 1)`. The body expression utilizes `f`, applying it to `n`, the number of the factorial to compute, and `1`, the initial product. The result of the body expression is the result of the `letrec`.

Lists

A commonly used data object in AGNA, as well as other functional languages, is a list. For example, a list `l` containing the numbers `1` through `5` is shown in Figure 2.2. Each cell in the list consists of two components: a *head*, which contains a number in this case, and a *tail*, which points at the remainder of the list. The tail of the last cell contains `nil`, the empty list, which is depicted as a diagonal line in the figure.

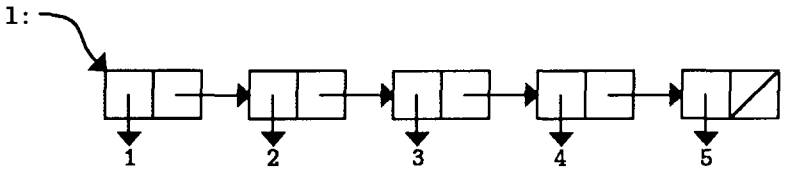


Figure 2.2: List of integers from 1 to 5.

Lists are constructed with `cons`, which takes a head and tail as arguments, and returns a new list. The following procedure, for example, constructs a list of integers in a specified range:

```
(define ints-from (lambda (from to)
                    (if (> from to)
                        nil
                        (cons from (ints-from (+ from 1) to)))))
```

The result list is constructed in a recursive manner: during each invocation in which `from` is not greater than `to`, the function creates a new list in which the head is the current value of `from`, and the tail is the list returned by the recursive call. When the recursion bottoms out (*i.e.*, `from` is greater than `to`), the empty list `nil` is returned.

The following application of `ints-from` creates the list of Figure 2.2:

```
(ints-from 1 5)
```

Elements of the list may be accessed using functions `hd` and `tl`, which return the head and tail, respectively, of a list. For example, the following expression creates list `l` and returns its second element:

```
(letrec ((l (ints-from 1 5)))
  (hd (tl l)))
```

Local name `l` is bound to the list returned by `ints-from`, and the body expression returns the second element of the list.

As we shall see in subsequent sections of this chapter, list data structures are used heavily in AGNA. For example, lists are a fundamental part of the high-level query notation (*i.e.*, list comprehensions) supported by the transaction language, and are used as a bulk data structure for organizing large collections of persistent objects.

Higher-Order Functions

Functions are first-class objects in the AGNA language which means, among other things, that they may be passed as arguments to other functions and returned as results. A function that takes a function as an argument, or returns a function as a result, is called a *higher-order* function. For example, the following function takes a function `f` and a list `l` as arguments, and applies the function to each element of the list, returning a list of the results:

```
(define map (lambda (f l)
              (if (nil? l)
                  nil
                  (cons (f (hd l)) (map f (tl l))))))
```

As in function `ints-from`, the result list is constructed in a recursive manner: during each invocation in which `l` is not empty (*i.e.*, `(nil? l)` is false), `map` creates a new list in which the head is the result of `f` applied to the current element of `l` (*i.e.*, `(hd l)`), and the tail is the list produced by the recursive call to `map`, which contains the results of `f` applied to the remaining elements of `l`. When the recursion bottoms out, `nil`, the empty list is returned.

Function `map` provides a powerful, high-level operator. For example, we can compute the squares of the integers from 1 to 5 as follows:

```
(map (lambda (x) (* x x)) (ints-from 1 5))
```

Here the squaring function is simply mapped over the list produced by `ints-from`. Function `map` may also be used to select a field value from each object in a collection, in much the same way that a field name in an SQL SELECT statement selects the field from each row in a relation.

Another powerful, higher-order function is `filter`, which takes a predicate function and a list, and returns a list of the elements in the input list that satisfy the predicate. `Filter` may be defined as follows:

```
(define filter (lambda (p l)
                 (if (nil? l)
                     nil
                     (letrec ((x (hd l)))
                          (if (p x)
                              (cons x (filter p (tl l)))
                              (filter p (tl l)))))))
```

The procedure body has the same overall structure as `map`, except that in the "else" branch of the outer `if` the current head of the list (`x`) is added to the result list only if the predicate expression `(p x)` is true. Function `filter` is similar to an SQL WHERE clause in that both may be used to select the data objects from a collection that satisfy some predicate.

Such functions as `map` and `filter` provide high-level list operators that may be composed easily. For example, here is a query that finds the names of all students in the list bound to `senior-class-officers` that satisfy predicate `honor-student?`:

```
(xact
  (map student-name
       (filter honor-student?
               senior-class-officers)))
```

Suppose identifier `senior-class-officers` is a top-level name in the database environment bound to a list that contains students that are officers in the senior class. The list is first filtered, producing an intermediate result list, over which the name selection function `student-name` is then mapped. `Student-name` and `honor-student?` are bound to functions in the database.

Yet another useful higher-order function is `foldr` ("fold right"), which takes a binary accumulating function `f`, an initial value `v`, and a list of values `l` as arguments, and returns an accumulated value. `Foldr` may be defined as follows:

```
(define foldr (lambda (f v l)
                (if (nil? l)
                    v
                    (f (hd l) (foldr f v (tl l))))))
```

During each recursive call, `f` is applied to the value at the head of the list and the value returned by the recursive call to `foldr`. When `l` is empty, value `v` is returned. `Foldr` may be used to sum a list of integers as follows:

```
(foldr + 0 (ints-from 1 10))
```

Here `ints-from` produces a list of integers from `1` to `10`, which `foldr` then sums using `+` as a combining function and `0` as an initial value. The sum is accumulated from the right, or end of the list: `10` is first added to `0`, then `9` is added to the intermediate sum, then `8`, and so on. A similar function `foldl` may be used to accumulate a value from the left.

It is important to note that functions in AGNA such as `map`, `filter`, and `foldr` operate uniformly on both persistent objects (*i.e.*, objects in the database) and ephemeral objects (*i.e.*, objects such as lists used only within a single transaction, and not accessible from the database environment). In other words, `map`, `filter`, `foldr`, and all other functions may access both kinds of objects in exactly the same way, and thus need not distinguish between the two. Similarly, both functions stored in the database such as predicate function `honor-student?`, and temporary computing functions such as `(lambda (x) (* x x))`, are treated uniformly. For example, both kinds of functions may be passed as arguments to `map` and `filter`.

2.3 Object Model

Objects in the language include scalars (such as numbers and booleans), primitive and user-defined functions, lists, arrays, and objects of user-defined types. Objects of any type may be stored in the database.

2.3.1 User-Defined Object Types

User-defined types are introduced into a database environment via the `type` form. For example, here are definitions of `STUDENT`, `COURSE`, and `ENROLLMENT` types.

```
(type STUDENT (extent)
  ((name    <=> STRING)
   (status  => STRING)
   (gpa     => FLOAT)
   (address => STRING)
   (bdate *<=> INTEGER)))

(type COURSE (extent)
  ((name     <=>  STRING)
   (prereqs *<=>* COURSE)
   (units    =>  INTEGER)))

(type ENROLLMENT (extent)
  ((grade     => STRING)
   (student *<=> STUDENT)
   (course  *<=> COURSE)))
```

For each field, the forms define the base type, whether it is single- or multiple-valued (=> or =>*), and whether it supports a single- or multiple-valued inverse (<= or *<=). For example, the student `name` field records a single string value, and supports a unique inverse that maps strings to students. The course `prereqs` field records a collection of course objects, and supports a multiple-valued inverse that maps a course object to the collection of courses for which it is a prerequisite.

A field of an object may be read using the `select` form. For example, if `s` references a student object, then the name field may be accessed by the expression:

```
(select s STUDENT NAME)
```

which evaluates to a string. It is important to realize that `s` is an expression that is evaluated, while `STUDENT` and `NAME` are literals that are *not* evaluated.[3] The reason for the `STUDENT` qualifier is that `NAME` is not unique—the `COURSE` type, for example, also has a `NAME` field. By qualifying each field fully, certain optimizations in AGNA such as determining field offsets at compile-time instead of at run-time, can always be performed.

Selection of a multiple-valued field produces a *bag* (or *multi-set*) of objects, packaged as a list. For example, if `c` references a course, then the expression:

```
(select c COURSE PREREQS)
```

evaluates to a list of course objects. The ordering of elements in the list is not significant.

A `type` form also specifies whether a persistent or "base" extent (*i.e.*, a collection of all objects of the type) is to be maintained automatically by the database system. Since `(extent)` declarations, which are optional, were included in all definitions above, persistent collections of all students, courses, and enrollments are maintained automatically by the system. Thus, whenever a student, course, or enrollment object is created, the system automatically inserts it into the appropriate collection. These collections are available to the programmer via the expressions `(all STUDENT)`, `(all COURSE)`, and `(all ENROLLMENT)`, which evaluate to lists of all students, courses, and enrollments, respectively. Again, the ordering of elements in the lists is not significant.

The rationale for system-maintained base extents is twofold: programmer convenience, and efficiency. It is very common that applications require such collections to be maintained and while the programmer could do this explicitly by inserting each new object into the appropriate extent list, it is more convenient for the programmer if this is done automatically. System-maintained sets are also less error-prone: a programmer, for example, may forget (or not know) to insert a new object into the type's extent. Finally, because base extents are maintained by the system, it is free to choose efficient internal representations. As we shall see in Chapters 5 and 6, scanning and filtering can be performed much more efficiently on such internal representations than on the general list representation.

The programmer, of course, is free to maintain additional collections of objects explicitly. For example, in the university database the top-level name `seminar-courses` may be bound to a

[3] Throughout this dissertation, we use all uppercase letters for such literals.

collection of all seminar courses. When an object is created that describes a seminar course, it must be inserted manually into this collection.

The `invert` form is used to apply inverse-mappings. It takes a type name, a field name and a field value as arguments, and searches the type extent for the object(s) with the desired field value (invertible fields are only allowed in types with a persistent extent, so there will always be an extent to search). The following expression, for example, evaluates to a student object with the indicated name, or a special null object if no such student exists:

```
(invert STUDENT NAME "John Smith")
```

Application of a multiple-valued inverse-mapping produces a collection of objects. The following expression, for example, evaluates to a list of enrollments for course `c`:

```
(invert ENROLLMENT COURSE c)
```

Again, the ordering of elements in the list is not significant.

Specifications of field inverse-mappings are included in the transaction language for both semantic and pragmatic reasons. The only semantic issue is automatic enforcement of uniqueness constraints on <= fields. For example, because of the unique inverse on student name, two student objects may not exist in the database at the same time with the same name. As we shall see in Section 2.4, a transaction that attempts to add to the database a student object with a non-unique name is automatically aborted, and the database is not updated. While it is certainly possible to express such constraints in other ways, it is very convenient to do so in the type declarations.

The pragmatic reason for specification of inverse-mappings is efficiency. As we shall see in Chapter 4, indexes are automatically constructed and maintained for invertible fields, and are used in the implementation of `invert` and list comprehension queries (to be described in Section 2.5). Efficiency considerations such as indexing are very important in persistent systems because databases tend to be large and objects long-lived. We will see in Chapter 6 the dramatic impact indexes have on performance.

2.3.2 Pre-Defined Object Types

Pre-defined object types include `list` and `array`. The list type is specified by the following definition:

```
(type list ()
  ((hd => ANY)
   (tl => ANY)))
```

The empty list is `nil`. Note that because the `extent` keyword is not specified, a collection of all list objects is not maintained automatically in the database. Thus, list objects only become persistent by being referenced from some persistent object. Lists may be manipulated with the following procedures:

```
(define hd (lambda (l) (select l LIST HD)))

(define tl (lambda (l) (select l LIST TL)))

(define cons
  (lambda (x y)
    (letrec ((c (allocate LIST)))
      (update c LIST HD x)
      (update c LIST TL y)
      c)))
```

Procedures `hd` and `tl` select the head and tail field values of a list, and `cons` constructs a new list object[4]. We defer a discussion of `allocate` and `update` to Section 2.4.

Arrays in AGNA are zero-indexed, and come with the following primitive operations:

```
(allocate-array size)

(select-array array index)

(update-array array index value)
```

`Allocate-array` allocates a new, empty array, and `select-array` and `update-array` read and write individual array elements, respectively. Multi-dimensional arrays are constructed by nesting one-dimensional arrays; one could extend AGNA to handle multi-dimensional arrays, but we have not because such constructs are not a focus of this work.

[4] If the body of `letrec` contains multiple expressions, as in the definition of `cons`, then all expressions are executed in parallel, with the value of the textually last expression returned as the result.

2.3.3 Type Checking

Type checking of data objects is performed dynamically at run-time by the operators that manipulate them, and not statically at compile-time. For example, `select` interrogates the type of an object at run-time and reports an error if it does not match the type indicated by the transaction programmer in the `select` form. While static type checking may be desirable in AGNA, our current focus is on issues of persistence and parallelism, which are somewhat orthogonal to type checking. We have chosen dynamic type checking simply because it is well understood and easier to implement. Exploration of the issues involved in building a static type system for the AGNA language is a possible direction for future research.

In a type declaration, only the "top-level" type of a field with a nested structure may be specified. For example, we may wish to include in the student type a field `new-courses` to record the collections of courses (*i.e.*, list of lists of courses) that a student is considering taking the next term. Such a field could be declared as follows:

```
(new-courses =>* LIST)
```

We may store in the field lists of lists of courses (*i.e.*, the collections of courses being considered) but the type checking performed will only ensure that the field value is a list of lists. The type "language" for describing fields (=>, =>*, *etc.* plus the base type name) is simply not capable of describing the entire structure of the field. While it would certainly be desirable to have a type system capable of expressing the entire nested field structure, again, our main focus in this work is on persistence and parallelism and not type systems.

2.4 Database Updates

An update transaction is a declarative specification of the new version of a database, expressed as a function of its current state. Both the top-level environment and objects in the heap may be updated. Conceptually, update specifications are collected during transaction execution, and occur instantaneously at transaction commit time. Thus, updates are only visible to subsequent transactions, and *not* to the transaction in which they were performed.

2.4.1 Changing the Top-Level Environment

The simplest forms of update involve the installation of a new value or type definition in the database environment. We have already seen the `define` form to introduce new value bindings. For example, the following transaction introduces three new value bindings:

```
(xact
    (define honors-student-gpa 3.9)

    (define senior-class-president (invert STUDENT NAME "John Smith"))

    (define student-address (lambda (s) (select s STUDENT ADDRESS))))
```

The first definition binds the name `honors-student-gpa` to the number 3.9. The second binds the name `senior-class-president` to the John Smith student object, while the third binds the name `student-address` to a function that, when applied to a student object `s`, returns its address.

Note that the three expressions producing the values (*i.e.*, `3.9`, `invert`, and `lambda`) associated with these names are each evaluated exactly once, in the environment existing at the time of definition. When these names are used in subsequent transactions, the expressions are *not* re-evaluated in the environment extant at the time. If that is the desired behavior, then the expression must be placed in a procedure, where it will be evaluated each time the procedure is applied.

To avoid the non-determinism of read-write race conditions, changes to the top-level environment are not immediately visible to an update transaction. A type or value definition in transaction T_i only becomes visible from transaction T_{i+1} onward. When we execute a reference to a top-level name during transaction T_i it is always looked up in the database environment prior to T_i (*i.e.*, the old top-level environment). For example:

```
(define x (+ x 1))
```

means: evaluate `x` in the current version of the database, add 1 to it, and bind `x` to this value in the new version of the database. On the other hand, consider:

```
(define length
  (lambda (l) (if (nil? l)
                  0
                  (+ 1 (length (tl l))))))
```

The reference to `length` in the body is not evaluated in the current transaction, since it is inside a procedure. When `length` is applied to some list in some later transaction, it will pick up the correct (*i.e.*, latest) binding.

As the definition of `length` indicates, we have taken the position that top-level names in procedure bodies are looked up in the environment current at the time the procedure is applied. This is also the approach taken by current Lisp systems. An alternative, perhaps equally plausible, position, is that such identifiers in procedure bodies are always looked up in the environment existing at the time the procedure was defined. With this approach, however, there is no way to "track" the most recent binding of an identifier. This means, for example, that when a procedure bound to a top-level identifier is redefined (say, to fix a bug), all callers of the procedure must also be redefined to pick up the corrected version. While this situation could be avoided by using special syntax such as a quote in front of an identifier, indicating that it should be looked up in the environment current at the time the procedure is applied, we feel this introduces unnecessary complexity.

Note that the lookup rule that we have chosen does not preclude this alternative approach. For example, if in the following definition:

```
(define f (lambda (x) (g x)))
```

we always want the current binding of `g` and not some later one that might exist when `f` is applied, we can rewrite the definition as follows:

```
(define f ((lambda (g x) (g x)) g))
```

Here `g` is added as a formal parameter, and the function is partially applied to the value bound to `g` in the current environment. All applications of `f` in subsequent transactions then use this value of `g`. Thus, top-level names in procedure bodies, which are looked up when the procedure is applied, can be looked up when the procedure is defined via this simple transformation.

We have also seen another simple form of update, the `type` form to introduce new type bindings:

```
(type <type-name> optional-extent-spec
  (( field-name  field-spec )
   ...
```

```
( field-name  field-spec )))
```

As with new value bindings, new type bindings are not visible until the subsequent transaction. Types introduced by a transaction may be recursive and mutually recursive. In other words, a field specification in one type introduced by a transaction may include the name of another type introduced by the same transaction. For example, the `prereqs` field in the `COURSE` type described in Section 2.3 is itself of type `COURSE`. Redefinition of types is not allowed, so a type name used in a field specification will either refer to a type introduced by the same transaction or one in the old version of the database, but not both.

A top-level definition may be removed with the phrase:

```
(undefine identifier)
```

Again, the update is not visible until the subsequent transaction.

The `type`, `define` and `undefine` forms are *top-level* phrases, *i.e.*, they cannot be nested inside expressions.

2.4.2 Object Manipulation

A new object may be allocated using the expression:

```
(allocate type)
```

If *type* was declared with an automatic persistent extent, the system also inserts the new object into the type's extent list. The updated extent list is not visible until the subsequent transaction. All fields of the newly allocated object are initialized to a special *undefined* value.

We have already seen the `select` form to read a field value:

```
(select object type-name field-name)
```

A field may be written using the expression:

```
(update object type-name field-name new-value)
```

The semantics of `select` and `update` depend on whether they are applied to a new object (*i.e.*, one allocated by the current transaction), or a *persistent* object (*i.e.*, one in the old version of the database, allocated by a previous transaction).

Selects and Updates on New Objects

Let O be an object of type T allocated in the current transaction. Initially, all fields of O are undefined. Let us focus on a particular field F. In the current transaction, at most one (`update` O T F v), but any number of (`select` O T F)'s may be executed on field F. The relative execution order of these operations is unspecified. If `select` is executed while the field is still undefined (*i.e.*, before `update` has been executed), it is automatically blocked until the field becomes defined. This is known in the parallel computing literature as *I-structure semantics*.

Here is an expression that defines a new course object:

```
(letrec ((c (allocate COURSE)))
  (update c COURSE NAME "Introduction to Algorithms")
  (update c COURSE PREREQS nil)
  (update c COURSE UNITS 12)
  c)
```

The expression allocates a course object `c`, defines its three fields, and returns `c` as its value. Any attempt to redefine a field of `c` in the same transaction is viewed as an inconsistent specification, and causes a run-time error. This *single-assignment* requirement ensures that all readers of the new object see a consistent view of it, *i.e.*, all readers of a field in the transaction receive the same value. If a `select` (outside of the `letrec`) tries to read a field value before it is defined, then it simply blocks. When the corresponding update is executed, all blocked readers are enabled.

All new course objects must satisfy the unique inverse on course name (recall from Section 2.3.1, the definition of `COURSE` includes a unique field-inverse mapping strings to courses). If two new courses have the same name, or a new course has the same name as pre-existing course, then the system aborts the transaction automatically.

Additional constraints may be imposed by the transaction programmer on object field values through explicit use of conditionals and `abort-transaction`, a primitive procedure that aborts the current transaction. For example, the following course constructor adds a constraint on the units field value:

```
(define make-course
  (lambda (name prereqs units)
    (if (or (< units 0) (> units 15))
        (abort-transaction "Value of course units out of legal range")
        (letrec ((c (allocate COURSE)))
          (update c COURSE NAME name)
          (update c COURSE PREREQS prereqs)
          (update c COURSE UNITS units)
          c))))
```

If the units value is out of the legal range (0 - 15), then an error message is printed and the transaction is aborted.

The single-assignment and I-structure semantics described above apply also to objects of pre-defined types such as `LIST` and `ARRAY`. (For arrays, though, `select-array` and `update-array` are used instead of `select` and `update`.)

Selects and Updates on Persistent Objects

Let O be a *persistent* object of type T and, again, let us focus on a particular field F. In the current transaction, as before, at most one (`update` O T F v), and any number of (`select` O T F)'s may be executed. The update, however, is not visible in the current transaction, and occurs only at transaction commit time. The value returned by `select` is always the value of F in the old version of the database. If the old value is undefined, then a run-time error is raised.

Here is an example of a persistent object update that changes the name of student "John Smith" to "John E. Smith":

```
(letrec ((s (invert STUDENT NAME "John Smith")))
  (update s STUDENT NAME "John E. Smith"))
```

The expression utilizes `invert` to locate the desired student object, and then updates its `name` field. The new name is visible only in subsequent transactions; all readers in the current transaction still see the old value (*i.e.*, `"John Smith"`). Also, the new name must satisfy the unique inverse mapping on student name.

For multiple-valued fields, the value of the field is a list of objects. While we may also use `update` on such fields to replace the current value by a new one, it is more often the case that we just want to insert a new member into the list or delete an existing member. These operations are expressed as follows:

```
(insert object type-name field-name new-member)

(delete object type-name field-name old-member)
```

Such operations are only allowed on fields of persistent objects, *i.e.*, not on fields of objects allocated in the current transaction, where `update` must be used. The reasons for this restriction are as follows. Clearly `delete` must be disallowed on a field of a new object because the deletion of an element from the field collection, defined by whatever means, would violate the single-assignment semantics and, therefore determinacy. For example:

```
(letrec ((c (allocate COURSE)))
  (update c COURSE PREREQS eecs-intro-courses)
  (delete c COURSE PREREQS (invert COURSE NAME "Introduction to Programming"))
  (select c COURSE PREREQS))
```

Here a new course is allocated, an `update` and a `delete` are performed on its `prereqs` field, and the prerequisites are returned. The deletion is not allowed in this case because otherwise the value returned by `select` is ambiguous, as it may be the collection of courses bound to `eecs-intro-courses`, either with or without Introduction to Programming. Similarly, insertion of an additional course into the `prereqs` field is disallowed because it also violates the single-assignment semantics.

Definition of a field value in a new object solely via `insert` is also problematic because somehow the field collection must be "closed". Consider the following modification of the previous example:

```
(letrec ((c (allocate COURSE)))
  (insert c COURSE PREREQS (invert COURSE NAME "Introduction to Programming"))
  (select c COURSE PREREQS))
```

Here a new course object is allocated, a course is added to its `prereqs` field, and the prerequisites are returned. The insertion is not allowed because, in general, it is not possible to determine when all insertions have executed (there may be more outside the `letrec`) and thus, when it is safe to return the field collection to `select`.

Multiple `insert` and `delete` operations may be performed in a single transaction. As usual, the effects of these operations are not visible until the subsequent transaction. Remember that `deletes` must refer to an *existing* member of the collection (if not, they are silently ignored).

It is perfectly all right to insert an object more than once into a collection— the collections are not sets. At transaction commit time, it is as if all `deletes` were performed followed by all `inserts`. This ensures that `deletes` are performed against only existing members of a field collection, and guarantees determinacy, because otherwise the final field value can depend on the order in which insertions and deletions are performed (*e.g.*, if object O is both inserted into and deleted from an empty field collection, then two final field values are possible, depending on whether the insertion or deletion is performed first).

While multiple insertions and deletions may be performed on a field in a transaction, they may not be mixed with an `update` on the same field. As outlined above, `insert` and `delete` when mixed with `update` on a field F of a new object O violate the single-assignment semantics. Insertions and deletions mixed with `update` on a field of a persistent object are incompatible because `update` defines the new field value *completely*, independent of the old value, while `insert` and `delete` define the new value relative to the old. Thus, the final field value is ambiguous, as it may either be the new value supplied by `update`, or the old value modified by the insertions and deletions.

Here is an example of an expression that uses `insert` and `delete` to change the prerequisites for the Advanced Algorithms course.

```
(letrec ((c1 (invert COURSE NAME "Advanced Algorithms"))
         (c2 (invert COURSE NAME "Introduction to Algorithms"))
         (c3 (invert COURSE NAME "Theory of Computation")))
  (insert c1 COURSE PREREQS c2)
  (delete c1 COURSE PREREQS c3))
```

The `letrec` expression binds local name `c1` to the Advanced Algorithms course object, `c2` to Introduction to Algorithms, and `c3` to Theory of Computation. `Insert` adds `c2` to the list of prerequisites for `c1`, and `delete` removes `c3` from the list (see Figure 2.3). The expression also updates the inverse on course `PREREQS` that maps a course to the collection of courses for which it is a prerequisite (see Figure 2.4). `Insert` adds `c1` (Advanced Algorithms) to the list of courses for which `c2` (Introduction to Algorithms) is a prerequisite, and `delete` removes `c1` from the list for `c3` (Theory of Computation).

Multiple insert and delete operations may be performed by repeated individual insertions and deletions; however, for convenience, the following forms are also available:

```
(insert-list object type-name field-name list-of-new-members)
(delete-list object type-name field-name list-of-old-members)
```

Deletion of Objects

Objects of types without automatic extents do not require any explicit removal from the database. They are garbage-collected automatically when they are no longer reachable from the top-level environment.

Objects of types with automatic extents will never get garbage-collected because there is at least one reference to such objects, from the automatic extent. Thus, they must be removed explicitly using the following construct:

```
(drop object)
```

2.5 List Comprehensions—An SQL-Like Notation

A popular notation used in many functional programming languages is the list comprehension.[5] A list comprehension has the form:

```
(all body-expression
     generator-or-filter
     ...
     generator-or-filter)
```

Each generator has the form:

```
(identifier list-expression)
```

and can be read as: "For each *identifier* in the list *list-expression* ...". The identifiers bound in the generators come into scope from top to bottom, so that each *list-expression* and filter can use identifiers bound by previous generators. Each filter has the form:

```
(where boolean-expression)
```

[5]List comprehensions are also called ZF-expressions and set expressions, and were popularized by David Turner in his language KRC[78]. We believe they were originally invented by Burstall and Darlington in their language NPL at Edinburgh.

Identifier bindings that do not satisfy a filter are discarded. The meaning of the overall list comprehension is to evaluate the *body-expression* for each combination of generator bindings that satisfies all filters, and to return a list of the corresponding values. For example, the following list comprehension is a query that finds the names of all special-status students enrolled in Software Engineering.

```
(all (select s STUDENT NAME)
  (s (all STUDENT))
     (where (== "special" (select s STUDENT STATUS)))
  (c (all COURSE))
     (where (== "Software Engineering" (select c COURSE NAME)))
  (e (all ENROLLMENT))
     (where (and (== c (select e ENROLLMENT COURSE))
                 (== s (select e ENROLLMENT STUDENT)))))
```

In words:

> for each `s` in the list of students,
> where `s`'s status is special,
> for each `c` in the list of courses,
> where `c`'s name is "Software Engineering",
> for each `e` in the list of enrollments,
> where `e`'s course is `c` and student is `s`,
> return the list of names of all such students.

List comprehensions are well-integrated with other parts of the transaction language, and thus may be embedded in procedures, may use recursion, may utilize arbitrary procedure calls (including user-defined procedures) in the body, generator, and filter expressions, *etc.* For example, here is a procedure that uses a list comprehension to compute the total number of units taken by a student `s`:

```
(define total-units (lambda (s)
                      (foldl + 0 (all (course-units (enrollment-course e))
                                      (e (all ENROLLMENT))
                                      (where (== s (select e ENROLLMENT STUDENT)))))))
```

The body expression of the list comprehension uses selector functions `enrollment-course` and `course-units` to access the units of each course associated with the enrollments for student `s`. Procedure `foldl` is used to sum the list of units. The selector functions are defined as follows:

```
(define course-units (lambda (c) (select c COURSE UNITS)))
```

```
(define enrollment-course (lambda (e) (select e ENROLLMENT COURSE)))
```

Function `total-units` may be utilized in other list comprehensions. For example, the following query finds the names of all students taking more than forty-eight units:

```
(all (student-name s)
  (s (all STUDENT))
  (where (> (total-units s) 48)))
```

Selector function `student-name` is applied to each student `s` that satisfies the predicate expression.

Of course, list comprehensions are not restricted to computing on lists of persistent objects, and may be used on lists of ephemeral objects as well. For example, the following procedure returns a list of pairs of relatively prime numbers between 1 and n:

```
(define relatively-prime-pairs
  (lambda (n)
    (all (cons x (cons y nil))
      (x (ints-from 1 n))
      (y (ints-from x n))
      (where (== 1 (gcd x y))))))
```

Generator identifier `x` is bound to elements in the list of integers from `1` to `n`, and for each `x`, identifier `y` is bound to elements in the list of integers from `x` to `n`. The body expression is executed for each binding in which the greatest common divisor of `x` and `y` is one.

2.6 Comparison With SQL

Data in the relational model are organized into collections of records (called *relations* or *tables*), where each record consists of a number of fields containing scalar values such as strings and numbers. The relational structure of the student-course database introduced in Section 2.3.1, along with some sample data, is shown in Figure 2.5. Four tables are needed, one each for students, courses, and enrollments, and one to record course prerequisites.

Note that the relational representation includes ID fields in the STUDENT and COURSE tables, which are not present in the corresponding AGNA type declarations. This is because in AGNA, an object may refer to another object by storing a (system-maintained) reference to it. In the relational model, however, the programmer must encode object references by explicitly storing

STUDENT

ID	NAME	STATUS	GPA	ADDRESS	BDATE
17	"John Smith"	"GRAD"	3.95	"31 Elm Road"	640910
21	"Peter Sinclair"	"UGRAD"	3.2	"14 Main St."	710412
	⋮				

COURSE

ID	NAME	UNITS
23	"Data Structures"	12
25	"Complexity Theory"	12
	⋮	

ENROLLMENT

STUDENT_ID	COURSE_ID	GRADE
17	25	"A"
21	23	"B"
⋮		

COURSE_PREREQS

COURSE_ID	PREREQ_ID
25	23
25	27
⋮	

Figure 2.5: Relational structure of student-course database.

a unique *key* value of the referenced object. The ID fields provide the keys by which students and courses are referenced. For example, the enrollment table records student and course ids, and a grade.

Forcing the SQL programmer to encode object references in this manner is similar to the way in which one must encode tree structures in arrays in Fortran. While this is certainly possible, it makes programming more complex and error-prone. This can be seen by examining two common operations: (1) creating a new entity, such as a student, and (2) dereferencing an object "pointer". The extra complexity of the first operation, creating a new object, is due to the explicit programmer management of the id field. For example, allocating a new, unique id requires a scheme such as maintaining a "high-water" mark, perhaps in an auxiliary table, or alternately searching for the highest id in current use, from which a unique id can be generated. In AGNA, unique object identifiers are also assigned during object creation, but they are used only internally and are managed entirely by the system.

The extra complexity of the second operation, dereferencing an object pointer, is due to the associative lookup that must be specified. For example, to access the GPA of a student with id 23 in SQL [31], the standard relational database language, the programmer must specify the id of the desired student record, and then select the GPA field:

```
SELECT gpa
FROM   student
WHERE  id=23
```

In AGNA, on the other hand, field values may be read directly from an object reference—no explicit lookups are required.

Of course, if a suitable unique object attribute already exists, then an artificial key such as course id is not necessary. We might, for example, consider using student and course names as keys. This is not a good idea, however, for two reasons. First, pointer dereferencing is a common operation, so we would like it to be as efficient as possible. In this case, lookups on integer id fields, and the associated key comparisons, are more efficient than lookups on string names. The second, and more fundamental reason, is that even though the names are unique, they may change (*e.g.*, as a result of a student getting married), in which case we must update the student object, as well as locate and update all references to it. It is unlikely that artificial keys will ever change.

An additional fundamental difference between the relational and AGNA representations of the student-course database is the way in which multi-valued fields such as course prerequisites are represented. In AGNA, prerequisites are implemented via a multiple-valued field in the course object:

```
(type COURSE (extent)
  ((name     <=>  STRING)
   (prereqs *<=>* COURSE)
   (units    =>  INTEGER)))
```

In the relational model, a collection of values cannot be stored in a field, so the prerequisites of a course must be encoded in the auxiliary table COURSE_PREREQS, which records one entry for each course prerequisite. As before, this encoding complicates and makes more error-prone the programmer's task. For example the definition of a course's prerequisites involves multiple insertions into the COURSE_PREREQS table. In AGNA, on the other hand, the list of courses can be stored directly in the prereqs field in a single update operation.

Fetching course prerequisites in SQL is also more abstruse. For example, here is an SQL query that finds the prerequisites for Software Engineering:

```
SELECT prereq_id
FROM   course, course_prereqs
WHERE  name="Software Engineering" and id=course_id
```

In the WHERE clause, we must specify explicitly the condition linking the course and course_prereqs

tables (`id=course_id`), as well as the condition on course name. In AGNA, we can access the prerequisites directly after the desired course object is located:

```
(letrec ((c (invert COURSE NAME "Software Engineering")))
  (select c COURSE PREREQS))
```

Finally, referential integrity is a serious concern in the relation model. For example, for every COURSE_ID in the COURSE_PREREQS table, there must be a course with that id in the COURSE table. Current relational systems either don't deal with this issue at all, in which case it is entirely the responsibility of the programmer to ensure the integrity of such references, or they layer on top of the basic system a separate mechanism for describing and checking integrity constraints. Both approaches complicate the programmer's task. In AGNA, on the other hand, the integrity of object references is not an issue because an object is only garbage-collected when no more references to it exist.

List Comprehensions

The reader may discern a strong similarity between list comprehension notation and SQL. For example, here is an SQL query to find the names of all students with a GPA of at least 3.9:

```
SELECT name
FROM   student
WHERE  gpa>=3.9
```

The corresponding list comprehension is:

```
(all (select s STUDENT NAME)
  (s (all STUDENT))
  (where (>= (select s STUDENT GPA) 3.9)))
```

The body expression in a list comprehension is analogous to SQL's SELECT clause, the generator lists are analogous to tables in the FROM clause, and the `where` clause is analogous to SQL's WHERE clause.

While list comprehensions are similar to SQL, they are more general. The expressive power of list comprehensions is at least as great as SQL because: (1) SQL can be translated to the relational calculus [81], and (2) the relational calculus can be translated to list comprehensions

[77]. List comprehensions are more general than SQL because:

- The generator lists may be arbitrary computed lists, unlike SQL, where they must be existing, named relations.
- The `where` predicates may be arbitrary boolean expressions that may include arbitrary function calls, whereas SQL allows only a fixed repertoire of operations.
- The *body-expression* may be an arbitrary expression, not just projections on the fields of base relations.

We have already seen an example of how `where` predicates may include arbitrary function calls. Recall from the previous section the following query that finds the names of all students taking more than forty-eight units:

```
(all (student-name s)
  (s (all student))
  (where (> (total-units s) 48)))
```

User-defined procedure `total-units`, also introduced in the previous section, utilizes a list comprehension to compute the total units for a student `s`. This sort of procedural abstraction is very natural in AGNA and functional programming, but simply not possible in SQL, where all queries must be encoded directly in the fixed, pre-defined set of operations.

Declarativeness of List Comprehensions and SQL

Both list comprehensions and SQL are declarative in the sense that they allow one to pose queries at a high level, and to ignore details such as the specific algorithms used to implement a query, and how data are organized on disk. For example, indexes are used automatically and transparently by AGNA and SQL systems. The transaction programmer need not mention them explicitly in queries, nor update them manually in update transactions. Also, the compiler picks efficient methods for data access, based on its knowledge of indexes, data-set sizes, distributions of data values, *etc.*

Embedded SQL

Simple operations on record-oriented data can be expressed elegantly and concisely in SQL. Equivalent operations in AGNA can also be expressed in a similar manner using list comprehensions. As we consider more complicated operations and data, the level of complexity in SQL

programs escalates rapidly due to the lack of abstraction, encoding of object pointers, encoding of multiple-valued attributes, and so on. In AGNA, on the other hand, the user may easily move to more complex operations by exploiting the richer object modeling and the full functional language included in the transaction language. For example, consider the following procedure that returns a list of the (direct and indirect) prerequisites of a course:

```
(define all-course-prereqs
  (lambda (c)
    (letrec ((cs (course-prereqs c)))
      (if (nil? cs)
          nil
          (foldl append cs (map all-course-prereqs cs))))))
```

Procedure `all-course-prereqs` takes a course `c` and returns a list of its prerequisites (duplicates are not removed). The procedure binds local name `cs` to the list of `c`'s direct prerequisites. If `cs` is empty (*i.e.*, `c` has no prerequisites), then `nil` is returned. Otherwise, `foldl` is used to append `cs` to the results of recursively calling `all-course-prereqs` on the elements of `cs`.

The transitive closure operation performed by `all-course-prereqs` is expressed easily and succinctly in AGNA. It is not possible to express this operation directly in SQL—it simply does not have adequate expressive power. To gain the necessary expressive power, SQL must be embedded into a host language such as C or Ada. Here is a C procedure that contains embedded SQL statements (prefaced by `EXEC SQL`) that finds the prerequisites of the course with id `courseId`:

```
int allCoursePrereqs( courseId, idArray )
EXEC SQL BEGIN DECLARE SECTION;
int courseId, idArray[];
EXEC SQL END DECLARE SECTION;
{
    EXEC SQL BEGIN DECLARE SECTION;
    int index=0, j=0, id=courseId;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL BEGIN TRANSACTION;
    do {
        EXEC SQL SELECT prereq_id
                 INTO   :idArray[index]
                 FROM   course_prereqs
                 WHERE  course_id=:id;
        EXEC SQL BEGIN;
                 index++;
        EXEC SQL END;
        id = idArray[j];
    } while ( j++ < index );
```

```
    EXEC SQL END TRANSACTION;

    return index;
}
```

Result prerequisite ids are stored in array `idArray` and the total number of ids retrieved is returned.[6]. The `SELECT` statement in the `do-while` loop fetches from the database all direct prerequisites of the course whose id is stored in host variable `id` (host variables in the `SELECT` statement are prefaced by :). Initially `id` is set to the value of `courseId`, the course whose prerequisites the procedure computes. The result of each invocation of the query is returned to the host language one value at a time. For each `prereq_id` returned, the actual id value is stored into host variable `idArray` (specified in the `INTO` clause) and the `BEGIN - END` block is executed, incrementing the index into `idArray`.

After the first execution of the `SELECT` statement, `id` is assigned, in sequence, to prerequisite ids stored in `idArray`. The loop is exited when the SQL query has been invoked for each such binding of `id` (*i.e.*, `j >= index`), and the number of prerequisites fetched is returned. The loop is enclosed in SQL statements to begin and end the transaction, which are necessary to force the database to consider all invocations of the `SELECT` statement part of the same multi-statement transaction, rather than a series of individual single-statement transactions. As in the AGNA version, duplicates are not eliminated.

The embedded SQL version of this operation is significantly more complicated than the corresponding AGNA version. To program in embedded SQL one must contend with two sets of incompatible data structures (*e.g.*, array `idArray` in C and relation `course_prereqs` in the database), two sets of incompatible control constructs, two type systems, *etc.* Difficulties inherent in embedded SQL make it a viable option only for expert applications programmers. In AGNA, on the other hand, even novice users can quickly and easily escalate to sophisticated queries.

Finally, it is important to realize that it is not the embedding of SQL, per se, that causes the complexity of `allCoursePrereqs`. For reasons of interoperability, it is desirable for *all* databases, including AGNA, to be accessible from languages such as C. The complexity of `allCoursePrereqs` arises because SQL does not have adequate control structures, and thus part of the operation

[6]We have glossed over the thorny issue of storage allocation for the result array `idArray` — the caller of `allCoursePrereqs` must allocate storage for it. For this version of `allCoursePrereqs` to work correctly on databases of all sizes, an additional `length` argument must be included to allow detection of the end of the array and extension of it. Alternatively, a linked list result data structure could be used. In either case, construction of the result is significantly more complex than in AGNA, where storage management is performed automatically by the system.

must be implemented in SQL (*e.g.*, SELECT) and part in C (*e.g.*, `do - while`). This also results in run-time inefficiencies, as the embedded program and database system repeatedly exchange small pieces of the operation and small pieces of the result.

AGNA, on the other hand, does have suitable control structures, and thus the entire operation can be coded in the transaction language. An embedded program may invoke this operation in the database with a single call and receive back from it the entire result. Thus, embedded AGNA programs may interact with the database in a clean and high-level manner.

2.7 Discussion

As stated earlier, we have focused mainly on expressive power, persistence and parallelism, and have given little or no attention to elegant syntax. The expressive power in AGNA arises from several features:

- The user may define new object types with no restrictions on the fields. Thus, arbitrary graph structures may be constructed, so that it is easy to directly model real-world structures— no complex encodings are necessary.

- There is no dichotomy between persistent objects and ephemeral objects. An object persists as a result of being (1) bound directly to a top-level identifier; (2) an object of a type with an automatic extent; or, recursively, (3) reachable from a persistent object. There is no need for the programmer to manage the migration of objects from the volatile heap to the persistent heap, or to map the persistent heap to disk files (or some other persistent storage medium). These tasks are performed automatically by the system.

- The system provides support for a family of field types that are very useful in databases: single-valued and multiple-valued, with optional inverses.

- The transaction language includes a full, higher-order, functional programming language [8, 20, 78].

- The transaction language includes list comprehension notation, which may be used as a declarative query language. List comprehensions are structurally similar to SQL, but more general.

- Parallelism is implicit in the language. The transaction programmer does not have to worry about partitioning data, about partitioning processes, about mapping data and processes to nodes of a parallel machine, *etc.* The single-assignment semantics makes it easy to reason about what state is read by each sub-expression, even in the presence of parallel execution. The single-assignment semantics ensures *determinacy* of execution, *i.e.*, a unique answer.

Chapter 3

Transaction Language Semantics

In the previous chapter, we gave a very informal description of the AGNA transaction language. In this chapter, we present a formal operational semantics for the language. We begin with a discussion of the implicit parallelism present in the language, which is the primary motivation for the semantics that we have chosen. We then give the formal semantics, by first presenting a core subset of the language, then giving a translation from the full language to this core subset, and finally describing how a program in the core language is reduced to a value and, for update transactions, how the database is updated. We conclude with an example reduction.

3.1 Implicit Parallelism

Parallelism in the transaction language is implicit in its semantics:

- In a block:

```
(letrec ((x1 e1)
         ...
         (xN eN))
  eBody)
```

  all expressions `e1`, ..., `eN` and `eBody` are evaluated in parallel, and the value of `eBody` may be returned as soon as it is available, even if the other expressions have not finished evaluating.

- In primitive applications:

```
(+ e1 e2)
```

```
(cons e1 e2)
...
```

all arguments are evaluated in parallel, and some primitives may even return a result value before the arguments have finished evaluating (for example, `cons` and other object constructors).

- In a function application:

```
(ef e1 ... eN)
```

all expressions are evaluated in parallel; the function (value of `ef`) may be invoked as soon as it is known, and it may even return a result, before the arguments are known.

- In a conditional expression:

```
(if e1 e2 e3)
```

the predicate `e1` is evaluated to a boolean value, after which one of the expressions `e2` or `e3` is evaluated and returned as the value of the expression.

In other words, *everything* is evaluated in parallel, except as controlled by conditionals and data dependencies. The semantics are:

Non-strict: procedures and object constructors may be invoked and even return results before all arguments are evaluated to values.

Not lazy: expressions may be evaluated even if they are not needed for the final result.

Eager: expressions are evaluated eagerly and in parallel.

Implicitly parallel: the programmer does not specify what must be done in parallel.

These semantics are borrowed from the Id programming language [58]. Programs evaluated under this regime often show massive amounts of parallelism. As an illustration of the parallelism that results from non-strict, eager evaluation, consider the following expression that finds the names of all students with a GPA of 4.0.

```
(map (lambda (s) (select s STUDENT NAME))
     (filter (lambda (s) (== (select s STUDENT GPA) 4.0))
             (all STUDENT)))
```

In a strict implementation, we must build the entire student list (*i.e.*, evaluate `(all STUDENT)`), filter it, and then perform the `map`. Each operation must execute entirely before the next one may begin. With the non-strictness in AGNA, as soon as the first cons-cell in the student list is allocated, a reference to it can be returned as the result of `(all STUDENT)` (shown in Figure 3.1). A reference to a student object is stored in the head of the list, while the tail is left empty ($\perp$ in the figure).

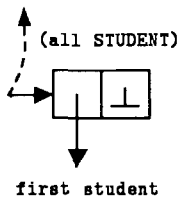


Figure 3.1: First cons cell in student list.

Construction of the remainder of the student list and the filter operation may then proceed in parallel. If the filter operation attempts to read the tail of a cons-cell that is empty (*i.e.*, one that contains $\perp$), then it simply blocks, waiting for a value to be stored there. When the write finally arrives, the blocked read operation is notified, allowing the list traversal to continue.

A similar kind of parallelism also exists between the filter operation, which produces a filtered list of student objects, and the map operation, which consumes it. In fact, this form of parallelism is possible in AGNA between *any* computation which produces a data object, and a computation that consumes it, not just those involving extent lists, or filtered extent lists. In [3], this producer/consumer parallelism, and other forms of parallelism due to non-strictness, are shown to be pervasive, even in programs that use traditional algorithms.

3.2 Formal Semantics

The formal semantics of the AGNA transaction language are defined via an abstract reduction machine called the FDB Machine[1]. The interface to the FDB Machine is exactly that of function `dbeval` from Section 2.1: it takes a transaction and a database as input, and produces an answer

[1] "FDB" stands for "Functional Database".

and a new database as output. Figure 3.2 depicts the overall structure of the process by which the FDB Machine reduces (or "executes") a transaction. Input function *in* maps transaction X and database D to initial machine configuration γ_0. Rewrite rules are then repeatedly applied, reducing sub-expressions within the transaction, and updating the database. A final configuration γ_f is reached when no more rewrite rules are applicable. Finally, output function *out* maps γ_f to a value V (*i.e.*, the answer) and database D'.

Rewrite rules used by the machine operate only on a core subset of the transaction language. This allows the rules to be specified more simply, and to reduce the total number needed by the machine. Translation from a transaction in the full language to this "kernel" language is performed by the input function.

