Automatic Partitioning and Scheduling on a Network of Personal Computers

David A. Hornig
November 1984

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University
Automatic Partitioning and
Scheduling on a
Network of Personal Computers

David A. Hornig
9 December 1984

Carnegie-Mellon University
Computer Science Department

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Copyright © 1984 David A. Hornig


This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.
Automatic Partitioning and Scheduling on a Network of Personal Computers

David A. Hornig
November 1984

Carnegie Mellon University
Computer Science Department

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Copyright 1984 David A. Hornig
ACKNOWLEDGEMENTS

I am very grateful to my advisors Rick Rashid and Raj Reddy for their help with everything from my first tentative ideas to my last colored viewgraph. The experimental results in this thesis were made possible by Rick's Accent operating system. I thank my committee members Bill Scherlis and Bert Halstead for their comments and suggestions, and particularly for their careful and critical reading.

Thanks to Bob Fitzgerald for explaining Accent subtleties, believing my bug reports, and cofounding AARG; Jon Webb for helping me with the mysteries of microcode and suffering with me through the early months of Accent; Larry Rudolph for the molecular modelling example and for encouragement when I needed it most; Robert Sansom for helpful discussions and the code that gathered the message passing statistics; and Mike Kazar for a stack of references and comments on the final draft. Jon Webb, Stephen Brookes, Scott Fahlman, Roger Dannenberg, Al Spector, and Mike Kazar gave me very helpful comments on my proposal. Robin Ruppel is responsible for the impressive calligraphy on the title page.

My family, all of them, were always there to lean on and renew my spirits.

Finally, my most special thanks to Carol Hammer for her friendship and support throughout the last six years. You were there when I needed you.
ABSTRACT

This dissertation investigates automatic partitioning, scheduling, and fault tolerance for distributed applications on a network of personal computers. Previous distribution systems in loosely coupled environments have used a special process control language or special language features to describe the task structure of a program while the lower level algorithms were defined in a standard algorithmic language. Like many of the systems proposed for tightly coupled processors, we use a single applicative language for both the task structure and the algorithm, doing away with the distinction between tasks and functions by making every function a potential task. However, the network environment has deeply influenced the design: the long communication latencies commit the system to large grain parallelism and carefully planned communication strategies.

The product of the research is a language and execution environment called STARDUST. STARDUST programs are written in a general-purpose applicative language and marked with an estimate of each function's execution time. The system partitions the programs by expanding user function calls with high execution time estimates and by breaking up calls to the system's list operators. The segments are scheduled on the available processors, taking into account both load balancing and the costs of message passing. Failed tasks can be restarted by moving them to surviving processors.

Three sets of experiments run on Perq computers and an Ethernet gave two successes and one failure. The quick sort experiment failed to achieve significant speedups due to some poor scheduling heuristics and large amounts of overhead that could not be transferred to other processors. The six-processor versions of the signal processing and molecular modelling experiments showed speedups of about two when the exported function calls took one second to evaluate, and speedups of over three when the exported calls took five seconds. Message passing and interpreter overhead are the main reasons that such a large granularity is needed; partitioning and scheduling do not contribute significantly to overall execution time.

The experimental results also include a demonstration of automatic failure recovery and run-time redundant subcomputation elimination.
# Table of Contents

1. Introduction  
   1.1. Summary of Results 3  
   1.2. Outline of The Dissertation 3  

2. The Language  
   2.1. Description  
      2.1.1. Commands to the Interpreter 6  
      2.1.2. Type Declarations 6  
      2.1.3. Function Definitions 7  
      2.1.4. Infix Operators 8  
      2.1.5. Structure Manipulation Constructs 9  
      2.1.6. List Operators 10  
      2.1.7. Other Constructs 11  
   2.2. Sample Programs  
      2.2.1. Prime Numbers 12  
      2.2.2. Towers of Hanoi 12  
      2.2.3. Matrix Multiply 13  

3. The Implementation  
   3.1. Overview 15  
   3.2. Background  
      3.2.1. Overview of Termination Semantics 16  
      3.2.2. Overview of Parameter Passing 17  
   3.3. Representation of Expressions  
      3.3.1. Forms 18  
      3.3.2. Nodes 19  
      3.3.3. Handles 19  
      3.3.4. Discussion 19  
   3.4. Representation of Functions  
      3.4.1. Function Descriptors 21  
      3.4.2. Function Definitions 22  
      3.4.3. Parse Tree Representation 22  
      3.4.4. Discussion 22  
   3.5. Redundant Subcomputation Elimination  
      3.5.1. Implementation of RSCE 23  
      3.5.2. Discussion 25  
      3.5.3. Example 26  
      3.5.4. Importance of Redundant Subcomputation Elimination 27  
      3.5.5. Comparison With Other Forms of RSCE 29  
      3.5.6. Why RSCE at Run Time is Sometimes Superior to Static RSCE 31  
      3.5.7. Costs and Limitations 32
List of Figures

Figure 3-1: Execution in a Graph Reduction Language 15
Figure 3-2: Expansion in a Graph Reduction Language: \( \text{choose}(x,y) = \text{div}(\text{fac}(x),\text{fac}(x-y)\times\text{fac}(y)) \) 16
Figure 3-3: Three list storage mechanisms 21
Figure 3-4: Tree and DAG Representations of \((2\times(3 + 4)) + (3