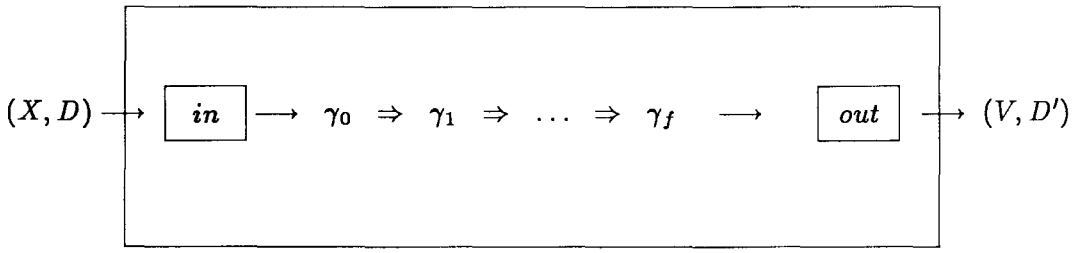


Figure 3.2: Operation of the FDB Machine.

It is important to realize that the translations and reductions included in the semantics are part of a machine-independent *specification* of the language, and not an actual implementation. As we shall see in subsequent chapters, the actual implementation will be quite different. In the development of the operational semantics, we have chosen clarity over efficiency.

3.3 The Kernel Language

The syntax of the kernel language is shown in Figure 3.3. Eliminated are features of the full language that make programs easier to write, but that provide no new expressive power. Below we give a scheme for translating from the full language to the kernel language.

Transaction	::=	Block
Block	::=	`(letrec ((Identifier Statement)+) SimpleExpr)`
Statement	::=	Definition | Expression
Definition	::=	`(define` Identifier SimpleExpr`)` `(undefine` Identifier`)`

Expression	::=	SimpleExpr | (SimpleExpr+)
		(Primitive SimpleExpr*)
		`(lambda` (Identifier*) Expression`)`
		`(if` SimpleExpr Expression Expression`)`
		Block

SimpleExpr	::=	Constant | Identifier
Primitive	::=	`+` | `-` | `drop` | `abort-transaction` |...
Constant	::=	Boolean | String | Number | `nil` | `abort` | `()`

Figure 3.3: Grammar of the kernel language.

3.4 Translation of Transactions into the Kernel Language

The algorithm for translating an input transaction into the kernel language first rewrites the transaction to a top-level `letrec` block, and then recursively translates all nested sub-expressions.

Xact Forms

The input transaction is rewritten to a `letrec` block as follows:

```
(xact                                  (letrec
  (define f e1)                            ((x1 (define f e1'))
   ...                                      ...
  (define-local x e2)      ==>             (x e2')
   ...                                      ...
  (type ...)                               (x2 (type ...)')
   ...                                      ...
  e3)                                      (x3 e3'))
                                       x3)
```

where `e1'`, `e2'`, `(type ...)'`, and `e3'` are the kernel language translations of `e1`, `e2`, `(type ...)`, and `e3`, respectively. Identifiers `x1`, `x2`, and `x3` are new and unique. Note that the `define-local` construct is eliminated by rewriting it to a binding in the `letrec` block.

Letrec Blocks

Multiple expressions in the body of a `letrec` block are eliminated. For example:

```
(letrec ((c (allocate LIST)))
  (update c LIST HD 1)
  (update c LIST TL nil)
  c)
```

is rewritten to:

```
(letrec ((c  (allocate LIST))
         (x1 (update c LIST HD 1))
         (x2 (update c LIST TL nil)))
  c)
```

Identifiers `x1`, and `x2` are new and unique.

Simple Expressions

In certain constructs, `letrec` blocks are introduced to simplify sub-expressions to simple sub-expressions (identifiers or constants). For example, the conditional:

```
(if e1 e2 e3)
```

is rewritten to:

```
(letrec ((x e1))
  (if x e2 e3))
```

Also translated in this manner are: arguments to primitive functions, all expressions of applications of user-defined functions, and the value expressions of `define` forms. Also, the body expression of a `letrec` block such as:

```
(letrec ((x1 e1))
  e2)
```

is rewritten to:

```
(letrec ((x1 e1)
```

```
        (x2 e2))
x2)
```

Identifier `x2` is new and unique. The motivation for these transformations is that they simplify the specification of rules (given in Section 3.6) for error propagation and reduction of `letrec` blocks. For example, we no longer need to specify error propagation from the predicate of a conditional.

Higher-Order Primitive Functions

All higher-order uses of primitive functions are eliminated. For example, a partial application of a primitive function such as `(+ 5)` below:

```
(map (+ 5) l)
```

is rewritten to:

```
(map ((lambda (x y) (+ x y)) 5) l)
```

This transformation ensures that primitive functions in the kernel language are always applied to a full set of arguments.

List Comprehensions

List comprehensions are rewritten according to the following scheme (for full details, please refer to [47]):

(all e (x e_{gen}) $q_1 \ldots q_n$) ⇒ (flatmap (lambda (x) (all e $q_1 \ldots q_n$)) e_{gen})

(all e (where e_{pred}) $q_1 \ldots q_n$) ⇒ (if e_{pred} (all e $q_1 \ldots q_n$) nil)

(all e) ⇒ (cons e nil)

Procedure `flatmap`, which flattens a list of lists into a single list by appending all the components, is defined as follows:

```
(define (flatmap f l)
  (if (nil? l)
      nil
```

```
(append (f (hd l)) (flatmap f (tl l)))))
```

Object Allocation

An `allocate` expression such as:

```
(allocate LIST)
```

is rewritten to:

```
(allocate-object list 2)
```

`Allocate-object` is the primitive procedure that performs object allocation, *list* is the unique numeric identifier of the `LIST` type, and `2` is the number of fields in the new object. The unique identifier associated with each type and the number of fields of a type are part of the meta-information maintained by the system, which we describe in Section 3.7. For now, type ids and field offsets can be viewed as constants inserted by the translation process.

`All` Expressions

An `all` expression such as:

```
(all ENROLLMENT)
```

is rewritten to:

```
(extent-of enrollment)
```

Procedure `extent-of` is a primitive that returns the extent list of a type.

`Select`, `Update`, `Insert`, and `Delete` Forms

A `select` expression such as:

```
(select c LIST HD)
```

is rewritten to:

```
(select-field c list 0)
```

`Select-field` is the primitive procedure that performs field selection, *list* is the numeric type identifier, and `0` is the (zero-based) field index. `Update`, `insert`, and `delete` are translated in a similar manner, using primitives `update-field`, `insert-in-field`, and `delete-from-field`, respectively.

For efficiency, multi-valued fields in persistent objects in the AGNA implementation do not use the general list representation, but rather a compact internal format.[2] This has implications with regard to sharing. For example, when a multi-valued field f of a persistent object `o` is updated to, say, a persistent list `l`, the elements of `l` are stored in `o` using this compact internal representation (see Figure 3.4). When the field value is read in subsequent transactions, a new list is built in the volatile heap and returned. Because the value stored for f is not simply a reference to `l`, but a separate representation of the collection, field insertions and deletions in subsequent transactions do not modify the contents of `l`. Also, updates to the structure of `l` (*e.g.*, adding a new element to the end of the list in a subsequent transaction) do not alter the value of f.

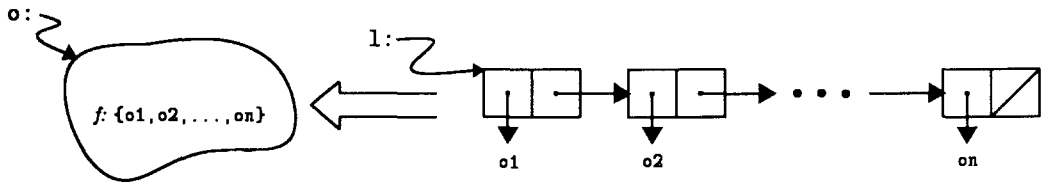


Figure 3.4: Building of multi-valued field collection.

Unlike the actual AGNA implementation, we *do* use the general list representation for field collections in the formal semantics because here we are more concerned with clarity than efficiency. We model the lack of sharing illustrated by the example above by explicitly copying lists before they are installed in multi-valued fields, and also before they are returned to field readers. For example, an `update` such as:

```
(update c COURSE PREREQS l)
```

is rewritten to:

```
(update-field c course 1 (copy-list l))
```

[2] We describe this internal format and the rationale for it in Chapter 5.

and a field selection such as:

```
(select c COURSE PREREQS)
```

is rewritten to:

```
(copy-list (select-field c course 1))
```

Array Manipulation

Array primitives are rewritten as follows:

(allocate-array *expr*) ⇒ (allocate-object *array expr*)

(update-array *object index value*) ⇒ (update-field *object array index value*)

(select-array *object index*) ⇒ (select-field *object array index*)

These translations rewrite operations on arrays to the corresponding generic object manipulation primitives.

Invert Forms

An `invert` form such as:

```
(invert STUDENT NAME "John E. Smith")
```

is rewritten to:

```
(invert1 student name-position "John E. Smith")
```

`Invert1` is the primitive function that implements <=> field inverses. Its arguments are a type id, field offset and field value, and it returns the object with the specified field value, if one exists, or a special null object otherwise. A similar function `invert2` implements multi-valued field inverses.

Insert-list and Delete-list Forms

An `insert-list` form such as:

```
(insert-list c COURSE PREREQS eecs-core-courses)
```

is rewritten to:

```
(letrec ((object c))
  (foreach eecs-core-courses
           (lambda (new-member) (insert object COURSE PREREQS new-member))))
```

Procedure `foreach` is like `map` in that it applies a procedure to each element in a list, but unlike `map` in that it returns `()` instead of a list of results.[3] `Delete-list` is rewritten in a similar manner.

Nested Function Definitions

All nested `lambda` expressions are named and lifted out into closed, local bindings in the top-level `letrec` block by a process known as *lambda-lifting* [46]. For each `lambda` expression, this entails: (1) adding to its formal parameter list all free variables (except references to top-level database names), (2) lifting the function to a local binding of a new, unique name, and (3) replacing all uses of the function by an application of it to its free variables. For example, a `letrec` block such as:

```
(letrec ((f (lambda (x) (+ x y)))
         (y e1))
  f)
```

is rewritten to:

```
(letrec ((f (f1 y))
         (y e1))
  f)
```

Also, a binding of new, unique identifier `f1` to:

```
(lambda (y x) (+ x y))
```

[3] `()` is the only value of type `void`, and has no useful operations defined on it. It is used as the return value of an expression that is executed for its side-effects, not the value it produces.

is added to the top-level `letrec` block.

Summary

The FDB Machine maps an input transaction X and database D to the answer V and a new database D'. Operation of the machine consists of three separate phases: (1) translation of X and D to γ_0, the initial machine state; (2) reduction of the transaction via repeated application of rewrite rules to expressions in the machine state; and (3) formation of the answer V and new database D' from the final machine state. The first step is performed by the input function, and includes the rewriting of X to an equivalent transaction in the kernel language, a core subset of the full language. In this section we have described the translations used by the input function in the rewriting of X. In the remainder of this chapter we describe the machine state, additional actions performed by the input function, the rewrite rules, and the output function.

3.5 The FDB Machine

As stated earlier, the FDB Machine repeatedly applies rewrite rules to reduce a transaction relative to a database. Each application of a rewrite rule maps a machine state γ_i to γ_{i+1}, reducing some sub-expression of the transaction. The state of the machine consists of four components:

1. The transaction. This describes the computations that remain to be performed.
2. The unique identifier associated with the transaction, assigned by the input function. Transactions are executed in a serializable order and the identifier associated with a transaction can be thought of as its position in this ordering.
3. The database against which the transaction is executed.
4. Five collections used to accumulate deferred updates during reduction of the transaction. These updates are installed in the new version of the database by the output function.

The third component, the database, consists of: (1) a top-level environment ρ, which maps names to values; and (2) a heap σ, which maps object identifiers to values. Object identifiers are used only internally, and are *not* the same as program identifiers. Data objects in σ are represented as follows:

$$\langle t\ i\ \phi\ v_{f_0}\ \ldots\ v_{f_{i-1}} \rangle$$

where t is a type tag[4], i is the number of fields, ϕ is the unique identifier of the transaction that allocated the object, and v_{f_0} through $v_{f_{i-1}}$ are field values.

Both the top-level environment and heap of a database are tables that map identifiers to values (see Figure 3.5). The tables also include a *new value* column used in update transactions to record identifier and object values in the new version of the database. We use notation $\rho_v(x)$ to refer to the value bound to top-level identifier x, and $\rho_n(x)$ to refer to the new value. Similarly, $\sigma_v(o)$ refers to the value bound to object identifier o, and $\sigma_n(o)$ to the new value.

ρ :

identifier	*value*	*new value*
x	259	$\perp$
y	o_{23}	$\perp$
...		

σ :

object-id	*value*	*new value*
o_{46}	$\langle list\ 2\ \phi\ v_{hd}\ v_{tl} \rangle$	$\langle list\ 2\ \phi\ \perp\ \perp \rangle$
o_{19}	$\langle course\ 3\ \phi\ v_{name}\ v_{prereqs}\ v_{units} \rangle$	$\langle course\ 3\ \phi\ \perp\ \perp\ \perp \rangle$
...		

Figure 3.5: Top-level database environment (top) and object heap (bottom).

The fourth component of the machine state contains collections that are used during reduction of the transaction to accumulate deferred updates. Database updates performed via `define`, `undefine`, and `update` are recorded directly in (the *new value* columns of) ρ and σ during the reduction process, while objects to be inserted into or dropped from type extents, and objects to be inserted into or deleted from field collections are accumulated in collections *adds*, *drops*, *inserts*, and *deletes*, respectively. Possible violations of the uniqueness of inverse field-mappings, resulting from new values of object fields for which unique inverses are maintained, are accumulated in collection *constraints*.

These deferred updates and constraint checks are processed by the output function after the machine reaches a halting configuration. Such updates and constraint checks are *not* performed during the reduction process for two reasons. First, deferring them to the output function greatly simplifies specification of the rewrite rules. Second, some sort of mechanism for deferring updates is required to implement such things as field insertions and deletions correctly. (Recall

[4]While t is actually a numeric type tag, here we will use symbolic tags such as *list* or *student* for readability.

from Section 2.4, it must be as if all `deletes` in an update transaction were performed prior to the first `insert`.)

Initial machine state γ_0 is produced by input function *in*, whose pseudo-code is shown in Figure 3.6. For a transaction (`xact` $s_1 \ldots s_m$) and database $\langle \rho, \sigma \rangle$, *in* returns the following initial state:

$$\langle \$(\texttt{letrec}\ ((x_1\ s'_1) \ldots (x_n\ s'_n))\ x_n),\ \Phi,\ \langle \rho', \sigma' \rangle, C \rangle$$

where:

- \$ is a marker used by the rewrite rules to track expressions to be evaluated;
- (`letrec` $((x_1\ s'_1) \ldots (x_n\ s'_n))\ x_n$) is the kernel language version of the original transaction in which all letrec-bound identifiers have unique names;
- Φ is the unique identifier assigned to the transaction;
- ρ' and σ' are updated versions of ρ and σ in which all top-level identifier and object field values in the new version of the database are undefined; and
- C is the set of collections for deferred updates and constraint checks, all initially empty.

Reduction proceeds from this initial state, and terminates when the machine reaches a configuration to which no rewrite rules apply. The transaction expression is initially a `letrec` block and, as we shall see in the next section, it remains a `letrec` block throughout the reduction process, unless the transaction is aborted, in which case it is rewritten to a special abort value. We use the top-level block as a place to accumulate transient bindings introduced via nested blocks and function applications (after appropriate renaming of identifiers). In the final machine state γ_f of a successful transaction, all expressions in the top-level block are reduced to values (*i.e.*, expressions for which no further reduction is possible).

Given the final machine state as input, the output function produces the answer (*i.e.*, the value of the transaction) and a new database. Determining the answer is straightforward: it is the value in the body of the final top-level `letrec` block. The new version of the database is formed in the *new value* columns of ρ and σ by processing updates in the *adds*, *drops*, *inserts*, and *deletes* collections, and migrating to the new version old values of top-level identifiers and persistent object fields not redefined by the transaction. For pure queries (no updates), the

```
in(X,⟨ρ,σ⟩)
{
    ;; Set values of identifiers and object fields to "undefined"
    ;; in new version of database.
    For each identifier x in ρ:  ρ_n(x) ← ⊥;
    For each identifier o in σ:  σ_n(o) = σ_v(o);
                                 For each field f in σ_n(o):  σ_n(o).f ← ⊥;

    ;; Initialize deferred update and constraint collections.
    C.adds ← ∅;
    C.drops ← ∅;
    C.inserts ← ∅;
    C.deletes ← ∅;
    C.constraints ← ∅;

    ;; Assign unique id to current transaction, translate
    ;; transaction to kernel language, rename letrec-bound identifiers
    ;; uniquely, and return initial machine configuration.
    Let Φ = ρ_v(transaction-id) + 1
        Let E = alpha-rename-letrec-ids(ρ,to-kernel-language(X))
            Return ⟨$E,Φ,⟨ρ,σ⟩,C⟩;
}
```

Figure 3.6: Pseudo-code of input function.

new database is equal to the old one. If the transaction expression in the final machine state is the abort value, then that is the answer returned, along with the old database. We give a precise definition of the output function in Section 3.8.

3.6 Rewrite Rules

Motivated by the Contextual Rewriting System of Ariola and Arvind [2], we use the following notation to describe a rule:

$$\frac{C_1,\ C_2,\ \ldots\ C_n}{E \Rightarrow E';U_1,\ U_2,\ \ldots\ U_m}$$

which reads: "If preconditions C_1 through C_n hold, then an expression E may be rewritten to expression E', with additional machine state updates U_1 through U_m." We omit the horizontal line when n is zero. We use meta-variable v to denote any value (constant or object identifier); c any constant; e any expression; r and s any statement; x and y any identifier; t any numeric type identifier; o any object identifier; and n, m, and j any integer.

Two markers are used in the reduction process. We have already seen one, \$, which is used to mark expressions to be reduced. Such expressions can be thought of as "scheduled" for

reduction. Some expressions marked with \$, such as `lambda`, can be rewritten directly to values. For others, however, such as a function application, the \$ is first propagated to sub-expressions, some or all of which must be reduced to values before the expression itself can be reduced. For this latter type of expression we mark it with a • after the \$ is propagated to sub-expressions. One can think of • as marking expressions that are "executing".

Definitions

$$\$(\texttt{define}\ x\ e) \Rightarrow \bullet(\texttt{define}\ x\ \$e)$$

$$\bullet(\texttt{define}\ x\ v) \Rightarrow ();\ \rho_n(x) \leftarrow v$$

$$\$(\texttt{undefine}\ x) \Rightarrow ();\ \rho_n(x) \leftarrow \top$$

In the first rule, the marker is simply propagated to component expression e. In the second and third rules, the value of x in the new version of the database is set to v and $\top$, respectively. Identifiers bound to $\top$ in ρ_n are removed from ρ by the output function. Inconsistencies such as multiple definitions of the same identifier are detected statically, and thus do not have to be handled here.

Constants

$$\$c \Rightarrow c$$

This rule states that constants are self-evaluating.

Procedures

$$\$(\texttt{lambda}\ (x_1 \ldots x_n)\ e) \Rightarrow \langle closure\ (\lambda x_1 \ldots x_n.e)\rangle$$

A lambda expression is rewritten to a closure object, which encapsulates a function. As we shall see soon in the rules for applications, closure objects may also include an environment containing arguments supplied via partial applications. No environment is needed in closures produced by the rule above because the lambda-lifting transformation performed in the translation to the kernel language ensures that procedure bodies contain no free variables.

To enhance readability, for closures we use a representation different from the standard $\langle t\ i\ \phi\ v_{f_0}\ \ldots\ v_{f_{i-1}}\rangle$.

Conditionals

$$\$(\texttt{if}\ e_1\ e_2\ e_3) \Rightarrow \bullet(\texttt{if}\ \$e_1\ e_2\ e_3)$$

$$\bullet(\texttt{if true}\ e_2\ e_3) \Rightarrow \$e_2$$

$$\bullet(\texttt{if false}\ e_2\ e_3) \Rightarrow \$e_3$$

The first rule simply propagates \$ to the predicate expression, while the second and third rules select the *then* (e_2) and *else* (e_3) expressions of a conditional after the predicate has been reduced to a boolean value. Note that the rules prevent evaluation of e_2 or e_3 until the conditional has been rewritten by the first rule, and the predicate is reduced to a value.

Letrec Blocks

$$\$(\texttt{letrec}\ ((x_1\ s_1)\ldots(x_n\ s_n))\ s) \Rightarrow \bullet(\texttt{letrec}\ ((x_1\ \$s_1)\ldots(x_n\ \$s_n))\ \$s)$$

This rule propagates the marker to all component statements, thus demonstrating a major source of parallelism in the language, as all statements may execute concurrently. An additional rule for `letrec` blocks given below allows a block immediately nested within another to be flattened into the outer block:

$$\begin{array}{l}\bullet(\texttt{letrec}\ \ ((x_1\ s_1)\\ \qquad\quad \ldots\\ \qquad\quad (x_i\ \bullet(\texttt{letrec}\ \ ((y_1\ r_1)\\ \qquad\qquad\qquad\qquad\quad \ldots\\ \qquad\qquad\qquad\qquad\quad (y_n\ r_n))\\ \qquad\qquad\quad r))\\ \qquad\quad \ldots)\\ \quad s)\end{array} \quad \Rightarrow \quad \begin{array}{l}\bullet(\texttt{letrec}\ \ ((x_1\ s_1)\\ \qquad\quad \ldots\\ \qquad\quad (x_i\ r)\\ \qquad\quad (y_1\ r_1)\\ \qquad\quad \ldots\\ \qquad\quad (y_n\ r_n)\\ \qquad\quad \ldots)\\ \quad s)\end{array}$$

Name clashes between the two blocks are not possible because of the unique renaming of `letrec`-bound identifiers in the original transaction body and in procedure bodies during application (described below).

Identifiers

$$\frac{\rho_v(x) = v}{\$x \Rightarrow v}$$

$$\frac{\gamma_i = \langle \bullet(\texttt{letrec}\ (\ldots(x\ v)\ldots)\ y),\ \Phi,\ \langle \rho, \sigma\rangle,\ C\rangle}{\$x \Rightarrow v}$$

These rules allow a program identifier to be replaced by the value to which it is bound. In the first rule, x is a top-level database name; note that this rule encodes the principle that all database identifiers are looked up in the *old* environment (*i.e.*, ρ_v). The second rule replaces a local identifier bound in the top-level `letrec` block with the corresponding value. At most one of these rules can apply for a given identifier x because of the unique renaming of local identifiers.

Applications

$$\$(e_1 \ldots e_n) \Rightarrow \bullet(\$e_1 \ldots \$e_n)$$

The \$ is propagated to all component expressions, which may execute in parallel. An application may be rewritten when the first subexpression reduces to a closure. If the arity of the function is not satisfied, then a new closure is created:[5]

$$\frac{0 < j < m}{\begin{array}{l}\bullet(\langle \mathit{closure}\ (\lambda x_1 \ldots x_m.e)\ e_1 \ldots e_i \rangle\ e_{i+1} \ldots e_j) \\ \Downarrow \\ \bullet(\texttt{letrec}\ ((y_{i+1}\ e_{i+1}) \ldots (y_j\ e_j)) \\ \quad \langle \mathit{closure}\ (\lambda x_1 \ldots x_m.e)\ e_1 \ldots e_i\ \$y_{i+1} \ldots \$y_j \rangle)\end{array}}$$

Here the arity is not satisfied because j, the number of arguments received so far (e_1 through e_i from previous applications, and e_{i+1} through e_j from the current application), is less than m, the number of arguments to the function. The application is rewritten to a `letrec` block in which argument expressions e_{i+1} through e_j are bound to new, unique identifiers y_{i+1} through y_j. The value of the block is a closure object whose environment includes the previous set of supplied arguments (e_1 to e_i), as well as the new identifiers bound to the new argument expressions.

If the function arity is satisfied, then execution of the body may begin:

$$\bullet(\langle \mathit{closure}\ (\lambda x_1 \ldots x_m.e)\ e_1 \ldots e_i \rangle\ e_{i+1} \ldots e_m) \Rightarrow \begin{array}{l}\bullet(\texttt{letrec}\ \ ((x'_1\ e_1) \ldots (x'_m\ e_m) \\ \qquad\qquad\quad (x'\ \$e')) \\ \quad \$x')\end{array}$$

Expression e' is a copy of e in which (1) formal parameters x_i are replaced by new, unique identifiers x'_i, and (2) all local variables bound in letrec blocks inside e are α-renamed consistently to new, unique identifiers.

These rules capture the eager, non-strict behavior of procedure applications: all expressions of an application are evaluated eagerly and in parallel; the body expression may be evaluated

[5] To prevent this rule from becoming too wide, the expression to which the application is rewritten is shown below the application instead of alongside it.

as soon as it is available, and it may even return a result, before all argument expressions have reduced to values.

Primitive Functions

$$\$(primop\ e_1 \ldots e_n) \Rightarrow \bullet(primop\ \$e_1 \ldots \$e_n)$$

All arguments of a primitive operation may execute in parallel. While we don't list them all here, we assume rules for the reduction of all primitive operations. For example, the following two rules describe how `not` is rewritten:

$$\bullet(\texttt{not true}) \Rightarrow \texttt{false}$$

$$\bullet(\texttt{not false}) \Rightarrow \texttt{true}$$

Similar rules exist for other primitive arithmetic, relational, and logical functions. If a primitive function is applied to an argument of the wrong type (*e.g.*, `not` applied to a number), then the machine becomes "stuck" in the sense that it will eventually reach a state in which some sub-expressions of the top-level block are not reduced to values (the erroneous application will be one such sub-expression), but to which no rewrite rules apply. Other errors, such as undefined identifiers, violations of the single-assignment semantics, *etc.* also cause the machine to get stuck in a similar manner. The output function recognizes these situations and aborts the transaction.

Object Allocation

$$\bullet(\texttt{allocate-object}\ t\ i) \Rightarrow o;\quad \sigma_v(o) \leftarrow \langle t\ i\ \Phi\ \ \perp_0 \ldots \perp_{i-1}\rangle,\quad adds \leftarrow adds + o$$

An object of type t with i fields, all initially undefined, is allocated in the heap and bound to new, unique identifier o. Note that the transaction id of the object is set to Φ, the id of the current transaction. A reference to the new object is inserted into collection *adds*. Output function *out* examines this collection of newly-allocated objects at the end of the transaction, and for each object of a type with an associated persistent extent, it inserts the new object into the extent. Thus, the updated extent is only visible to subsequent transactions.

Dropping An Object

$$\bullet(\texttt{drop}\ o) \Rightarrow (); \quad drops \leftarrow drops + o$$

A reference to the object to be dropped is added to collection *drops*; deletion of the object is performed by the output function.

Field Selection

$$\frac{\sigma_v(o) = \langle t\ i\ \phi\ \ldots v_j \ldots\rangle,\ 0 \leq j < i}{\bullet(\texttt{select-field}\ o\ t\ j) \Rightarrow v_j}$$

Here the jth field value is selected from object o. Note that the precondition precludes application of this rule when the jth field is undefined (*i.e.*, $\perp$). In other words, reduction of `select-field` is permitted only after the field has been assigned a value. Also note that there is no way for `select-field` to access field values of objects in the new version of the heap (*i.e.*, σ_n). Finally, this rule illustrates the non-strictness of objects in AGNA, as selection of the jth field of an object is allowed even though other fields may not yet be defined.

Field Update

A field update of an object allocated in the current transaction (*i.e.*, transaction id Φ) inserts the field value into the object in the old version of the heap (*i.e.*, o_v), making it accessible to field readers in the current transaction:

$$\frac{\sigma_v(o) = \langle t\ i\ \Phi\ \ldots \perp_j \ldots\rangle,\ 0 \leq j < i}{\begin{array}{l}\bullet(\texttt{update-field}\ o\ t\ j\ v) \Rightarrow ();\ \ \sigma_v(o) \leftarrow \langle t\ i\ \Phi\ \ldots v \ldots\rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad if\ \ unique\text{-}inverse(t,j)\ \ then \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad constraints \leftarrow constraints + \langle t,j,v\rangle\end{array}}$$

If the field at offset j has a unique inverse (*i.e.*, a field declared `<=>` or `<=>*`, identified by predicate *unique-inverse*), then a three-tuple consisting of the type id, field offset, and field value is added to collection *constraints*. The check to ensure that the new field value is in fact unique is performed by the output function. If a violation of the uniqueness constraint is found, then the transaction is aborted.

A field update of a persistent object, on the other hand (*i.e.*, one with a transaction id *not* equal to Φ), inserts the field value into the object in the new version of the heap (*i.e.*, o_n), where it is not visible to the current transaction.

$$\frac{\sigma_n(o) = \langle t\ i\ \phi\ \ldots\ \perp_j \ldots\rangle,\ 0 \le j < i,\ \phi \neq \Phi}{\bullet(\texttt{update-field}\ o\ t\ j\ v) \Rightarrow ();\ \ \sigma_n(o) \leftarrow \langle t\ i\ \phi\ \ldots v \ldots\rangle \\ \quad \textit{if unique-inverse}(t, j)\ \textit{then} \\ \quad\quad \textit{constraints} \leftarrow \textit{constraints} + \langle t, j, v\rangle}$$

Note that in both rules, the preconditions require the jth field to be undefined, thus enforcing the single-assignment semantics. Here also, a three-tuple is added to *constraints* if the field has a unique inverse.

Field Insertion and Deletion

$$\frac{\sigma_n(o) = \langle t\ i\ \phi \ldots \perp_j \ldots\rangle,\ 0 \le j < i,\ \phi \neq \Phi}{\bullet(\texttt{insert-in-field}\ o\ t\ j\ v) \Rightarrow ();\ \textit{inserts} \leftarrow \textit{inserts} + \langle o, j, v\rangle}$$

A reference to the object, the field offset, and the field value are packaged into a tuple, and inserted into collection *inserts*. Deletion from a field collection (using `delete-from-field`) is rewritten in a similar manner. As with other deferred updates, insertions and deletions are performed by the output function at the end of the transaction.

Again, note that the preconditions require the jth field to be undefined, thus preventing insertion into or deletion from a field that has already been updated via `update`. They do *not* prevent `update` from being used on a field after an insertion or deletion, however. Inconsistent updates such as these are detected by the output function, and the transaction is aborted. Finally, note that the preconditions also prevent insertions and deletions on fields of ephemeral objects (transaction id Φ).

Explicit User Abort

$$\bullet(\texttt{abort-transaction}\ v) \Rightarrow \texttt{abort}$$

$$\bullet(\texttt{letrec}\ (\ldots(x_i\ \texttt{abort})\ldots)\ e) \Rightarrow \texttt{abort}$$

The first rule rewrites `abort-transaction` to a special `abort` value. This value is propagated via the second rule to the top-level `letrec` block where it ultimately becomes the value of the transaction. For our purposes here, the error message argument to `abort-transaction` is not important and thus we ignore it. Additional error values such as those for type errors and violations of the single-assignment semantics, could be introduced explicitly and propagated to top-level in a similar manner, though we have not done so here.

Summary of Parallelism in Transaction Language

There are three main sources of parallelism in the transaction language: `letrec` blocks, applications of user-defined functions, and applications of pre-defined functions. As illustrated by the rewrite rules, all sub-expressions of these constructs are evaluated eagerly and in parallel. Furthermore, this parallel evaluation regime is applied recursively to each sub-expression, thus resulting in an abundance of fine grain parallelism.

3.7 Meta-Data

The translation of AGNA transactions into the kernel language given in Section 3.4 utilizes information describing the types and fields of a database (*i.e.,* the meta-data) such as unique type ids and field offsets. We assume that this information is available to the translation process. In practice, the meta-data may be stored in a separate database, but for our purposes here, we have chosen to store it in the main database in the form of TYPE and FIELD objects. This is also the approach that we have taken in the actual implementation of AGNA.

A type form such as:

```
(type DEPARTMENT (extent)
  ((id   <=> INTEGER)
   (name <=> STRING)))
```

is translated to:

```
(letrec ((f1 (make-field 0 "id" "<=>" INTEGER ...))
         (f2 (make-field 1 "name" "<=>" STRING ...)))
  (make-type "DEPARTMENT" true (cons f1 (cons f2 nil)) ...))
```

This translation is meant only to sketch a rough picture of how type forms may be translated; for our purposes here, the precise details are unimportant. The translation utilizes `make-field` to create field objects from the field positions, names, kinds, *etc.* and `make-type` to create a type object from the type name, whether an extent is to be maintained, the list of fields, *etc.* The extent list for the type is stored in a field of the type object named `extent-list`. We assume that each type and field object has associated with it a unique numeric identifier, which is generated and stored in the `id` field of the object by constructor `make-field` or `make-type`.

3.8 The Output Function

Output function *out* maps a final machine configuration to a composite result consisting of the answer and a new database. Pseudo-code for the function is shown in Figure 3.7. *Out* first checks the machine state for the following three conditions which cause the transaction to be aborted:

1. Final expression `E` is itself the value `abort`. This is the case when a transaction is explicitly aborted via `abort-transaction`.

2. `E` is a `letrec` block, but not all sub-expressions are reduced to values. This is the case when an error (*e.g.*, a type error) prevents the machine from reducing all sub-expressions fully.

3. Both `update` and `insert`/`delete` were used on the same field of a persistent object field. For the reasons outlined in Section 2.4, this is viewed as an inconsistent specification.

If at least one of these conditions holds, the transaction is aborted by returning the `abort` value and the old database, after a garbage-collection to remove heap objects inaccessible from σ.

Next, all undefined components of the new version of the database are assigned the corresponding values from the old version. For example, if a top-level identifier in the old version of the database were not redefined by the transaction, then in this step the new version automatically inherits the old value. Deferred updates accumulated in the drop, add, delete, and insert collections are then performed in the new version of the database.

Installation of deferred updates may cause the uniqueness of inverse field-mappings to be violated, *e.g.*, two new student objects may have the same name. Such violations are detected by performing inversions for all new values of unique fields which, along with the object type and field position, are stored in the *constraints* collection. If more than one object is found with a given field value, then the transaction is aborted.

At this point in *out*, we are assured that the transaction will not need to be aborted. Next, all identifiers undefined by the transaction in the new environment are located and their values in ρ_n are set to $\perp$.[6] Finally, the new version of the database is installed in ρ_v and σ_v, the transaction id seed is incremented by one, and the result value and new database are returned.

[6] If the rewrite rule for `undefine` had set the value of such identifiers to $\perp$ in ρ_n, then they would not be distinguished from identifiers in the old version of the database that were not redefined. Thus, the rule binds undefined identifiers to $\top$.

```
out(⟨E,Φ,⟨ρ,σ⟩,C⟩)
{
    ;; Abort transaction if: (1) E is "abort"; (2) E not reduced fully (e.g.,
    ;; because of an undefined identifier); and (3) insert/delete - update
    ;; inconsistency.
    if ( E==abort OR E!=(letrec ((x1 v1)...(xn vn)) v) OR
         update-insert-conflict?(ρv(insert-list),ρ) OR
         update-delete-conflict?(ρv(delete-list),ρ) ) Then
        Return (abort,gc(⟨ρ,σ⟩));

    ;; Move to new version of database identifier and field values
    ;; not redefined by transaction.
    For each id x in ρ:  if ρn(x) = ⊥ Then ρn(x) ← ρv(x);
    For each id o in σ:  if σn(o) = ⊥ Then σn(o) ← σv(o);
                         For each field f in σn(o):
                             if σn(o).f = ⊥ Then σn(o).f ← σv(o).f;

    ;; Process deferred updates in C: drops, adds, inserts, and deletes.
    ;; Drops:
    For each object o in drops:  Let t = (invert TYPE ID type-id-of-object(o))
                                     If type-extent(t) Then
                                         σn(t.extent-list)←remove(o,σn(t.extent-list));

    ;; Similarly for adds, inserts, and deletes.

    ;; Check uniqueness of inverse-mappings; abort if not unique.
    For each triple (t,pos,v) in constraints:
        if inverse-not-unique?(t,pos,v) Then
            Return (abort,gc(⟨ρ,σ⟩));

    ;; Set to ⊥ all identifiers undefined by transaction (i.e.,
    ;; identifiers bound to ⊤ in ρn).
    For each identifier x in ρn:  If ρn(x) = ⊤ Then
                                      ρn(x) ← ⊥;

    ;; Finally, install new version of database, gc, increment transaction
    ;; id "seed", and return result v (assuming E=(letrec ((x1 v1)...(xn vn)) v)).
    For each identifier x in ρ:  ρv(x) ← ρn(x);
    For each identifier o in σ:  σv(o) ← σn(o);
    ρv(transaction-id) ← ρv(transaction-id) + 1;

    Return (v,gc(⟨ρ,σ⟩));
}
```

Figure 3.7: Pseudo-code of output function.

3.9 Example: Persistent Object Update

Let us now examine the reduction of the following transaction that increments by three the units value of the course object bound to `c1`:

```
(xact
  (update c1 COURSE UNITS (+ 3 (select c1 COURSE UNITS))))
```

An equivalent kernel language expression is:

```
(letrec ((x1 (select-field c1 course 2))
         (x2 (+ x1 3))
         (x3 (update-field c1 course 2 x2)))
  x3)
```

Here we use the notation *course* to represent the numeric identifier of the course type. Constant 2, the third argument in the calls to `select-field` and `update-field`, is the offset of the course `units` field. Let us call the expression above `E`. The initial machine state γ_0, produced by the input function, is:

$$\langle \$\mathrm{E}, 43, \langle \rho, \sigma \rangle, C \rangle$$

Components of the initial machine state are: expression `E`, marked for reduction; 43, the unique id assigned to the transaction, generated from the seed bound to `transaction-id` in ρ (shown below); the database; and C, collections for deferred updates and constraint checks, all initialized to $\emptyset$. The initial state of the database is:

ρ :

identifier	*value*	*new value*
c1	o_{23}	$\perp$
transaction-id	42	$\perp$
...		

σ :

object-id	*value*	*new value*
o_{23}	$\langle$*course* 3 ϕ "Algorithms" nil 9$\rangle$	$\langle$*course* 3 ϕ $\perp$ $\perp$ $\perp\rangle$
o_{19}	$\langle$*enrollment* 3 ϕ "B" o_{14} $o_{23}\rangle$	$\langle$*enrollment* 3 ϕ $\perp$ $\perp$ $\perp\rangle$
o_{11}	$\langle$*enrollment* 3 ϕ "C" o_{15} $o_{23}\rangle$	$\langle$*enrollment* 3 ϕ $\perp$ $\perp$ $\perp\rangle$
...		

The first reduction step propagates the \$ to all sub-expressions of the top-level block:

```
•(letrec ((x1 $(select-field c1 course 2))
          (x2 $(+ x1 3))
          (x3 $(update-field c1 course 2 x2)))
  $x3)
```

Next we propagate the marker into the right-hand sides of `letrec` bindings:

```
•(letrec ((x1 •(select-field $c1 $course $2))
          (x2 •(+ $x1 $3))
          (x3 •(update-field $c1 $course $2 $x2)))
   $x3)
```

Then, we rewrite the references to top-level identifier `c1` and the numeric constants:

```
•(letrec ((x1 •(select-field o23 course 2))
          (x2 •(+ $x1 3))
          (x3 •(update-field o23 course 2 $x2)))
   $x3)
```

Note that the identifier lookups were performed in ρ_v, the old version of the database. The field selection can now be rewritten:

```
•(letrec ((x1 9)
          (x2 •(+ $x1 3))
          (x3 •(update-field o23 course 2 $x2)))
   $x3)
```

Next we rewrite the addition and substitute for `x2`:

```
•(letrec ((x1 9)
          (x2 12)
          (x3 •(update-field o23 course 2 12)))
   $x3)
```

Finally, after reducing the field update and substituting for `x3` we get:

```
•(letrec ((x1 9)
          (x2 12)
          (x3 ()))
   ())
```

with the following heap in which the units field of course o_{23} is updated:

σ :

object-id	*value*	*new value*
o_{23}	⟨*course* 3 ϕ "Algorithms" nil 9⟩	⟨*course* 3 ϕ $\perp$ $\perp$ 12⟩
o_{19}	⟨*enrollment* 3 ϕ "B" o_{14} o_{23}⟩	⟨*enrollment* 3 ϕ $\perp$ $\perp$ $\perp$⟩
o_{11}	⟨*enrollment* 3 ϕ "C" o_{15} o_{23}⟩	⟨*enrollment* 3 ϕ $\perp$ $\perp$ $\perp$⟩
...		

Note that the update is recorded in the new version of the heap, while the old value still remains accessible to the transaction. The output function returns the answer () and the updated database shown below.

ρ :

identifier	*value*	*new value*
c1	o_{23}	$\perp$
transaction-id	43	$\perp$
...		

σ :

object-id	*value*	*new value*
o_{23}	$\langle course\ 3\ \phi$ "Algorithms" nil 12$\rangle$	$\langle course\ 3\ \phi$ "*Algorithms*" *nil* 12$\rangle$
o_{19}	$\langle enrollment\ 3\ \phi$ "B" $o_{14}\ o_{23}\rangle$	$\langle enrollment\ 3\ \phi$ "*B*" $o_{14}\ o_{23}\rangle$
o_{11}	$\langle enrollment\ 3\ \phi$ "C" $o_{15}\ o_{23}\rangle$	$\langle enrollment\ 3\ \phi$ "*C*" $o_{15}\ o_{23}\rangle$
...		

This example also illustrates the sharing of objects in AGNA: enrollment objects bound to o_{19} and o_{11} contain references to the course object, and the update is visible to both in the new database.

Chapter 4

Compilation of Agna Transactions

In the previous chapters, we described the AGNA language model and argued that it contains much implicit parallelism. In this chapter, we outline our approach to exploiting this parallelism by compilation to fine grain threads. Compilation of AGNA transactions occurs in three major phases— source-to-source translation of the original transaction text, translation into dataflow program graphs, and translation into code for a multi-threaded abstract machine called P-RISC. Substantial code optimizations are performed at each stage.

We begin this chapter with a discussion of the motivation for fine grain parallelism and data-driven computation. Then, we describe each of the three phases of compilation, in turn, including a presentation of the P-RISC abstract machine. In the next chapter, we shall see how the P-RISC machine, the target of compilation, is mapped to a concrete multiprocessor.

4.1 Rationale For a Fine Grain, Data-Driven Model

Central to our strategy for exploiting the implicit parallelism of the transaction language is a computational model that utilizes fine grain parallelism with data-driven execution, both for normal computation as well as for disk I/O. Research into dataflow architectures indicates that this is an effective way to mask the long memory latencies inherent in a parallel computer [3, 4]. The analysis and experimental results may be summarized as follows.

Parallel MIMD machines are dominated by asynchronous events. Even on uniprocessors, it is already well recognized that asynchronous events are more efficiently handled by an interrupt-driven model rather than one that uses polling, because it avoids busy-waiting. An interrupt is a simple example of data-driven scheduling— when data becomes available, an interrupt occurs, and a *continuation* is specified (via an interrupt vector), which is the thread to be activated to

accept the data.

Dataflow models of computation take this idea to the limit— *all* scheduling is uniformly data-driven. All long-latency (asynchronous) operations are structured as *split-phase* actions. Examples of long-latency operations include: memory reads across a parallel machine, disk transfers, procedure calls, requests for resources (such as heap allocation), *etc.* In the first phase, a request is sent from point A to point B in the machine, carrying with it three pieces of information:

(1) the continuation `contB` in B that will handle the request;

(2) the arguments for the request, and

(3) the continuation `contA` in A that will handle B's response.

When the request arrives at B, the continuation `contB` is activated (scheduled), which uses the arguments to perform the remote computation. When this is complete, the second phase occurs: B sends the result back to A, accompanied by `contA`. When the response arrives at A, the continuation `contA` is scheduled to compute using the result. Thus, a natural coroutining structure is inherent in the model.

The benefit of split-phase actions is that processor A does not have to block while waiting for B's response— it is free to perform other computations in the interim. Further, the benefit of passing continuations around is that a continuation directly identifies the thread to be activated, making scheduling very efficient. Finally, a processor can have multiple split-phase actions outstanding, and messages do not have be processed in any particular order. These features are invaluable in achieving high processor utilization.

Traditionally, this model has been used in dataflow machines to mask the long latencies of inter-node messages. In our model, we also use it to mask the long latencies of disk requests by allowing a processor to execute other threads while some are blocked on disk I/O. A further possibility, which we do not explore in this work, is in a model that permits multiple outstanding disk requests, it may be possible to reorder them to improve average access time.

Fine grain threads are useful for scalability and load balancing. The performance of a parallel system should improve if we can provide more parallel resources (more processors, memories, and disks). Having small threads ensures that even with more processors, each processor still has enough threads to keep it busy while some threads are blocked. Further, small threads give more flexibility in the distribution of work across the machine.

Connection With The AGNA Language Model

The above argument for fine grain threads can be made independently of the language model. However, it is very difficult to compile to such threads from traditional languages. Since they are usually tied inextricably to an imperative model, partitioning into parallel threads without introducing read-write races (due to side-effects) requires complex *dependency analysis*, which is difficult in all but the simplest programs.

For declarative languages, on the other hand, the lack of side-effects makes it particularly easy to compile into very fine grain threads. In the terminology of parallelizing compilers, there are no *anti-dependencies* in the language. We refer the reader to [3] for substantial evidence that compilers for declarative languages can effortlessly extract orders of magnitude more parallelism than is possible with traditional languages.

4.2 Phase One: Source-to-Source Translation

Phase one of compilation rewrites an input transaction in the full language to one in a core internal language similar to the kernel language of Section 3.3. This rewriting of a transaction to the core language simplifies the remainder of the compiler by reducing the number of constructs it must handle. All of the translations described in Section 3.3 are actually performed by the first phase of the compiler, except that the `xact` form is not eliminated, and more sophisticated strategies are employed to translate list comprehensions, inverse-mappings, and operations on multi-valued fields and persistent extents to more efficient code. Phase one also performs additional translations to eliminate `define` and `undefine`, and to insert code to begin and end a transaction, and print the result.

In this section, we describe these additional translations, and discuss the more sophisticated strategies for handling list comprehensions, inverse-mappings, and operations on multi-valued fields and persistent extents. We also describe a number of significant optimizations that are used in AGNA to enhance the performance of list comprehension database queries.

4.2.1 Sequencing of Transaction Execution

Added to the original transaction text are calls to library routines to begin the transaction, print the result, and end the transaction. A transaction (`xact` *body*), for example, is translated as follows:

```
(xact
  (seq
    (begin-transaction)
    (print body')
    (end-transaction)))
```

`Seq` is a new construct part of the internal intermediate language that sequentializes execution of expressions[1], and *body'* is a translation of the original transaction body. Top-level expressions in the `seq` form define the three phases of execution of an AGNA transaction: the prologue, during which transaction-specific initialization is performed; the body, during which the user-supplied portion of the transaction is actually executed and the result printed; and the epilogue, during which the transaction's updates, if any, are installed in the database, making them visible to the next transaction. We will have more to say in the next chapter regarding the specific actions performed during the transaction prologue and epilogue.

4.2.2 Define, and Undefine

Bindings in the top-level environment are recorded in the database via objects of the following type:

```
(type BINDING (extent)
  ((name <=> STRING)
   (value => ANY)))
```

The database contains one such object for each top-level name. `Define` and `undefine` constructs in a transaction are rewritten to code that manipulates these objects. For example, a definition such as:

```
(define x 10)
```

is rewritten to:

```
(add-new-binding "x" 10)
```

where procedure `add-new-binding` (and helper `make-binding`) are defined as follows:

[1] As we shall see in Section 4.3, `seq` is *hyper-strict* in that *all* computation associated with a component expression must complete before execution of the next one begins.

```
(define add-new-binding
  (lambda (name value)
    (letrec ((b (invert BINDING NAME name)))
      (if (null? b)
          (make-binding name value)
          (update b BINDING VALUE value)))))

(define make-binding
  (lambda (name value)
    (letrec ((b (allocate BINDING)))
      (update b BINDING NAME name)
      (update b BINDING VALUE value))))
```

`Add-new-binding` applies the `name` inverse-mapping to search for the binding object with the desired name. If the binding does not exist in the database (*i.e.*, `(null? b)` is true[2]), then a new one is created by constructor `make-binding`. If the binding does exist, then its `value` field is updated. In either case, the binding of `name` (`"x"`) to `value` (`10`) is added to the new version of the database.

An `undefine` such as:

```
(undefine x)
```

is translated to:

```
(letrec ((b (invert BINDING NAME "x")))
  (if (null? b)
      ()
      (drop b)))
```

Again, the inverse field-mapping on `name` is applied to search for the desired binding object. If the object is not found, then the `undefine` is silently ignored and `()` is returned. If it is found, then the binding object is dropped from the database.

Finally, references in a transaction to top-level database names are also rewritten to code that manipulates binding objects. A reference to top-level name `x`, for example, is translated to:

```
(lookup "x")
```

[2]Recall that `(invert T F v)`, for a `<=>` field F, returns a special null object when no object of type T in the database has value `v` in field F.

where `lookup` is a primitive procedure that locates the appropriate binding object and returns its value. Procedure `lookup` could be written in the transaction language and defined as a non-primitive procedure in the database, but that would introduce a boot-strapping problem as the lookup procedure itself is required to find the value bound to `lookup`. The problem could be avoided by statically mapping pre-defined objects (including the lookup procedure) to heap locations, and resolving all references to such objects at compile-time. While this eliminates entirely run-time lookups of pre-defined objects, the overall impact on performance is small because lookups are fairly inexpensive (especially repeated lookups of the same identifier). In the current version of AGNA, we chose to avoid the complexity of statically mapping pre-defined objects to heap locations, and thus `lookup` is implemented as a primitive procedure.

The use of ordinary database objects instead of special-purpose data structures to implement the top-level environment simplifies both compilation and the run-time system. For example, the part of the run-time system that installs updates in the database does not have to handle the top-level environment specially. A possible disadvantage is efficiency, since special-purpose structures can be tailored to support operations on the top-level environment most efficiently. However, as we shall see soon, several techniques used in the AGNA implementation, such as indexing, allow operations on the top-level environment to be performed efficiently.

4.2.3 Inverse Field-Mappings

In the translation to the kernel language given in Section 3.3, field inversions were rewritten to exhaustive searches on extent lists. For an extent with n objects, the complexity of this approach is $O(n)$ for both single- and multi-valued inverses. The translation performed by the compiler utilizes ***index structures*** to increase the efficiency of field inversions. An index is a data structure that efficiently maps field values to objects.

The object storage system on which AGNA is based supports both hash and Btree ("balanced tree") indexes. By default, hash indexes are created for all fields with inverse-mappings (*i.e.*, `*<=` and `<=` fields). This default behavior can be changed, however, by annotations in the type declaration. For example, if the student type were defined as follows:

```
(type STUDENT (extent)
  ((name    <=> STRING)
   (status  => STRING)
   (gpa     => FLOAT)
   (address => STRING)
```

```
(bdate *<=> INTEGER (index btree))))
```

then two indexes would be created: a hash index on `name`, and a Btree index on `bdate` (see Figure 4.1). Both index types are *dense* in the sense that they contain one entry for each student object, recording a field value and a pointer to the associated student. In the hash index, entries are maintained in a hash table, while in the Btree index, entries are maintained in a balanced tree. Non-leaf nodes of the Btree provide a multi-level index to the leaves, where the entries are stored in sorted order.

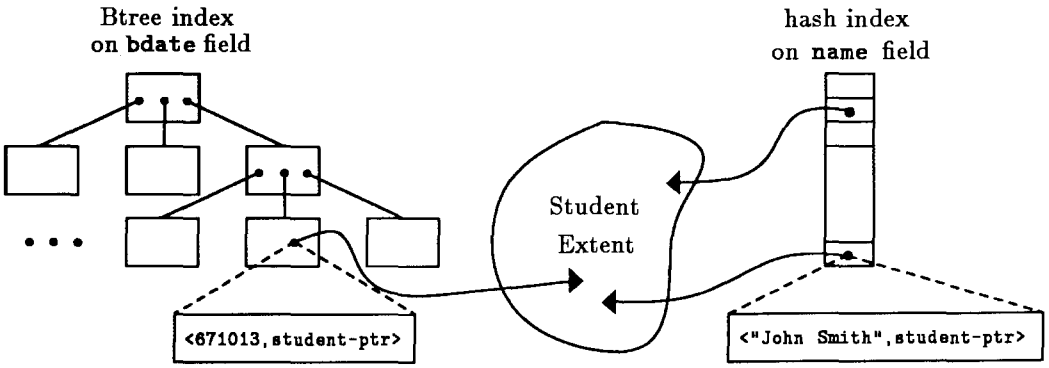


Figure 4.1: Btree and hash indexes on student objects.

In the student type above, a Btree index on `bdate` may be preferred over a hash index because of its ability to support range queries as well as "exact match" lookups. For example, the Btree index can be used in the implementation of a list comprehension query (as we shall see shortly) that finds all students with birthdates in a particular range of values, while a hash index is of no use in such a query. This is because the entries in a Btree are maintained in sorted (field) order, while entries in the hash table are maintained in hash order. For the `name` field, on the other hand, range queries are probably less likely and the increased efficiency of a hash index over a Btree for unique field lookups makes it a more suitable choice.

Inverse field-mappings are translated to `sv-invert` and `mv-invert`, pre-defined procedures which use these indexes to implement single-valued and multi-valued inversions, respectively. For example, the expression:

```
(invert STUDENT NAME "John Smith")
```

is translated to:

```
(sv-invert student-id name-id "John Smith")
```

Procedure `sv-invert` takes a type id, a field id, and a field value, and returns the object with the field value, if one exists, or a null object otherwise. Using a hash index, the complexity of `sv-invert` above is $O(1)$. Using a Btree index, the complexity of `sv-invert` is, for almost all practical purposes, $O(1)$ also, though with a larger constant factor.[3] For `mv-invert`, the complexity is $O(m)$ for both types of index, where m is the number of objects with the desired field value.

A quick check is performed at run-time by `sv-invert` and `mv-invert` to determine which type of index exists on a given field. Though this information could be added as a parameter at compile-time, the overhead of the run-time check is insignificant, and by not compiling it in, we retain the freedom to change the index structures without forcing recompilation.

In Chapter 6, we will see the dramatic impact that indexes have on performance.

4.2.4 Multi-Valued Fields and Type Extents

In the translation to the kernel language given in Section 3.3, explicit copying of multi-valued field and extent lists was introduced to prevent the lists from being shared and modified indirectly through some other part of the database. Such copying is not actually necessary in AGNA because field collections and extents are stored in a compact internal form, and the general list representation is built in the volatile heap only when needed (*e.g.*, when a multi-valued field is selected and traversed). We will describe this more compact representation and the rationale for it in the next chapter.

4.2.5 List Comprehensions

The translation scheme given in Section 3.3 rewrites a list comprehension to nested map and append operations. For example, the comprehension:

```
(all (cons x y)
  (x (ints-from 1 n))
  (y (ints-from x n))
  (where (== 1 (gcd x y))))
```

[3]For example, if nodes in the tree are 8 Kbytes in size (the current node size used in AGNA) and field values average thirty-two bytes in length, then a four-level Btree contains almost three billion entries, while a five-level tree contains over six-hundred billion entries. Therefore, the number of levels that must be traversed (and hence the complexity) is, for all practical purposes, constant.

that returns a list of pairs of relatively prime numbers between 1 and n, is translated to:

```
(flatmap (lambda (x)
           (flatmap (lambda (y)
                      (if (== 1 (gcd x y))
                          (cons (cons x y) nil)
                          nil))
                    (ints-from x n)))
         (ints-from 1 n))
```

Recall that `flatmap` applies a list-producing procedure to each element of a list, and returns an appended list of the results. The procedure applied by the outer `flatmap` produces, for a given `x`, the list of pairs that satisfy the filter (*i.e.*, are relatively prime). As these intermediate result lists are produced for each value of `x`, they are also appended together to form the final result. The procedure applied by the inner `flatmap` evaluates the filter expression for a given `x` and `y`, and produces a singleton list containing the pair if the filter is true, or `nil` if it is false. These singleton lists are appended together by the inner `flatmap` to form the intermediate result lists which are, in turn, appended together by the outer `flatmap`.

While the translation scheme above is simple and elegant, it includes costly construction and appending of intermediate result lists. The translation scheme actually used in the AGNA compiler avoids this overhead by building the result list directly. This is accomplished by extending the tail of the result list as each output element is generated. For example, let us assume that `lc` points to the tail (*i.e.*, last cons-cell) of the result list, and that we wish to add a new output element `(x,y)`. As shown in Figure 4.2, the result list is extended by allocating a new cons-cell `lc'`, storing a reference to it in the tail of `lc`, and defining its head to be `(x,y)`. Note that the tail of `lc'` is left undefined. This kind of list extension is performed by the following function, which adds an element `z` to the last cons-cell of a list:

```
(define extend-list
  (lambda (z lc)
    (letrec ((c (allocate LIST)))
      (update c  LIST HD z)
      (update lc LIST TL c)
      c)))
```

After all elements of the result list have been added in this manner, the list is terminated by storing `nil` in the tail of the last cons-cell.

We can use `extend-list` to translate a simple list comprehension such as:

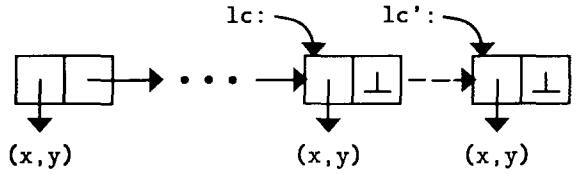


Figure 4.2: Extension of result list.

```
(all x
  (x (ints-from 1 n))
  (where (odd? x)))
```

to:

```
(letrec ((r (allocate LIST))
         (f (lambda (l lc)
              (if (nil? l)
                  lc
                  (f (tl l) (letrec ((x (hd l)))
                              (if (odd? x) (extend-list x lc) lc)))))))
  (update (f (ints-from 1 n) r) LIST TL nil)
  (tl r))
```

The translation allocates an empty cons-cell `r`, from which the result list is grown. Tail-recursive procedure `f` is applied to the list of integers returned by `ints-from`, and `r`. During each iteration, `f` evaluates the filter expression for the current `x`. If the predicate evaluates to true, then the result list is extended (by `x`) in the next iteration. When the iteration is complete (*i.e.*, `(nil? l)` is true), the last cons-cell of the result list is returned to the body of the `letrec`, where it is terminated by storing `nil` in the tail. The final result list is then simply the tail of `r`, the initial cons-cell.

Non-strictness in the AGNA transaction language is exploited in two important ways by this translation. First, non-strictness of data objects is what allows us to extend the result list incrementally. Under strict evaluation, this translation is simply not possible because there is no way for `extend-list` to allocate and return a new cons-cell before defining *both* of its fields. Second, as soon as the tail of initial cons-cell `r` is defined, then the result of the list comprehension (*i.e.*, `(tl r)`) can be returned and passed along to consumer computations. Such consumer computations, then, may execute concurrently with construction of the remainder of the list.

The translation scheme used above can be extended easily to handle comprehensions with nested generators. We simply need to pass the last cons-cell into the innermost iteration where, again, `extend-list` is used to grow the result list. For example, the list comprehension:

```
(all (cons x y)
  (x (ints-from 1 n))
  (y (ints-from x n))
  (where (== 1 (gcd x y))))
```

is translated to:

```
(letrec ((r (allocate LIST))
         (f2 (lambda (x l lc)
               (if (nil? l)
                   lc
                   (f2 x (tl l) (letrec ((y (hd l)))
                                  (if (== 1 (gcd x y))
                                      (extend-list (cons x y) lc)
                                      lc))))))
         (f1 (lambda (l lc)
               (if (nil? l)
                   lc
                   (f1 (tl l) (letrec ((x (hd l)))
                                (f2 x1 (ints-from x n) lc)))))))
  (update (f1 (ints-from 1 n) r) LIST TL nil)
  (tl r))
```

This translation has the same top-level structure as the previous one, except that two local functions are used (`f1` and `f2`) instead of one (`f`). Local procedure `f1` iterates over the list of integers from 1 to n and, for each `x` in the list, calls `f2` to iterate over the associated list of `y`'s. The current tail of the result list (`lc`) is threaded through `f1` to `f2`, where it is extended for each binding of `x` and `y` that satisfy the filter expression. As before, the list is terminated in the body of the top-level `letrec`, and the final result is the tail of initial cons-cell `r`.

4.2.6 Phase One Optimizations

While the translations given above are optimal in the sense that they allocate exactly one cons-cell per element of the result list (after allocation of initial cell `r`), they can be improved considerably by both algebraic and implementation-based transformations. In this subsection, we describe these improvements. In the next chapter, we describe additional list comprehension improvements that are performed within the run-time system.

Performing Filters As Soon As Possible

An important improvement, especially in comprehensions that involve base extents, is to apply filters as soon as possible. For example, the comprehension:

```
(all e_x,y
  (x (all T1))
  (y (all T2))
  (where (and e_x e_y)))
```

is translated to:

```
(all e_x,y
  (x (all T1))
  (where e_x)
  (y (all T2))
  (where e_y))
```

Here $e_{x,y}$ is an expression in `x` and `y`, and e_x and e_y are predicate expressions in `x` and `y`, respectively. Since e_x does not involve `y`, it is inserted as a separate filter qualifier immediately after the generator introducing `x`. The advantage of this transformation is that it eliminates as soon as possible bindings of `x` that do not satisfy predicate e_x, thus avoiding bindings of `y` and the associated predicate evaluations in the original expression that can't possibly contribute to the result. For this translation to be valid in AGNA, predicate expressions e_x and e_y and generator expression `(all T2)` must have no side-effects.

Combination of Unary Operations

A well-known algebraic transformation performed in relational database systems combines sequences of unary operations, applying them as a group, in order to avoid multiple traversals over large collections of data [79]. This general transformation is also useful for improving list comprehensions and, in fact, is performed automatically in certain cases by the default translation scheme. For example, consider the following query to find the names of all students with a GPA of at least 3.9:

```
(all (select s STUDENT NAME)
  (s (all STUDENT))
  (where (>= (select s STUDENT GPA) 3.9)))
```

A straightforward translation of the query first filters the list of all students, producing an intermediate list, over which the name selection function is then mapped.

```
(map (lambda (s) (select s STUDENT NAME))
     (filter (lambda (s) (>= (select s STUDENT GPA) 3.9))
             (all STUDENT)))
```

While this translation is simple and elegant, it can be improved by eliminating the construction and traversal of the intermediate list. This is accomplished by combining the list filtering and mapping, two unary operations, and performing them both in a single pass over the student list. Here is a translation using the scheme described in Section 4.2.5 that includes this improvement.

```
(letrec ((r (allocate LIST))
         (f (lambda (l lc)
              (if (nil? l)
                  lc
                  (f (tl l) (letrec ((s (hd l)))
                              (if (>= (select s STUDENT GPA) 3.9)
                                  (extend-list (select s STUDENT NAME) lc)
                                  lc)))))))
  (update (f (all STUDENT) r) LIST TL nil)
  (tl r))
```

Local tail-recursive procedure `f` iterates over the student list, performing both the list filtering and name selection in a single pass. The result list is grown incrementally from initial cons-cell `r` using `extend-list`, and terminated in the body by storing `nil` in the last cell of the list.

Low-Level Filtering and Projection of Base Extents

For the filtering and transformation of an arbitrary list of objects, the previous translation is optimal with respect to the number of cons-cells used to construct the result list (exactly one per result element after allocation of initial cell `r`). The value of the expression `(all STUDENT)` is obtained by scanning over the file that stores student objects and building a list in the volatile heap. When this list is available, it is then filtered and transformed in one scan.

However, a substantial improvement in performance can be obtained when the generator expression is a base extent and the filters are simple predicates on the object fields. In this case, the filtering may be performed during file scanning, avoiding even the construction of the original list.

As in all database systems, AGNA is based on an "object storage system" that implements

files and file scanning. The services provided by this module are similar to those provided in the Research Storage System (RSS) in System R [6], and WiSS, the Wisconsin Storage System [26]. The object storage system implements sequential object files, secondary Btree and hash indexes, sequential and index object scans with predicates, and management of the cache of file pages. All persistent data access is performed through the object storage system, thereby insulating higher levels of the system from details of secondary storage such as data layout, whether access is through the OS file system or directly to a raw disk, *etc.*

The scan predicates supported by the object storage system are lists of conditions of the form $F\ \theta\ v$, where F is a field name, θ is a relational operator such as equality, and v is a value. We do not allow arbitrary AGNA predicates to be evaluated during the file scan because we would like predicate evaluation to be "quick", *i.e.,* matched to the speed at which the file scan is performed.

In the example query above, the condition describing the students of interest (GPA $\geq$ 3.9) is suitable for translation to such a low-level predicate, which we can then use in the scan of the student file.[4] By performing the filtering within the storage system, a compact internal representation of the student extent is scanned and filtered in an efficient manner. Also, the file scanning function has the capability of performing simple projections on object fields. Thus, we may also push the final projection on the student name field down into the scan operation. Here is a translation that incorporates these improvements:

```
(letrec ((pred (cons (make-condition gpa-id ">=" 3.9) nil)))
  (filter-extent student-id name-id access-path pred))
```

The extent filtering and projection are performed by primitive procedure `filter-extent`, which takes type and field identifiers, an access path (to be described soon), and a predicate, *i.e.,* a list of condition objects. In this case the predicate consists of a single condition, which is created by procedure `make-condition`. `Filter-extent` scans the student extent and produces a list of names of students that satisfy the condition on GPA.

Use of Indexes

One of the most important optimizations performed by relational systems is the use of efficient index structures. Studies of relational systems have shown that the effective exploitation of

[4] In Chapter 5, we describe in detail the mapping of persistent objects to disk files.

indexes is essential for achieving good performance for a range of queries [17]. The experimental results presented in Chapter 6 indicate that the effective use of index structures is equally important for implementing list comprehension queries efficiently.

As discussed earlier, the object storage system in AGNA supports two types of indexes—Btree and hash. For example, if the student type were defined as follows:

```
(type STUDENT (extent)
  ((name    <=> STRING)
   (status  => STRING)
   (gpa     => FLOAT)
   (address => STRING)
   (bdate *<=> INTEGER (index btree))))
```

then two indexes would be created: a hash index on `name`, and a Btree index on `bdate`. For a query which accesses the student extent, there may be multiple "access paths" to the data. For example:

```
(all (select s STUDENT NAME)
  (s (all STUDENT))
  (where (and (>= (select s STUDENT GPA) 3.9)
              (<  (select s STUDENT BDATE) 720101))))
```

There are at least two ways to implement this query: (1) scan the student extent applying a predicate consisting of both conditions; and (2) use the index on `bdate` to locate students with birthdates before 1/1/72, then apply the condition on GPA to the corresponding objects in the base extent.

Our compiler uses the following heuristics, listed in order of preference, to select an implementation strategy.

1. If a condition of the form $F = v$ exists on a field with a hash index, then use the index to find all objects with value v. Apply the remaining conditions to the objects returned. If conditions exist for more than one such field, then choose a <= field (unique inverse) over a *<= field. If more than one possibility still exists, then pick one arbitrarily.

2. If a condition of the form $F\ \theta\ v$ exists on a field with a Btree index and θ is not the inequality operator, then use the index to find objects satisfying all such conditions on F. Apply the remaining conditions to the objects returned. If conditions exist for more than one such field, say F_1 and F_2, then use the following four steps to select one. (1) If a

condition involving the equality operator exists for one field and not the other then choose the field with the equality condition. (2) If one field has a single-valued inverse (<=) and the other has a multiple-valued inverse (*<=), then choose the one with the single-valued inverse. (3) Choose more restrictive condition sets over less restrictive ones. For example, $F_1 > v_1$ and $F_1 < v_2$ is more restrictive than $F_2 > v_1$. (4) Pick a field arbitrarily.

3. If Rules 1 and 2 are not applicable, then simply scan the entire extent for objects which satisfy all conditions.

More sophisticated strategies are certainly possible, taking into consideration such things as the number of objects in the base extent, histograms describing distributions of field values, *etc.*; AGNA does not currently implement them.

The access path selected by the compiler is passed as arguments to `filter-extent` indicating the unique field identifier and the type of index to use (there may be more than one). For the example query above, all students with birthdates in the desired range are located first using the index, and then the condition on GPA is applied. Here is the translation:

```
(letrec ((pred (cons (make-condition gpa-id ">=" 3.9)
                     (cons (make-condition bdate-id "<" 720101) nil))))
  (filter-extent student-id name-id bdate-id BTREE pred))
```

`Filter-extent` takes the id of the extent to filter, the field onto which the result objects are projected, the field and index of the access path, and the predicate. As described previously, both the filtering and projection are implemented within the object storage system, rather than explicitly materializing lists and then filtering and transforming them.

4.3 Phase Two: Translation to DFPGs

Phase two of compilation translates the text output of phase one to a *dataflow program graph* (DFPG) [76]. A DFPG is, roughly, a "data-driven" representation of the abstract syntax tree. Many other parallelizing compilers start with a *control flow graph* of a sequential program and, using extensive dependence analysis, attempt to extract some form of dataflow graph, because this is widely recognized to be the "most parallel" representation of the program (see [9, 14, 36]). Unfortunately, this analysis is very complicated, primarily due to the underlying imperative model of computation.

In AGNA, it is possible to go directly from the source language to a dataflow graph, precisely because of its non-imperative model. For example, a `select` operation in procedure A that reads from a particular field of a data object does not have to be sequenced by graph edges with the corresponding `update` operation, which may even be in another procedure B, because the compiler can assume that the `select` automatically blocks until the corresponding `update` has executed. This assumption depends critically on single-assignment semantics.

Translations used in phase two of compilation are based largely on methods described by Traub in [76]. Readers familiar with this material may wish to skip to Section 4.3.6, where we describe a new dataflow graph optimization involving tail-recursive functions.

4.3.1 Simple Expressions

Expressions involving primitive operations such as arithmetic, logical, and relational operators are translated to dataflow graphs in a straightforward manner. For example, the graph for the following expression:

```
(* (+ x y) (- x y))
```

is shown in Figure 4.3. The graph consists of three instructions, each specifying an *opcode*, and *input* and *output arcs*. Data values are carried on *tokens*, which flow along the output arc of one instruction to the input of another. In AGNA, the data values are either constants, such as numbers and booleans, or references to heap-resident objects.

Instructions may execute only when their *firing rule* is satisfied. The firing rule for strict primitives such as + states that the instruction may execute only when both inputs are present, *i.e.,* tokens have been placed on both input arcs. Execution of an instruction consumes input tokens, possibly produces side-effects, such as allocation of a new heap object, and generates new tokens that are placed on output arcs. The +, -, and * instructions produce output tokens that carry the sum, difference, and product, respectively, of their inputs.

Dataflow graphs capture all of the fine grain parallelism of the source language, and make explicit any data dependences and multiple uses of a variable. When `x` and `y` are available, the addition and subtraction may execute either serially or in parallel—the relative execution order of the two instructions is left unspecified. The * instruction, because it depends on data produced by + and -, may not execute until both have placed tokens on their output arcs.

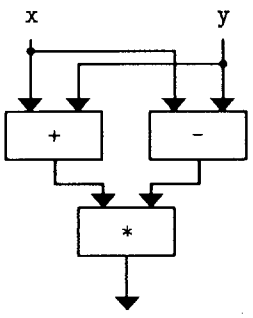


Figure 4.3: Dataflow graph for `(* (+ x y) (- x y))`.

4.3.2 Object Manipulation

Object allocation, and field update, insertion and deletion, are translated to the primitive graph instructions that perform these operations. For example, the dataflow graph for the following expression:

```
(letrec ((c  (allocate-object list 2))
         (x1 (update-field c list 0 h))
         (x2 (update-field c list 1 t)))
  c)
```

is shown in Figure 4.4. The expression allocates a list object `c`, updates its fields, and returns a reference to the new object. Inputs to the graph provide `h` and `t`, the head and tail of list, and `CONSTANT` instructions provide the necessary constants. Outputs are the list-carrying result token produced by `ALLOCATE-OBJECT`, and the tokens produced by each of the `UPDATE-FIELD` instructions.

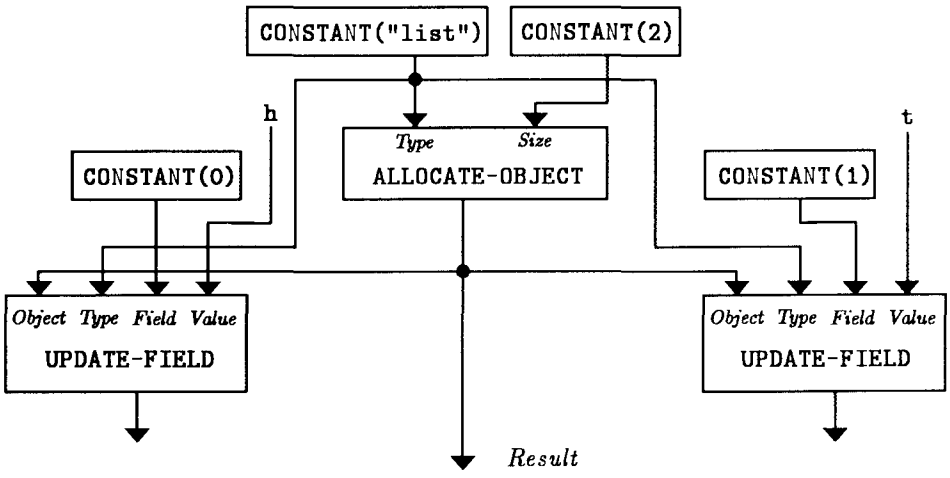


Figure 4.4: Dataflow graph for list construction.

Synchronization between the UPDATE-FIELD instructions and any SELECT-FIELD instructions that read fields of the new object is performed automatically by the instructions themselves. In other words, if a field selection happens to execute before the corresponding update operation, then it automatically blocks until the field is written, after which the field value is returned. Thus, we are free to execute field update and selection instructions in any order.

4.3.3 Triggers and Signals

Some instructions, such as the CONSTANT instructions of Figure 4.4, have no "normal" inputs. But without inputs, how can we give a firing rule for such instructions? To address this issue, we add to the graph arcs along which special *trigger* tokens flow. Such tokens carry no meaningful value, but serve only to initiate execution of an instruction. When the graph is augmented with trigger arcs (see Figure 4.5), the firing rule for CONSTANT may be stated simply: the instruction executes when its trigger input token is available.

Note that the result of the graph (*i.e.*, list object `c`, the output of ALLOCATE-OBJECT) does not depend on field values `h` and `t`. As soon as the trigger input is available, CONSTANT(2) and CONSTANT("list") may execute and place tokens on their output arcs. These tokens, in turn, initiate execution of ALLOCATE-OBJECT, which allocates a list object in the heap, and produces the result token that carries a reference to the new object. This result can be produced before fields in the new object are updated, and even before field values `h` and `t` are available.

Another issue that arises from the graph of Figure 4.4 is what to do with outputs of the UPDATE-FIELD instructions? It is clear that the output arc of ALLOCATE-OBJECT is to be connected to the instruction that consumes the result, *i.e.*, the expression in which the original `letrec` is embedded. But what about the outputs of UPDATE-FIELD? While such outputs do not contribute directly to the result of the expression, it is nevertheless useful to know when they are available. For example, if the `letrec` expression is in the body of a procedure, then we may wish to free resources allocated to the procedure when all computation in the body has terminated, *i.e.*, when the result and both UPDATE-FIELD outputs are available. To enable this detection of termination, we add to the graph a SIGNAL-TREE instruction that collects the UPDATE-FIELD outputs, and emits a *signal* token when they are both present (see Figure 4.5). Like trigger tokens, signal tokens carry no meaningful value. When both the result and signal outputs are present, all computation in the graph is terminated.

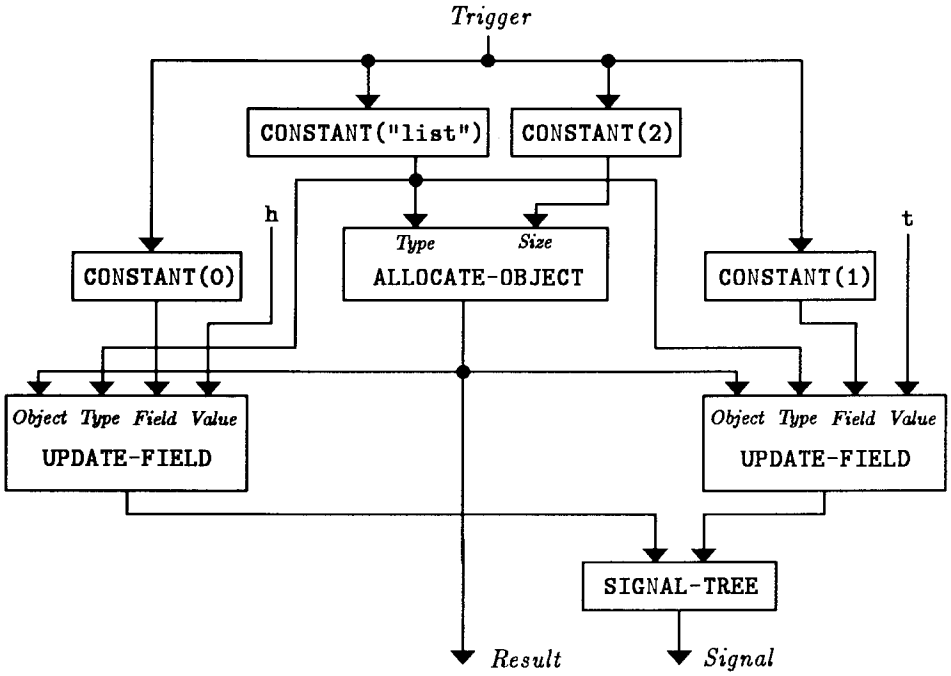


Figure 4.5: Dataflow graph for list construction with triggers and signals.

4.3.4 Procedure Definition and Application

Procedure definitions are translated to LAMBDA instructions. For example, the definition of cons:

```
(lambda (h t)
  (letrec ((c  (allocate-object list 2))
           (x1 (update-field c list 0 h))
           (x2 (update-field c list 1 t)))
    c))
```

is translated to the dataflow graph shown in Figure 4.6. The LAMBDA instruction encapsulates the body expression (shown previously in Figure 4.5), providing it with a trigger input, and arguments h and t. The RESULT-RETURN instruction receives the result of the body expression (*i.e.*, a reference to the new list object), and returns it to the caller of the procedure. When both the result and signal outputs are present, SIGNAL-RETURN propagates the termination signal back to the caller, and resources allocated to the procedure are freed. Resources are deallocated by the callee, instead of the caller, because this enables optimization of tail-recursive calls, as we shall see in Section 4.3.6.

A LAMBDA instruction is connected to the graph in which it is embedded via external *Trigger* and *Result* arcs (see Figure 4.7). The firing rule is: when the trigger token arrives, an object

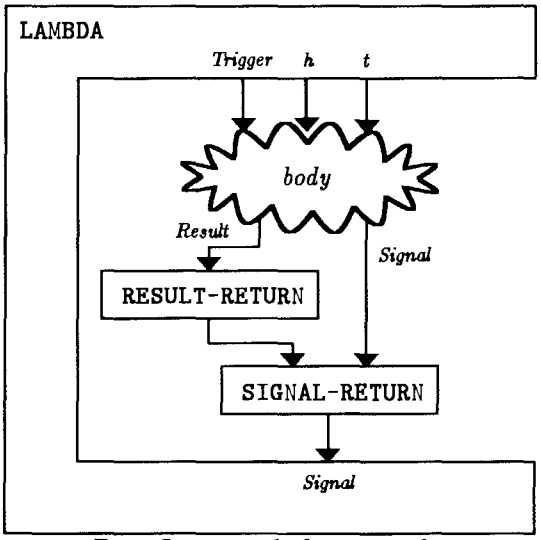


Figure 4.6: Dataflow graph for procedure cons.

representing the encapsulated procedure is created in the heap, and a token carrying a reference to it is placed on the *Result* output arc.

Figure 4.7: External view of LAMBDA instruction.

Procedure applications are translated to APPLY.[5] For example, translation of the list construction:

```
(cons 10 nil)
```

is shown in Figure 4.8. APPLY takes as input the procedure object and two arguments, and produces as output the result and signal. The non-strictness of applications is embodied in the firing rule for APPLY, which requires only that the procedure input be present before the instruction begins executing. Thus, when the cons input is present, APPLY allocates a new instance of the procedure body, and initiates execution with a trigger token. When argument

[5]In this work, we consider only the application of a procedure to *all* of its arguments, *i.e.*, a *full* application. Compilation methods for partial applications or curried functions are well understood for declarative languages such as ours (see [59, 76]), and we do not explore them here.

inputs are available, they are simply passed on to the procedure body. When `cons` returns a value (which, as we have already seen, may be before field values `h` and `t` are available), it is placed on the *Result* output arc. When the signal token is returned, it is placed on the *Signal* output.

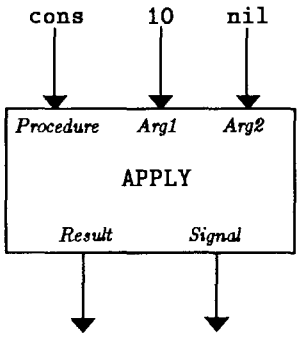


Figure 4.8: Dataflow graph for `(cons 10 nil)`.

In our description of procedure application, we have been quite vague about the linkage mechanism, saying things like the caller "initiates execution" of the body, and arguments are "passed on to the procedure". The reader can rest assured that we will make all of these details explicit in Section 4.5, when we describe the translation of dataflow graphs to P-RISC instructions.

4.3.5 Miscellaneous

Conditionals

Conditional expressions are translated to `IF`, an instruction that encapsulates the "then" and "else" clauses. For example, the conditional:

```
(if (p x) 0 (+ x x))
```

is translated to the graph shown in Figure 4.9. Incoming external arcs provide the value of predicate expression `(p x)` and free variable `x`. A trigger token and the free variable are routed either to the *then* or *else* arm of the conditional, depending on the predicate value. Result and signal outputs from the selected arm are placed on the corresponding external *Result* and *Signal* output arcs.

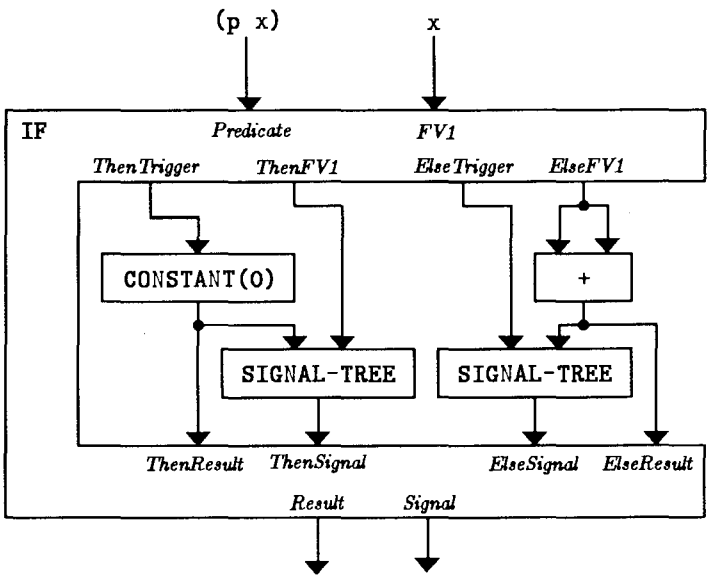


Figure 4.9: Dataflow graph for `(if (p x) 0 (+ x x)))`.

Like procedures, conditionals are non-strict. The firing rule for `IF` requires only that the predicate value be present before an arm of the conditional is selected and execution initiated via the trigger token. When free variable inputs arrive, they are passed on to the appropriate arm. In the graph of Figure 4.9, for example, if the predicate is true, then `0` can be produced immediately by the conditional, even before `x` arrives.

Top-Level Name Lookups

Lookups in the top-level database environment are translated to `LOOKUP`. For example, a reference to top-level name `begin-transaction` is translated to the graph shown in Figure 4.10.

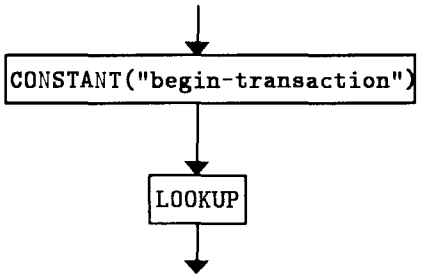


Figure 4.10: Lookup of top-level name `begin-transaction`.

Seq Forms

Expressions in a `seq` form are sequenced through the use of SIGNAL-TREE and IDENTITY instructions. For example, the expression:

```
(seq (f x) x)
```

is translated to the graph shown in Figure 4.11. Procedure `f` is applied to argument `x`, and the result and signal outputs are collected by SIGNAL-TREE, which passes a signal on to IDENTITY when both inputs are present. When its two input tokens are available, IDENTITY passes on its first (*i.e.*, `x`), which is the value of the expression. Note that `seq` is *hyper-strict* in that the second expression (`x`) may not begin executing until the first expression produces *both* a result and a signal, not just a result.

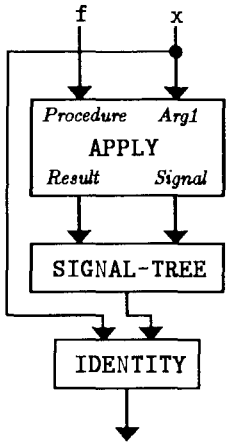


Figure 4.11: Dataflow graph for `(seq (f x) x)`.

Transactions

Transactions are translated to XACT, an instruction that encapsulates the transaction body. For example, transaction:

```
(xact
  (seq
    (begin-transaction)
    (print x)
    (end-transaction)))
```

is translated to the graph shown in Figure 4.12. The XACT instruction provides a trigger input that initiates lookup of top-level identifier `begin-transaction`, and a signal input port that accepts the signal indicating that all execution in the transaction body has terminated.

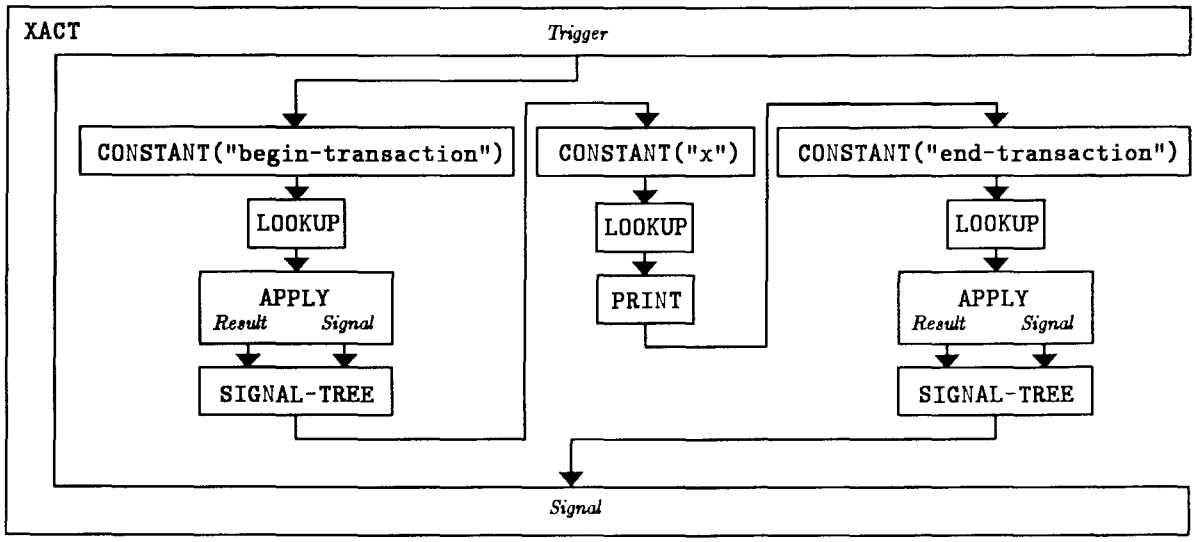


Figure 4.12: XACT dataflow graph.

4.3.6 Phase Two Optimizations

Tail Recursion

A significant optimization performed by the AGNA compiler on dataflow graphs involves tail-recursive functions. Consider the definition of procedure `foldl`:

```
(define foldl
  (lambda (f v l)
    (if (nil? l)
        v
        (foldl f (f v (hd l)) (tl l)))))
```

Recall from Chapter 2 that `foldl` takes a binary combining function `f`, an initial value `v`, and a list of values `l`, and produces an accumulated value. During each invocation in which `l` is not empty, `foldl` recursively calls itself, passing `f`, the new accumulated value, and the tail of `l`.

The dataflow graph of `foldl` is shown in Figure 4.13. Let us focus on the APPLY instruction in the body that implements the recursive call to `foldl`. APPLY allocates a new instance of `foldl`, passes to it trigger and argument tokens, and receives from it result and signal tokens. The result and signal tokens, in turn, are passed across the bottom arm of the IF instruction to

RESULT-RETURN and SIGNAL-RETURN, which return them to the appropriate input ports of the APPLY instruction that invoked the procedure. Thus, the computation unfolds as shown in the example of Figure 4.14: the first instance of the body invokes the second, and so on, until the recursion ends in the nth instance, at which point the final accumulated value and termination signal are propagated back through each instance to the original caller. Note that even though `foldl` is written in a tail-recursive manner, its execution, as just described, consumes $O(length(l))$ resources, *i.e.*, one instance of the procedure body for each element of `l`.

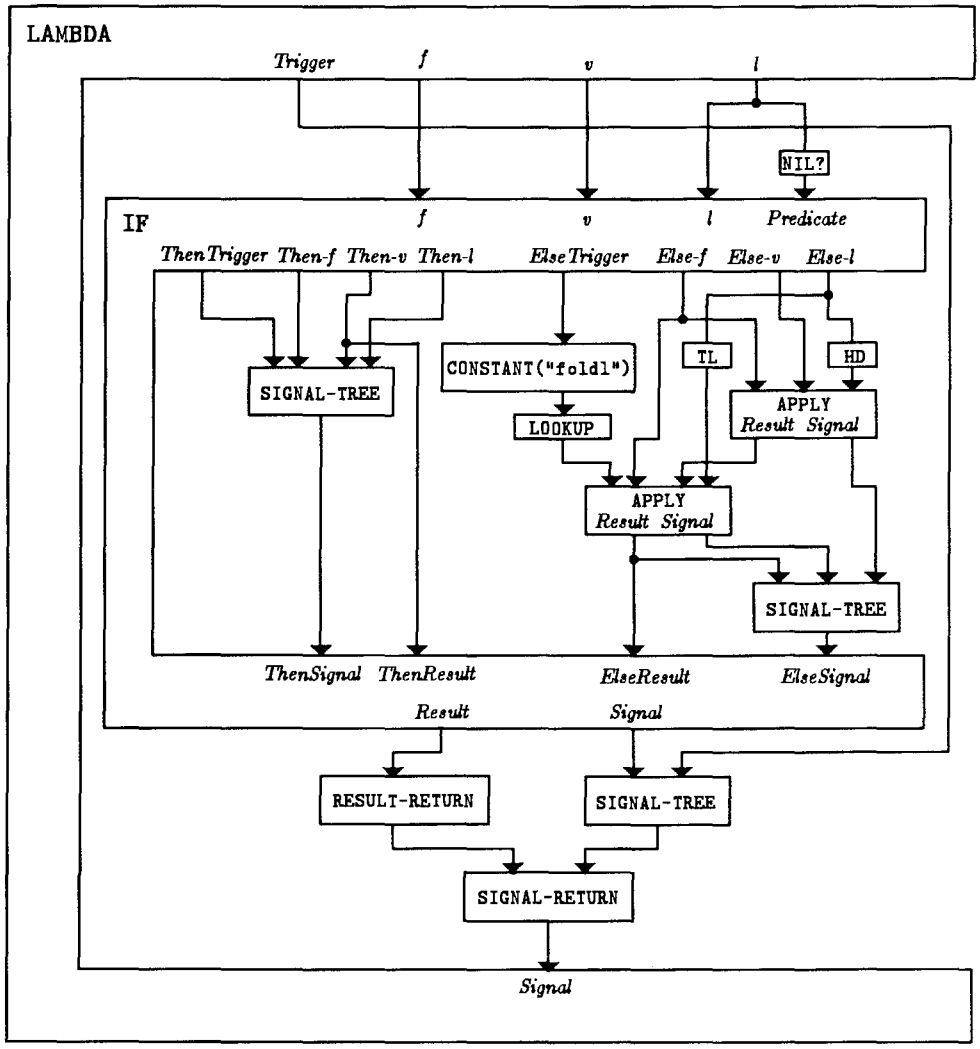


Figure 4.13: Dataflow graph of procedure `foldl`.

A modification of the procedure linkage mechanism enables `foldl` to execute using less than $O(length(l))$ resources. As before, the APPLY in instance i allocates instance $i+1$, and passes to it trigger and argument tokens. However, instead of receiving from it and propagating a result token as before, APPLY instructs instance $i+1$ to return its result directly to the instruction

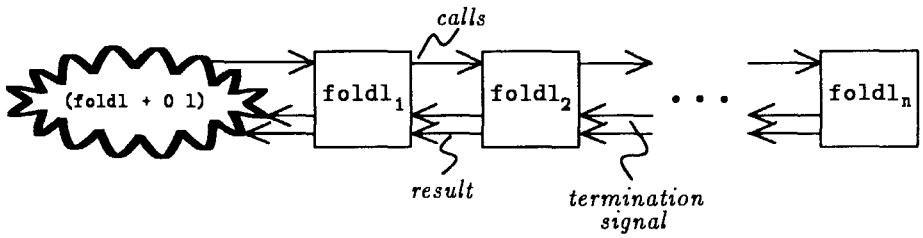


Figure 4.14: Unfolding of computation for `(foldl + 0 l)`.

waiting for the result of instance i. Also, the SIGNAL-RETURN instruction in instance i is instructed to send its termination signal *forward* to instance $i+1$, rather than back to instance $i-1$ as before. With this new linkage mechanism between instances of `foldl`, the computation unfolds as shown in Figure 4.15: the first instance invokes the second, and so on, until the recursion ends, at which point the accumulated value is returned *directly* to the original caller of `foldl`. The termination signal is propagated forward, and also returned directly to the original caller when the recursion ends.

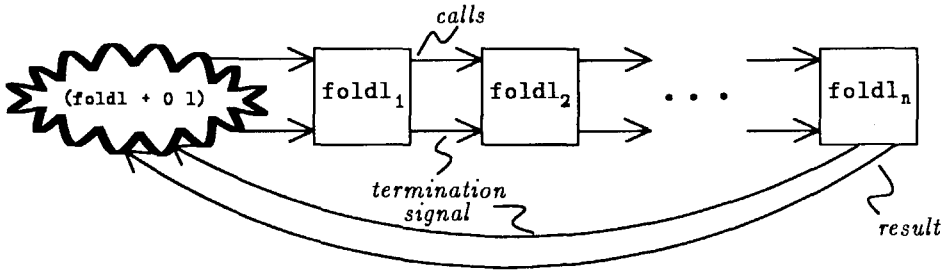


Figure 4.15: Modified unfolding of computation for `(foldl + 0 l)`.

This new `foldl` computation is potentially more efficient than the old one because instance i, after invoking instance $i+1$ and sending its termination signal, may free resources allocated to it. Thus, regardless of the length of list `l`, the entire computation can execute using only a fixed set of resources. Whether or not this actually happens in an implementation is a function of the order in which instructions are actually executed. As we shall see in the next chapter, the scheduling strategy used by the AGNA implementation ensures that a common class of tail-recursive functions does execute using a constant amount of resources.

To implement the linkage scheme described above, we introduce a new dataflow graph instruction called TAIL-APPLY. How do we know when to use this new instruction? After the graph is generated, using APPLY for all applications, we simply search it and replace all subgraphs matching the left side of Figure 4.16 with the instruction shown on the right. In other

words, we use TAIL-APPLY in those situations where the result of an application is immediately returned as the result of the procedure in which the application is performed. Note that the body of foldl, shown previously in Figure 4.13, does *not* contain a sub-graph of this form, even though the result of the recursive application inside the conditional is ultimately returned as the result of foldl. To accommodate such tail-recursive functions, we preprocess the graph before searching it, and propagate RESULT-RETURN inside encapsulator instructions such as IF. Inside the IF, a RESULT-RETURN is inserted into each branch of the conditional between the result-producing instruction and the internal result input port.

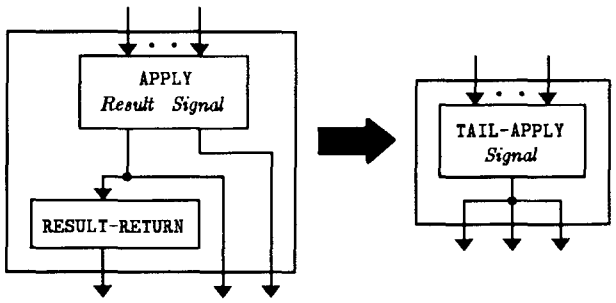


Figure 4.16: Introduction of TAIL-APPLY.

Like APPLY, TAIL-APPLY is non-strict in that it may fire as soon as the procedure input is available. Execution of TAIL-APPLY involves the following actions (say f calls g):

1. Allocation of a new instance of g's body and initiation of execution via a trigger token. The input ports to which g is to return its result and signal (which are stored in the execution environment established by f for g) are set to the input ports to which f is to return its result and signal. Finally, the input port to which f is to send its termination signal is updated to a new signal input in g (to be described soon). The signal token is not actually generated, however, until f executes SIGNAL-RETURN, which determines the destination by accessing return information stored in f's execution environment.

2. When argument inputs are available, they are passed on to the procedure body.

3. When step one is complete and all arguments are transmitted, a token is placed on the *Signal* output arc of the TAIL-APPLY instruction, indicating that the tail call has completed.

Finally, we need to add a new signal input port to each procedure, as mentioned in step one above, to ensure that a procedure invoked via TAIL-APPLY does not complete and send a termination signal prior to completion of its caller. We do this by requiring that a token be

present on the new signal input before SIGNAL-RETURN is allowed to fire. The final graph of foldl, including this additional signal input and the TAIL-APPLY optimization, is shown in Figure 4.17. Note that the signal input must be present even if the procedure is not invoked by TAIL-APPLY; thus, we need to modify APPLY to generate a trigger *and* a signal token after allocating a new instance of the body. While generation and handling of the signal token adds a small overhead to normal procedure applications, the benefits of TAIL-APPLY far outweigh this cost.

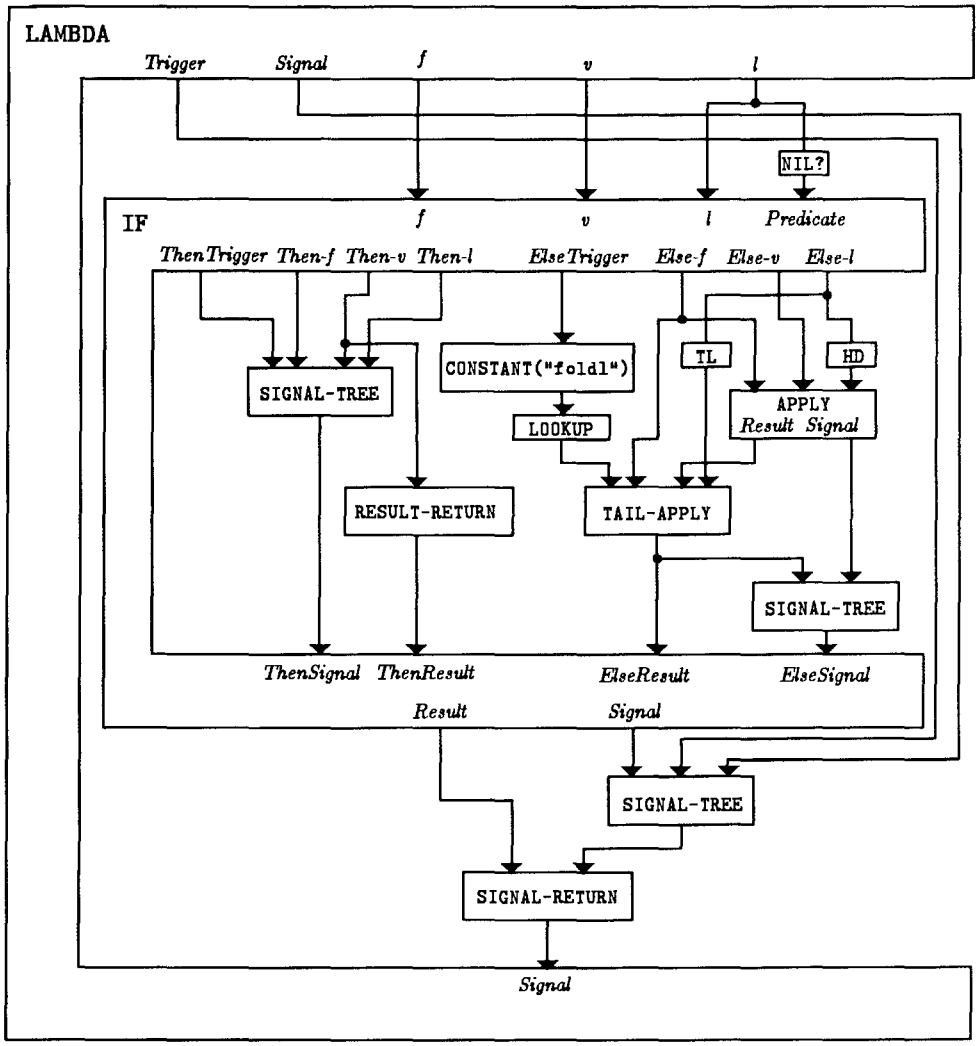


Figure 4.17: Graph of foldl including TAIL-APPLY and *Signal* input.

Common Sub-Expressions

Two types of common sub-expressions are identified during translation to dataflow graphs: constants and references to top-level database names. Lists of constants and top-level names

already translated are maintained for each scope, and subsequent uses of a constant or name present in these lists simply attach a new arc to the appropriate graph instruction, rather than generating an entirely new instruction. Thus, all uses of a constant or top-level name in a scope are connected to the same graph instruction.

4.4 The P-RISC Abstract Machine

In the preceding sections, we described the first two phases of compilation: source-to-source translation of the original transaction text, and then translation to dataflow graphs. We now describe the final target of compilation, which is a multi-threaded RISC-like abstract machine called P-RISC (for "parallel RISC"). In Section 4.5, we complete our description of the AGNA compilation process by describing how dataflow graphs are translated to P-RISC code.

At this point the reader may wonder why we include the P-RISC machine in the compilation process. For example, why not compile dataflow graphs directly into native machine code? We chose the P-RISC machine as the target of compilation because at the time the compiler was written, we had not yet decided on a multiprocessor platform on which to run AGNA, and even if we had, we wanted the flexibility to experiment easily with different platforms. The P-RISC model provides a complete, machine-level description of the program which makes explicit the key features of our computational model: fine grain multi-threading, data-driven execution, and split-phase actions to tolerate long latencies. A program expressed as P-RISC code can be translated to native machine code for execution, interpreted in software, or perhaps even executed directly in hardware.

Many machine-level details hidden by dataflow graphs are made explicit in the P-RISC abstract machine. For example, we shall now see the details of how the caller of a procedure establishes an execution context for the body, initiates execution, and passes arguments. Other details, however, such as the organization of secondary storage, and the distribution of computation and data across a parallel machine, are not addressed here. These will be addressed in the next chapter, where the abstract machine is mapped to a concrete architecture.

The P-RISC machine consists of a pool of active thread descriptors and separate memories for frames and the heap (see Figure 4.18). Each thread is described by a pair: an instruction pointer (IP) and a frame pointer (FP). IP points to the current instruction, which resides in the code section of a heap-based procedure or transaction object, and FP points to a frame.

Like conventional activation records, frames are allocated and deallocated as part of procedure call and return, and provide local storage for arguments and computations in a procedure body. Frames are organized into a tree: there is a root frame for the transaction body, and a frame for each outstanding procedure call. Multiple frames all throughout the tree can be active simultaneously, and each frame can have many simultaneously active threads. This is in contrast to the traditional "cactus stack" model, where there can be only one thread active for each branch of the cactus, which may contain several frames.

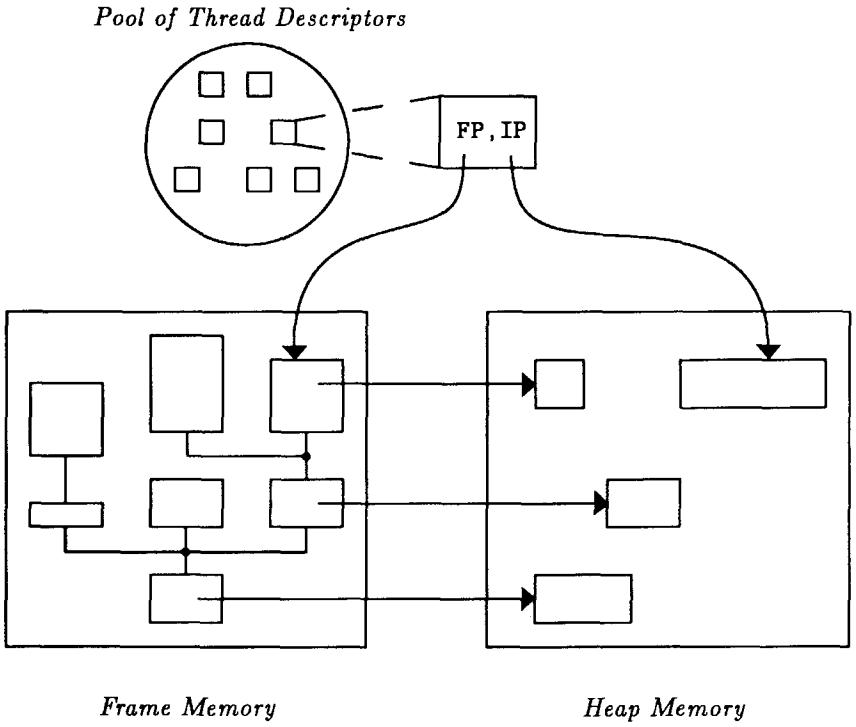


Figure 4.18: Organization of P-RISC abstract machine.

The machine operates by repeatedly extracting an active thread, executing its current instruction, and adding to the thread pool zero or more successor descriptors. The strategy for choosing the next thread to execute is not specified, and multiple threads may execute concurrently.

Heap memory is a single-level, global store, part of which is volatile, and the remainder of which is persistent. Thus, heap memory accommodates persistent objects part of the database, as well as transient objects only used during execution of a single transaction. Heap memory locations include extra status bits that indicate whether a location is full or empty. As we shall see in a moment, the heap may be read and written using normal loads and stores, which ignore

these status bits, and synchronized loads and stores, which are used to implement the deferred read, single-assignment semantics of the AGNA object model.

4.4.1 P-RISC Instructions

Following [59], we describe the semantics of P-RISC instructions in terms of state transitions on frame memory, heap memory, and the pool of thread descriptors. In other words, for a thread descriptor (FP,IP), we describe how execution of the instruction referenced via IP modifies frame and/or heap memory, and the new descriptors, if any, that are added to the thread pool. We use the notation Frames[FP+r] to refer to offset r in the frame referenced via FP, and Heap[j] to refer to the jth location of heap memory.

Arithmetic, Logical, and Relational

The usual complement of arithmetic, logical, and relational instructions such as ADD, AND, NOT, *etc.* are supported. Their syntax and semantics are standard:

Syntax	*Semantics*
binop r1 r2 r3	Frames[FP+r1] <- Frames[FP+r2] *binop* Frames[FP+r3]
unop r1 r2	Frames[FP+r1] <- *unop* Frames[FP+r2]
LOADC r1 *c*	Frames[FP+r1] <- *c*

Thread descriptor (FP,IP+1) is added to the active pool in all three cases.

Control Flow

Both conditional and unconditional jumps are also standard:

Syntax	*Semantics*
JMP L	Add descriptor (FP,L) to active pool
JMPF r L	If Frames[FP+r]==0 Add descriptor (FP,L) to active pool Else Add descriptor (FP,IP+1) to active pool

Other conditional jumps such as JMPT (jump "true"), are also supported.

The abstract machine supports three basic thread manipulation primitives:

Syntax	*Semantics*
FORK L	Add descriptors (FP,L) and (FP,IP+1) to active pool
DIE	Add no descriptor to active pool
JOIN r bn	If bit n of Frames[FP+r] is one Add descriptor (FP,IP+1) to active pool Else Add no descriptor to active pool Toggle bit n of Frames[FP+r]

FORK spawns a new fine grain thread for the current frame by adding both (FP,L) and successor descriptor (FP,IP+1) to the active pool. DIE terminates execution of the current thread. JOIN is used to combine and synchronize parallel threads, generating a successor descriptor only if the join bit is set to one. For example, in the code below, threads T1 and T2 are combined and synchronized by the JOIN instruction at label L1:

```
T1: % r1 ← e1
    JMP L1

T2: % r2 ← e2

L1: JOIN r3 b0      % wait for e1 and e2
    ADD  r4 r1 r2
    .
    .
```

T1 and T2 compute the values of expressions e_1 and e_2, place them in frame slots r1 and r2, and transfer control to L1. T1 transfers control via JUMP, while T2 simply "falls through" to L1. The join bit is initially set to zero. When JOIN is first executed (say, by T1), the bit is set to one, and no successor descriptor is generated, *i.e.*, the thread is terminated. When T2 executes JOIN, the bit is set back to zero, and the thread is allowed to continue with the addition. Note that the two instances of the JOIN instruction must execute atomically to avoid a situation in which both see the zero value of the join bit, and neither is allowed to continue.

Heap Access

The heap may be accessed via normal loads and stores, or synchronized loads and stores. Normal loads and stores have the usual syntax and semantics:

Syntax	*Semantics*
LOAD r1 r2	Frames[FP+r1] <- Heap[Frames[FP+r2]]
STORE r1 r2	Heap[Frames[FP+r2]] <- Frames[FP+r1]

Successor thread descriptors (FP,IP+1) are added to the active pool in both cases. The semantics of synchronized loads and stores depend on the status bits of the location being accessed. If a synchronized load (ILOAD) attempts to read an empty location, then it adds to the deferred reader list stored at that location a triple consisting of its FP, IP+1, and frame offset. If the location is full, on the other hand, then ILOAD behaves the same as LOAD and simply returns the value stored there. A synchronized store (ISTORE) writes a value in an empty location (an error is raised if it is full), and unblocks any deferred ILOADs waiting there.

Syntax	*Semantics*
ILOAD r1 r2	Let A = Frames[FP+r2] Case Heap[A] of (Empty,l) ⇒ Heap[A] <- (Deferred,(cons (FP,IP+1,r1) l)) (Full,v) ⇒ Frames[FP+r1] <- v; Add (FP,IP+1) to active pool
ISTORE r1 r2	Let v = Frames[FP+r1] Let A = Frames[FP+r2] Case Heap[A] of (Empty,l) ⇒ Heap[A] <- (Full,v); For each (FP',IP',r) in l Frames[FP'+r] <- v Add (FP',IP') to active pool (Full,-) ⇒ Error

The reading and writing of heap location A in each case must be performed atomically.

Inter-Frame Transfers

In procedure call and return, it is necessary for the caller and callee to transfer between them both control and data. The caller, for example, has to transfer arguments and return information to the callee's frame, and initiate one or more threads of execution in the body. The callee, on the other hand, has to transfer back to the caller frame the result of the procedure, and also initiate execution of threads receiving the result and signal of the application. Inter-frame transfers of this kind are performed by STARTO and START1, which are defined as follows:

Syntax	*Semantics*
STARTO r1 r2	Let FP' = Frames[FP+r1] Let IP' = Frames[FP+r2] Add (FP,IP+1) to active pool Add (FP',IP') to active pool
START1 r1 r2 r3 r4	Let FP' = Frames[FP+r1] Let IP' = Frames[FP+r2] Let r = Frames[FP+r3] Let v = Frames[FP+r4] Frames[FP'+r] <- v Add (FP,IP+1) to active pool Add (FP',IP') to active pool

4.4.2 P-RISC Managers

We also include in the abstract machine an extensible set of high-level "manager" instructions performing such tasks as frame and object management. Each manager performs some function relative to zero or more inputs, and produces zero or more outputs. Managers may be viewed either as complex instructions, or macros which the compiler expands to a sequence of primitives. Below we describe only a few managers; a description of the complete set used by the AGNA compiler is given in Appendix B.

Object allocation is encapsulated by ALLOCOBJ, which allocates and initializes a new object in the volatile heap:

Syntax	*Semantics*
ALLOCOBJ ri rj	Let T = Frames[FP+ri] Let S = Frames[FP+ri+1] Allocate and initialize object in volatile heap of type T and size S Let A be the address of this object Frames[FP+rj] <- A Add (FP,IP+1) to active pool

The initialization performed includes defining the object header and setting to "empty" all field status bits. Like all managers, ALLOCOBJ receives its arguments in contiguous frame slots beginning at ri, and returns its results in contiguous slots beginning at rj.

For convenience, field selection and update are also performed by managers. For example, field selection is performed by SELECTF, defined as follows:

Syntax	*Semantics*
SELECTF ri rj	Let Obj = Frames[FP+ri] Let ObjType = Frames[FP+ri+1] Let Offset = Frames[FP+ri+2] Error checking: 1. Type check 2. Bounds check 3. Obj persistent and field undefined? If error found raise error Frames[FP+rj] <- Obj + Offset ILOAD rj rj

Arguments to SELECTF are the object, its type, and the field offset; the result returned is the field value. SELECTF first checks for three error conditions: (1) an object not of the correct type; (2) a field reference that is out of bounds (this condition may hold only if Obj is an array because this is the only offset that is computed and not supplied by the compiler); and (3) a persistent object with field at Offset that is undefined[6]. If one of these conditions is found to hold, then a run-time error is raised. Otherwise, the field address is built in slot rj and ILOAD is used to access the field. If the field is not yet defined, then ILOAD will defer the operation as described previously.

Frame allocation is also performed by a manager instruction:

[6]This last condition is an error because field updates of persistent objects are only visible to subsequent transactions and thus if SELECTF is allowed to continue, ILOAD will cause the transaction to deadlock.

Syntax	*Semantics*
ALLOCFRAME ri rj	Let CallerFP = Frames[FP+ri] Let ResultIP = Frames[FP+ri+1] Let SignalIP = Frames[FP+ri+2] Let ResSlot = Frames[FP+ri+3] Let NumSlots = Frames[FP+ri+4] Let FP' be address of new frame Zero-out frame slots in FP' Store in FP': CallerFP, ResultIP, SignalIP, and ResSlot Frames[FP+rj] <- FP' Add (FP,IP+1) to active pool

Manager ALLOCFRAME allocates a frame as part of procedure application. Its arguments are: the caller's FP, the IPs of the threads to receive the result and signal, the slot where the result of the procedure is to be stored, and the size of the new frame. ALLOCFRAME allocates a frame of the desired size, initializes its slots to zero, stores the linkage information, returns a pointer to the new frame, and adds successor descriptor (FP,IP+1) to the active pool.

At the end of a transaction, all objects in the volatile heap that are reachable from the top-level database environment are moved to the persistent heap. Manager MKPERSISTENT performs this low-level moving of objects to the persistent heap.

Syntax	*Semantics*

MKPERSISTENT ri rj	Let Obj = Frames[FP+ri]
	Frames[FP+rj] <- Obj
	Frames[FP+rj+1] <- false
	If volatile?(Obj)
	  If alreadyMoved(Obj)
	  Frames[FP+rj] <- lookupPersistentAddr(Obj);
	  Else
	  LetA be address of persistent storage
	  copy(Obj,A);
	  Frames[FP+rj] <- A;
	  Frames[FP+rj+1] <- true;
	Add (FP,IP+1) to active pool

A reference to the object to be moved is passed in slot ri and the persistent address is returned in slot rj. An additional (boolean) result is returned in slot rj+1 indicating whether the persistent address returned in rj was allocated by the current call to MKPERSISTENT (true) or a previous one (false).[7] The first result is initialized to the object itself, and the second result to false. If the object has already been moved to the persistent heap (tested by predicate alreadyMoved), then the persistent address is looked up via lookupPersistentAddr and returned. Otherwise, storage is allocated in the persistent heap, the object is copied, and the new persistent address and true are returned.

[7] MKPERSISTENT may be invoked on the same volatile object more than once if it is reachable from multiple points in the database. For example, if a transaction binds the same object to two different names in the top-level environment, then MKPERSISTENT is called two times on the object. To preserve the sharing of objects, the persistent address of the object established in the first call must also be returned as the result of the second call. The boolean result returned to the second caller (false) indicates that it is not necessary to search for volatile objects in the graph of objects rooted at Obj. This is performed by the first caller of MKPERSISTENT. In the next chapter, we give a complete description of the process by which objects are moved to the persistent heap.

4.5 Phase Three: Translation to P-RISC Code

The third and final phase of compilation generates P-RISC code from dataflow graphs via a three-step process:

1. Analysis of the graph to determine which pieces are worth doing in parallel and which are best done sequentially.

2. Mapping of temporary storage implicit in the graph to frame slots. Such storage consists of slots for synchronization, slots for values that are carried on tokens from one instruction to another, and any slots needed internally by an instruction.

3. Nearly context-free expansion of each graph instruction to P-RISC code.

We now examine each of these steps in turn.

4.5.1 Graph Analysis

The graph analysis performed by AGNA is based on a heuristic developed by Iannucci called the Method of Dependence Sets (MDS) [44]. We first describe the motivation for such analysis and then present MDS. Readers familiar with MDS may wish to skip to Section 4.5.2.

Motivation

Instructions in a dataflow graph may be viewed as independent tasks that execute when and only when their required inputs are available. Synchronization of input values is performed *implicitly* by the instructions themselves, *i.e.,* when the necessary inputs are present, an instruction schedules itself for execution. At a given instant of time, all instructions for which inputs are available may execute.

In the generation of P-RISC code, we can preserve all of the parallelism present in dataflow graphs by translating each graph instruction to a corresponding P-RISC thread that performs: (1) *explicit* synchronization of its inputs; (2) the operation specified by the graph instruction; and (3) transfer of control to consumer threads. Using this translation scheme, the dataflow graph of Figure 4.19 is translated to the following P-RISC code:

```
X:  % r10 ← x
    FORK T1
    JMP T2
```

```
Y:  % r11 ← y
    FORK T1
    JMP T2

T1: JOIN r9  b0          % wait for x and y
    ADD  r12 r10 r11
    JMP T3

T2: JOIN r9  b1          % wait for x and y
    SUB  r13 r11 r12

T3: JOIN r9  b2          % wait for sum and difference
    MUL  r14 r12 r13
    ...
```

Computations that produce `x` and `y` place their values in frame locations `r10` and `r11`, respectively, and then transfer control to threads `T1` and `T2`. When both inputs are available, the threads perform the addition and subtraction, and transfer control to `T3`, where the multiplication is performed.

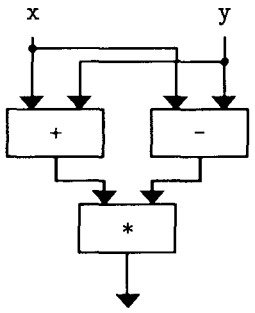


Figure 4.19: Dataflow graph for `(* (+ x y) (- x y))`.

An alternative translation strategy is to combine `T1`, `T2`, and `T3` as follows:

```
X:  % r1 ← x
    JMP T

Y:  % r2 ← y

T:  JOIN r9  b0
    ADD  r12 r10 r11
    SUB  r13 r10 r11
    MUL  r14 r12 r13
    ...
```

Here, explicit synchronization is again performed for inputs `x` and `y`, but then all three arithmetic operations are performed in sequence. While in this translation we give up the flexibility

to execute the addition and subtraction operations in any order (and possibly in parallel), the instruction count is reduced considerably through the elimination of synchronization overhead and transfers of control. Also, by reducing the number of transfers of control, pipeline performance is improved, and locality is enhanced so that it may be easier to store intermediate results in high speed memory.

When is it beneficial to perform such sequentialization of operations? In the example above, it is clearly worthwhile. If the addition and subtraction, however, were replaced by more complex operations involving, say, disk accesses, then it may be advantageous to allow them to be executed in parallel. In the AGNA compiler, we utilize a simple heuristic developed by Iannucci called the Method of Dependence Sets (MDS) [44] to address this question. The net result is, roughly, that parallelism is preserved between long-latency operations, and sequential code is generated for connected subgraphs that do not involve such operations.

We decided to use MDS in AGNA because (1) it is simple and well understood; (2) it is provably deadlock-free; and (3) its latency-directed approach seemed particularly appropriate given the many long-latency operations in a parallel, persistent system. While a possible area of future research, we have not undertaken in this work a thorough analysis of its strengths and weaknesses relative to other approaches.

Method of Dependence Sets

MDS operates by dividing graph instructions into partitions in such a way that all instructions in a partition P_i depend, either directly or indirectly, on the same set of long-latency outputs. There are three basic kinds of long-latency outputs in the graph instructions used in the AGNA compiler:

- Internal outputs of LAMBDA. Such arcs are considered to have a long latency because of the non-strictness of procedure calls in AGNA.
- Outputs of APPLY.
- Outputs of manager instructions such as LOOKUP, SELECT-FIELD, UPDATE-FIELD, *etc.*

An example partitioning produced by MDS, shown in Figure 4.20, divides the body of the procedure into partitions P_1, P_2, and P_3. Partition P_1 contains the LOOKUP and CONSTANT instructions, which depend only on the *Trigger* arc. Partition P_2 contains the arithmetic instructions

and RESULT-RETURN, which depend on the long-latency outputs of the two LOOKUPs and the trigger. Finally, partition P_3 contains SIGNAL-RETURN, which depends on all of the long-latency outputs in the procedure body.

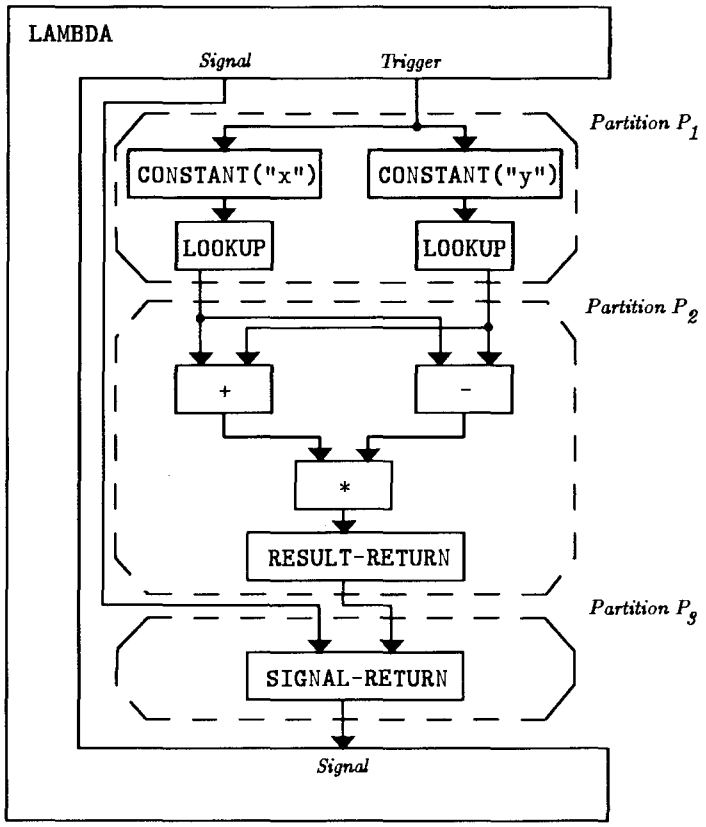


Figure 4.20: Partitioning of graph for `(lambda () (* (+ x y) (- x y)))`.

This kind of partitioning is performed on the graph in the body of each procedure definition (*i.e.*, the instructions encapsulated by each LAMBDA), as well as the body of the top-level XACT. The partitioning algorithm, applied to either LAMBDA or XACT, proceeds as follows:[8]

1. Topologically sort instructions in encapsulator body.

2. Give each long-latency output in the graph a unique name.

3. For each instruction i, in topological order:

   (a) Compute $LLDS(i)$, the long-latency dependence set, as follows. Let J be the set of instructions from which i receives input. Let O be the set of output arcs which

[8] As described in [44], MDS works only on *acyclic* graphs, and thus cannot accommodate the graphs generated by expressions such as `(letrec ((c (f a b c))) c)`. Here, we assume all graphs are acyclic.

connect instructions in J to i.

$$LLDS(i) = (\bigcup_{j \in J} LLDS(j)) \cup \{o \mid o \in O \wedge \textit{long-latency}(o)\}$$

In words: $LLDS(i)$ includes $LLDS(j)$, for all instructions j from which i receives input, plus all long-latency output arcs o which feed i directly.

(b) Place i in the partition with dependence set $LLDS(i)$. If no such partition exists, then create one.

After partitioning (and allocation of frame slots, to be described next), code generation is performed within a partition by expanding, in topological order, each graph instruction to the P-RISC instructions that implement it. Explicit transfers of control to consumer instructions and synchronization of inputs are performed only for data flowing along inter-partition arcs. For example, for partitions P_1, P_2, and P_3, the order in which instructions are expanded, and the explicit control transfers and synchronization are shown in Figure 4.21. In P_1, FORK is used prior to the first LOOKUP to initiate concurrent execution of the remainder of the partition, and both LOOKUPs transfer control to the start of P_2 after execution. In P_2, after synchronization of inputs, execution is completely sequential. Note that the - instruction does not need to perform additional synchronization of its inputs x and y, nor do the LOOKUPs need to transfer control directly to -. The sequentialization of instructions in P_2 ensures that both inputs are present when the subtraction is performed. Finally, P_3 synchronizes its inputs and executes SIGNAL-RETURN.

An important issue to consider when introducing additional execution constraints (*e.g.*, sequentiality) not present in the source language is deadlock. For example, if we introduce a new constraint that forces a SELECT-FIELD graph instruction to execute before the corresponding UPDATE-FIELD, then the program will deadlock because the SELECT-FIELD will block indefinitely. In [44] it is shown that the partitioning performed by MDS does not introduce deadlock.

Finally, we note that enhancements to the basic MDS Algorithm have been developed by David Culler and his research group at U.C. Berkeley [67]. Additional analysis is utilized to produce larger partitions, and extensions have been developed to partition cyclic graphs.

4.5.2 Frame Slot Allocation

After graph analysis, the next step in the generation of P-RISC code from dataflow graphs is the mapping of temporary storage implicit in the graph to frame slots. This storage includes

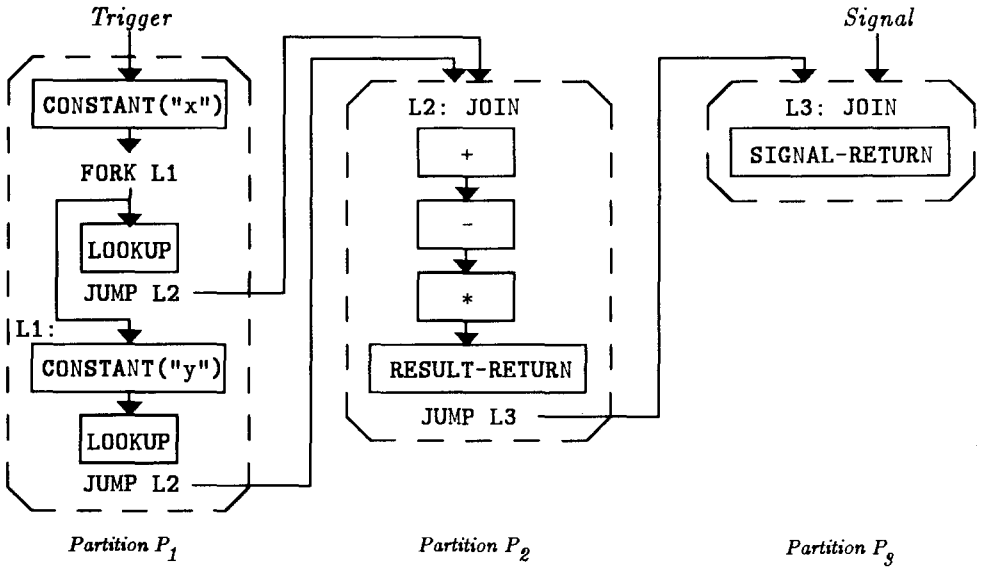


Figure 4.21: Synchronization and control transfers for P_1 and P_2.

slots for: (1) synchronization; (2) values that are carried on tokens; and (3) internal use by the expansion of an instruction. We first describe the organization of frames, and then present the algorithm for frame slot allocation.

Frame Organization

There are two different types of frames: *transaction* and *procedure*. Exactly one of the first type is allocated per transaction execution, while one of the second type is allocated for each application of a user-defined procedure. Both consist of a chunk of linearly-addressed memory. Unlike the heap, frames are not accessed by synchronized reads and writes, so extra status bits are not needed in frame memory.

The first two slots in a transaction frame (shown in Figure 4.22) contain constant `0`, and a pointer to the frame itself. Because these values are used so frequently, slots for them are allocated statically and they are written by the frame allocation manager. Instructions in the body of the transaction may read from but not write to these slots. The remaining slots in the frame provide dynamic storage for instructions in the transaction body.

The first six slots in a procedure frame (also shown in Figure 4.22) are allocated statically. Again, the first two slots contain constant `0` and a pointer to the frame itself. The next four contain the following return information: the caller's FP, the caller's result and signal IPs, and the slot in the caller's frame where the result is to be stored. The next set of contiguous

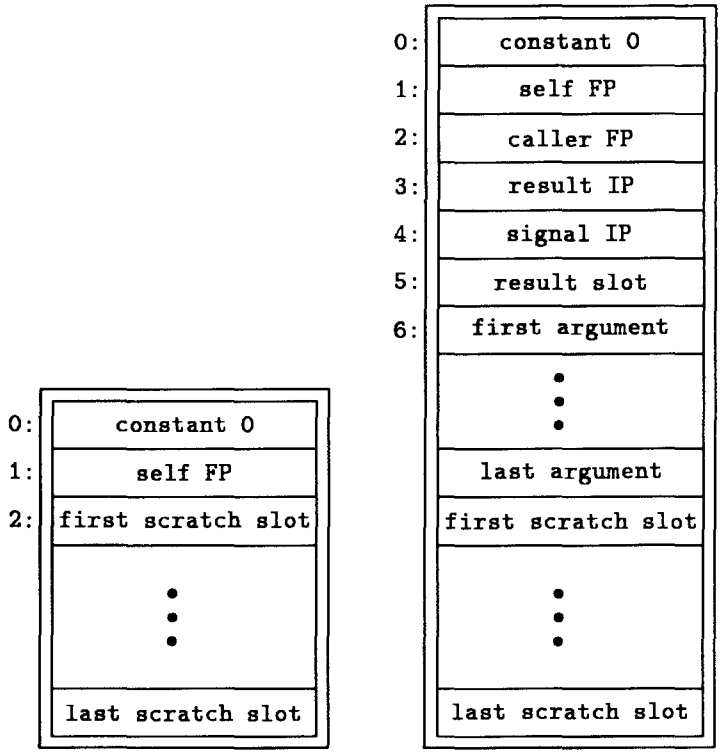


Figure 4.22: Organization of transaction frame (left) and procedure frame (right).

slots contains arguments to the procedure and, as before, the remaining slots provide dynamic storage for instructions in the body.

Algorithm

The algorithm for mapping graph storage to frame slots traverses the graph in a transaction or procedure body and allocates, for each instruction, slot bits for synchronization, slots for internal use, and slots for outgoing, value-carrying arcs. Simple static analysis is employed to determine when an allocated frame slot is no longer accessible and thus may be reused safely. Static techniques, however, may not always be sufficient to determine at what point in the graph it is safe to reuse a frame slot, because the relative execution order of instructions may be left unspecified by the compiler and determined only at run-time. For example, consider slot `ri` which holds the result of ALLOCATE-OBJECT, shown in Figure 4.23. It is safe to reuse `ri` only after all instructions J_i have executed and consumed its value, and for the reason just described, this may not be determinable at compile-time. In the AGNA compiler, we identify two common cases for which static analysis can determine when it is safe to reuse `ri`:

- There is only one consumer instruction, J_1; slot `ri` may be reused safely when J_1 has consumed its value.
- All J_i reside in the same partition and are executed sequentially; slot `ri` may be reused after the last J_i has executed.

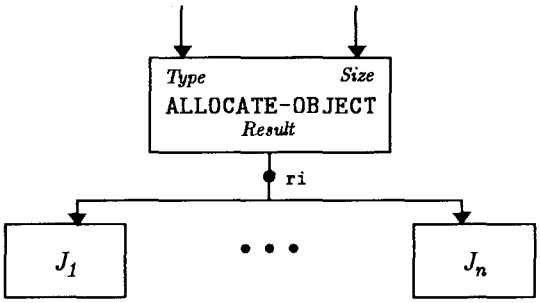


Figure 4.23: Reuse of result value in slot `ri`.

Another issue to consider is when slots used internally by an instruction may be reused. For a strict instruction such as ALLOCATE-OBJECT, internal slots may be reused as soon as a result token is produced. For a non-strict instruction such as APPLY, internal slots may be safely reused as soon as a signal token is generated.

In AGNA, each input port of an instruction has associated with it two sets of slots, called *free* and *to-free*, that are used by the compiler to propagate across the graph information describing when a slot may be reused. The *free* set contains slots that are available for immediate reuse, while *to-free* contains slots that are available for reuse by an instruction's successors. For example, consider the graph shown in Figure 4.24. Result slot `r12` and internal slot `r13` are allocated for use by ALLOCATE-OBJECT from the incoming free set. The *free* set passed on to APPLY consists of unused free slot `r14` and *to-free* slots `r15` through `r18`. Slots `r12` and `r13` used by ALLOCATE-OBJECT are placed in the *to-free* set passed to APPLY which, in turn, will place them in the *free* set passed to its successors.

For non-strict instructions such as APPLY, unused free slots are passed on to consumers of the result output. If there is more than one consumer, as shown in Figure 4.25, the slots are divided evenly amongst them. Thus, free slots `r16`, `r17`, and `r18` not used by APPLY are split between I_1 and I_2. APPLY's internal slots are added to the *to-free* set of the input port connected to the signal output. In the figure, it is assumed that the relative execution order of I_1 and I_2 is *not* known at compile-time, and thus APPLY result slot `r15` is *not* added to the *to-free* set of I_1 or I_2.

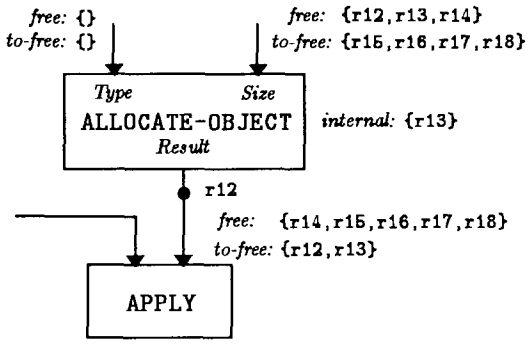


Figure 4.24: Propagation of *free* and *to-free* sets across ALLOCATE-OBJECT.

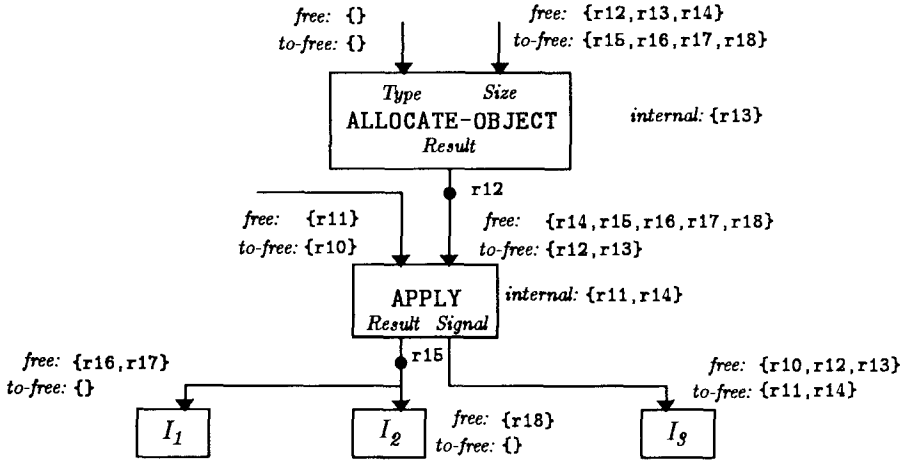


Figure 4.25: Propagation of *free* and *to-free* sets across APPLY.

The rules for propagating *free* and *to-free* information can be summarized as follows. For a strict instruction: *to-free* and unused *free* slots are split amongst successor *free* sets; internal slots are split amongst successor *to-free* sets; and the result slot is added to the *to-free* set of the last successor to execute, if this instruction can be identified statically (according to the analysis described above). For a non-strict instruction: unused *free* slots are split amongst *free* sets of consumers of the result; *to-free* slots are split amongst *free* sets of consumers of the signal; internal slots are split amongst *to-free* sets of consumers of the signal; and the result slot is added to the *to-free* set of its last consumer, again, if the instruction can be identified statically.

The algorithm for allocating frame slots traverses the graph in an XACT or LAMBDA body in topological order, and performs the following for each instruction I:

- Allocation of slot bits for explicit synchronization of inputs. Input ports requiring such synchronization are exactly those identified by the graph analysis described previously in Section 4.5.1. A unique label is also associated with each input port; these are used later in the actual generation of P-RISC code.

- Compute set $FREE(I)$ of slots available for reuse as follows. Let IP be I's input ports.

$$FREE(I) = \bigcup_{ip \in IP} ip.free$$

  Allocate from $FREE(I)$ slots needed for results and internal use; if set is exhausted, then allocate new, unused frame slots as necessary. Allocation of such slots is performed by incrementing a "high water" mark, initially set to the first free slot in the frame.

- Propagate to I's successors *free* and *to-free* information as described above.

Allocation of synchronization bits in the first step is straightforward and proceeds as follows. Before the graph traversal begins, the first scratch slot is allocated for synchronization, and a marker indicating the number of bits used within the slot is set to zero. When the algorithm needs to allocate synchronization bits, it simply increments the marker by the appropriate number. If the marker happens to exceed the bit length of a slot (*i.e.*, all bits in the current slot are allocated), then a new, unused frame slot is assigned for synchronization, and the marker is reinitialized to zero. Frame slots allocated for synchronization, and bits in such slots, are never reused.[9]

[9]For the programs that we have compiled and run on our prototype system, which has 64 bit frame slots, only rarely does a procedure or transaction body require more than one slot for synchronization.

The second step is complicated somewhat by the requirement of some instructions that slots for internal use be contiguous. The reason for this requirement is that P-RISC manager instructions, as described in Section 4.4, are passed arguments in contiguous frame slots.

4.5.3 Code Generation

After partitioning and frame slot allocation, the next and final step translates the dataflow graph to P-RISC code by traversing the graph in the transaction body in topological order and invoking an expander function on each instruction to generate the equivalent P-RISC code. Results of the partitioning and slot allocation steps are stored in data structures representing the graph and thus are available to the expander functions, of which there is one for each type of instruction. An expander is solely a function of the instruction and its attributes (inputs, outputs, partition, *etc.*) and has no global knowledge of the graph. Code produced by expander functions is accumulated in partition objects created and added to the graph in the partitioning step.

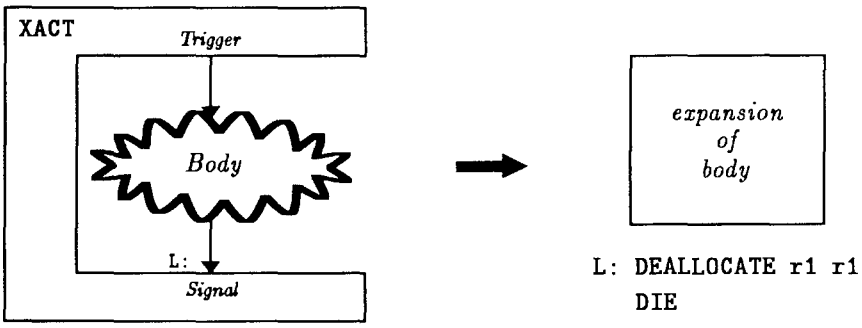


Figure 4.26: Expansion of XACT.

P-RISC code of the transaction consists of the results of expanding the body, followed by frame deallocation and thread termination as shown in Figure 4.26. Allocation of a frame for the transaction is performed by the run-time system when the transaction object is loaded into memory, and not explicitly by the transaction itself. Code associated with each partition in the body is terminated with a DIE instruction prior to the appending of partitions to form the final code sequence, so there is no danger that sequential flow of control in one partition will incorrectly fall through and begin executing in another. The final transaction object, *i.e.*, the result of compilation, is structured as follows.

- Standard object header. This contains things like a pointer/data flag, type tag, *etc.*

- Frame size and base address. The frame size is used by the run-time system to allocate a frame of the appropriate size prior to execution of the transaction body. The base address, initially zero, is the heap address at which the transaction object is assumed to reside, and is used by the compiler to construct intra-object pointers such as references to objects in the static data area (described below). When the transaction object is assigned an actual heap address at run-time, the base address and all intra-object pointers are updated.
- Code area. This contains P-RISC code of the transaction.
- Static data area. This contains embedded data objects such as strings and procedures.

Synchronization and Transfers of Control

As described in Section 4.5.1, explicit synchronization of inputs and transfers of control to consumer instructions are needed only for inter-partition arcs. Since these aspects of code generation are identical for all strict instructions, they are not repeated in each instruction-specific expander, but rather factored out and performed by procedures `pre-expand` and `post-expand`. Thus, code generation for a strict instruction I actually consists of:

(1) `pre-expand(`I`)`;

(2) *instruction-specific expansion*; and

(3) `post-expand(`I`)`.

For example, consider the translation of SELECT-FIELD in Figure 4.27. Since SELECT-FIELD is a long-latency operation, procedure `pre-expand` adds a FORK to label L2 to initiate concurrent execution of the remainder of partition P_2, and a JOIN instruction to synchronize the *Object* input using the label, frame slot, and bit assigned to the input port during slot allocation. Note that synchronization is *not* needed for the *Type* and *Field* inputs because they are produced within the same partition, P_2. Line (3) contains code produced by the expander for the SELECT-FIELD instruction; we will see soon how this and other object-manipulation instructions are translated. Finally, `post-expand` adds a JMP to the consumer instruction in partition P_3 and a NOOP as the target of FORK in line (1).[10]

For non-strict instructions (APPLY, TAIL-APPLY, and IF), `pre-expand` and `post-expand` initiate concurrent execution of the remainder of the thread as for strict instructions (lines (1) and (5)

[10] All such NOOPs are eliminated during peep-hole optimization.

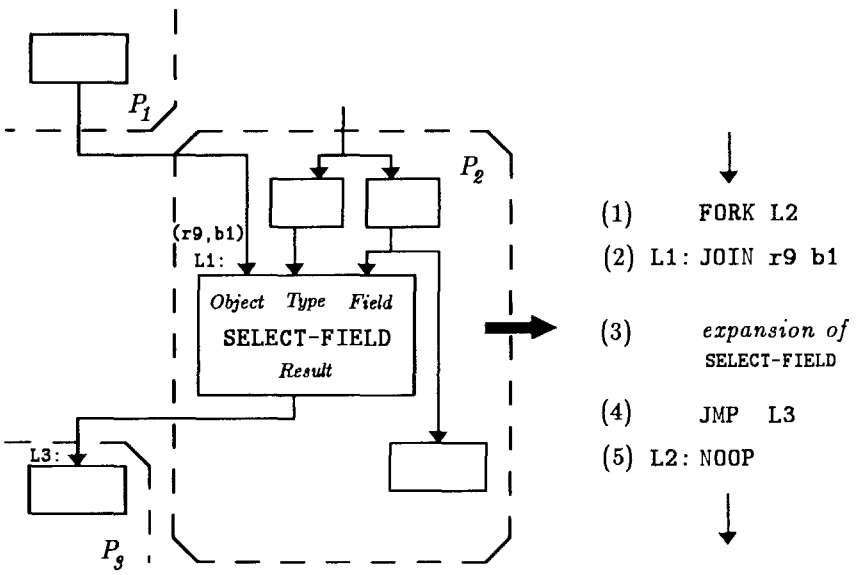


Figure 4.27: Explicit synchronization and control transfers.

in Figure 4.27), but do not add explicit synchronization or transfers of control to consumers. These are handled by the expander functions themselves, to be described soon, because they are different for each of the three non-strict instructions.

Constants

A CONSTANT graph instruction is expanded to LOADC, the P-RISC instruction that loads a data value into a frame slot. If a constant can fit in a single frame slot, such as an integer, then it is included directly in the instruction as shown in Figure 4.28. Other constants, such as long strings, are placed in the static data area and a pointer to them is loaded, also via LOADC.

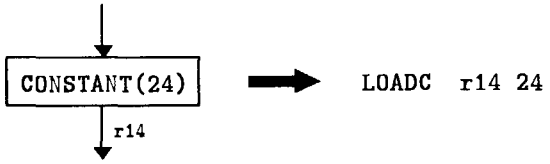


Figure 4.28: Expansion of CONSTANT(24).

Primitive Applications

Expressions involving primitive operations such as arithmetic, logical, and relational operators are expanded in the obvious way. For example, the + instruction is translated as shown in Figure 4.29.

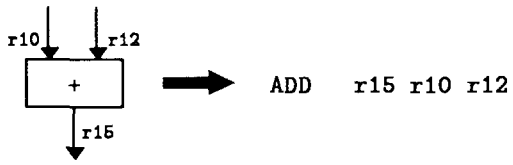


Figure 4.29: Expansion of +.

Object Manipulation

Object allocation, field update, field insertion and deletion, field inverse-mappings, and other kinds of object manipulation are expanded to code which places the arguments in contiguous frame slots, and then calls the appropriate manager. For example, the SELECT-FIELD instruction shown in Figure 4.30 calls manager SELECTF after copying its arguments to frame slots r11, r12, and r13, the slots allocated for internal use.

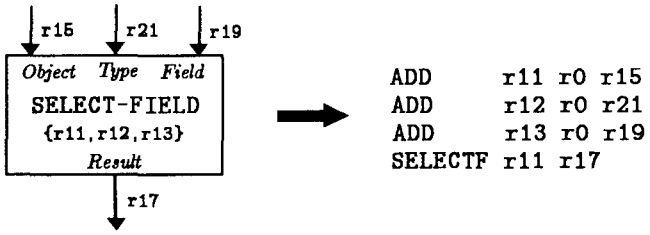


Figure 4.30: Expansion of SELECT-FIELD.

Procedure Definition

Expansion of LAMBDA recursively expands instructions in the body and packages the resulting code into a procedure object with the same overall structure as the transaction object described previously. The procedure object is placed in the transaction's static data area, and a reference to it (*i.e.*, the result of LAMBDA) is placed in the result slot via LOADC.

Result and termination signals produced by a procedure are returned via RESULT-RETURN and SIGNAL-RETURN, respectively, whose expansions are shown in Figure 4.31. RESULT-RETURN is translated to START1, which stores the result of the procedure in the caller's frame, and activates the thread to receive this value. SIGNAL-RETURN is translated to START0, which activates the signal-receiving thread. Both instructions use the linkage information stored in frame slots r2 through r5.

The structure of the code area of a procedure with two arguments is shown in Figure 4.32. At the beginning is a thread-initiation table that contains one entry for each input to the body

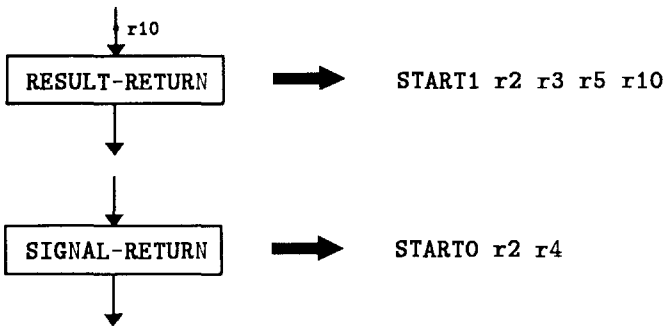


Figure 4.31: Expansion of RESULT-RETURN and SIGNAL-RETURN.

expression. As we shall see soon, callers of the procedure transfer control to instructions in this table to initiate execution of trigger, signal, and argument threads in the body. Each entry in the table is a JMP, whose target is either the consumer instruction itself, if there is only one, or a dispatch area immediately following the table for initiating execution of multiple consumers of an input. For example, argument x has multiple consumers as shown in the figure, and execution is initiated by the FORK and JMP following the table. While the FORK and JMP could be included directly in the thread-initiation table, this would complicate matters at the call site as the offset for a thread in the body would, in general, have to be determined at run-time. If the table contains only one entry per input, however, then offsets can be computed at compile-time.

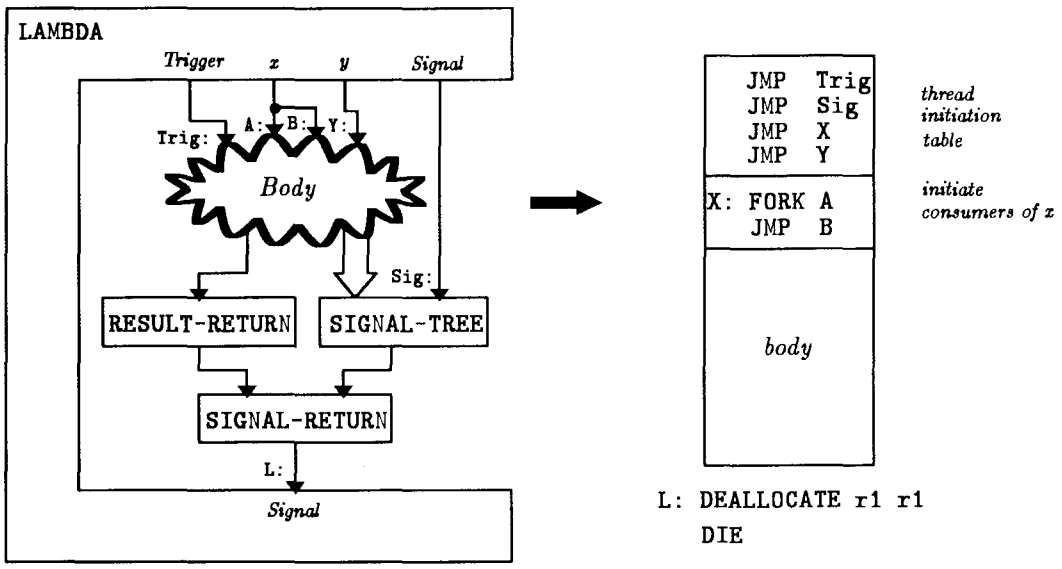


Figure 4.32: Organization of code in procedure body.

Code associated with each partition in the procedure body is terminated with a DIE instruction prior to the appending of partitions to form the code sequence labeled *body* in the figure,

so as with transactions, there is no danger that sequential flow of control in one partition will incorrectly fall through and begin executing in another. At the *Signal* input port of the bottom arm of LAMBDA, the procedure frame is deallocated, and execution is terminated.

P-RISC code generated for the SIGNAL-TREE instruction used in Figure 4.32 is produced entirely by pre-expand and post-expand; its expander function generates no code. In other words, the code for SIGNAL-TREE consists entirely of synchronization and transfers of control to successor instructions.

Procedure Applications

Procedure applications in AGNA perform: synchronization of long-latency inputs; allocation of a new frame; transmission of linkage information and argument values; initiation of threads in the procedure body; and setup of threads to receive the result and termination signal. Consider the APPLY instruction in Figure 4.33 that applies a procedure to two arguments; its expansion consists of three contiguous segments of code. The first segment, given below, performs explicit synchronization of the *Procedure* input and allocates a new frame.

```
(1) Proc: JOIN  r3  b9          % synchronize Procedure input
(2)       LOADC r18 24          % fetch frame size at address r10+24
(3)       ADD   r18 r18 r10
(4)       LOAD  r18 r18
                                % load args and call frame allocator
(5)       ADD   r14 r0  r1      % caller FP = self FP, in r1
(6)       LOADC r15 Res         % result IP = label Res
(7)       LOADC r16 Sig         % signal IP = label Sig
(8)       LOADC r17 20          % result slot = r20
(9)       ALLOCFRAME r14 r14    % allocate new frame, r14 ← FP
(10)      FORK  Arg2            % new FP available, go store and start arg 2
```

After synchronization in line 1, the frame size and linkage information are loaded into slots r14 through r18 and the frame allocation manager is invoked. The frame size is read in lines 2 through 4 from a known offset (24) from the start of the procedure object. Linkage information is loaded in lines 5 to 8: the caller FP is the current frame's FP, stored in r1; the result and signal IPs are labels Res and Sig, respectively; and the result slot is r20. Manager ALLOCFRAME (in line 9) allocates a new frame, stores the linkage information, and returns a pointer to the frame in slot r14. After frame allocation, a thread is spawned in line 10 at label Arg2 (code given below) to store the procedure's second argument and initiate its thread of execution in the body.

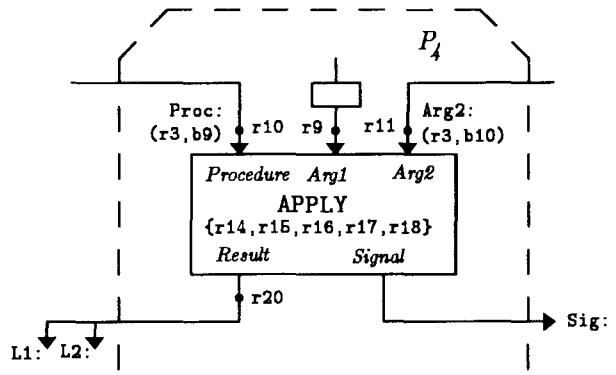


Figure 4.33: APPLY instruction, annotated with labels and slot information.

The second segment of code transmits the first argument and initiates execution of threads in the procedure body corresponding to the trigger, signal, and first argument inputs.

```
(11)      LOADC  r15 32            % start trigger thread at r10+32
(12)      ADD    r15 r15 r10
(13)      START0 r14 r15
(14)      LOADC  r15 40            % start signal thread at r10+40
(15)      ADD    r15 r15 r10
(16)      START0 r14 r15
(17)      LOADC  r15 48            % start arg1 thread at r10+48
(18)      ADD    r15 r15 r10
(19)      LOADC  r16 6             % first arg stored in slot 6
(20)      START1 r14 r15 r16 r9
(21)      DIE
```

Execution of both the trigger and signal threads is initiated via START0 in lines 13 and 16. The FP used in both cases is the new frame pointer in slot r14. The target IPs for the threads are computed by adding known offsets (32 and 40) to the address of the procedure stored in slot r10. Lines 17 to 20 store the first argument of the procedure in slot six in the new frame, just past the linkage information, and activate the thread to receive it.

The final segment of code in the expansion of APPLY in Figure 4.33 transmits the second argument and sets up a thread to receive the procedure's result.

```
(22) Arg2: JOIN   r3  b10          % wait for both arg value and frame pointer
(23)       LOADC  r17 56           % start arg1 thread at r10+48
(24)       ADD    r17 r17 r10
(25)       LOADC  r18 7            % second arg stored in slot 7
(26)       START1 r14 r17 r18 r11
(27)       DIE

(28) Res:  FORK   L1               % dispatch to consumers of result
(29)       JMP    L2
```

Explicit synchronization is performed first in line 22, waiting for both the argument value and new frame pointer to be computed. When they are available, the argument value (in r11) is stored in slot r7 and the corresponding thread is started via START1. The thread receiving the result of the application (at label Res, line 28) branches to consumer instructions at labels L1 and L2. Since the termination signal has only a single consumer (at label Sig), control is transferred directly to it from the procedure body.

While the expansion given above is for the particular APPLY instruction shown in Figure 4.33, the general translation scheme used for applications should be clear. First, synchronization is provided for the procedure input, if necessary, after which a new frame of the appropriate size is allocated. When the frame is available, threads in the procedure body receiving the trigger, signal, and all arguments not requiring explicit synchronization are started. Arguments requiring synchronization are started when both the argument value and new frame pointer are available. Finally, threads are set up to dispatch to multiple consumers of the result or termination signal.

Tail Calls

Expansion of TAIL-APPLY is similar to APPLY, but different in several important aspects. Recall from Section 4.3.6 that a procedure f called in a tail-recursive manner from a procedure g returns its result and termination signal directly to the computations waiting for the result and signal of g (say C_1 and C_2), instead of passing them first to g and then to C_1 and C_2, as in a normal application. In a tail call, procedure g also propagates its termination signal forward to f, rather than back to its caller.

Consider the TAIL-APPLY instruction shown in Figure 4.34. As with APPLY, the first segment of the expansion performs explicit synchronization of the *Procedure* input and allocates a new frame.

```
(1) Proc: JOIN  r3  b9            % synchronize Procedure input
(2)       LOADC r18 24            % fetch frame size at address r10+24
(3)       ADD   r18 r18 r10
(4)       LOAD  r18 r18
                                  % load args and call frame allocator
(5)       ADD   r14 r0  r2        % caller FP = return FP, in r2
(6)       ADD   r15 r0  r3        % result IP = return result IP, in r3
(7)       ADD   r16 r0  r4        % signal IP = return signal IP, in r4
(8)       ADD   r17 r0  r5        % result slot = return result slot, in r5
(9)       ALLOCFRAME r14 r14      % allocate new frame, r14 ← FP
(10)      FORK  Arg2              % new FP available, go store and start arg 2
```

```
(11)      ADD   r2  r0  r14        % return FP ← new FP
(12)      LOADC r15 40
(13)      ADD   r4  r10 r15        % return signal IP ← callee signal IP
```

After synchronization in line 1, the frame size and linkage information are loaded into slots r14 through r18 and the frame allocation manager is invoked as before in APPLY. Here, however, the linkage information in the new frame (r2 through r5) is set to return to the current procedure's caller, rather than the procedure itself: in line 5, the caller FP is set to the current frame's return FP, stored in slot r2; in lines 6 and 7, the result and signal IPs are set to the current frame's result and signal IPs, respectively, stored in r3 and r4; and in line 8, the result slot is set to the contents of r5, the current frame's result slot. Next, a thread is spawned in line 10 at label Arg2 to start the second argument, as before. Finally, lines 11 through 13 update the return FP and signal IP of the current procedure to the new FP and signal IP of the procedure being applied so that SIGNAL-RETURN propagates the termination signal to the callee as desired.

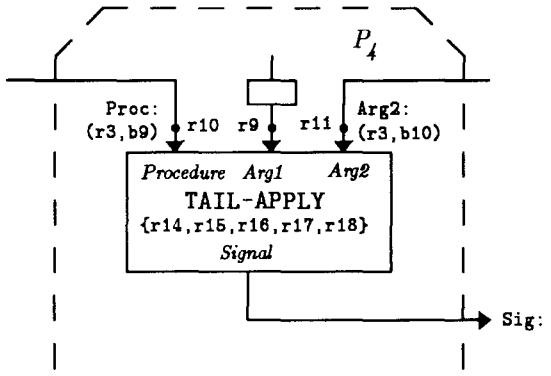


Figure 4.34: Annotated TAIL-APPLY instruction.

The second segment of code transmits the first argument and initiates execution of threads in the procedure body corresponding to the trigger and first argument inputs.

```
(14)      LOADC  r15 32            % start trigger thread at r10+32
(15)      ADD    r15 r15 r10
(16)      START0 r14 r15
(17)      LOADC  r15 48            % start arg1 thread at r10+48
(18)      ADD    r15 r15 r10
(19)      LOADC  r16 6             % first arg stored in slot 6
(20)      START1 r14 r15 r16 r9
(21)      JMP    Done
```

As before, the trigger and first argument are started in lines 14 through 20. Note that here we do not start the signal thread, however, because it is started later by SIGNAL-RETURN. The final

instruction in line 21 transfers control to label Done which, as we shall see below, generates the *Signal* output of TAIL-APPLY when the application is complete.

The last segment of code:

```
(22) Arg2: JOIN   r3  b10          % wait for both arg value and frame pointer
(23)       LOADC  r17 56           % start arg1 thread at r10+48
(24)       ADD    r17 r17 r10
(25)       LOADC  r18 7            % second arg stored in slot 7
(26)       START1 r14 r17 r18 r11

(27) Done: JOIN   r3  b9
(28)       JMP    Sig
```

As before, explicit synchronization is performed in line 22 and the second argument is started in lines 23 to 26. Here, however, synchronization is performed in line 27, waiting for the trigger and all argument inputs to be started, after which control is transferred to label Sig, the consumer of TAIL-APPLY's signal output.

Conditionals

Conditionals are expanded to code that performs: explicit synchronization of long-latency inputs; triggering and routing of free variable inputs to the appropriate arm of the conditional; and transfers of control to consumers of the result and termination signal. For example, expansion of the IF instruction shown in Figure 4.35 is given below.

```
(1)  Pred:  JOIN r9  b1        % synchronize Predicate input
(2)         FORK FV2           % predicate available, go handle FV2
(3)         JMPT r10 L         % if predicate is true, go to L
(4)         FORK EFV1          % start consumer of FV1 in Else branch
(5)         JMP  ETrig         % trigger Else branch
(6)  L:     FORK TFV1          % start consumer of FV1 in Then branch
(7)         JMP  TTrig         % trigger Then branch
(8)  FV2:   JOIN r9  b2        % synchronize FV2 input
(9)         JMPT r10 TFV2      % if predicate is true, route FV2 to Then branch
(10)        JMP  EFV2          % route FV2 to Else branch

(11) TRes:  ADD  r20 r0 r13    % move Then result to result slot r20
(12)        FORK L1            % branch to consumers of Result
(13)        JMP  L2
(14) ERes:  ADD  r20 r0 r12    % move Else result to result slot r20
(15)        FORK L1            % branch to consumers of Result
(16)        JMP  L2
(17) TSig:  NOOP
(18) ESig:  JMP  L3            % branch to consumer of Signal
```

After synchronization of the incoming predicate in line 1, a thread is spawned in line 2 at label FV2 to handle long-latency input *FV2* when it is available. Next, control is transferred to TFV1 and TTrig in lines 6 and 7 (the *Then* branch) or EFV1 and ETrig in lines 4 and 5 (the *Else* branch), depending on whether the predicate is true or false. Note that the consumer of *FV1* can be safely started at this point because the value of *FV1* is delivered on an intra-partition arc. The next three instructions in lines 8 to 10 wait for both the *Predicate* and *FV2* inputs, and then branch either to TFV2 or EFV2, again, depending on the predicate value.

Lines 11 to 13 and 14 to 16 receive the results of the *Then* and *Else* branches, copy them to final result slot r20, and transfer control to consumer instructions at labels L1 and L2. Lines 17 and 18 provide branch targets for producers of *ThenSignal* and *ElseSignal*, and transfer control to the consumer instruction at label L3.

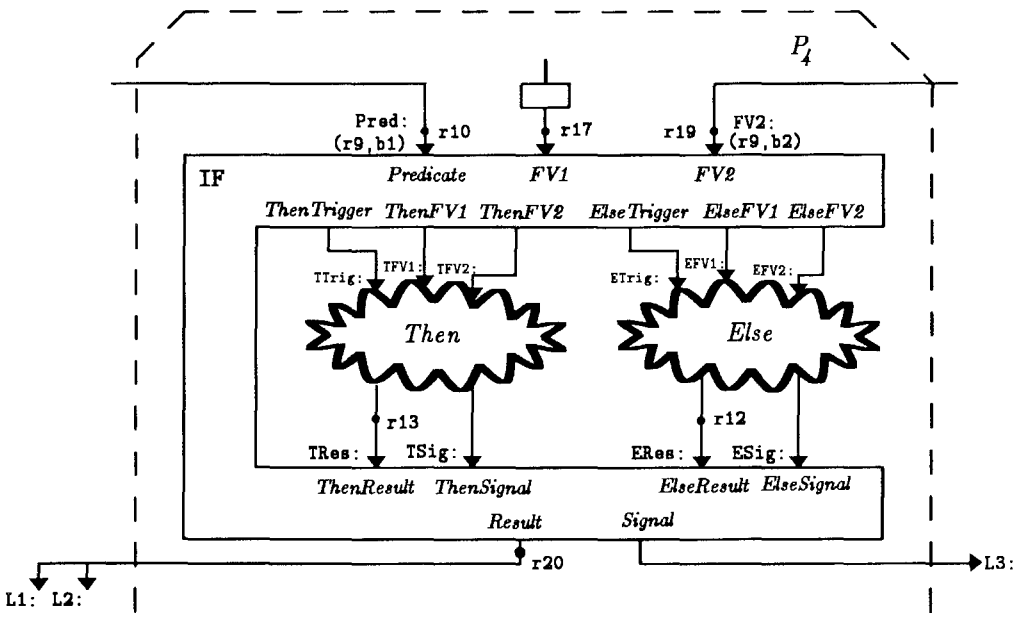


Figure 4.35: Annotated IF instruction.

The general translation scheme for IF is summarized is follows. First, explicit synchronization is performed for the predicate input, if necessary, after which the *Then* or *Else* branch is triggered, according to the value of the predicate. Next transfers of control are made to *Then* or *Else* consumers of free variables arriving on intra-partition arcs. Then, explicit synchronization is performed for free variables arriving on inter-partition arcs, and transfers of control are again made to corresponding *Then* or *Else* consumers. Finally, code is generated that receives the result and signal outputs of both arms of the conditional, copies the results to the final result

slot, and transfers control to successor instructions.

4.5.4 Phase Three Optimizations

Peep-hole optimization is performed on code in the transaction body and code for each embedded procedure. Four types of optimization are performed:

1. NOOP instructions are eliminated. Such instructions are not necessary and were introduced only to simplify code generation.

2. FORK instructions for which the target is a DIE instruction are eliminated. This situation may result, for example, from a FORK generated by pre-expand prior to a long-latency operation. The FORK is intended to initiate concurrent execution of the remainder of the thread, but if the instruction happens to be the last one expanded in the partition, the FORK target may be DIE.

3. JMPs to successor instructions are eliminated. This situation often appears when code from the various partitions in a transaction or procedure are appended together. For example, the last instruction in one partition may transfer control to a consumer at the beginning of the next partition.

4. Transfers of control to JMP instructions are "forwarded". In other words, the label to which a JMP instruction transfers control is inserted directly in all control-transfer instructions for which it is the target.

Chapter 5

Implementation of the P-RISC Abstract Machine

In the previous chapter, we described how transactions are compiled to code for the P-RISC abstract machine. In this chapter, we describe how the abstract machine is implemented on a real multiprocessor.

5.1 Overview

We are targeting our implementation to MIMD multiprocessors consisting of one or more processor-memory elements (PMEs), interconnected via a high-speed network (see Figure 5.1). Each PME consists of a processor, some local memory, and an attached disk. The network may be a bus, as is typical in small multiprocessors, or a switching network in larger machines.

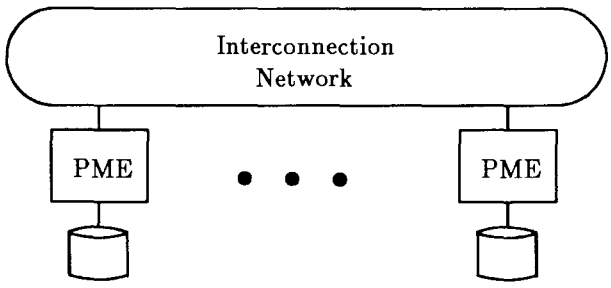


Figure 5.1: Target machine organization.

We feel that this particular architecture is an appropriate choice for two reasons. First, we believe that a distributed instead of shared model of physical memory is easier to scale to larger

machines. Second, the clustering of a processor, memory, and disk at a PME allows the locality of persistent data to be exploited. For example, a transaction may filter an enormous amount of persistent data (*e.g.*, terabytes), and having processing power close to each disk allows filters to be applied locally. If all persistent data must be read from a remote disk and transferred through the network, then the latency of access is increased and valuable network bandwidth is consumed, much of it potentially by transfers for data that are quickly discarded by the filter.

The P-RISC abstract machine may be implemented in a number of different ways. One possibility is to build a P-RISC processor directly in hardware [69], while another is to translate P-RISC instructions into native machine code of a real multiprocessor [59]. In AGNA we have taken yet another approach, implementing the P-RISC machine via software emulation. The motivation for this decision is twofold. First, this approach provides a great deal of flexibility and portability (the emulator is written in C), since we can easily change platforms, add run-time metering, *etc.* Second, this requires less effort than the other approaches, and before pursuing a more complicated (and efficient) implementation, we wanted to verify that the overhead of emulation was indeed a bottleneck. If the limiting factor were the disk or network, for example, then more efficient execution of P-RISC instructions would have little impact on overall performance.

Software for the AGNA system is structured into three separate programs, as shown in Figure 5.2 (a similar system organization is used by Culler [29]). Two of the programs, the compiler and a command interpreter, run on the front-end machine, while a single, identical copy of the third program, the P-RISC emulator, runs continuously on each processor of the back-end machine. The user interacts with the system as follows. He first submits the source of a transaction to the compiler, which translates it as described in the previous chapter, and then writes the output to a P-RISC "executable" file on the front-end machine. He then issues a "load" command to the command interpreter to download the compiled transaction into the heap memory of the machine. Finally, he issues a "run" command, again to the command interpreter, to execute the transaction and print the result. Additional commands supported by the command interpreter allow the user to create and destroy databases, review statistics gathered during execution, *etc.*

A prototype implementation of the AGNA software was developed on workstations running the Unix operating system[1], interconnected via a local area network. The compiler is written in

[1] Unix is a registered trademark of AT&T.

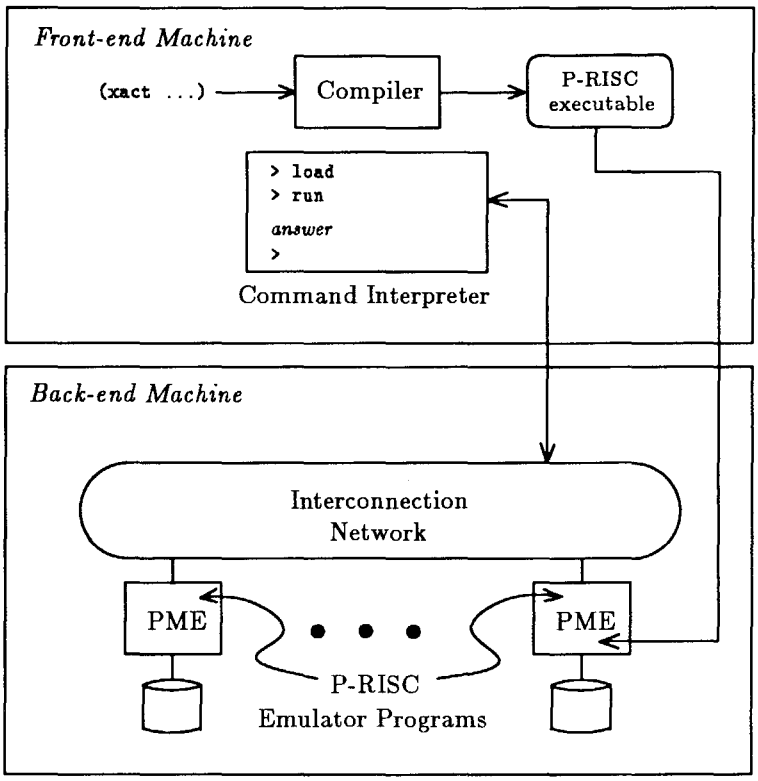


Figure 5.2: AGNA system structure.

Common Lisp, while the command interpreter and P-RISC emulator are written in C. Several P-RISC emulators may run on a single workstation, or each emulator may be mapped to its own physical machine. The development environment and programming languages (*i.e.*, Unix, Lisp, and C) were chosen because of their widespread availability, their flexibility, and the many program development tools such as profilers and debuggers that they support. We have also ported our software to an Intel iPSC/2 Hypercube with 32 processors and 32 disks. We give a detailed description of the iPSC/2 in the next chapter.

Outline of Chapter

In the next two sections we describe how the heap and frame memory of the P-RISC machine are implemented in the distributed memory and disks of the parallel machine. Then we describe the organization of the emulator process and how it schedules and executes instructions. Next we discuss the representations used for indexes and multi-valued fields. Then we explore the issues of how computation and data are distributed across the machine, and finally, we end with a description of the different phases of execution of an AGNA transaction.

5.2 Mapping the Heap to the Physical Machine

Heap memory of the P-RISC machine provides the abstraction of a single-level, virtual, global store, part of which is volatile, and the remainder of which is persistent. Heap addresses in AGNA are forty-two bits wide. To facilitate organization of the heap and its mapping to PMEs of the machine, it is divided into 2^{10} *segments* of size 2^{32}. Segments are the unit of distribution across PMEs, or said another way, each segment is contained entirely within one PME. A segment-to-PME mapping table (many-to-one) is replicated on all PMEs. The first stage of heap address translation, therefore, involves consulting this table to determine which PME holds the target heap location.

The number of segments was chosen to be larger than the number of processors in most current MIMD machines (so at least one segment may be mapped to each PME), but yet small enough to allow the segment-to-PME mapping table to be cached entirely within all local memories of the machine for fast address translation. The segment size was chosen to be fairly large to accommodate big databases, and because thirty-two bits is a convenient unit to manipulate on most current machines.

When a database is created, only a single segment is initially mapped to each PME. Segments not assigned to a PME at database creation time are available for dynamic allocation at some later point. In other words, when the volatile or persistent heap in all segments in a PME is exhausted, then a fresh segment is allocated dynamically to the PME.

Within a PME

Each segment is divided into *pages* of size 8 Kbytes each, which are the units transferred between disk and physical memory. Heap addresses therefore consist of a segment number, a page number, and a page offset as shown in Figure 5.3. On each PME, a subset of all the virtual pages in that PME are cached in physical memory in a collection of *page frames*. The mapping is maintained in a hash table:

(segment,page) $\rightarrow$ page-frame

Thus, the second step of heap address translation involves probing this table. If successful, *i.e.*, the page is in memory, then the heap address can be accessed immediately

The page size used in AGNA was chosen as a compromise between efficiency of access to large chunks of the heap, and good overall paging behavior. A large unit of transfer is more efficient

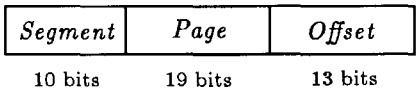


Figure 5.3: Structure of heap address.

than a smaller one in the sense that for the fixed costs such as seek time and rotational latency associated with each read operation, more data is actually transferred.[2] On the other hand, if objects in the working set are clustered poorly in the heap, larger page sizes may actually be less efficient because of the time required to transfer additional page data into the cache when it is unlikely to be needed. Also, this may result in more page-outs from the cache, since a greater portion of it will be consumed by objects not used by the current computation.

AGNA currently imposes the restriction that the total size of an object must be less than the size of a page. Further, an object is placed in contiguous locations in the heap so that it is contained entirely within a page. The rationale for these decisions is that they simplify management and manipulation of objects. A number of proposals in the literature for handling objects larger than a page (*e.g.,* [22]) could be adopted in AGNA; this is an area for future investigation.

Pages within a segment are partitioned into a persistent part and a volatile part—the high-order page bit distinguishes between the two. Thus, no mapping tables or lookups of any kind are required to differentiate between addresses in the persistent and volatile parts of the heap. This is important for efficiency reasons, as the two kinds of addresses have to be identified and handled differently in a number of frequently-performed operations such as installation of database updates at the end of a transaction, field update, field selection, *etc.* While other encodings may be equally suitable, we use the high-order bit because it can be accessed quickly and efficiently. A consequence of this decision is that the persistent and volatile portions of the heap are of equal size.

If the second step of address translation, described above, finds that the target page is not present in the cache, then it is fetched from disk. All volatile pages in all segments in a PME are mapped, via a hash table, to a single paging file. Thus, if the target address is in the volatile heap, then the remainder of address translation involves: (1) probing the hash table to determine the location of the associated page in the paging file; (2) reading the page into a free

[2]For example, if the cost to read a 1 Kbyte block is 27 milliseconds (18 seek, 8 rotate, 1 transfer), then the cost to read an 8 Kbyte block is only 34 milliseconds (18 seek, 8 rotate, 8 transfer). Thus, while the latter operation takes roughly twenty-five percent longer, eight times the data is transferred.

page frame; and (3) accessing the target location.

The mapping of persistent pages to disk files is more complex. To facilitate this mapping, we define a ***physical extent*** to be a contiguous set of 512 persistent pages. Thus, the page number in a persistent heap address consists of a physical extent number and a page offset within the extent (see Figure 5.4). We impose a further restriction that an extent can only contain objects of a single type.

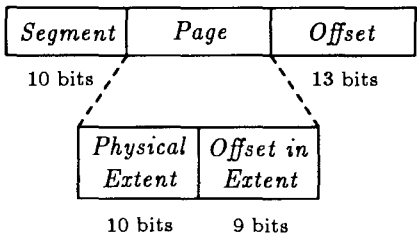


Figure 5.4: Structure of persistent heap address.

All physical extents of a particular type (*e.g.*, student) in all segments in a PME are mapped to the same disk file (`student.dat`). The correspondence is maintained in a hash table:

(segment,extent) → (file,page-offset)

Thus, if the second step of address translation finds that the target page is persistent and not present in the cache, then the following actions are performed (see Figure 5.5): (1) the hash table is probed to determine the file and page offset at which the extent begins; (2) the page offset within the extent, stored in the low-order page bits, is added to the offset within the file and the desired page is read into a free page frame; and (3) the offset within the page is used to compute the target location.

The reason for mapping contiguous page groups (*i.e.*, physical extents) rather than individual pages to files is to keep the mapping tables from becoming too large. For example, the table below shows the number of mapping table entries required for physical extents of various sizes.

Unit of Mapping (Physical Extent Size)	*Entries in Mapping Table (Per Segment)*
1 page	262,144
512 pages	512
262,144 pages	1

With an extent size of only one page, storage utilization in the persistent heap and associated files is high because pages are allocated individually. The size of the corresponding mapping

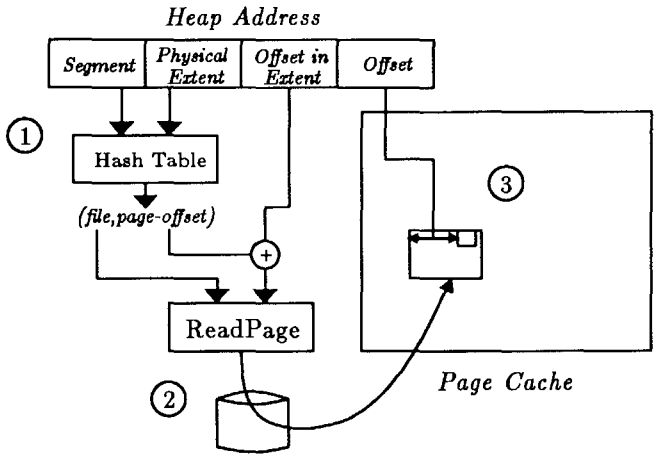


Figure 5.5: Handling of persistent page fault.

table, however, is quite large and thus it may be difficult to keep it entirely memory-resident, in which case costly paging of the mapping table is required. Tradeoffs at the other extreme (the last row of the table) are just the opposite: low storage utilization, but a small mapping table. We chose a physical extent size of 512 pages as a compromise between these two extremes.

When a new type is declared via the transaction language, a file is created and a single physical extent is allocated on each PME on which objects of the type are to be stored (we will see in Section 5.8 how this set of PMEs is determined and how objects are distributed amongst PMEs in the set). For example, when type STUDENT is introduced into a database environment, local files named `student.dat` are created and initial physical extents are allocated on each PME implementing the type (see Figure 5.6). Each PME may implement multiple segments, and each segment consists of multiple physical extents. Extents within a PME are allocated sequentially, so the next available extent is allocated and mapped to the student file.

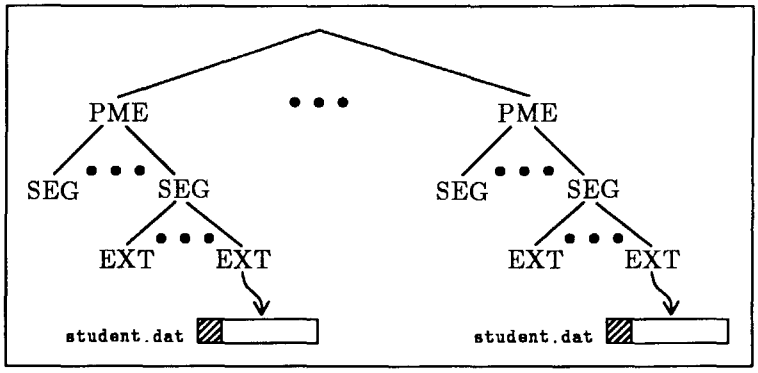


Figure 5.6: Allocation of initial physical extents in student files.

While an entire physical extent is mapped to each student file, only the first page in each file is actually allocated on disk. A free list of storage on allocated pages is maintained in each file. Thus, when a new student object is to be allocated in the persistent heap, the free list in the local student file is searched for a suitable chunk of storage. If one is found, then the object is placed at that location. If one is not found, then a new file page is allocated. When the initial student extent in a PME is full, then the next available one is allocated and mapped to the student file. Again, only the first page of the new extent is actually allocated on disk. When all extents in all segments in a PME are full, then a new segment is allocated to the PME. (A new segment is also allocated when the volatile heap becomes full.)

Objects in a type extent are clustered and stored compactly in the associated physical extents; the general list representation is not stored in the physical extents or anywhere else in the persistent heap. The compact storage of objects allows bulk operations on type extents to be performed efficiently (*e.g.*, finding all students with GPA $\geq$ 3.9), and facilitates building of indexes on files to support the inverse field-mappings specified in the source-language type declarations. While cons-cells of the list representation could be maintained elsewhere in the heap without affecting the compact storage of (say) student objects, this is not done for two reasons. First, this avoids the storage overhead of one cons-cell per student object. Second, as we saw in Section 4.2.6, in many cases the entire extent list is not needed anyway because a (generally) much smaller filtered list of objects can be produced efficiently from a scan over the associated physical extents. When the entire extent list is required, a list is built in the volatile heap.

Heap Reorganization

Apart from the normal requirement of fast translation from virtual heap addresses to physical memory addresses, we also require fast reconfigurability of a database to a machine with a different number of PMEs. For example, it should be easy to reconfigure a database originally constructed on a sixteen PME machine to a fifteen PME machine (perhaps a PME failed) or a thirty-two PME machine (perhaps the machine was upgraded).[3]

In the first case, we need only: (1) map segments implemented by the failed PME to other PMEs of the machine by updating the segment-to-PME mapping table; and (2) append physical extents stored at the failed PME to type files of other PMEs and add new entries to the tables

[3]For now, we are only considering static, off-line reconfiguration.

mapping physical extents to files. In the second case (*i.e.*, the machine grows from sixteen to thirty-two PMEs), either unused or existing segments are allocated to the new PMEs. In any case, reorganization consists only of modifying mapping tables and moving around blocks of persistent data; no exhaustive searches of the database are required to update inter-object references.

5.3 Frame Memory and the Pool of Active Threads

We have chosen to implement frames as objects in the volatile heap, and thus the emulator does *not* contain a separate memory for frames. The motivation for this decision is that both frame and heap memory of the P-RISC machine require storage management routines, including some sort of mechanism for moving objects to and from a backing store. Rather than duplicate the bulk of this functionality for frame and heap memory, we decided simply to store frames in the heap. Note, however, that not all of the generality of heap memory is required for frames (*e.g.*, synchronized reads and writes aren't needed), so while this decision reduces complexity, there may be some performance penalty.

We have taken the position that each frame resides entirely within a PME, and that all threads of a frame on PME i must execute on PME i. The rationale for this is that it enables a procedure or transaction body to access its frame slots efficiently, since all such accesses are entirely local. We shall see in Section 5.8 how the PME on which a frame is placed is selected.

Finally, the pool of active thread descriptors of the P-RISC machine is not represented as a separate collection, as described in the previous chapter, but rather the IP of each active thread of a frame is stored in a stack inside the frame itself, as shown in Figure 5.7. The stack grows "up" from the last slot in the frame, which records the depth of the stack. Since the FP part of a thread descriptor (*i.e.*, (FP,IP)) is available implicitly in this organization, only the IP part is actually stored in the stack. The maximum number of threads that may be active simultaneously in a procedure or transaction body, and hence the maximum depth of the IP stack in the associated frame, is determined statically by the compiler. The required frame size stored in procedure and transaction objects by the compiler is large enough to ensure that the IP stack never overflows into the data area.

This particular representation of the pool of active threads was chosen for three reasons. First, it groups thread descriptors by FP, which is desirable for scheduling purposes. Second,

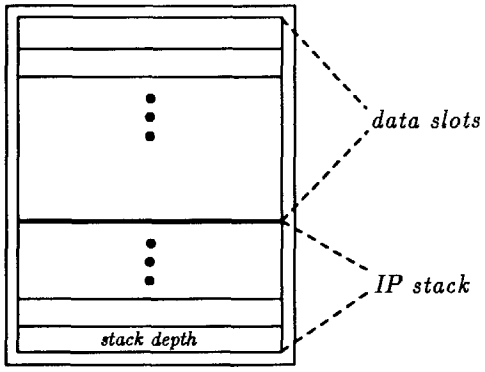


Figure 5.7: Stack of active IPs in frame.

it is storage efficient, since the FP part of descriptors is not stored explicitly. Third, storage management is simplified, since allocation and deallocation of the IP stacks is performed as part of normal frame manipulation.

References to all frames in an emulator with at least one active IP are stored in an "active frame" stack. When an IP is pushed onto the IP stack in an inactive frame (*i.e.*, one in which the stack depth is zero), a reference to the frame itself is also pushed onto the active frame stack. If the stack becomes full, then additional active frames are stored in an overflow chain. A stack was chosen as the primary data structure because it is simple and efficient, yet flexible enough to accommodate the order in which active frames are accessed by the emulator (to be described soon).

5.4 Organization of the Emulator Process

The emulator process is structured into a single *interpreter* thread and one or more *manager* threads, all of which are lightweight, user-level threads running in the same operating system process.[4] The interpreter thread and manager threads are spawned when the emulator is started, and exist for the lifetime of the process. Such threads are never created or destroyed after system startup time. In the next chapter, we address the issue of the number of manager threads used in an emulator process.

The rationale for using multiple lightweight threads is as follows. At any given instant of time, an emulator process may contain many P-RISC micro-threads that are ready to execute.

[4]The interested reader is referred to [16] for an introduction to such threads, which are supported by many operating systems today such as Mach and Sun Unix OS. To avoid confusion here, we will refer to these threads as "lightweight threads" and to P-RISC threads as "micro-threads" when the meaning is not clear from context.

If execution of one of them involves disk I/O—say a LOAD tries to access a local but non-resident heap location—then we do not want the entire process to block awaiting completion of the I/O, because the emulator can execute other micro-threads while the disk transfer is in progress.

Ideal operating system support for this, with respect to our model of computation, is a non-blocking, asynchronous, disk-read routine. With such a routine, LOAD can be executed in a split-phase manner. In the first phase, the disk transfer is initiated via the disk-read routine, after which the emulator continues executing other micro-threads. In the second phase, the emulator is notified (*e.g.*, via an interrupt) that the I/O is complete, and the desired heap location is accessed and the micro-thread to receive the result is activated.

Unfortunately, this kind of non-blocking, asynchronous disk I/O is not supported by the operating systems to which we have ported AGNA (two versions of Unix). We use an interpreter thread and multiple manager threads to achieve the desired overlap of computation and disk I/O. The interpreter thread is responsible for scheduling P-RISC instructions and executing those that do not involve long-latency operations, while those that do involve such operations are executed by manager threads. Communication between the interpreter and manager threads is via message queues, as shown in Figure 5.8.

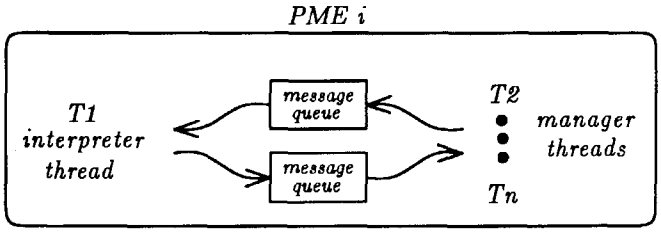


Figure 5.8: Interpreter and manager threads on PME *i*.

The interpreter thread initiates execution of instructions that involve disk I/O, such as the LOAD described above, by enqueueing a manager request that contains the operation to be performed, its arguments, and the micro-thread (*i.e.*, (FP,IP)) and frame slot to receive the result. All manager threads are capable of handling all messages, so in the case of LOAD, the next available manager dequeues the message and issues the I/O request. The request causes only the manager thread to suspend, and the emulator may switch to other lightweight threads to continue execution. When the disk transfer is complete, the manager responds with a message containing the FP, IP, result frame slot, and contents of the desired heap location. At some later point, the interpreter handles the message by storing the value into the result frame slot, and

activating the micro-thread (FP,IP).

The Interpreter Thread

Operation of the interpreter thread consists of repeated selection of an active micro-thread, execution of its current instruction, and generation of successor micro-thread descriptors, if any. Its C code is shown below.

```
interpreter()
{
    HeapAddr *fp=0, *ip=0;

    while ( TRUE ) {                    /* loop infinitely */

        if ( ip==0 ) {                  /* no current IP */
            if ( fp!=0 ) {              /* got current FP, try to get IP */
                ip = getNextActiveIP(fp);
                if ( ip!=0 ) goto L;    /* if active IP found, go execute instruction */
            }
            checkForMsgs();             /* handle any queued messages */

            fp = getNextActiveFP();     /* switch to new frame */
            if ( fp==0 ) waitForMsgs(); /* no active frames; wait for messages */
            continue;
        }

        L: switch ( opcode(ip) ) {
            case ADD:    ...perform addition...
                         ip->address += ADD_LENGTH;
                         break;
            case FORK:   ...perform fork...
                         ip->address += FORK_LENGTH;
                         break;
            case LOAD:   ip=doLoad_1(fp,ip,readFrame(fp,loadAddressSlot(ip)),
                                     resultSlot(ip));
                         break;
            case LOOKUP: ip=doLookup_1(fp,ip,readFrame(fp,lookupNameSlot(ip)),
                                       resultSlot(ip));
                         break;
            ...
        }
    }
}
```

The top-level `if` statement in the first half of the `while` loop assigns to variables `fp` and `ip` the heap addresses of the frame and instruction pointers, respectively, of the thread to be executed. The current instruction is executed in the `switch` statement in the second half of the `while` loop. Instructions that do not involve long-latency operations, such as `ADD`, are executed entirely within the `switch` statement. If the successor instruction is immediately available for execution,

which is the case for ADD and FORK shown above, then ip is updated to reference it.[5] In the next section, we give the code which performs ADD and FORK operations.

For an instruction that may involve long-latency operations, the interpreter invokes a procedure to handle the instruction, passing the current FP and IP, and any instruction arguments. For example, procedure doLoad_1, which handles LOAD instructions, takes as arguments an fp, ip, heap location, and result frame slot. Three outcomes are possible for doLoad_1:

1. The entire LOAD operation is executed immediately, and the IP of the next sequential instruction is returned. This is the case when the target heap address is local and resident.

2. A LOAD message is sent to a local manager, and zero is returned indicating that the current micro-thread is terminated. This is the case when the target location is local but non-resident. A new micro-thread is selected for execution inside procedure interpreter.

3. A LOAD message is sent to another PME, and again, zero is returned. This is the case when the target location is non-local.

All other procedures which handle instructions that may involve long-latency operations have three similar possible outcomes, though the conditions which distinguish one outcome from another may be different.

Manager Threads

Manager threads all execute a copy of the same code, which is given below.

```
managerThread()
{
    char *msg;

    while ( TRUE ) {
        msg = dequeue(mgrMsgQ);
        switch ( msgOpcode(msg) ) {
            case LOAD:   doLoad_2(msg);   break;
            case LOOKUP: doLookup_2(msg); break;
            ...
        }
    }
}
```

[5] HeapAddr, the type of the C structure to which ip points, contains fields segment and address, recording the segment number and address within the segment, respectively. Thus, assigning ip to the next instruction involves only incrementing the address field by the length of the current instruction.

Manager threads repeatedly dequeue messages from `mgrMsgQ` (the queue of messages from the interpreter) and execute the specified operations. Procedure `dequeue`, which implements the dequeue operation, is blocking and atomic. In other words, if no messages are present in `mgrMsgQ`, then the manager thread blocks. When the interpreter thread enqueues a message, exactly one of the managers blocked on a dequeue operation will receive it. After implementing the specified operation, a manager thread sends a reply message either to the local interpreter thread or a remote PME.

All instructions that may contain long-latency operations have two procedures which implement them, one that executes in the interpreter thread, and one that executes in a manager thread. Above we saw examples of the first kind: `doLoad_1`, `doLookup_1`, *etc.*, and also of the second kind: `doLoad_2`, `doLookup_2`, *etc.*. We will give the definitions of these procedures in the next section.

Intra-PME Messages

As we saw earlier, messages sent from the interpreter to local managers contain the operation to be performed, any arguments, and the micro-thread and frame slot to receive the result. The number of message types handled by manager threads is equal to the number of P-RISC instructions (including manager instructions) that may involve long-latency operations.

Only two types of messages are sent to the interpreter thread from local managers:

```
<START0,fp,ip>
<START1,fp,ip,r,v>
```

The interpreter handles these messages via procedure `handleMsg`, defined as follows.

```
handleMsg(msg)
char *msg;
{
    HeapAddr *fp = startMsgFP(msg), *ip = startMsgIP(msg);

    if ( msgOpcode(msg)==START1 ) {
        int   r = start1MsgSlot(msg);
        long *v = start1MsgValue(msg);
        writeFrame(fp,r,v);            /* fp[r] <- v */
    }
    pushIP(fp,ip);                     /* activate (fp,ip) */
}
```

If the message type is START1, then the value contained in the message (v) is stored in the indicated frame slot (r). For both message types, micro-thread (fp,ip) is activated by procedure pushIP, which pushes an IP onto the stack of active IPs in a frame.

All queued messages are handled by the interpreter thread in two situations. First, after all active IPs in the current frame are exhausted, but before switching to a new frame, procedure checkForMsgs is called, as shown earlier in procedure interpreter. Here is the definition of checkForMsgs.

```
void checkForMsgs()
{
    while ( !queueEmpty(interpreterMsgQ) )
        handleMsg(dequeue(interpreterMsgQ));
}
```

The procedure simply extracts and handles all queued messages. The motivation for the interpreter handling messages before switching to a new frame is that this enhances locality in the emulator. We comment further on the issue of locality in Section 5.6.

The second situation arises when the interpreter has no work to do, *i.e.*, the local pool of active micro-threads is empty. In this case, it calls waitForMsgs, again from the body of procedure interpreter. Here is the definition of waitForMsgs.

```
void waitForMsgs()
{
    handleMsg(dequeue(interpreterMsgQ));
    checkForMsgs();
}
```

First, the interpreter waits for a message via dequeue, which blocks until one is available. After the first message is handled via handleMsg, any additional messages in the queue are handled through the call to checkForMsgs.

Inter-PME Messages

Inter-PME messages have the same structure as messages passed between the interpreter and local manager threads. Inter-PME messages are sent via procedure sendMsg, which takes the target PME number and message text. When a message arrives at a PME, the emulator process is interrupted, and the message is enqueued. Messages of type START0 and START1 are added to

queue `interpreterMsgQ`, while all others are added to queue `mgrMsgQ`.

`START0` and `START1`, the only messages that activate micro-threads, are placed in the interpreter's message queue because we have taken the position that all manipulation of the stack of active frames and IP stacks in frames is performed only in the interpreter thread. The motivation for this is as follows. First, this means that no exclusion mechanism, such as semaphores, is needed when manipulating these stacks to ensure consistency. This enhances both the simplicity and efficiency of access, which is important because these stacks are manipulated frequently.

The second reason has to do with what the interpreter thread does when no micro-threads are available for execution. As described previously, the interpreter calls procedure `waitForMsgs`, which blocks until a message is available. If manager threads could activate micro-threads, then some mechanism would be needed to allow a manager to unblock the interpreter from its wait for incoming messages. Perhaps a null "wake up" message could be used. Or perhaps the interpreter thread could be restructured to idle when no work is available by repeatedly checking for both incoming messages and active threads.[6]

In any case, we believe that these schemes are more complicated and less efficient than the one we have chosen. A possible drawback of our approach, however, is that if the frame in which an IP is to be stored is not resident, then the interpreter thread will suspend until the desired heap page is transferred to physical memory. While manager threads may execute during the transfer, active micro-threads may not.

5.5 Execution of P-RISC Instructions

Arithmetic, Logical, and Relational Instructions

In the previous section we saw that instructions that do not involve long-latency operations, such as `ADD`, are executed entirely in the `switch` statement in procedure `interpreter`. Here is the implementation of `ADD`:

```
case ADD: {
    long *v1 = readFrame(fp,source1(ip)),
         *v2 = readFrame(fp,source2(ip)),
         r[2];

    r[0] = v2[0];
```

[6]For this to be efficient, the interpreter thread would have to lower its priority, relative to manager threads, before idling.

```
        r[1] = v1[1]+v2[1];
        writeFrame(fp,sink(ip),r);
        ip->address += ADD_LENGTH;
        break;
    }
```

Procedure `readFrame` is used to assign to `v1` and `v2` the (C) memory addresses of the operands. The result value is constructed in two-word array `r` and written via `writeFrame`. Integers and other scalars in AGNA are represented as two-word quantities: the first word contains the standard object header (type tag, data/pointer flag, *etc.*), while the second word contains the value. Thus, `r[0]` contains the new header, taken from `v2`, while `r[1]` contains the sum. Finally, the new IP is set to the next sequential instruction.

All other arithmetic, logical, and relational instructions are executed in a similar manner.

Control Flow

Control flow instructions, such as `FORK`, also execute entirely within procedure `interpreter`:

```
case FORK: {
    HeapAddr tgtIP;

    tgtIP.segment = ip->segment;
    tgtIP.address = ip->address+forkOffset(ip);
    pushIP(fp,&tgtIP);
    ip->address += FORK_LENGTH;
    break;
}
```

A new active IP `tgtIP` (*i.e.*, the `FORK` target) is constructed and pushed into the current frame via `pushIP`. As before, `ip` is set to the next instruction. The `FORK` target is computed by adding the offset contained in the `FORK` instruction to the segment address of the current IP.

Other control flow instructions, such as conditional and unconditional jumps, are executed in a similar manner. `JOIN` is executed as follows:

```
case JOIN: {
    long *v = readFrame(fp,joinSlot(ip));
    int bit = joinBit(ip);

    if ( getBitValue(v,bit)==0 ) {
        ip = 0;
        setBitValue(v,bit,1);
    } else {
        ip->address += JOIN_LENGTH;
```

```
            setBitValue(v,bit,0);
        }
        break;
    }
```

If the join bit is zero, then `ip` is set to zero (*i.e.*, the current micro-thread is terminated) and the bit is toggled. If the bit is one, then `ip` is set to the next instruction, and again, the bit is toggled.

Heap Access

Heap access instructions include `LOAD`, `ILOAD`, `STORE`, and `ISTORE`. Here is the definition of procedure `doLoad_1`, called by the interpreter to handle `LOAD` instructions.

```
HeapAddr *doLoad_1(fp,ip,address,resultSlot)
HeapAddr *fp, *ip, *address;
int resultSlot;
{
    int pme=pmeOfAddress(address);
    char *msg;

    ip->address += LOAD_LENGTH;
    if ( pme==localPME )
        if ( addressResident(address) ) {
            long *v = translate(address);
            writeFrame(fp,resultSlot,v);
            return ip;
        }

    msg = buildMsg(LOAD,fp,ip,2,address,resultSlot);
    if ( pme==localPME )
        enqueue(mgrMsgQ,msg);
    else
        sendMsg(pme,msg);
    return 0;
}
```

First the PME on which the target location resides is determined via procedure `pmeOfAddress` and stored in `pme`, and `ip` is set to the next instruction. If the location is implemented by the current PME (whose number is stored in global variable `localPME`) and the address is resident in the local cache of page frames, then the value stored in the desired location is written into the result frame slot, and the updated `ip` is returned. If the target location is non-local or non-resident, then a `LOAD` message is constructed and sent either to a local manager thread or to the PME in which the location resides, after which zero is returned.

The procedure which handles the LOAD message, either in a local or remote manager thread, is defined as follows:

```
void doLoad_2(msg)
char *msg;
{
    long *value;

    ...Destructure message into fp, ip, resultSlot, and address.

    value = mapAndTranslate(address);
    msg = buildMsg(START1,fp,ip,2,resultSlot,value);
    pme = pmeOfAddress(fp);
    if ( pme==localPME )
        enqueue(interpreterMsgQ,msg);
    else
        sendMsg(pme,msg);
}
```

After the message is destructured into its component fields, the target heap location is mapped and translated by procedure `mapAndTranslate`. The procedure first checks to see if the desired page is already resident in the cache, and if it is not present, then it is read into a free page frame. A pointer to the value stored at the heap address is returned. Semaphores are used within `mapAndTranslate` to ensure that the calling thread sees a consistent view of shared data structures such as the cache of page frames and its hash table index.

Other heap access instructions are handled in a similar manner. For example, `doStore_1` first checks to see if the target address is local and resident. If so, then it performs the operation and returns a pointer to the next instruction. If not, a STORE message is sent either to a local manager or a remote PME, and zero is returned. ILOAD and ISTORE are identical to LOAD and STORE, except that they also perform implicit synchronization. If the execution of ILOAD finds, by examining the presence bits, that the target location is empty, then it adds a triple consisting of the FP, IP, and frame slot to the list of deferred readers rooted at the desired location. When ISTORE writes a value into an empty location (it is an error if full), all waiters are enabled via START1 messages.

Inter-Frame Transfers

Instructions for transferring control and data from one frame to another are START0 and START1. In the previous section we saw how the interpreter handles incoming START0 and START1 messages. Procedure `interpreter` calls `doStart1_1`, defined below, to implement START1 instructions.

```
HeapAddr *doStart1_1(fp,ip,tgtFP,tgtIP,value,resultSlot)
HeapAddr *fp, *ip, *tgtFP, *tgtIP;
long *value;
int resultSlot;
{
    int pme=pmeOfAddress(tgtFP);

    if ( pme==localPME ) {
        writeFrame(tgtFP,resultSlot,value);
        pushIP(tgtFP,tgtIP);
    } else
        sendMsg(pme,buildMsg(START1,fp,ip,4,tgtFP,tgtIP,value,resultSlot));
    ip->address += START1_LENGTH;
    return ip;
}
```

If the target frame is local, then the value is stored in the desired slot and the micro-thread (`tgtFP,tgtIP`) is activated. Otherwise, a `START1` message is sent to the PME on which the frame resides. In either case, the next instruction is scheduled for execution.

`START0` instructions are handled via procedure `doStart0_1` in a similar manner.

Manager Instructions

Execution of many manager instructions is similar to that of heap access instructions. For example, `ALLOCOBJ`, which allocates a new object in the volatile heap, decides whether the allocation is to be performed locally or on a remote PME[7], and if locally, whether in the interpreter or a manager thread. Other instructions which are executed in this manner include `ALLOCFRAME`, `SELECTF`, and `UPDATEF`.

Other manager instructions are used solely to implement local operations. Examples of this sort include: `SVINVERT` and `MVINVERT`, which perform local single- and multiple-valued field inversions, respectively; and `FILTEREXTENT`, which builds a filtered list of all local objects of a specific type. `SVINVERT`, which searches locally for the object with a particular field value (*e.g.*, the student object with a certain name), is handled in the interpreter thread via procedure `doSvinvert_1`:

```
HeapAddr *doSvinvert_1(fp,ip,type,field,value,resultSlot)
HeapAddr *fp, *ip, *value;
int type, field, resultSlot;
{
    ip->address += MGR_LENGTH;
    enqueue(mgrMsgQ,buildMsg(SVINVERT,fp,ip,4,type,field,value,resultSlot));
```

[7] We will see in Section 5.8 how it makes this decision.

```
    return 0;
}
```

Since it is difficult to predict whether the associative search required to implement SVINVERT will involve long-latency operations (*i.e.*, accesses to non-resident pages), doSvinvert_1 always builds a SVINVERT message and passes it to a local manager.

Inside a manager thread, SVINVERT is handled by doSvinvert_2, defined below.

```
void doSvinvert_2(msg)
char *msg;
{
   long *v;

    ...Destructure message into component fields...
    ...Use field index to search for object with desired field value...

    if ( object found )
        v = object;
    else
        v = buildNullObject(type);
    enqueue(interpreterMsgQ,buildMsg(START1,fp,ip,2,resultSlot,v));
}
```

After the message is destructured, a search for the desired object is performed using the field index. The return value, which is either the desired object or a null object, is placed in a START1 message that is passed to the local interpreter thread.

The reader may wonder at this point why we include such a high-level manager instruction as a primitive in the emulator. For example, why isn't it expanded to more basic P-RISC instructions? Or why isn't the operation coded as a procedure in the transaction language, or some extension thereof? We currently implement high-level instructions as primitives for two reasons: efficiency of execution and ease of development. First, by implementing such operations in C instead of multiple P-RISC instructions, we avoid the overhead of interpretation in our current implementation. Second, compilers for C are more robust and efficient than the current AGNA compiler. Note, however, that regardless of the implementation strategy (*e.g.*, primitive, macro, or procedure call), the interface and split-phase execution remain the same.

5.6 Ordering of Instructions

In Chapter 4, we saw that the P-RISC machine does not specify the order in which micro-threads are to be executed. In the interpreter, however, we made a (static) ordering decision in which (1) all threads of the current frame are executed to completion before switching to a new frame, and (2) all instructions in a micro-thread are executed sequentially until the thread is terminated (*e.g.*, via `JOIN`).

The motivation for this is that it enhances *locality* of execution. First, this may increase performance by increasing the effectiveness of caches in the underlying hardware (*e.g.*, processor caches). Second, this locality allows the emulator to avoid many translations of the heap addresses of the current FP and IP. While not all of the details were shown in procedure `interpreter` earlier, switching to a new frame actually entails:

1. "Unpinning" the old code and frame objects. In other words, the heap pages on which the objects reside are unlocked from their physical memory addresses in the cache, and thus are available for replacement.

2. The new FP and IP are selected and stored in variables `fp` and `ip`.

3. The new code and frame objects are pinned and their physical memory addresses are determined and stored in variables `fpPA` and `ipPA`. These physical addresses are used inside the interpreter whenever possible, thus avoiding unnecessary translations of frame and instruction pointers.

As we shall see in the next chapter, address translation, which is currently performed in AGNA in software, is relatively expensive, so this improvement has a significant impact on performance. Of course, the virtual addresses, stored in `fp` and `ip`, are used in messages sent to local manager threads and remote PMEs.

5.7 Representation of Indexes and Multi-Valued Fields

In this section, we describe the representation of indexes and multi-valued fields.

Indexes

Indexes in AGNA are currently not represented as objects in the global heap, but rather are separate file-based structures that may be manipulated only through calls to the object storage system. For example, addition of a new element to a hash index is not performed via `allocate`, `update`, and so on, but through a call to the appropriate routine in the object storage system (made from within a P-RISC manager), which is written in C. While our long-term goal is to represent indexes uniformly as persistent objects, we currently do not for the same two reasons that we treat high-level manager instructions as primitives: efficiency and ease of development.

An index on a field `f` of objects in an extent of type `T` is mapped to a collection of files, one per PME on which objects of type `T` are stored. For example, the hash index on student `name` described in Section 4.2.3 is stored in local files named `student-name.idx`. Each index file in a PME maps names to local student objects as shown in Figure 5.9.

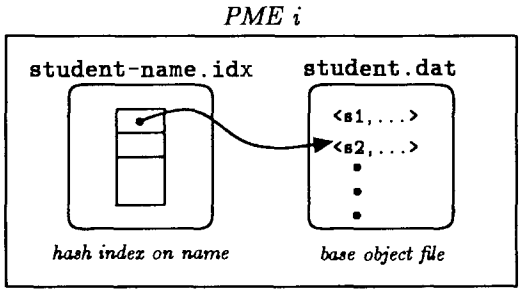


Figure 5.9: Index on name field of local student objects.

Indexes could be mapped to files differently, *e.g.,* the entire index on student `name` could be stored in a single file, rather than in a collection of files. The mapping used in AGNA was chosen because it supports a high degree of parallelism, as local searches that utilize the index may execute concurrently on all PMEs of a type. Also, locality is enhanced because an index refers only to local objects.

Multi-Valued Fields

As discussed in Chapter 3, multi-valued fields in objects in the persistent heap do not use the general list representation, but rather a compact internal form. Consider the definition of type `COURSE` introduced in Chapter 2 that includes multi-valued field `prereqs`:

```
(type COURSE (extent)
  ((name      <=>  STRING)
```

```
(prereqs *<=>* COURSE)
(units     =>  INTEGER)))
```

Like all new objects, `allocate-object` allocates new course objects in the volatile heap, with all fields undefined. When the `prereqs` field of a new course object (call it *o*) is defined via `update`, a reference to the field list is stored in the appropriate field "slot" of *o*.

When course *o* is added to the persistent heap at transaction commit time, the representation of the `prereqs` field is changed from a list of courses to a collection of records of the form:

```
<o,prereq>
```

In other words, installation of *o* in the persistent heap involves construction of a *new* representation of its `prereqs` field collection. Each record in the new representation stores a reference to *o* and a reference to one of its prerequisites; there is one such record for each element of the field collection. These records are stored in file `course-prereqs.dat` on the PME on which *o* itself resides. A similar `course-prereqs.dat` file exists on all PMEs on which course objects are stored, with the file on PME *i* holding records that describe the `prereqs` field collections of all course objects residing on PME *i*.

When the `prereqs` field of a persistent course object is read, records in the `course-prereqs.dat` file are located, and a list of prerequisites is constructed in the volatile heap. Two indexes are maintained on the records that describe prerequisites, one on the course field, and the other on the prerequisite field. The first index allows all prerequisites for a given course to be located quickly, while the second index allows the inverse field-mapping on `prereqs` (that maps a course to all the courses for which it is a prerequisite) to be implemented efficiently.

All of the files that relate to type `COURSE` in a PME are shown in Figure 5.10. At the top of the figure are the base object file (`course.dat`), which contains all course objects that reside in segments stored in the PME, and the file holding the hash index on the `name` field (`course-name.idx`). At the bottom of the figure is the file holding the `prereqs` field collections of all course objects in the PME (`course-prereqs.dat`), along with its two index files.

Field collections of persistent objects are stored in this internal form, and do not use the general list representation, for the following reasons. First, by not storing the cons-cells of the list representation, less storage is required. Second, the clustering of field collections in the internal form facilitates indexing to support the inverse field-mappings specified in type

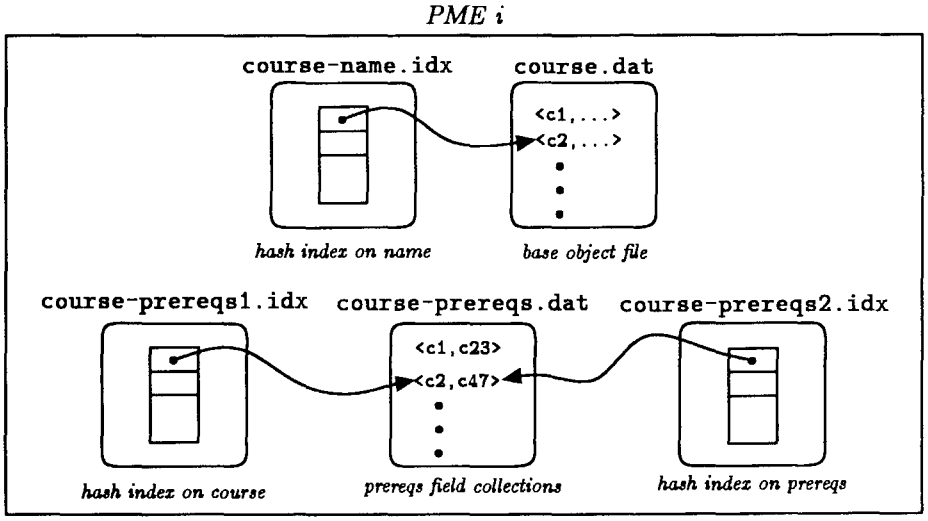


Figure 5.10: All files related to courses on PME i.

declarations. Finally, this clustering also facilitates efficient scanning of all field collections of a type. A possible disadvantage in not storing the general list representation explicitly is that it must be constructed in the volatile heap when needed.

As with indexes, records in multi-valued field files are not represented as persistent objects in the heap, but rather they use a representation internal to the object storage system. These records may be manipulated only via calls to the object storage system. Again, the motivation for this decision is efficiency and ease of programming.

5.8 Distribution of Data and Computation

In this section, we describe how data and computation are distributed to PMEs in a parallel machine.

5.8.1 Data

We saw in Section 5.2 how segments of the heap are mapped to PMEs. The issue addressed here is: when a new object is allocated, in what segment, and therefore on what PME, should it be placed? This issue can be broken down into two separate questions. First, for a given type (*e.g.*, student), on what PMEs should instances of the type be stored? For example, objects may be clustered on some subset of the PMEs, or distributed over all nodes of the machine. Second, after this "home" set of PMEs is determined for a type, how does the object allocation

manager choose (at run-time) the PME on which to place a new object?

In the current version of AGNA, the answer to the first question is: objects of all types are distributed to all PMEs of the machine. For a type T, this placement information is stored as a collection of PME numbers in the associated type object (*i.e.*, the object describing the type, and part of the database's meta-information) in multi-valued field `pmes`. In some cases, it may be desirable to restrict distribution of objects of some type to a subset of all PMEs, based on expected extent size, frequency of access, or other factors. While not currently implemented, AGNA could accommodate such cases by allowing hints or annotations in type declarations describing the PMEs over which objects are to be distributed, or a tool by which the database administrator could modify the home set of PMEs of a type.

The answer to the second question is: a new object is placed on a PME selected randomly from the type's home set. This placement strategy is carried out, in part, by the compiler in its translation of `allocate` expressions. For example, an expression such as:

```
(allocate STUDENT)
```

is translated to:

```
(allocate-object RANDOM student size)
```

This is identical to the translation described previously for `allocate`, except that here we have added argument `RANDOM`, a hint describing where the new object is to be placed.[8] This hint is ultimately passed to P-RISC manager `ALLOCOBJ`, which allocates the new student object in the volatile heap of a PME selected randomly from the set of PMEs assigned to the student type. When the object is moved to the persistent heap at transaction commit time, it is moved to a location in the same PME, thus becoming part of the database stored in the selected PME.

This random placement of objects over all PMEs can be expected to provide a fairly uniform distribution of objects over the machine. Depending on the nature and frequency of access to objects of a particular type, alternative placement strategies may be more effective. For example, consider objects of type `BINDING` that record identifier bindings in the top-level database environment. Recall from Chapter 4 that the `BINDING` type has fields `name` and `value`:

[8] We did not describe this placement hint in the compilation chapter because the compiler has no special understanding of it, and also because it is primarily an implementation-level issue.

```
(type BINDING (extent)
  ((name <=> STRING)
   (value => ANY)))
```

The most frequent operation performed on binding objects involves locating the object with a particular name, and then extracting its value (*i.e.*, an identifier lookup). If binding objects are distributed randomly to PMEs of the machine, then lookup on the `name` field requires local searches on all PMEs. If, however, binding objects were mapped to PMEs via a hash value computed from the name, then the lookup operation need search only at one node (*i.e.*, the PME to which the hash value of a name maps). Thus, the lookup operation is better supported by hash distribution than by random placement.

While not accessible to objects of user-defined types, AGNA uses this hash-based placement strategy internally for binding objects. Binding constructor `make-binding` is defined as follows:

```
(define make-binding
  (lambda (name value)
    (letrec ((pmes (pmes-of-type "BINDING"))
             (j    (hash-to-pme pmes (string-hash name)))
             (b    (allocate PME j BINDING)))
      (update b BINDING NAME name)
      (update b BINDING VALUE value))))
```

This definition is identical to the one given in Chapter 4, except for the placement hint `PME j` included in `allocate`. The computed PME number, `j`, is generated by procedure `hash-to-pme` from `pmes`, the set of PMEs on which binding objects reside, and a hash value computed from the `name`. This expanded use of `allocate` that includes a placement hint is not available to the transaction programmer, but only internally for the coding of system functions. The placement hint is passed on to ALLOCOBJ, which allocates the new binding object on the desired PME.

With binding objects distributed in this manner, LOOKUP, the identifier lookup manager, hashes an identifier to a PME, locates the appropriate binding object on that PME, and selects and returns the `value` field. To make identifier lookups even more efficient, a special cache of top-level identifiers and their values is maintained in physical memory on each PME. Thus, repeat lookups of an identifier on a PME are handled very efficiently, involving no non-local searching or disk I/O.

Note that while this distribution strategy does increase the efficiency of lookups, it requires redistribution of all binding objects whenever the set of home PMEs for type BINDING changes.

This may happen, for example, if a PME fails and the database is reconfigured to run on a smaller machine. In the case of binding objects, though, they are easily moved from one heap location to another because they are used only internally and can have no references to them from other objects. Thus, we do not have to worry about creating dangling pointers when moving such objects.

Two additional placement hints may be specified in `allocate`: `LOCAL` and `REMOTE`. The former places the new object on the PME on which the object allocation manager is invoked, while the latter places the new object on a remote PME. These hints, along with `RANDOM` and `PME j`, provide sufficient low-level mechanisms by which to implement a variety of high-level distribution strategies. In addition to the hashing and random placement strategies described above, a number of other methods for distributing objects are possible: round-robin, "minimum object load", associating a range of field values with each PME, *etc.* Parallel relational systems, such as Gamma [33], have utilized many of these data distribution strategies. Evaluation of such strategies and a full assessment of their impact on performance is a topic for future investigation in AGNA.

One issue that must be examined closely in AGNA in strategies such as hashing is the relocation of objects. For example, if objects of type `T` are mapped to PMEs via hashing on field `f`, then when the value of `f` in an object o is updated, o may need to be relocated if the new value of `f` hashes to a PME different from the old one. But if o is moved from one heap location to another, then all references to it must also be updated. A scan of the entire database to locate such references is not an acceptable solution, since it would be prohibitively expensive, even in medium-sized databases.

One possible approach is to move o, but leave in its place an object containing the new address. Pointers to the old location of o are automatically "forwarded" to and replaced by the new address whenever they are dereferenced. The issue of moving objects while maintaining referential integrity is intimately tied up with garbage-collection, which we have not addressed in this work. Garbage-collection of both the persistent and volatile heaps is an important topic of future investigation in AGNA.

5.8.2 Computation

Computation spreads from a PME to other nodes of the machine via two basic mechanisms: object manipulation and procedure calls. Examples of the first mechanism include allocation

of a new object on another PME, and selection of a field value in a remote object.

In the second mechanism (procedure calls), the PME on which the procedure body executes is determined via the placement of its associated frame. As described in Section 5.3, the code of a procedure or transaction and its associated frame must reside on the same PME. The compiled transaction and initial transaction frame are always placed on a PME designated at system startup time. Procedure frames are placed on PMEs in one of two ways. The first method, which is the default strategy, is to select a PME randomly at run-time. The intent of this simple approach is to balance the computational load evenly across the machine.

Like the placement of objects described earlier, placement of procedure frames and code is controlled by a hint that is either introduced by the compiler, or used explicitly in internal system functions. All procedure calls are translated by the compiler to APPLY, a special form for procedure application that includes a hint on where the procedure body is to be executed. It is in this translation that the default hint is generated. For example, expression:

```
(f x y)
```

is translated to:

```
(APPLY RANDOM f x y)
```

The hint (*i.e.*, RANDOM) is ultimately passed to the P-RISC frame allocation manager, which places the new frame on a PME selected randomly. This, in turn, determines the PME on which the procedure executes.

After the frame for procedure f is allocated, but before threads are started in the body, a copy of the procedure object is placed on the same PME as the frame. This can be seen in the P-RISC code for the application of f, sketched below:

```
(1) f: JOIN       r9  b3          % wait for f
       .                          % load into r14-r18: caller FP, result IP,
       .                          % signal IP, result slot, and frame size
(2)    LOADC      r19 1           % r19←encoding of hint RANDOM
(3)    ALLOCFRAME r14 r14         % r14←new FP
(4)    OBJECTPME  r14 r15         % r15←PME of new object
(5)    ADD        r16 r0  r22     % r16←copy of proc. ptr. in r22
(6)    LOCALIZE   r15 r15         % r15←copy of proc. on same PME as frame
       .                          % start threads in body of proc. r15
       .
```

After synchronization in line 1, arguments to the frame allocator are loaded into slots r14 to r18, as described in Chapter 4. Here an additional argument, an encoding of the placement hint, is loaded into slot r19. In line 3, the new frame is allocated and a pointer to it is placed in slot r14. Then in line 4, manager OBJECTPME is used to determine the number of the PME on which the frame resides. Line 5 copies the procedure pointer to r16, and then in line 6 manager LOCALIZE is used to place a copy of the procedure on the selected PME (stored in r15) and return a pointer to it. LOCALIZE utilizes a special table on each PME describing objects that have been stored locally, so if a copy of the procedure already exists on the PME, then a pointer to it is returned and a new copy is not made. After this, threads in the procedure body are started as described in the previous chapter.

Other hints allowed in the APPLY form are: LOCAL, REMOTE, and PME n. In the remainder of this section we describe how these hints are used to exploit the locality of data in the building and filtering of extent lists. Before we do that, however, we describe a special representation of lists used by the system functions that build and filter extent lists.

Open Lists

The non-strictness of lists in AGNA can be used to append them together efficiently. Say, for example, that we want to append lists L1 and L2. By choosing representations for L1 and L2 that embody two key ideas, we can append the lists in $O(1)$ time while using no additional storage. The first idea is to leave the tail of the last cons-cell in each list empty. This will allow us to link the end of L1 directly to the head of L2 using no additional storage *i.e.,* we don't have to build any intermediate lists. The second key idea is to maintain a direct reference not only to the first cons-cell in each list, but also to the last one. This will allow us to locate the end of each list in constant time, so that we can link L1 to L2, and terminate L2 by storing nil in the tail of its last cons-cell.

A representation for lists which includes these two ideas is called an *open list*. Open lists, which are closely related to difference lists in logic programming, can be represented as a cons-cell containing two references A1 and A2, as shown in Figure 5.11. Internally, the structure consists of a list in which the tail of the last cons-cell is empty. A1 points at the first cell and A2 points at the last cell (these may in fact be the same cell, if there is only one element in the list). With L1 and L2 represented as open lists, we can append them and return the result list as follows.

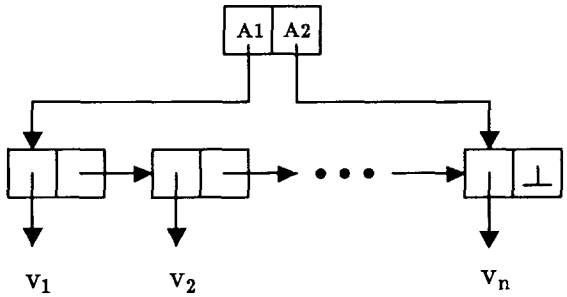


Figure 5.11: The structure of an open list.

```
(letrec ((x1 (update (tl L1) LIST TL (hd L2)))
         (x2 (update (tl L2) LIST TL nil)))
  (hd L1))
```

As shown in Figure 5.12, the first update stores a reference to the head of `L2` in the last cons-cell of `L1`, while the second one terminates the list by storing `nil` in the tail of the last cons-cell in `L2`. `X1` and `x2` are dummy identifiers. The body of `letrec` returns the result list, which is not represented as an open list.

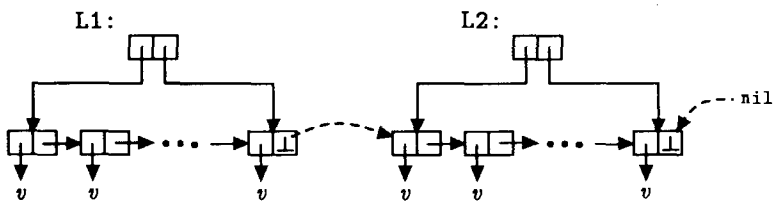


Figure 5.12: Appending of open lists L1 and L2.

Building and Filtering Extent Lists

Consider the following example:

```
(all (select s STUDENT NAME)
  (s (all STUDENT))
  (where (>= (select s STUDENT GPA) 3.9)))
```

i.e., an expression that evaluates to the list of names of students with a GPA of at least 3.9. As described earlier, student objects are distributed over the various PMEs of the parallel machine. The strategy for filtering and building the result list is for each PME to construct a local list of all the names of its students that satisfy the filter, and then to append these lists together. However, we would like the PMEs to work in parallel, and for the list-appending to be very

efficient, *i.e.*, we wish to avoid constructing any intermediate lists.

Each PME executes the following expression (we will see in a moment how it is persuaded to do this):

```
(letrec ((pred (cons (make-condition gpa-id ">=" 3.9) nil)))
  (local-filter-extent student-id name-id BASEFILE NOINDEX pred rest))
```

producing the result shown in Figure 5.13. The arguments to `local-filter-extent` are: the type id of the extent to filter; the object field to be included in the result; the access path and index—constants `BASEFILE` and `NOINDEX` indicate that use of an index in this query is not possible, so the filter must be performed by scanning over all student objects in the base file; the filter predicate, which consists of a list of conditions; and `rest`, which is used to link the local lists together. Because of the non-strict semantics in AGNA, each PME can construct its local list of student objects before it knows the value of the `rest` argument. The last cons-cell simply remains empty. As soon as the first cons-cell is allocated, a reference to it can be returned as the result of `local-filter-extent`. When the value of `rest` arrives, which may be much later, it gets stored into the last cons-cell.

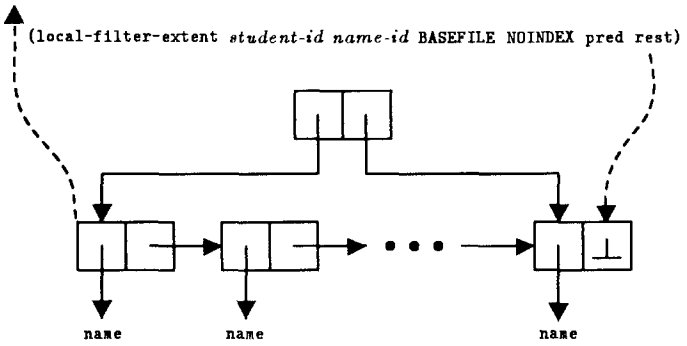


Figure 5.13: Local list of student names constructed on each PME.

Here is the definition of `local-filter-extent`:

```
(define local-filter-extent
  (lambda (type field access-path index predicate rest)
    (letrec ((res (FILTEREXTENT type field access-path index predicate)))
      (if (nil? res)
          rest
          (letrec ((x1 (update (tl res) LIST TL rest)))
            (hd res))))))
```

Local result lists are produced by FILTEREXTENT, which is passed the type and field ids, the access path, the index to use, and the predicate. FILTEREXTENT returns either nil, if no local objects are found which satisfy the predicate, or an open list containing the student names. In the first case, the rest argument is returned as the result. In the second case, the local result list and rest are appended together, and a reference to the local list is returned. The use of an open list by FILTEREXTENT allows this list appending to be performed efficiently, as described earlier.

Now, we step up one level to see the implementation of the entire list comprehension. Conceptually, the extent filtering and appending of lists is accomplished by structuring the computation thus:

```
(letrec ((pred (cons (make-condition gpa-id ">=" 3.9) nil)))
  (foldr
    (lambda (j rest)
      (APPLY PME j local-filter-extent student-id name-id BASEFILE NOINDEX pred rest))
    nil
    pmes))
```

Recall from Chapter 2 that foldr takes a binary combining function, an initial value, and a list of values as arguments, and returns an accumulated value as its result. Pmes is the list of PME numbers on which student objects reside. The overall structure of the computation may be seen in Figure 5.14. The foldr computation executes on various PMEs, and initiates the local-filter-extent computations on each PME j in list pmes. Because of non-strictness, all these computations can proceed in parallel, even though each one does not yet have the value of its rest parameter. Further, each PME j can return the reference to the head of its sublist as soon as it is allocated; this reference is passed on by foldr as the rest parameter for the computation in PME $j-1$, where it is stored in the tail of the last cons-cell.

5.9 Transaction Execution

As we saw at the beginning of this chapter, the compiler runs on the front-end machine and writes its output to a file, also on the front-end machine. The compiler obtains type and field information from a local shadow copy of the meta-data of a database. This copy is kept up-to-date by the command interpreter, which queries the back-end and installs any updates after each transaction execution.

After a transaction is compiled, it is down-loaded into the back-end machine. Before execu-

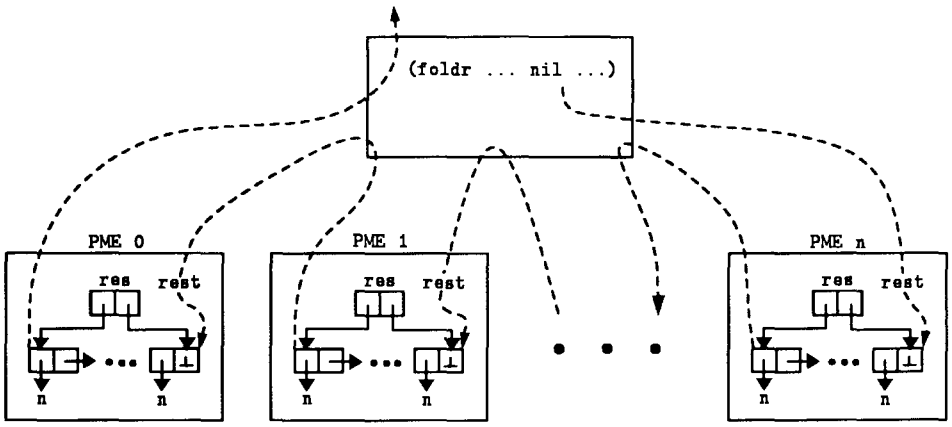


Figure 5.14: Local filtered and transformed extent constructed on each PME.

tion begins, each PME performs a number of local initializations, such as reclaiming all storage in the volatile heap, clearing the paging file, and initializing deferred update collections and various caches. While a number of read-only transactions may be loaded together and executed concurrently, update transactions must be executed serially. This is because AGNA does not currently include a concurrency control mechanism, such as two-phase locking [30], to prevent harmful interactions between queries and update transactions. Concurrency control is a major topic for future investigation in AGNA.

As we saw in the compilation chapter, execution of an AGNA transaction occurs in three phases: prologue, body, and epilogue. For a transaction `(xact e)`, the three phases correspond to the top-level expressions in the `seq` form introduced by the compiler to sequence execution properly:

```
(xact
  (seq
    (begin-transaction)
    (print e')
    (end-transaction)))
```

Expression e' is a translated version of e.

Prologue

In the prologue, an identifier is assigned to the transaction and propagated to all PMEs of the machine. Here is the definition of `begin-transaction`:

```
(define begin-transaction
  (lambda ()
    (letrec ((id (next-transaction-id)))
      (foreach (lambda (j) (APPLY PME j local-begin-transaction id))
               all-pmes))))
```

The new id is assigned via `next-transaction-id`, from a seed stored in the database. The id is passed to procedure `local-begin-transaction`, which is executed on each PME. Identifier `all-pmes` is bound to a list of all PME numbers.

Body

In the next phase of execution, the user-supplied portion of the transaction is executed and the answer is printed. All updates to the database are stored in collections `adds`, `drops`, `updates`, `inserts`, and `deletes`, while possible violations of field uniqueness constraints are accumulated in `constraints`. Each collection is represented as multiple local sublists, one on each PME. For example, when the object allocation manager creates a new student object on PME i, it also adds a reference to the student to the `adds` sublist on PME i.

If during this phase a run-time error occurs or the transaction is explicitly aborted via `abort-transaction`, then an appropriate message is generated and printed on the front-end machine and execution is terminated. This is currently accomplished in a heavy-handed manner by an automatic aborting and restarting of all emulator processes. Note that because of non-strictness, the result returned by a transaction that ultimately aborts may be printed as the answer before execution is terminated. In such cases, the message describing the abort condition printed on the front-end machine informs the user that the transaction was aborted, and therefore the answer is to be discarded.

Epilogue

The final phase of execution, implemented via `end-transaction`, installs deferred updates in the database and brings disk files up-to-date. In the first phase, procedure `precommit` is invoked on each PME to install in the database the local collections of deferred updates. The collections are processed on each PME as follows:

- **adds**: Each object in the list is copied to the persistent heap in the local PME. All objects in the volatile heap reachable from these new persistent objects are also copied

to the persistent heap.

- **drops**: In each object in the list, a flag is set to "false" in the object header indicating that the object is not part of the type's extent. The routines which manipulate type extents implicitly include a predicate that filters out objects not part of the extent. Note that dropping an object does not delete it from the database, but only removes it from the type extent. Though not implemented in the current prototype, the object may be garbage-collected separately when no references to it exist.

- **updates**: New field values in the list are installed in persistent objects. All objects in the volatile heap reachable from these new persistent objects are also copied to the persistent heap.

- **deletes**: Elements in the list are deleted from multi-valued field collections. All `deletes` are processed completely before processing of `inserts` begins.

- **inserts**: Elements in the list are added to multi-valued field collections. Again, all objects in the volatile heap reachable from the new persistent objects are copied to the persistent heap.

Two errors that may arise during the processing of updates are (1) a field of a persistent object is updated multiple times, and (2) a field is updated via both `update` and `insert/delete`. Such errors are detected as follows. Maintained in the header of each object is the id of the last transaction to update the object, and a field bit mask. When a persistent object is first updated in a transaction, this update-id is set to the id of the current transaction and the bit mask is cleared. Whenever a field is updated, the corresponding bit in the mask is set. Multiple update errors are identified when an update is performed on a field for which the bit is already set.

For multi-valued fields, additional update bits are maintained in each field slot indicating whether the field has been updated via `update` or `insert/delete`. Such bits are cleared along with the object header bit mask. Updates to fields via both `update` and `insert/delete` are detected by examining these field update bits.

The search for reachable volatile objects in the processing of `adds`, `updates`, and `inserts` is performed via procedure `make-object-persistent`, which is defined as follows:

```
(define make-object-persistent
  (lambda (object)
    (if (and (volatile? object) (object-reference? object))
        (letrec ((x (make-persistent-copy object)))
          (if (hd x)
              (tl x)
              ...Call make-object-persistent on each field value...
              ...Install updated values...
              ...Return (tl x)...))
        object)))
```

First, if the object is already persistent or a scalar such as an integer, then there is nothing to do and the object is returned. Otherwise, `make-persistent-copy` is called to move the object to the persistent heap. The return value is a cons-cell in which the head is a boolean indicating whether the object has already been moved to the persistent heap via a previous call, and the tail is the persistent address. As described in the previous chapter in the discussion of P-RISC manager instruction `MKPERSISTENT`, this is needed to preserve the sharing of objects. In other words, if `make-persistent-copy` is called twice with the same object, the second time the object is *not* copied to another persistent location, but rather the persistent address established in the first call is returned to the second caller as well. `MKPERSISTENT` utilizes a hash table to maintain pairs of volatile and persistent addresses.

If `object` is not a new persistent object (*i.e.*, `(hd x)` is true), then the persistent address is returned. Otherwise the graph traversal continues, and `make-object-persistent` is called recursively on each field value of the object. The (possibly) updated values are installed in the persistent copy of the object.

After all `precommit` operations have completed, all updates are installed in the persistent heap, but not necessarily written to disk, *i.e.*, some dirty persistent heap pages may still be in the page cache. `Precommit` procedures each return a composite result indicating whether any local updates were processed, and whether any errors were encountered. If no local updates and no errors were reported, then the transaction is committed.

If errors were reported, then the transaction is aborted. Some sort of recovery mechanism, such as write-ahead logging [30], is needed in this situation to restore the original state of the database. Recovery is not implemented in the current version of AGNA, and is a topic for future investigation.

If no errors were reported, then the list of constraint checks is executed to ensure that the uniqueness of field inverse-mappings still hold. If no errors are reported in this step, then all

persistent pages are written to disk and the transaction is committed. If errors are reported, then the transaction is aborted.

Chapter 6

Analysis

The universe of database transactions can be divided into two sets, one containing transactions that can be expressed in SQL, and the other containing transactions that can't (see Figure 6.1). An example transaction in the former set is T1, while an example in the latter set is T2:

- T1: *Find the names of all students with a GPA of at least 3.9.*
- T2: *Find all direct and indirect prerequisites of course "Advanced Algorithms".*

As discussed in Chapter 2, T1 can be expressed succinctly and elegantly in SQL. Many sophisticated implementation techniques such as indexing and clustering of data are used by relational systems to deliver top-notch performance for such transactions. Both ease-of-expression and good performance are important for queries such as T1 because they form a common and useful class of transactions.

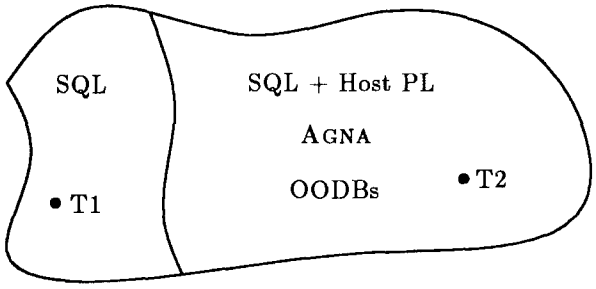


Figure 6.1: Division of universe of database transactions.

It is important for more expressive persistent systems to provide ease-of-expression and performance comparable to SQL for T1-like transactions. One of the primary reasons why relational databases displaced earlier CODASYL systems as the DBMS of choice is because transactions such as T1 could *not* be expressed easily in such systems. One had to write

a program that explicitly navigated through the collection of student records, and selected the name field of each record that satisfied the predicate—no high-level query language was supported.

In Chapter 2, we saw that list comprehensions in AGNA can be used to formulate T1 as concisely as in SQL. Most other object-oriented systems also provide high-level query languages with which to formulate T1 concisely. In the first section of this chapter, we present experimental results from a preliminary performance evaluation of both uniprocessor and multiprocessor versions of AGNA on T1-like queries, along with comparisons to similar results for commercial and experimental relational systems. The results show that the performance of AGNA on such queries approaches that of state of the art relational database systems.

The AGNA transaction language contains a full higher-order programming language, and thus the user may easily move from T1 to more complicated queries such as T2. This is also the case in most current object-oriented systems, since they also contain full programming languages (*e.g.*, C++ or Smalltalk). In Section 6.3, we compare the uniprocessor performance of AGNA on T2-like queries to that of relational and object-oriented systems. The results show that AGNA is competitive with other OODBs, and that both AGNA and other OODBs provide superior performance relative to relational systems on queries such as T2.

6.1 Measurement Methodology

For each query execution, we report elapsed (wall clock) time in seconds until *all* computation in the query is complete. Query results are built in the volatile heap, and are *not* traversed by the `print` function, *i.e.*, times for printing results are not included. Execution times are automatically reported by AGNA for all transactions. The emulator on which a transaction is initially loaded records the time immediately before execution commences, and again after it completes. The difference between the two times is reported as the elapsed time. In the Sun Unix implementation of AGNA, the C library function `ftime` is used to record times, while in the hypercube implementation, iPSC/2 C library function `mclock` is utilized.

For some transactions, we also give an execution profile showing how much time was spent in various activities such as I/O and heap management. On both the Sun and iPSC/2 platforms, the I/O time for a transaction was determined by tracing its I/O requests, and then timing the execution of a separate program that performed no computation, but issued a similar sequence

of I/O operations. We were careful to ensure that no disk pages were cached in the operating system prior to execution of these separate programs.[1]

On the Sun platform, a profile of activities other than I/O was obtained via the Unix profiling tool `gprof` [39]. On the iPSC/2, suitable profiling tools were not available, so we used analytical methods such as counting inter-PME messages, and simple experimental techniques such as timing successive executions of a node program, with each one adding a single new activity. In all cases, no other jobs were allowed to run concurrently on the machines on which we conducted our experiments.

6.2 Relational Queries

Transactions used in the first set of experiments were taken from the "selection" queries in the Wisconsin benchmark [17], a standard set of relational queries executed against a synthetic database. All of our experiments used the following object type:

```
(type WISC (extent)
  ((unique1  => INTEGER)
   (unique2 <=> INTEGER (index Btree))
   (filler   => (STRING 200))))
```

While the relation structure specified in the Wisconsin benchmark has more fields than type `WISC`, only `unique1` and `unique2` were utilized by the part of the benchmark that we used. Thus, we combined all other fields into the `filler` field for simplicity.[2] The total length of all fields was the same as the relation width specified in the Wisconsin benchmark. Fields `unique1` and `unique2` were assigned unique values from the range 1 to n, where n was the number of objects used in a particular test.

Test queries selected a subset of objects in the `WISC` extent. Parameters varied were: extent size, predicate selectivity, object field used by the predicate, query structure (list comprehension vs. explicit use of procedure `filter`) and, for multiprocessor experiments, machine size.

[1] On the Sun platform, this was accomplished by rebooting the machine prior to program execution. On the iPSC/2, AGNA accesses a raw disk directly, so no caching is performed by the operating system.

[2] The notation (STRING 200) in the specification of `filler` defines a fixed-length field of length 200. If a length is not specified in a string field, then values of unlimited length may be stored in the field.

6.2.1 Uniprocessor Results

Our uniprocessor experiments were designed to investigate the effectiveness in AGNA of low-level filtering and indexing, and to compare absolute performance with state of the art relational systems. The platform used was a Sun 4/490 configured as follows.

- **main memory**: sixty-four megabytes
- **operating system**: Sun UNIX OS 4.1.1
- **disk**: ipi-1000/2HP with an ips-80 controller

For all uniprocessor experiments, we used an extent size of 50,000 objects, and a 16 megabyte page cache.

Low-Level Filtering

The first experiment involved execution of the following two queries:

```
T3: (xact (filter (lambda (w) (and (> (select w WISC UNIQUE1) v1)
                                   (< (select w WISC UNIQUE1) v2)))
                  (all WISC)))

T4: (xact (all w
            (w (all WISC))
            (where (and (> (select w WISC UNIQUE1) v1)
                        (< (select w WISC UNIQUE1) v2)))))
```

Values $v1$ and $v2$ were chosen so that each query returned the same five hundred objects or one percent of the extent. The first query builds an extent list in the volatile heap, and then filters it. The second query performs the filter in the object storage system during object retrieval, building only a list of objects in the result. In both cases, no persistent heap or file pages were present in either the AGNA or operating system caches prior to query execution. The timings were:

T3: 148.5 seconds

T4: 034.5 seconds

Here is a breakdown:

T3	• 23% build extent list (34.2 secs)
	  • 10.2% list-cell allocation and field definition (3.5 secs)
	  • 24.6% object storage system (8.4 secs)
	  • 65.2% I/O (22.3 secs)
	• 77% filter and build result list (114.3 secs)

T4	• 00.1% result list construction, begin/end transaction (.04 secs) • 35.3% object storage system (12.2 secs) • 64.6% I/O (22.3 secs)

In T3, 23% of the time is spent scanning over the WISC file and building an extent list, while 77% of the time is spent filtering it and building the result list. The extent list construction consists of I/O (65.2%), page and object handling in the object storage system (24.6%), and allocation of list-cells and definition of their head and tail fields (10.2%). In T4, .1% of the time is spent constructing the result list and beginning and ending the transaction, 35.3% is spent in the object storage system manipulating pages and applying the predicate, and 64.6% is spent in disk I/O.

P-RISC manager instruction FILTEREXTENT is used to perform the file scanning and initial list construction in both T3 and T4, the only difference being that a file-scan predicate (*i.e.*, a list of simple conditions on field values) is utilized by T4 and not by T3. Since FILTEREXTENT is currently implemented as a strict primitive for simplicity, no overlap is possible in T3 between the extent list building and filtering. In both T3 and T4, the amount of time spent in I/O is the same, 22.3 seconds. Virtually all of this results from the WISC file scan performed in FILTEREXTENT. The WISC data file consists of 1516 pages; each page transfer, therefore, takes about 14.7 milliseconds, and the total transfer rate is .56 megabytes/second.

From the additional time spent in the object storage system in T4 (12.2 seconds vs. 8.4 seconds in T3), we can determine that 3.8 seconds (the difference) is spent in T4 checking whether WISC objects satisfy the predicate. When this checking is performed via the more general procedure `filter` in T3, it takes significantly longer—virtually all of the 114.3 seconds spent filtering and building the result list.[3] While some of this wide difference is attributable to compilation technology and our decision to emulate the P-RISC machine in software, the results of this experiment nevertheless demonstrate the significant impact on performance of efficient, low-level filtering.

[3]We shall see in the next section a detailed breakdown of a computation similar to `filter`, thus we don't examine it further here.

Indexing

The next experiment was designed to test the effectiveness of indexing in AGNA. It involved T4, and a new query T5, which is identical to T4 except that field unique2 is used instead of unique1:

```
T5: (xact (all w
          (w (all WISC))
          (where (and (> (select w WISC UNIQUE2) v1)
                      (< (select w WISC UNIQUE2) v2)))))
```

As before, values $v1$ and $v2$ were chosen so that each query returned the same five hundred objects. T4, as we have already seen, scans the entire WISC data file, performing the filtering during object retrieval. T5, on the other hand, utilizes the Btree index on unique2, thus avoiding a scan of the entire file. The timings were:

T4: 34.5 seconds

T5: 01.1 seconds

Here is the breakdown:

T4	• 00.1% result list construction, begin/end transaction (.04 secs) • 35.3% object storage system (12.2 secs) • 64.6% I/O (22.3 secs)
T5	• 03.6% result list construction, begin/end transaction (.04 secs) • 41.9% object storage system (.46 secs) • 54.5% I/O (.6 secs)

Both T4 and T5 spend the same amount of time constructing the result list and beginning/ending the transaction, but the time spent in the object storage system and I/O is much less in T5 because the Btree index provides direct access to those objects with unique2 field values in the desired range. The results of this experiment underscore the importance of efficient indexing in AGNA.

Comparison with INGRES

We also executed the relational equivalents of T4 and T5 using INGRES version 6.3, a modern commercial relational database system. We built a corresponding database of the same size, and used the same hardware platform (Sun 4/490, 64 Mb of memory, ipi-1000/2HP disk with

an isp-80 controller). INGRES was configured to use a page cache of the same size (16 Mb), and locking and logging were turned off to eliminate the overhead of concurrency control and recovery, respectively, since these functions are not performed by AGNA.

The queries were written in SQL and embedded in a C program. Rather than build result lists as in AGNA, we stored each record id of the result in a pre-allocated C array. Here is query T4:

```
void doT4()
{
    EXEC SQL BEGIN DECLARATION
    int ids[500], i=0;
    EXEC SQL END DECLARATION

    EXEC SQL SELECT id
         INTO   :ids[i]
         FROM   wisc
         WHERE  unique1 > v1 and unique1 < v2;
    EXEC SQL BEGIN;
         i++;
    EXEC SQL END;
}
```

Query T5 is similar. The INGRES timings were:[4]

T4: 34.1 seconds

T5: 00.3 seconds

These results indicate that even on a uniprocessor, the performance of AGNA is competitive with that of commercial relational systems on comparable queries. Further, we have an advantage over SQL in that AGNA's list comprehensions are part of a full functional language, so that the user can smoothly extend queries to perform arbitrary computation.

6.2.2 Multiprocessor Results

Multiprocessor experiments were conducted on an Intel iPSC/2 with 32 nodes. Each node includes an Intel 80386 processor, 8 megabytes of physical memory, and a MAXTOR 4380 disk drive with an embedded SCSI controller that maintains a 45 Kbyte read-ahead cache. Also included in each node is a specialized hardware router module, all of which are interconnected

[4]INGRES applications code such as doT4 executes in a separate operating system process, which connects to and interacts with a back-end database server. The timings do *not* include the time to connect to the server and to perform initializations such as opening the database.

to form a hypercube. Routers support eight bit-serial, full-duplex channels that connect a node to its eight nearest neighbors.[5] Each channel is capable of sustaining a data transfer rate of 2.8 Mbytes/second, independent of other channels.

AGNA uses the interrupt-driven communication primitives in NX, the operating system of the iPSC/2. An interrupt handler and message buffer are always posted on a node to receive incoming messages. When a message arrives at a PME, it is written directly to the posted message buffer, after which an interrupt occurs and the handler is invoked. Thus, no unnecessary copying of the message is performed.

Two different underlying protocols are used by the iPSC/2 for message transmission [27]. Messages of one hundred bytes or less are sent as datagrams, using a one-trip protocol. Messages longer than one hundred bytes use a three-trip protocol. In the first step, a proxy message is sent from the source to the target node to establish a communications circuit between them. If a receive is posted for the message or enough free memory exists to hold it on the target node, then a reply is sent back immediately requesting transmission of the message. In the third step, the message is sent and, as the tail of the message moves through the circuit, it is released one link at a time. The table below shows the latency (from one emulator process to another) of inter-node messages of various sizes [33]:

Message size in bytes	*Latency*
50	0.74ms.
500	1.46ms.
1000	1.57ms.
4000	2.69ms.
8000	4.64ms.

We conducted four sets of experiments to evaluate the performance of simple list comprehensions in the multiprocessor version of AGNA. The first set of experiments compared the multiprocessor performance of AGNA, using a full machine configuration of thirty-two nodes, to the uniprocessor performance of both AGNA and INGRES. The next three sets of experiments involved only the multiprocessor version of AGNA, and varied the machine size (*i.e.*, number of PMEs), the problem size (*i.e.*, number of objects in the extent), or both. The second set of experiments evaluated performance relative to extent size by keeping the machine size constant while scaling up the problem size. The third set evaluated the scalability of the system by maintaining a constant problem size while increasing the number of PMEs in the machine configuration. Finally, the fourth set of experiments evaluated the ability of the system to maintain

[5] Configurations with both compute and I/O nodes use one of the channels exclusively for communication with the I/O subsystem.

a constant response time as the problem size and machine size were increased proportionally. The design of the latter three sets of experiments was motivated by the performance study reported in [33]. For all experiments, a 4 megabyte page cache was used; the remainder of physical memory was consumed by other system data structures, message buffers, and interpreter and manager thread code.

All of the queries used in our experiments involved the WISC type described previously and the following query template:

```
(xact
  (all w
    (w (all WISC))
    (where (and (> (select w WISC unique1-or-unique2) v1)
                (< (select w WISC unique1-or-unique2) v2)))))
```

Values v_1 and v_2 were varied to test queries that returned different numbers of objects. The field used in the predicate expression was varied between unique1 and unique2 to test both indexed and non-indexed access to WISC objects. For all experiments, WISC objects were distributed uniformly across all PMEs of the machine.

Effect of parallelism

In the first set of experiments, we selected 1% of an extent size of 100,000 objects, using both unique1 (the non-indexed field) and unique2 (the indexed field). We gathered performance results for AGNA on the iPSC/2 multiprocessor platform (all 32 PMEs), and both AGNA and INGRES on the Sun uniprocessor platform described in Section 6.2.1. The results are tabulated below.

	1% non-indexed	1% indexed
AGNA (Sun)	68.3	1.2
INGRES (Sun)	68.0	0.4
AGNA (Hypercube, 32 nodes)	4.1	2.2

Results for the non-indexed query demonstrate the exploitation of parallelism in AGNA to achieve significantly greater performance than that of INGRES. The performance difference is actually greater than it appears, because the Sun processor and disk are approximately three and two times faster, respectively, than the processor and disk used in the Hypercube. Performance of AGNA on the multiprocessor platform for the indexed selection is less than that of both AGNA and INGRES on the uniprocessor platform, because of the little computation and disk I/O performed by the query (*i.e.*, it is not a good candidate for parallel execution), and

the slowness of the Intel Hypercube network which, as we shall see below, is the performance bottleneck for this class of query.

Performance relative to extent size

In the second set of experiments, the machine configuration was kept constant at 32 PMEs while the extent size was increased from 100,000 to 10,000,000 objects. We executed two queries using `unique1` (the non-indexed field), selecting 1% of the objects in the base extent in the first one, and 10% in the second one. We also executed two queries using `unique2` (the indexed field), selecting 1% of the objects in the first one, and a single object in the second one.

Ideally, the increase in response time would not grow at a rate faster than the increase in extent size. The results are shown in the table of Figure 6.2. For the non-indexed selections, the increase in response time is almost perfectly matched with the increase in extent size from 1,000,000 to 10,000,000 objects (27.8 to 275 seconds, and 29.1 to 298 seconds). For the increase in extent size from 100,000 to 1,000,000 objects, this is not the case because with 100,000 objects the time it takes to begin a transaction, dispatch the local filter operations, append the local result lists, and end the transaction becomes more significant (almost half of the total time) relative to the time spent filtering on each node. Response time for the indexed selections is dominated by this overhead.

Query Description	100,000	1,000,000	10,000,000
1% non-indexed selection	4.1	27.8	275
10% non-indexed selection	4.3	29.1	298
1% indexed selection	2.2	2.6	7.0
1 object using index	2.1	2.1	2.1

Figure 6.2: Performance relative to extent size.

Let us examine two of the queries in more detail: selection of a single object from an extent of size 100,000 (lower left-hand corner of the table), and selection of 1% of an extent of size 10,000,000 objects (upper right-hand corner).

While appropriate run-time metering tools weren't available on the iPSC/2, an analysis of the number of messages in the critical path of execution of the first query suggests that it is network-bound. As described in Section 5.9, query execution is organized into four steps, each completed entirely before the next is allowed to begin: (1) prologue, (2) body, (3) epilogue precommit, and (4) epilogue commit. All steps involve distributing some computation to each

node of the machine, and steps (2) and (3) also include receipt and handling of results. Let us examine the messages exchanged during the prologue. Procedure `begin-transaction`, defined below along with two additional routines, implements the prologue.

```
(define begin-transaction
  (lambda ()
    (letrec ((id (next-transaction-id)))
      (foreach (lambda (j) (APPLY PME j local-begin-transaction id))
               all-pmes))))

(define local-begin-transaction
  (lambda (id) ...Perform local initializations...))

(define foreach
  (lambda (f l)
    (if (nil? l)
        "DONE"
        (letrec ((x (f (hd l))))
          (foreach f (tl l))))))
```

The following messages are transmitted in the call to `begin-transaction` from the transaction body:

Number of Messages	*Operation*
2	lookup identifier `begin-transaction`
2	read frame size from procedure object
2	allocate frame
4	store procedure on same PME as frame
2	start trigger and signal threads in body

Messages are sent in sequence because each depends on the result of the previous one, and all but one of the 12 messages (the one that starts the signal thread) are required before computation may begin in the body of `begin-transaction`.[6] By a similar analysis, up to 14 messages may be transmitted before the trigger and list argument threads are started in the body of procedure `foreach`. Each subsequent invocation of `foreach` may also require transmission of up to 14 messages. After the final invocation returns a result and signal, the next phase of the transaction (*i.e.*, the body) may begin. The maximum number of messages in the critical path during the epilogue step is tabulated below:

[6]This message count assumes a worst-case scenario. For example, the procedure may already reside on the same PME as the frame, in which case no messages are required.

Number of Messages	*Operation*
11	invocation of `begin-transaction`
14	initial invocation of `foreach`
14*32	all subsequent invocations of `foreach`
2	signal and result return
475	

The unfolding of computation in the prologue is shown in Figure 6.3. It is important to realize that the total number of messages in the table above does not represent the number of messages transmitted during the prologue phase, but rather an estimate of those in the critical path, *i.e.*, those required to "grow" the spine shown in the figure. In fact, many more inter-PME messages, such as those to invoke procedure `local-begin-transaction` on each PME, will be transmitted.

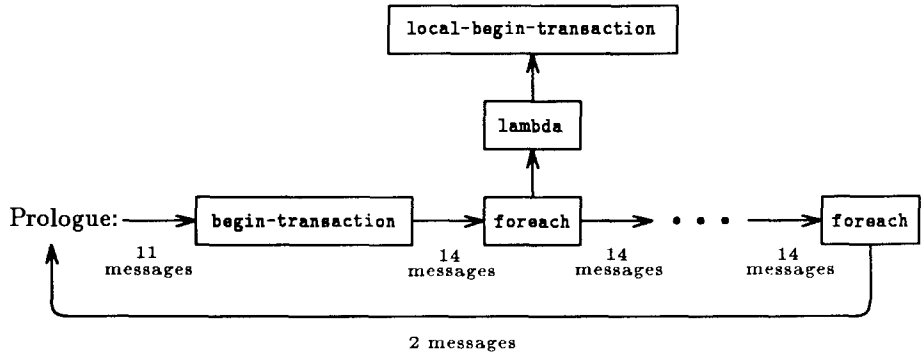


Figure 6.3: Messages in critical path of prologue execution.

Assuming that all messages are fifty bytes in length, the total transmission time for messages in the critical path is approximately .35 seconds (475 * .75 ms).[7] Roughly speaking, this same pattern of computation is used in the subsequent three steps, and hence approximately the same number of message transmissions are required. An estimate of the time to send all messages in the critical path of the entire transaction execution, then, is 1.4 seconds (4 * .35), which is 67% of the total execution time. Since few disk transfers and little computation are performed by the transaction, processors and disks are most likely idle the majority of the time.

While the distribution of computation to PMEs in AGNA could be modified to reduce the number of message transmissions in the critical path, one conclusion of the analysis above is that the latency of message transmission in the iPSC/2 is such that the network may quickly become a bottleneck in computations which involve much non-local communication. A non-

[7]This is a conservative assumption because messages which copy a procedure to a PME will always be much longer than fifty bytes.

local procedure call, for example, requires a minimum of five messages (two to read the frame size, two to allocate the frame, and one to start the trigger thread) to initiate execution in the procedure body. The message transmission time alone for this is almost four milliseconds. In the implementation of general purpose models of computation such as ours, many small messages are inevitable. Thus, we believe that the latency of inter-PME messages in the iPSC/2 must be reduced by at *least* an order of magnitude to provide acceptable overall performance for expressive models of computation.

Now let us examine the selection of 1% of an extent of size 10 million objects using `unique1`, the non-indexed field. In this case, the network is not the limiting factor. Approximately 255 of the 257 seconds of execution is spent with all PMEs executing (in parallel) a copy of procedure `local-filter-extent` which, as we saw in the previous chapter, builds a local list of objects satisfying the predicate. The bulk of procedure `local-filter-extent` is implemented by P-RISC manager FILTEREXTENT, whose execution profile for this query is given below.

69% computation
- 52.2% object manipulation
- 34.8% result list construction
- 13.0% page cache management

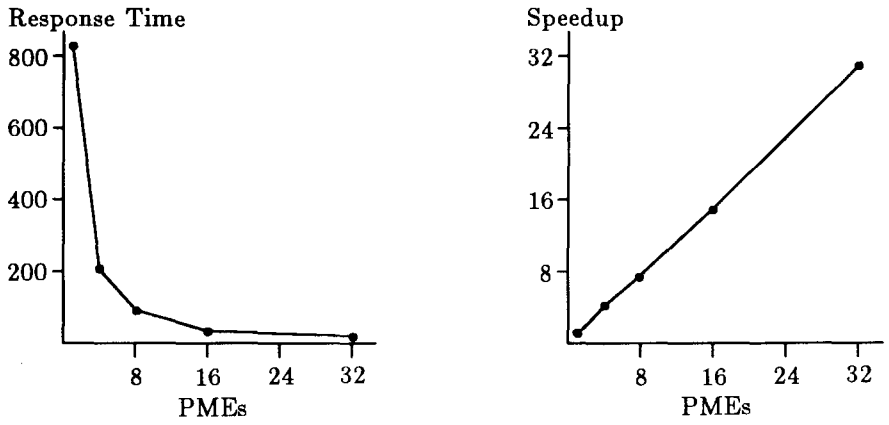
31% disk I/O

FILTEREXTENT, and hence the transaction, is compute-bound, as 69% of the time is spent in object manipulation (*e.g.,* applying the predicate), constructing the result list, and managing the cache of pages, while only 31% is spent performing I/O.

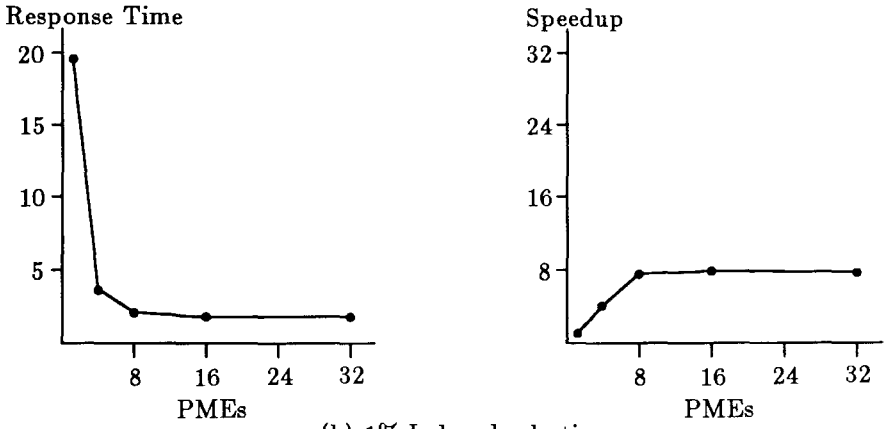
Speedup

In this set of experiments, we kept the extent size constant at 1,000,000 objects while the machine configuration was increased from 1 to 32 nodes. The ideal behavior in this case would be for response time to decrease (or speedup) proportionally with increases in machine size. Results for the 1% selection queries are shown in Figure 6.4. (We also executed the 10% selection query using the non-indexed field; the response time and speedup curves for it are almost identical to those for the 1% non-indexed selection.)

The speedup for non-indexed selections is practically linear from 1 to 32 nodes. We would like to emphasize that this linear speedup depends, in part, on the uniform distribution of WISC objects across the machine. If objects were distributed in a non-uniform manner, then it is



(a) 1% Non-indexed selection.



(b) 1% Indexed selection.

Figure 6.4: Speedup for 1% selection.

possible that the speedup would be less. For the indexed selection, good speedup is obtained from 1 to 8 nodes, but then response time levels off due to the additional communications overhead with larger machine configurations.

Scaleup

In the final set of experiments, we increased the machine and extent sizes proportionally. The ideal behavior in this case would be for response time to remain constant. The results are shown in Figure 6.5.

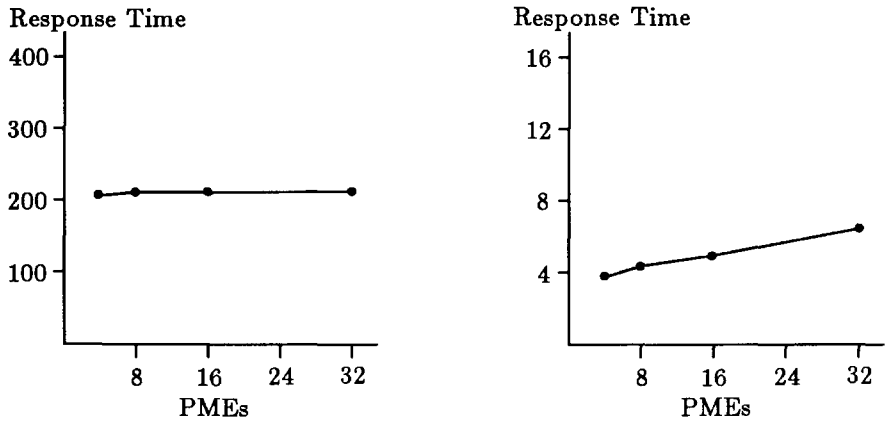


Figure 6.5: Scaleup for 1% selection without index (left) and with index (right).

Response time remains relatively constant for selection using the non-indexed field. For the indexed selection, response time increases slowly with increases in the machine size due, again, to the increased communications overhead.

Comparison with Gamma

The Gamma Parallel Relational Database Project at the University of Wisconsin has also reported results for the selection queries that we used in our experiments. A thorough performance evaluation of Gamma, on the same hardware platform, is reported in [33].

For queries that involve a significant amount of processing relative to the overhead of a transaction, *e.g.*, a non-indexed selection on all 32 nodes with an extent size of 10,000,000 objects, Gamma is anywhere from two to five times faster than AGNA. For very short transactions, the difference is more significant, often greater than a factor of ten. We believe that the differences in performance can be attributed to three things:

1. The P-RISC abstract machine, the target of the AGNA compiler, is currently emulated in software. Further, we know of numerous optimizations possible on P-RISC code that we have not yet had the opportunity to incorporate. From hand-coded examples and results reported for Culler's Threaded Abstract Machine [67], we believe that these optimizations and compilation to native code can easily increase overall performance by a factor of 10 or more.

2. AGNA implements objects in a segmented, paged, virtual heap. The management of this heap, including its paging to disk, is implemented entirely in software which has not been optimized very much yet. Further, our heap structure is more complicated due to the more complex object model supported by AGNA. For example, the storage model in Gamma is not a global heap, but rather a collection of files of records. The mapping of records to files is determined statically when a relation is created, and no direct inter-record references are allowed.

3. Much less effort has gone into AGNA to date to tune the system, particularly the object storage system, where additional optimizations could, we believe, increase the performance of low-level object scans by at least a factor of two. Also, additional optimizations to reduce the number of inter-PME messages have not yet been incorporated.

We are very encouraged by our results, and believe that we are within shooting distance of parallel relational systems on comparable queries. Again, we have the advantage that in AGNA, the programmer can smoothly escalate to more complex objects and queries that involve general networks of objects and arbitrary computation.

6.3 Extra-Relational Queries

We also performed a set of experiments using the Engineering Database Benchmark (EDB) [24] to evaluate the performance of AGNA on extra-relational transactions. We begin this section with a description of the benchmark database and operations, then we report uniprocessor results for AGNA. Next we compare our results to similar results for INGRES and a commercial object-oriented database system. Multiprocessor results for AGNA on this benchmark are not available, as an unresolved bug in the iPSC/2 communications subsystem has prevented us from running the benchmark operations successfully on that platform.

6.3.1 Engineering Database Benchmark

The Engineering Database Benchmark was designed to evaluate the performance of database systems on basic operations commonly performed by engineering applications. Object types in the benchmark database are PART and CONNECTION, defined as follows:

```
(type PART (extent)
  ((id          <=>  INTEGER)
   (type         =>  (STRING 10))
   (x            =>  INTEGER)
   (y            =>  INTEGER)
   (build-date   =>  INTEGER)
   (connections <=>* CONNECTION)))

(type CONNECTION (extent)
  ((part *<=> PART)
   (type   => (STRING 10))
   (length => INTEGER)))
```

Each part consists of a unique id, four other attributes, and connections to three parts; the connection information is stored in multi-valued field connections. Each connection consists of a part (*i.e.*, part B in a connection from part A to B), and two attributes. Thus, the database consists of a graph of objects as shown in Figure 6.6.

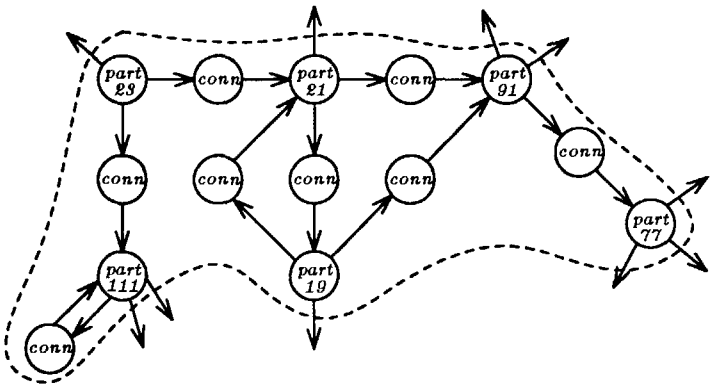


Figure 6.6: Part and connection objects in EDB database.

While databases of various sizes are included in the benchmark, we chose to use the smallest because more published results for other systems are available using this size than any of the others. The "small" database consists of 20,000 parts and 60,000 connections. As specified in the benchmark, connections between parts are chosen to enhance locality in the sense that 90% of all connections are made from a source part to a target part selected randomly from the "closest" 1%, *i.e.*, those 1% of the parts with id values numerically closest to the id of the

source part. The remaining 10% of the connections are made to target parts selected randomly from the entire part extent.[8]

Operations in the benchmark are:

- **Lookup.** 1000 random ids are generated and the corresponding parts are fetched. The `x`, `y`, and `type` field values of each part are passed to a null-bodied procedure.
- **Traversal.** Select a part randomly, and then visit all parts reachable in seven or fewer connections. (The total number of parts visited is 3280, with duplicates.) The `x`, `y`, and `type` field values of each part visited are passed to a null-bodied procedure. A reverse traversal is also performed, *i.e.,* the same operation except that the connections are traversed in the opposite direction.
- **Insert.** Create 100 new parts and their connections.

All operations are performed in each of ten iterations. Initially, no data pages may be present in the database or OS caches. The results of the first iteration are referred to as "cold" start results, and the asymptotic best times of the remaining iterations, which may access cached pages, are referred to as the "warm" start results. The EDB designers made the distinction between warm and cold start results to distinguish systems which enhance performance via efficient caching of data in applications programs. Engineering applications, the target domain of the benchmark, often access the same data repeatedly from different parts of the program, thus performance can be improved significantly if data is cached efficiently. The Wisconsin benchmark designers, on the other hand, did not make such a distinction, because relational queries typically do not exhibit this pattern of access, and thus the benchmark is focused more on testing query processing algorithms, use of indexes, *etc.* rather than particular caching strategies.

For each of the ten iterations, a different set of random ids and parts must be selected. The benchmark specification allows the database to be stored locally or remotely and accessed via a network. For our experiments, we used a local database since remote access, in the sense defined by the benchmark, is not supported by AGNA.

[8] A random number generator is utilized in construction of the database and in the benchmark operations. The random number generator used in AGNA, described in [65], is the one suggested by the EDB authors [24].

6.3.2 Experimental Results

We ran the EDB benchmark on the uniprocessor platform described in the previous section (Sun 4/490, 64 megabytes of memory, ipi-1000/2HP disk with an isp-80 controller). As before, a 16 megabyte page cache was used. The results (response times in seconds) are tabulated below.

Operation	*Cold*	*Warm*
Lookup	13.9	7.2
Forward Traverse	21.7	12.2
Reverse Traverse	42.4	32.5
Insert	9	8

Total size of the database was 11 megabytes and the database build time was 1180 seconds.

Warm-Start Execution of Forward Traverse Operation

Let us now examine in detail a warm-start execution of the forward traversal operation, *i.e.*, one in which all relevant data pages are in the cache prior to execution, thus no I/O is required. First, here are the AGNA procedures that perform the operation:

```
(define forward-traverse
  (lambda (part hops total-parts-visited)
    (letrec ((x  (select part PART X))
             (y  (select part PART Y))
             (t  (select part PART TYPE))
             (a  (null-proc x y t))
             (cs (select part PART CONNECTIONS)))
      (if (>= hops 1)
          (f-traverse-helper cs (- hops 1) (+ 1 total-parts-visited))
          1))))

(define null-proc (lambda (x y type) ()))

(define f-traverse-helper
  (lambda (cs hops n)
    (if (nil? cs)
        n
        (letrec ((part (select (hd cs) CONNECTION PART)))
          (f-traverse-helper (tl cs)
                             hops
                             (+ n (forward-traverse part hops 0)))))))
```

Procedure `forward-traverse` takes a part, the number of "hops" or connections to follow from the part, and the total number of parts visited so far. The `x`, `y`, and `type` fields are selected and passed to `null-proc`, as specified by the benchmark. If the number of hops remaining is at least one, then procedure `f-traverse-helper` is utilized to continue the traversal. Its arguments

are a list of connections, the number of hops, and the total number of parts visited. Procedure `f-traverse-helper` simply iterates over the list of connections, selecting the target part of each and passing it to `forward-traverse`.

The forward traversal operation is invoked as follows:

```
(xact
  (forward-traverse (invert PART ID (random 1 20000)) 7 0))
```

Procedure `random` returns a random number in the specified range, which is then used to select the part from which the traversal begins. The number of hops is initially seven, and the count of parts visited zero.

Here is the breakdown of a warm execution of the transaction above.

27.7%	frame pointer address translations, initiated via procedure `interpreter`
21.0%	procedure `interpreter`, but not procedures called by it
16.0%	building/sending/receiving messages
11.4%	allocation and deallocation of frames
06.5%	miscellaneous
06.3%	identifier lookups
05.6%	pinning/unpinning heap pages via procedure `interpreter`
05.5%	selection of object fields

The largest category in the breakdown, which consumes 27.7% of the time, involves only address translations of FPs initiated via procedure `interpreter`. Such translations, which total 93,025, are performed when the interpreter switches execution from one frame to another (25,172), and in the execution of START0 (28,459) and START1 (39,394) instructions. Including the additional address translations performed in object field selection, frame allocation, *etc.* a total of 32.4% of the time is spent translating AGNA heap addresses. Thus, it is clear that the implementation of the virtual heap in software in AGNA is a major source of overhead. Each translation of a (resident) heap address takes 166 native SPARC instructions. In Section 6.3.4, we discuss techniques used by some OODBs to reduce this overhead, and their applicability in AGNA.

Execution of procedure `interpreter`, the next largest category, consumes 21% of the time. This does not include time spent in any procedures called by the interpreter, but only in the interpreter procedure itself. The bulk of this time represents the overhead of interpretation of the P-RISC instruction set. For example, execution of a P-RISC ADD instruction in procedure interpreter takes 26 native SPARC instructions.[9] While this overhead may be reduced somewhat by more efficient compilation to P-RISC code and additional optimizations in procedure

[9]This assumes that the P-RISC frame and instruction pointers have already been translated.

`interpreter`, it is clear that the overhead of interpretation is significant. Compilation to native machine code is thus an important area of future research in AGNA.

The building, sending, and receiving of messages is the next largest category, consuming 16% of the time.[10] The primary reason why this category consumes so much time is that the interpreter and manager lightweight threads are each mapped to a separate OS process in the current uniprocessor implementation, thus sending an intra-PME message involves inter-process communication, which is relatively expensive. For example, a 100 byte message sent from one process to another takes approximately .5 ms. While we would have preferred to place all lightweight threads in the same OS process and thereby reduce the communication overhead considerably, the only thread package that we had access to (Sun's lightweight process library) did not support non-blocking I/O, *i.e.*, when a lightweight thread issues an I/O request, the entire OS process blocks, even if other lightweight threads are ready to execute. In the iPSC/2 implementation, on the other hand, a suitable thread package was available, and thus all lightweight threads are placed in the same OS process.

The next most time consuming activity (11.4%) is the allocation and deallocation of frames. Allocation involves a call to the heap storage allocator, which maintains a table of free blocks of various sizes, and initialization of the frame structure. Deallocation involves a call to the storage allocator to free storage occupied by the frame. Both allocation and deallocation involve address translations which, as we just saw, are quite expensive. A total of 10,946 frames are allocated and deallocated by the transaction.

Remaining categories in the breakdown above are: various miscellaneous tasks; top-level identifier lookups; pinning and unpinning of the heap pages on which the current frame and code objects reside; and the selection of object fields.

Cold-Start Execution of Forward Traverse Operation

We also examined a cold-start execution of the forward traverse operation to determine the degree to which the latency of disk I/O is masked by parallelism. We executed the query using a modified version of AGNA that included only one manager thread, whose execution was not

[10] In a warm execution, the interpreter thread only sends messages to manager threads for lookups of top-level names not in the identifier cache, and selection of multi-valued fields. As discussed in the previous chapter, such operations are always performed by manager threads because it is difficult to predict whether the associative searches that they perform will involve access to non-resident pages.

allowed to overlap with the interpreter thread.[11] We also executed the query using different numbers of manager threads that did overlap execution with the interpreter thread. The results are tabulated below.

Number of Manager Threads	*Overlap?*	*Response Time*
1	no	25.9
1	yes	22.5
2	yes	21.6
3	yes	21.7

With one manager thread and no overlap of disk I/O and computation, the response time is 25.9 seconds. This improves by approximately 16% (4.3 seconds) when two or three managers are used and overlap is allowed. Regardless of the number of manager threads used and whether overlap is allowed, 574 disk pages are read in manager threads. Using the average service time reported earlier (14.7 ms), the total I/O time is 8.3 seconds.

If 8.3 seconds are spent in I/O, though, why does response time only improve by about half that amount (4.3 seconds) when overlap is allowed? The answer involves the following three factors. First, while much parallelism is available throughout most of the transaction (see Figure 6.7[12]), there are short periods of time at the beginning and end of execution during which not enough work is available to cover I/O time completely. During these periods, the emulator is idle for a total of approximately .5 seconds waiting for I/O to complete.

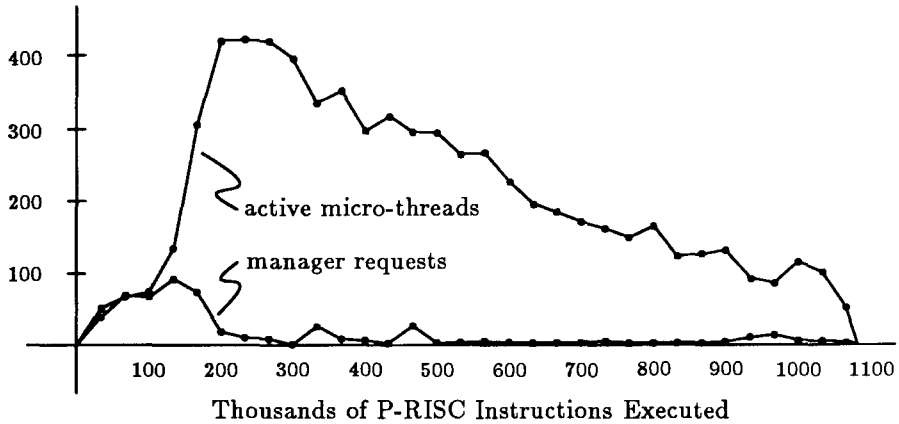


Figure 6.7: Parallelism profile of forward traverse operation.

[11] This was accomplished by having the interpreter thread, after sending a message to the manager thread, immediately block awaiting the reply.

[12] Raw data for the parallelism profile of Figure 6.7 was produced by the statistics-gathering facilities of AGNA. Data such as the number of active micro-threads and queued manager requests are recorded at regular intervals (every 10,000 P-RISC instructions), and may be viewed by the user after transaction execution is completed.

Second, not all of the 8.3 seconds of I/O time is available for other computations. For example, before the interpreter may begin execution after an I/O request is initiated by a manager thread, the operating system must switch execution from the manager to the interpreter process, thus incurring the overhead of a context switch. Also, the 14.7 ms estimate of I/O service time includes the time required to copy the disk page from a kernel buffer to user space, *i.e.*, to an AGNA page frame buffer. During this copying, which takes a total of about .33 seconds, the processor is not available for execution of other threads.

Third, execution of the transaction with overlap of I/O and computation involves the exchange of more intra-PME SELECTF messages than when no overlap is allowed. For example, a number of connection objects may reference parts that are on the same non-resident page. With no overlap, the interpreter thread issues a SELECTF request to a manager thread the first time an object on a non-resident page is referenced, and waits for the reply. All subsequent references to objects on the page find it resident, and thus may be handled entirely within the interpreter thread.

When overlap is allowed, the interpreter thread may issue multiple SELECTF manager requests involving objects on the same non-resident page before the page is made resident by a manager thread.[13] The I/O is performed only once, but the extra manager messages are expensive given the high cost of intra-PME messages in the current uniprocessor implementation of AGNA. In the forward traverse operation with one manager thread and no overlap, 1440 messages are sent from the interpreter to the manager thread. With three manager threads and overlap, an additional 700 SELECTF messages are sent. The additional messages, and their replies (START1), represent a substantial overhead, as each message requires construction, transmission, decoding, context switching, *etc.* Thus, the third reason why response time does not improve by more than 4.3 seconds when computation is overlapped with I/O is that more messages, and hence more computation, is required.

Warm-Start Execution of Forward Traverse Operation

We also used the statistics gathering facilities supported by AGNA to determine the parallelism available in the interpreter during a warm execution of the forward traversal operation. The profile is shown in Figure 6.8.

[13] While the interpreter could keep track of the pages for which I/O requests have been issued and thus avoid multiple manager requests, this is currently not done.

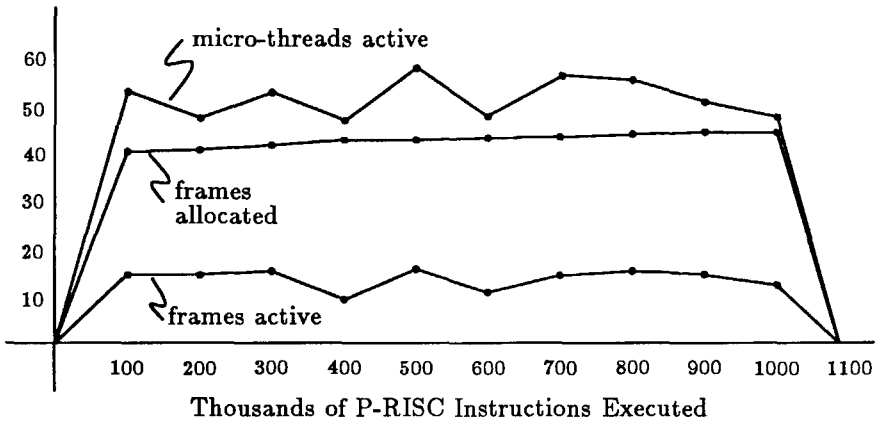


Figure 6.8: Parallelism profile.

For the most part, all three curves remain relatively flat throughout execution of the transaction. The reason for this is that the computation unfolds in a depth-first rather than a breadth-first manner. In the body of `forward-traverse`, the frame for the call to `null-proc` is allocated and threads in the body activated (but not executed) before the frame and threads for `f-traverse-helper`. In the body of `f-traverse-helper`, the frame for the recursive call is allocated and threads activated before the frame and threads for `forward-traverse`. As described in the previous chapter, active frames are maintained in a stack, and this causes the traversal to be performed in a depth-first manner.

For a warm execution on a uniprocessor, a depth-first traversal is optimal because it is the most resource-efficient. A breadth-first traversal, for example, may allocate more frames than can fit in physical memory at any one time, and thus may incur expensive paging overhead. The additional parallelism in a breadth-first approach, though, may be desirable in a multiprocessor environment, and even on a uniprocessor to help mask the latency of disk I/O. This, and other issues related to resource management, are an important topic of future investigation in AGNA.

6.3.3 Comparison With INGRES

We executed the Engineering Database Benchmark using INGRES on the same hardware platform. As before, INGRES was configured to use a 16 megabyte page cache, and logging and locking were turned off, since these functions are not performed in AGNA. The SQL representation of the database includes two tables, PART and CONNECTION, as shown in Figure 6.9. Hash indexes were built on the ID column in the PART table, and the FROMID and TOID columns in the CONNECTION table. All of the operations were coded in SQL embedded in C. Also, SQL queries

were compiled where possible to avoid the overhead of parsing, optimization, access path selection, *etc.* on each query execution. The INGRES results are tabulated below, along with the AGNA results presented earlier.

	Cold		*Warm*	
Operation	INGRES	AGNA	INGRES	AGNA
Lookup	22.2	13.9	18.0	07.2
Forward Traverse	66.5	21.7	57.0	12.2
Reverse Traverse	76.1	42.4	69.0	32.5
Insert	20.0	09.0	19.0	08.0

Total size of the INGRES database was 5 Mb and the database build time was 2855 seconds. AGNA provides superior performance in all cases; we believe there are three fundamental reasons for this. First, AGNA supports the overlap of execution and disk I/O which, as we just saw, improves the cold start results. In INGRES, each transaction executes in a single thread of control, so when a disk transfer is initiated, the entire transaction is suspended until it is complete.

PART

ID	TYPE	X	Y	BUILD_DATE
17	"part-type7"	23	1209	227488376
21	"part-type3"	98945	2246	755333
	⋮			

CONNECTION

FROMID	TOID	TYPE	LENGTH
17	21	"conn-type4"	345
34	3345	"conn-type0"	9
	⋮		

Figure 6.9: Relations in EDB benchmark.

Second, AGNA supports a richer object model, and in particular, objects may directly reference other objects. A connection object, for example, points at the target part of the connection. Traversing this link to a target part in AGNA involves only the dereferencing of a pointer. In INGRES, on the other hand, access to the target part referenced via a row in the CONNECTION table involves a general associative lookup in the PART table.

Finally, all of the benchmark operations are written entirely in the AGNA transaction language—no embedding in a separate language is necessary. This allows the AGNA compiler to see and thus optimize the entire operation. Also, this organization allows the entire transaction to be executed in the "database system". As we saw in Chapter 2, in INGRES one is forced to embed more complicated operations in a host language. For example, here is the embedding of the forward traversal operation in C:

```
int forwardTraverse( id, hops, totalPartsVisited )
```

```
EXEC SQL BEGIN DECLARE SECTION;
int id;
EXEC SQL END DECLARE SECTION;
int hops, totalPartsVisited;
{
    EXEC SQL BEGIN DECLARE SECTION;
    int x, y, toId;
    char partType[11], cursorName[8];
    EXEC SQL END DECLARE SECTION;

    if ( hops<1 ) return 1;

    /*
    ** Fetch part X, Y, and TYPE fields.  Call null procedure.
    */
    EXEC SQL REPEATED SELECT x, y, type
                      INTO :x, :y, :partType
                      FROM parts
                      WHERE id=:id;
    nullProc(x,y,partType);

    /*
    ** Continue traversal.
    */
    sprintf(cursorName,"level%d",hops);
    EXEC SQL DECLARE :cursorName CURSOR FOR
             SELECT toId
             FROM   connections
             WHERE  fromId=:id;

    EXEC SQL OPEN :cursorName;
    EXEC SQL WHENEVER NOT FOUND GOTO closeCursor;

    totalPartsVisited++;
    while ( 1 ) {
        EXEC SQL FETCH :cursorName INTO :toId;
        totalPartsVisited += forwardTraverse(toId,hops-1,0);
    }

  closeCursor:
    EXEC SQL CLOSE :cursorName;
    return totalPartsVisited;
}
```

Procedure `forwardTraverse` takes a part id, the number of hops, and the total number of parts visited. The first SQL query fetches the `x`, `y`, and `type` fields of the current part and passes them to a null procedure. Next, the traversal is continued by declaring and opening a cursor-scan over the parts to which the current part is connected. For each part returned, `forwardTraverse` is called recursively.

Note that here part of the operation is implemented by SQL, and part by the host language. This means that the SQL compiler cannot see and thus cannot optimize the entire operation—it

is only able to optimize small pieces of it. Also, this organization results in run-time inefficiencies, as the front-end OS process containing `forwardTraverse` must repeatedly exchange small pieces of the operation and replies with the back-end database system. The result is high communication overhead and inefficient use of the database system.

6.3.4 Comparison With an OODB

EDB benchmark results for commercial object-oriented and relational systems, whose identities were deliberately not revealed, are reported in [24]. The hardware platform used was a Sun 3/260 with 8 megabytes of memory, an SMD-4 disk controller, and two local Hitachi disks. The operating system used was Sun UNIX OS 4.0.3. The page cache was approximately 5 Mb in size, which was large enough to hold the entire database in both systems. The small *remote* database was used. The results reported in [24] are tabulated below.

	Cold		*Warm*	
Operation	OODB	RDB	OODB	RDB
Lookup	20.0	29.0	01.0	19.0
Forward Traverse	17.0	94.0	01.2	84.0
Insert	03.6	20.0	02.9	20.0

It is difficult to make precise comparisons between the OODB results and the AGNA results presented earlier because of the different hardware and OS platforms used, and the placement of the database (local vs. remote). However, since the RDB results given above are similar to the results that we obtained using INGRES, we believe that two general conclusions are justified. First, the cold-start performance of AGNA is competitive with OODB: the lookup results for AGNA are roughly 30% faster, the forward traverse results 30% slower, and the insert results two to three times slower. Again, this does not take into account the differences cited above.

Second, the warm-start performance of AGNA is at least an order of magnitude slower than that of OODB. We feel that the majority of this difference is attributable to the overhead inherent in AGNA's software-based emulation of the P-RISC machine. If we were to compile AGNA to native machine code, we believe that the warm-start performance of the forward traversal operation would improve by at least an order of magnitude. We base this on the performance results reported by Culler and his group at Berkeley [29, 67] for Id programs compiled to native code, which execute two orders of magnitude faster than Id programs compiled and interpreted under GITA, a graph interpreter for the MIT Tagged-Token Dataflow Architecture [5].

Another factor is the techniques used by OODB (and most other OODBs) to provide fast access to memory-resident data. While [24] does not mention the specific techniques employed

by OODB, those mentioned in the literature include converting all inter-object references in resident objects from logical identifiers to memory addresses, constructing special indexes on resident objects, *etc.* [32].

One strategy for implementing the first technique works as follows. When a page of objects is read into memory, all objects in the page are scanned and outgoing logical pointers are converted (or "swizzled") to memory addresses. If a logical pointer references a non-resident object, then it is converted to a memory address that will cause a trap if the pointer is dereferenced. A trap handler fetches the page on which the target object resides, and converts all logical identifiers in the page to memory addresses. While this approach is not very portable, it does support efficient pointer traversal, since no interpretation of pointers is required.

Before an updated page of objects can be written back to disk, all pointers must be converted back to logical identifiers. Also, it is necessary to ensure that pointers (in resident objects) to objects on the page will generate a trap when dereferenced. This is potentially very expensive, as conceptually it requires, for each object, maintaining the set of all objects which reference it.

The tradeoffs of this approach are high overhead when disk pages are read or written, but efficient access to memory-resident objects. Most commercial OODBs are targeted to design applications which [24] estimates have working set sizes of approximately 4 Mb (*e.g.,* a CAD drawing) and include an order of magnitude more reads than writes. For applications with these characteristics, the techniques described above are very effective at increasing performance.

While we could also utilize such techniques in AGNA, it is not clear that this is is a good idea, for two reasons. First, optimizations such as pointer swizzling are significantly more complicated in a multiprocessor implementation. For example, in both uniprocessor and multiprocessor systems, there are logical (*i.e.,* disk-based) and physical (*i.e.,* memory-based) pointers that must be manipulated. In multiprocessor systems, though, there is the additional complication of local vs. remote pointers, of both kinds. Say, for example, that a non-local object is referenced from PME i via its logical id. Should the object be copied to PME i and the pointer converted? If so, what if the object is also referenced from PME j? Maintaining consistency in such cases is equivalent to the multiprocessor caching problem.

The second reason why OODB optimization techniques might not be as effective in AGNA is that we have targeted our system to a wider class of databases and applications which often will not have the characteristics of design applications mentioned previously, for which

such techniques are most effective. The EDB benchmark shows systems that perform these optimizations in the best possible light because the entire database fits in memory, and thus such systems never have to pay the cost of unswizzling pointers. AGNA accesses resident data less efficiently, but will incur much less overhead when pages of objects are read from and written to disk.

Chapter 7

Concluding Remarks

In this chapter we summarize the present work, compare our work with related work of other researchers, and outline directions for future research.

7.1 Contribution and Summary of Present Work

The main contribution of this work is the design of a persistent object system that utilizes parallelism in a fundamental way to achieve competitive performance, and the techniques used in its implementation. Parallelism is incorporated into the system at all levels. We began with a declarative, implicitly parallel transaction language that includes a full higher-order programming language and list comprehensions, a notation that can be used as a high-level, declarative query language. We believe that such declarative languages, from which massive parallelism can be extracted easily, are essential to achieving scalable performance for complex transactions on large-scale parallel machines.

A novel aspect of the language is its non-destructive, single-assignment update of persistent objects. This permits even update transactions to be executed with a high degree of parallelism, and ensures a unique result, *i.e.*, transactions are determinate. Previous work on rewrite rule systems was extended and a formal semantics for the language specified.

AGNA transactions are compiled in three major phases— source-to-source translation of the original transaction text, translation into dataflow program graphs, and translation into code for a multi-threaded abstract machine called P-RISC, whose central feature is fine grain parallelism with data-driven execution. Coarse grain parallelism is used to distribute computations of a transaction over the nodes of a parallel MIMD machine, and fine grain parallelism is used within a node to mask long-latency operations such as disk I/O, remote memory accesses, and waits

for synchronization.

A number of significant optimizations are performed on list comprehensions, including use of Btree and hash indexes to combine generators and filters, reduction of the number of intermediate lists, and moving filters as close to disk I/O as possible. An additional, significant optimization performed by the compiler enables tail-recursive procedures to execute in a resource-efficient manner. Included in the abstract machine is a virtual heap that encompasses both persistent and ephemeral objects. A novel segmented, paged structure permits clustering of objects of a particular type, and easy mapping to the distributed memory and disks of a multiprocessor.

A prototype implementation was developed on workstations running the UNIX operating system, interconnected via a local area network. A port of the software is also operational on an Intel iPSC/2 Hypercube with thirty-two processors and thirty-two disks. To our knowledge, AGNA is the first real implementation of a *parallel* persistent object system, whether based on functional languages or not. Analysis and experimental results demonstrate:

- Performance approaches that of state of the art uniprocessor and multiprocessor relational systems on a simple but important class of queries. Performance relies heavily on optimization of list comprehensions, and aggressive pursuit of parallelism, based on fine grain, non-strict evaluation.

- Performance is superior to uniprocessor relational systems, and approaches that of state of the art object-oriented systems, on more complicated queries such as those involving graph traversal operations.

- Exploitation of parallelism on both uniprocessor and multiprocessor platforms. On a uniprocessor platform, fine grain parallelism is used to mitigate the effects of I/O latency by executing other threads while I/O is in progress. On a multiprocessor platform, parallel execution of both computation and I/O is used to achieve scalable performance of simple list comprehension queries.

7.2 Comparison With Related Work

In Chapter 1, we compared the approach taken in AGNA to those taken by the developers of other third-generation database systems. Here we include additional comparisons with specific

systems.

7.2.1 Functional Data Model

Some of the earliest connections between functional languages and database systems were made by Shipman in his work on the Functional Data Model [70], Buneman *et al.* in their work on FQL (Functional Query Language) [19] and Atkinson, Kulkarni *et al.* in their work on PS-Algol and the Extended Functional Data Model [7, 52]. Most of this work emphasized language design and all the implementations were sequential.

7.2.2 Trinder's Functional Database Model

Trinder, in his thesis [77], has also studied the use of a non-strict functional language as a parallel database language. He examined parallelism in transactions by executing them on a simulator (running on a sequential machine) that models parallel execution and disk I/O in certain idealized ways. His results confirm our belief (a belief long held by our fellow dataflow researchers and substantiated by numerous experiments in dataflow architectures [3]) that non-strict parallel evaluation is a very promising method to exploit parallelism and to overcome the long latencies of disk I/O and communication in parallel machines.

Trinder and his colleagues are currently implementing his functional database model on GRIP, a parallel machine [66]. One difference from our work is that since GRIP does not have a parallel I/O system, the database will have to reside in main memory. In AGNA, we have addressed the issue of implementing a virtual heap that is much larger than main memory and of distributing objects onto multiple disks. A second difference is that GRIP is a shared memory architecture; we believe that our distributed memory model is not only physically easier to scale to much larger machines, but also operationally, because we have taken into account the increased latencies of larger machines.

7.2.3 AGM

AGM (Active Graph Model) [15], developed at UC Irvine, was a parallel database system that used a language based on the Entity-Relationship data model. Memory was viewed as an active graph of tokens, which are used to represent both entities and relationships. Tokens are mapped to PMEs of the machine by a hashing function, and transaction processing consists of injecting and propagating special query and/or update tokens through the graph.

7.2.4 SPL

Kato *et al.* [48] have designed a database language SPL based on list comprehensions. However, their approach seems quite different from ours. They have a compilation scheme where each generator and filter in the list comprehension is treated as a function from streams to streams. Each function is treated as a sequential process in a parallel system, and the streams are directly implemented as communication channels between these processes. With these restrictions, they have not had to deal with persistent heaps, and they do not utilize indexes for efficient access. Some problems that we see with this approach are that streams themselves are not first class objects, and it is difficult to deal with higher-order functions and updates.

7.2.5 Gamma

Gamma is a parallel relational database system developed at the University of Wisconsin. It exploits intra-transaction parallelism to achieve roughly linear speedup for single-user execution of relational queries as the hardware is increased from 1 to 30 nodes, *i.e.*, for a constant-sized database, doubling the number of PMEs roughly halves the response time [33].

Gamma uses a "shared-nothing" [71] memory organization in which PMEs do not share main memory or disks, and communicate solely via message-passing. Relations are partitioned horizontally across all PMEs. A data-driven, coarse-grained model of parallel computation is utilized. For example, a relational query is compiled to a dataflow graph containing operators such as *select*, *project*, and *join*. An operator is implemented via one or more processes (or process threads) running on each PME which participates in the operator. For relational queries, the query processing strategy to be used and the process-to-PME mapping are determined largely at compile-time.

In contrast to the Gamma implementation, we use a much finer parallel grain size and a memory model consisting of a global heap and procedure frames (or activation records). Another major difference is the way in which parallel tasks are mapped to PMEs. In AGNA, tasks are often mapped to PMEs at runtime, based primarily on considerations of load balancing and object location. For an SQL-like subset of our language, however, the compiler is often able to select a query processing strategy which provides hints on where certain computations should be performed, thereby exploiting the locality of data. While such methods work well for relatively simple operations on regularly structured data, it is not clear that they can be easily

generalized to more sophisticated operations on complex data.

The final significant difference is the way in which parallel tasks are synchronized and controlled. In Gamma, a separate control process is used to inform an operator process of the identity of the processes to (from) which it is to send (receive) data. When the processes which implement an operator complete, they inform the control process, which in turn may initiate other operator processes. Consumer processes block while waiting for data from producer processes. In our approach, synchronization is performed explicitly via the `join` instruction (which combines parallel threads) or implicitly through heap access (readers of an empty location are automatically deferred until the location is written). These mechanisms make minimal use of OS synchronization primitives and are entirely data driven, involving no busy-waiting. Control is handled uniformly through the use of continuations, which can be thought of as the ultimate in lightweight threads. For operating systems which support non-blocking I/O, *all* threads on a PME, possibly from different transactions, may execute within the same OS process.

7.2.6 FAD and Bubba

FAD, designed at MCC [10], was another parallel functional database language. FAD did not have anything like list comprehensions; queries had to be composed explicitly using operations such as map and filter. In FAD, functions were not first class objects, and updates were completely imperative which, as discussed in Chapter 1, constrains parallelism significantly. FAD was to be implemented on Bubba, a parallel database machine [18] built on top of a Flex/32 multicomputer consisting of 40 PMEs (32 with local disks). We believe that only a relational subset of FAD (no inter-object references) was actually implemented before the end of the project; we do not know what optimizations were implemented or what performance was achieved.

7.3 Directions for Future Work

While we are pleased with the results and progress of the present work, many exciting opportunities exist for future work. In this section we outline some of these opportunities.

7.3.1 Language

In AGNA, we have obviously not spent any time on concrete syntax design— with its heavily parenthesized notation, the transaction language is by no means user-friendly. A more concise and elegant syntax, such as that used in [63], is clearly desirable. Further, AGNA does not have a static type system, something that is also desirable.

7.3.2 Concurrency Control and Failure Recovery

We have not yet addressed in AGNA the important issues of concurrency control (across transactions) and failure recovery, though we believe that conventional techniques such as two-phase locking and write-ahead logging could be applied easily in AGNA.

7.3.3 Retention of Historical Data

Perhaps a more interesting approach to recovery would be to extend AGNA's database model to include historical data, which is straightforward because of its non-destructive update model, and then to fold recovery into normal management of the database. Instead of modeling a database as a single environment of bindings, we can model it as a sequence of environments, each produced by an update transaction. Because old data is never overwritten, a conventional log is not necessary, and recovery is simplified greatly. Also, the entire history of the database is available to all transactions. This means that the programmer, or database administrator, does not have to decide a priori which object histories to maintain, since needed historical data will always be available.

If old database environments are to be accessible to the programmer, then the transaction language must be extended to include a way to refer to environments, and a mechanism for specifying the environment in which an expression is to be evaluated. As a practical matter, strategies will need to be developed for archiving and/or pruning old environments, since it may not be feasible to maintain on-line all old versions of the database.

7.3.4 Compiler

An obvious area of future work here is compilation to native machine code. Also, many additional optimizations are possible on list comprehensions. One area for improvement is comprehensions that include nested generators, the equivalent of join queries (cross-products) in

relational systems. We believe that traditional methods for implementing join queries, as well as new ones that can utilize the direct object references part of the AGNA object model, can be exploited. Finally, enhancements to the compiler and/or run-time system aimed at reducing the overhead of heap address translations, are desirable.

7.3.5 Resource Management

Much additional work is possible in the difficult area of resource management: automatic garbage-collection of both persistent and volatile parts of the heap; additional strategies for distribution of objects to PMEs; code-mapping policies; support for large data objects; frame management; coding of resource managers, including the object storage system, in a parallel language.

7.3.6 Analysis and Experimentation

Finally, more analysis and experimentation are needed to further assess the strengths and weaknesses of AGNA. One obvious area to be examined is the ability of AGNA to exploit parallelism between transactions. In the present work we have focused on parallelism within a single transaction, but parallelism may also be used in AGNA to increase total transaction throughput.

Appendix A

Syntax of the Agna Transaction Language

This appendix gives the syntax of the complete AGNA transaction language. Standard BNF notation is used.

Transaction	::=	(xact Statement+)
Statement	::=	Definition | Expression
Definition	::=	(define Identifier Expression)
	::=	(define-local Identifier Expression)
		(undefine Identifier)
		(type Identifier ([extent]) (Field-Spec*))
Field-Spec	::=	(Identifier {=>|<=>|=>*|<=>*|*<=>|*<=>*} Identifier)
Expression	::=	Constant | Identifier | (Expression+)
		(Primitive Expression*)
		(if Expression Expression Expression)
		(let ((Identifier Expression)+) Expression+)
		(lambda (Identifier*) Expression)
		(allocate Identifier)
		(invert Identifier Identifier Expression)
		(select Expression Identifier Identifier)
		(update Expression Identifier Identifier Expression)
		(insert[-list] Expression Identifier Identifier Expression)
		(delete[-list] Expression Identifier Identifier Expression)
		(all Identifier)
		(all Expression Qualification+)
Qualification	::=	(Identifier Expression)
		(where Expression+)
Primitive	::=	+ | - | drop | ...
Constant	::=	Boolean | String | Number | nil | ()

Appendix B

P-RISC Managers

This appendix describes the complete set of P-RISC manager instructions used in AGNA.

B.1 Heap Memory Allocation

ALLOCOBJ allocates and initializes a new object in the volatile heap:

Syntax	*Semantics*
ALLOCOBJ ri rj	Let T = Frames[FP+ri] Let S = Frames[FP+ri+1] Allocate and initialize object in volatile heap of type T and size S Let A be the address of this object Frames[FP+rj] <- A Add (FP,IP+1) to active pool

Object initialization performed includes defining the object header and setting to "empty" all field status bits.

ALLOCFRAME allocates and initializes a new frame.

Syntax	*Semantics*
ALLOCFRAME ri rj	Let CallerFP = Frames[FP+ri] Let ResultIP = Frames[FP+ri+1] Let SignalIP = Frames[FP+ri+2] Let ResSlot = Frames[FP+ri+3] Let NumSlots = Frames[FP+ri+4] Let FP' be address of new frame Zero-out frame slots in FP' Store in FP': CallerFP, ResultIP, SignalIP, and ResSlot Frames[FP+rj] <- FP' Add (FP,IP+1) to active pool

Arguments are: the caller's FP, the IPs of the threads to receive the result and signal, the slot where the result of the procedure is to be stored, and the size of the new frame. ALLOCFRAME allocates a frame of the desired size, initializes its slots to zero, stores the linkage information, returns a pointer to the new frame, and adds successor descriptor (FP,IP+1) to the active pool.

Allocation in the persistent heap is performed by manager MKPERSISTENT, which copies an object from the volatile to the persistent part of the heap.

Syntax	*Semantics*

MKPERSISTENT ri rj	Let Obj = Frames[FP+ri]
	Frames[FP+rj] <- Obj
	Frames[FP+rj+1] <- false
	If volatile?(Obj)
	  If alreadyMoved(Obj)
	  Frames[FP+rj] <- lookupPersistentAddr(Obj);
	  Else
	  LetA be address of persistent storage
	  copy(Obj,A);
	  Frames[FP+rj] <- A;
	  Frames[FP+rj+1] <- true;
	Add (FP,IP+1) to active pool

A reference to the object to be moved is passed in slot `ri` and the persistent address is returned in slot `rj`. An additional (boolean) result is returned in slot `rj+1` indicating whether the persistent address returned in `rj` was allocated by the current call to `MKPERSISTENT` (`true`) or a previous one (`false`). The first result is initialized to the object itself, and the second result to `false`. If the object has already been moved to the persistent heap (tested by predicate `alreadyMoved`), then the persistent address is looked up via `lookupPersistentAddr` and returned. Otherwise, storage is allocated in the persistent heap, the object is copied, and the new persistent address and `true` are returned.

`DEALLOCATE` frees the heap storage occupied by an object.

Syntax	*Semantics*
DEALLOCATE ri rj	Let Obj = Frames[FP+ri] Let Len = length(Obj) Add storage of length Len at address Obj to free list Add (FP,IP+1) to active pool

B.2 Object Manipulation

Selection of field values is performed via manager `SELECTF`.

Syntax	*Semantics*

SELECTF ri rj	Let Obj = Frames[FP+ri]
	Let ObjType = Frames[FP+ri+1]
	Let Offset = Frames[FP+ri+2]
	Error checking:
	  1. Type check
	  2. Bounds check
	  3. Obj persistent and field undefined?
	If error found
	  raise error
	Frames[FP+rj] <- Obj + Offset
	ILOAD rj rj

Arguments to `SELECTF` are the object, its type, and the field offset; the result returned is the field value. `SELECTF` first checks for three error conditions: (1) an object not of the correct type; (2) a field reference that is out of bounds (this condition may hold only if `Obj` is an array because this is the only offset that is computed and not supplied by the compiler); and (3) a persistent

object with field at `Offset` that is undefined. If one of these conditions is found to hold, then a run-time error is raised. Otherwise, the field address is built in slot `rj` and `ILOAD` is used to access the field. If the field is not yet defined, then `ILOAD` will defer the operation.

Update of field values is performed via manager `UPDATEF`.

Syntax	*Semantics*

`UPDATEF ri rj`	Let `Obj = Frames[FP+ri]`
	Let `ObjType = Frames[FP+ri+1]`
	Let `Offset = Frames[FP+ri+2]`
	Let `Value = Frames[FP+ri+3]`
	Error checking:
	1. Type check
	2. Bounds check
	If error found
	raise error
	`Frames[FP+rj] <- ()`
	If unique inverse exists on field
	Add `(ObjType,Offset,Value)` to *constraints* collection
	If `Obj` is ephemeral
	`Frames[rx] <- Obj + Offset;`
	`Frames[ry] <- Value;`
	`ISTORE ry rx;`
	Else
	Add `(Obj,Offset,Value)` to *updates* collection
	Add `(FP,IP+1)` to active pool

Arguments to `UPDATEF` are the object, its type, the field offset, and the field value. If a unique inverse exists on the field, then a triple consisting of the object type, field offset, and field value is added to collection *constraints*. If the object is ephemeral, then two internal frame slots `rx` and `ry` are loaded with the field address and value, respectively, and `ISTORE` is used to define the field. If the object is persistent, then a triple consisting of the object, the field offset, and the field value is placed in the *updates* collection, and the successor descriptor is added to the pool of active threads.

Insertion of values into field collections is performed via manager `INSERTF`.

Syntax	*Semantics*
`INSERTF ri rj`	Let `Obj = Frames[FP+ri]` Let `ObjType = Frames[FP+ri+1]` Let `Offset = Frames[FP+ri+2]` Let `Value = Frames[FP+ri+3]` Error checking: 1. Type check 2. Bounds check 3. Object ephemeral? If error found raise error `Frames[FP+rj] <- ()` Add `(Obj,Offset,Value)` to *inserts* collection Add `(FP,IP+1)` to active pool

Arguments to `INSERTF` are the object, its type, the field offset, and the new field element. If no errors are found, then a triple consisting of the object, the field offset, and the field element

is placed in the *inserts* collection, and the successor descriptor is added to the pool of active threads.

Deletion of values from field collections is performed via manager DELETEF.

Syntax	*Semantics*
DELETEF ri rj	Let Obj = Frames[FP+ri] Let ObjType = Frames[FP+ri+1] Let Offset = Frames[FP+ri+2] Let Value = Frames[FP+ri+3] Error checking: 1. Type check 2. Bounds check 3. Object ephemeral? If error found raise error Frames[FP+rj] <- () Add (Obj,Offset,Value) to *deletes* collection Add (FP,IP+1) to active pool

Arguments to DELETEF are the object, its type, the field offset, and the field element to be deleted. If no errors are found, then a triple consisting of the object, the field offset, and the field element is placed in the *deletes* collection, and the successor descriptor is added to the pool of active threads.

The actual addition of an element to a field collection during the transaction epilogue is performed via manager ADDMVFIELDELEMENT.

Syntax	*Semantics*
ADDMVFIELDELEMENT ri rj	Let Obj = Frames[FP+ri] Let Offset = Frames[FP+ri+1] Let Value = Frames[FP+ri+2] Frames[FP+rj] <- () Insert Value into field collection Add (FP,IP+1) to active pool

Arguments to ADDMVFIELDELEMENT are the object, field offset, and field element to be added. The new element is inserted into the field collection, and all appropriate indexes are updated. A successor descriptor is added to the pool of active threads.

The actual deletion of an element from a field collection during the epilogue is performed via DELETEMVFIELDELEMENT.

Syntax	*Semantics*
DELETEMVFIELDELEMENT ri rj	Let Obj = Frames[FP+ri] Let Offset = Frames[FP+ri+1] Let Value = Frames[FP+ri+2] Frames[FP+rj] <- () Delete Value from field collection Add (FP,IP+1) to active pool

Arguments to DELETEMVFIELDELEMENT are the object, field offset, and field element to be deleted. The element is removed from the field collection, and all appropriate indexes are updated. A successor descriptor is added to the pool of active threads.

Finally, new values of single-valued fields are installed in persistent objects during the epilogue via INSTALLFIELDVALUE.

Syntax	*Semantics*
INSTALLFIELDVALUE ri rj	Let Obj = Frames[FP+ri] Let Offset = Frames[FP+ri+1] Let Value = Frames[FP+ri+2] Frames[FP+rj] <- () Frames[rx] <- Obj + Offset; Frames[ry] <- Value; STORE ry rx; If index exists on field update index Add (FP,IP+1) to active pool

Arguments to INSTALLFIELDVALUE are the object, field offset, and field value. The field value is written via STORE, and the field index is updated, if one exists. A successor descriptor is added to the pool of active threads.

B.3 Associative Searches

Lookup of top-level identifiers is performed via LOOKUP.

Syntax	*Semantics*
LOOKUP ri rj	Let Name = Frames[FP+ri] Use index on name field in binding objects to locate object with desired name If object with name not found raise error Frames[rj] <- value bound to name; Add (FP,IP+1) to active pool

The collection of binding objects in the database is searched (via the index on name) for the object with the desired name. If it is not found, then an error is raised. Otherwise, the value bound to the name is returned, and a successor descriptor is added to the active pool.

Single-valued inverse field-mappings are implemented via SVINVERT.

Syntax	*Semantics*

SVINVERT ri rj	Let ObjType = Frames[FP+ri]
	Let Offset = Frames[FP+ri+1]
	Let Value = Frames[FP+ri+2]
	Use index on field to search for object with field value
	If desired object found
	  Frames[rj] <- object;
	Else
	  Frames[rj] <- the-null-object;
	Add (FP,IP+1) to active pool

The extent of objects of type ObjType is searched (via the field index) for the object with the desired field value. If it is found, then it is returned as the result, otherwise the null object is returned. In both cases, a successor descriptor is added to the active pool.

Multi-valued inverse field-mappings are implemented via MVINVERT.

Syntax	*Semantics*
MVINVERT ri rj	Let ObjType = Frames[FP+ri] Let Offset = Frames[FP+ri+1] Let Value = Frames[FP+ri+2] Use index on field to search for objects with field value Frames[rj] <- list of objects found; Add (FP,IP+1) to active pool

The extent of objects of type ObjType is searched (via the field index) for the objects with the desired field value. A list of the objects found is returned, and a successor descriptor is added to the active pool.

Searches of type extents are performed via FILTEREXTENT.

Syntax	*Semantics*
FILTEREXTENT ri rj	Let ObjType = Frames[FP+ri] Let ObjField = Frames[FP+ri+1] Let AccessPath = Frames[FP+ri+2] Let Index = Frames[FP+ri+3] Let Predicate = Frames[FP+ri+4] Search type extent, using indicated access path, for objects that satisfy the predicate Frames[rj] <- list of objects (projected onto ObjField); Add (FP,IP+1) to active pool

Arguments are the type of the extent to search, the field onto which the result is to be projected, the access path and index, and the predicate. A result list is returned, and a successor descriptor is added to the active pool.

B.4 Miscellaneous

An object is printed via manager PRINT.

Syntax	*Semantics*
PRINT ri rj	Let Obj = Frames[FP+ri] Print Obj Frames[rj] <- (); Add (FP,IP+1) to active pool

A top-level identifier binding is entered into the identifier cache via CACHEIDVALUE.

Syntax	*Semantics*
CACHEIDVALUE ri rj	Let Name = Frames[FP+ri] Let Value = Frames[FP+ri+1] Add (Name,Value) to cache Frames[rj] <- (); Add (FP,IP+1) to active pool

Managers ADDSLIST, DROPSLIST, INSERTSLIST, DELETESLIST, and CONSTRAINTSLIST return the lists of updates and constraints collected during execution of the transaction body. Manager ADDSLIST is given below; the others are similar.

Syntax	*Semantics*
ADDSLIST ri rj	Frames[rj] <- addsList; Add (FP,IP+1) to active pool

Two additional managers, OBJECTMGR and LOCALIZE, aren't part of the P-RISC machine per se, but are used in the AGENA implementation. OBJECTMGR returns the number of the PME on which an object is stored, and LOCALIZE makes a local copy of an object on a specified PME.

Bibliography

[1] Anon et al. A Measure of Transaction Processing Power. *Datamation*, 31(7):112–118, April 1985.

[2] Z. Ariola and Arvind. Contextual rewriting. Technical Report CSG Memo 323, MIT Laboratory for Computer Science, 1991.

[3] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. *International Journal of Supercomputer Applications*, 2(3), 1988.

[4] Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, Springer-Verlag LNCS 295*, June 25-29 1987.

[5] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1989.

[6] M. Astrahan et al. System R: Relational Approach to Database Management. *ACM Trans. on Database Systems*, 1(2), June 1976.

[7] M. P. Atkinson, K. Chisholm, and W. Cockshott. PS-Algol: An Algol with a Persistent Heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1981.

[8] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[9] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data- and demand-driven interpretation of imperative languages. In *Proc. SIGPLAN '90 Conf. on Programming Language Design and Implementation*, July 1990.

[10] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a Powerful and Simple Database Language. In *Proc. 13th. Intl. Conf. on Very Large Databases, Brighton, England*, pages 97–105, September 1-4 1987.

[11] P. S. Barth and R. S. Nikhil. Supporting state-sensitive computation in a dataflow system. Technical Report CSG Memo 294, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, March 1989. Describes how to deal with state (updatable objects) in a fine grain parallel system.

[12] C. Baru, O. Frieder, D. Kandlur, and M. Segal. Join on a cube: Analysis, simulation, and implementation. In *Proceedings of the 5th International Workshop on Database Machines*, 1987.

[13] D. Batory, T. Leung, and T. Wise. Implementation Concepts for an Extensible Data Model and Data Language. *ACM Transactions on Database Systems*, 13(3), Sept. 1988.

[14] M. Beck and K. K. Pingali. From control flow to dataflow. Technical Report TR89-1050, Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, USA, October 1989.

[15] L. Bic and R. L. Hartmann. AGM: A dataflow database machine. *ACM Transactions on Database Systems*, 14(1):114–146, March 1989.

[16] A. D. Birrell. An introduction to progamming with threads. Technical report, DEC Systems Research Center, 130 Lytton Ave., Palo Alto CA 94301, January 1989.

[17] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems: A Systematic Approach. In *Proceedings of the 1983 Conference on Very Large Data Bases*, August 1983.

[18] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[19] O. P. Buneman, R. E. Frankel, and R. S. Nikhil. An Implementation Technique For Database Query Languages. *ACM Transactions on Database Systems*, 7(2):164–186, June 1982.

[20] W. H. Burge. *Recursive Programming Techniques.* Addison Wesley, Reading, MA, 1975.

[21] M. J. Carey, D. J. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases.* Morgan Kaufmann, San Mateo, Calif., 1990.

[22] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 1986 Conference on Very Large Data Bases*, Kyoto, Japan, August 1986.

[23] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of the 1988 ACM SIGMOD Conference*, Chicago, June 1988.

[24] R. Cattell and J. Skeen. Engineering database benchmark. Technical report, Database Engineering Group, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, April 1990.

[25] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):76–90, March 1990.

[26] H.-T. Chou, D. J. DeWitt, R. H. Katz, and A. C. Klug. Design and Implementation of the Wisconsin Storage System. *Software—Practice and Experience*, 15(10):943–962, October 1985.

[27] P. Close. The iPSC/2 Node Architecture. In *Proceedings of Third Hypercube Conference (ACM)*, 1988.

[28] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proceedings of the 1984 ACM SIGMOD Conference*, Boston, MA, 1984.

[29] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[30] C. J. Date. *An Introduction to Database Systems, Volume II.* Addison-Wesley, Reading, Massachusetts, 1984.

[31] C. J. Date. *A Guide to the SQL Standard.* Addison-Wesley, Reading, Massachusetts, 1987.

[32] A. Dearle, G. M. Shaw, and S. B. Zdonik, editors. *Proc. of the Fourth International Workshop on Persistent Object Systems.* Morgan Kaufmann, September 1990.

[33] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[34] K. Dittrich, W. Gotthard, and P. Lockemann. DAMOKLES—The Database System for the UNIBASE Software Engineering Environment. *IEEE Database Engineering*, 10(1), 1987.

[35] S. Englert, J. Gray, T. Kocher, and P. Shah. A benchmark of NonStop SQL Release 2 demonstrating near-linear speedup and scaleup on large databases. Technical Report Tech. Report 89.4, Tandem Part No. 27469, Tandem Computers, May 1989.

[36] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[37] Object Databases Corp. GBase Technical Summary, 1990.

[38] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proceedings of the 1990 ACM SIGMOD Conference*, Atlantic City, NJ, May 1990.

[39] S. Graham, P. Kessler, and M. McKusick. gprof: A Call Graph Execution Profiler. *Proc. of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, 17(6):120–126, 1982.

[40] M. D. Guzzi, W. L. Harrison, III, and D. A. Padua. Programming languages for parallel processors. Technical Report CSRD report no. 623, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 104 S. Wright Street, Urbana, Illinois 61801, January 22 1987.

[41] W. L. Harrison III. *The Interprocedural Analysis and Automatic Parallelization of Scheme Programs.* PhD thesis, University of Illinois, Urbana-Champaign, Illinois, March 1989.

[42] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[43] M. L. Heytens and R. S. Nikhil. GESTALT: An Expressive Database Programming System. *ACM SIGMOD Record*, 18(1), March 1989.

[44] R. A. Iannucci. *A Dataflow/von Neumann Hybrid Architecture.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1988.

[45] Itasca Systems, Inc. ITASCA System Overview, 1990.

[46] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings Functional Languages and Computer Architecture*, Nancy, France, September 1985.

[47] S. L. P. Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, London, 1987.

[48] K. Kato, T. Masuda, and Y. Kiyoki. A Comprehension-Based Database Language and its Distributed Execution. In *Proc. 10th Intl. Conf. on Distributed Computing Systems, Paris, France*, pages 442–449, May 28-June 1 1990.

[49] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[50] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1), 1983.

[51] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[52] K. G. Kulkarni and M. P. Atkinson. Implementing Extended Functional Data Model Using PS-Algol. *Software: Practice & Experience*, 1986.

[53] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1990.

[54] J. R. Larus and P. N. Hilfinger. Restructuring Lisp Programs for Concurrent Execution. In *Proc. ACM/SIGPLAN PPEALS (Parallel Programming: Experience with Applications, Languages and Systems, New Haven, Connecticut*, pages 100–110, July 19-21 1988. (ACM Sigplan Notices 23:9).

[55] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to STARBURST: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34(10):94–109, October 1990.

[56] R. S. Nikhil. Functional languages, functional databases. In *Proceedings of the Workshop on Persistence and Data Types, Appin, Scotland*, Appin, Scotland, August 1985.

[57] R. S. Nikhil. The semantics of update in a functional database programming language. In *Proceedings of the ALTAIR-CRAI Workshop on Database Programming Languages, Roscoff, France*, Roscoff, France, September 1987.

[58] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[59] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Workshop on Massive Parallelism, Amalfi, Italy, October 1989.* Academic Press, 1990. Also: CSG Memo 313, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

[60] R. S. Nikhil and M. L. Heytens. Exploiting Parallelism in the Implementation of AGNA, a Persistent Programming System. In *Proc. of 7th IEEE Intl. Conf. on Data Engineering*, Kobe, Japan, April 8-12 1991.

[61] O. Deux *et al.* The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1990.

[62] Objectivity, Inc. Objectivity Database System Overview, 1990.

[63] A. Ohori, O. P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli— a Polymorphic Language with Static Type Inference. In *Proc. Intl. Conf. on the Management of Data, Portland, OR*, pages 46–57, June 1989.

[64] Ontologic, Inc. ONTOS Reference Manual, 1990.

[65] S. Park and K. Miller. Random number generators: Good ones are hard to find. *CACM*, 31(10), October 1988.

[66] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – A High Performance Architecture for Parallel Graph Reduction. In *Proc. 3rd. Intl. Conf. on Functional Programming and Computer Architecture, Portland, OR*, September 1987.

[67] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient languages. In *Proc. 5th Intl. Conf. on Functional Programming and Computer Architecture, Cambridge, MA*, August 1991.

[68] D. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the 1989 ACM SIGMOD Conference*, Portland, OR, June 1989.

[69] M. Sharma and R. S. Nikhil. PRISC-1: A Multi-threaded RISC Architecture. Technical Report (unpublished), MIT Laboratory for Computer Science, November 1989.

[70] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, March 1981.

[71] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[72] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The Design of XPRS. In *Proceedings of the 1988 Conference on Very Largbe Data Bases*, Los Angeles, CA, August 1988.

[73] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–93, October 1990.

[74] Tandem Performance Group. A Benchmark of Non-Stop SQL on the Debit Credit Transaction. In *Proceedings of the 1988 ACM SIGMOD Conference*, Chicago, IL, June 1988. ACM.

[75] Dbc/1012 database computer system manual release 2.0. Technical Report Document C10-0001-02, Teradata Corp., November 1985.

[76] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Master's thesis, Massachusetts Institute of Technology, 1986. Technical Report LCS TR-370.

[77] P. Trinder. A functional database. Oxford University D.Phil. Thesis, December 1989.

[78] D. A. Turner. The semantic elegance of applicative languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85–92, October 1981.

[79] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, Maryland, 1982.

[80] Versant Object Technologies, Inc. VERSANT Technical Overview, 1990.

[81] G. von Bultzingsloewen. Translating and Optimising SQL Queries Having Aggregates. In *Proceedings of the 13th Intl. Conference on Very Large Databases*, Brighton, England, September 1987.

[82] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

[83] N. Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, 1982.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Feb. 1992	

4. TITLE AND SUBTITLE

The Design and Implementation of A Parallel Persistent Object System

5. FUNDING NUMBERS

6. AUTHOR(S)

Michael L. Heytens

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

MIT/LCS/TR 529

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

N00014-89-J-1988

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT (A

It is widely recognized that the expressive power of relational database systems is inadequate for applications that manipulate complex, non record-oriented data. Much recent research has been focused on the design of more expressive database language models that seamlessly integrate the data modeling, abstraction, and general computation of full programming languages with the features of traditional database systems such as persistence, failure recovery, and security. Such additional flexibility gives expressive power to the programmer, but complicates matters for the compiler and run-time system in their efforts to implement database programs efficiently.

In this report we describe AGNA, an experimental persistent object system that we have designed and built that utilizes parallelism in a fundamental way to enhance performance. Parallelism is incorporated into the design of the system at all levels. We begin with an implicitly parallel transaction language that includes a full higher-order programming language and the "list comprehension," a notation similar to SQL but more general. Transactions are compiled into code for a multi-threaded abstract machine called P-RISC, whose central feature is fine grain parallelism with data-driven execution. P-RISC code is emulated on each processor of a MIMD machine with multiple disks. Coarse grain parallelism is used to distribute computations of a transaction over the nodes of a parallel machine, and fine grain parallelism is used within a node to overlap useful computation with long-latency operations such as disk I/O and remote memory accesses.

(con't.)

14. SUBJECT TERMS	15. NUMBER OF PAGES
persistent objects, functional languages, multi-threaded, object oriented databases, parallel databases systems	234
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

A prototype of AGNA is operational, running on both a network of workstations and an Intel iPSC/2 Hypercube with thirty-two processors and thirty-two disks. Experimental results demonstrate that parallelism is exploited on both uniprocessor and multiprocessor platforms. Performance of AGNA approaches that of state of the art relational and object-oriented database systems, and relies heavily on *compiler* optimisations and aggressive pursuit of parallelism.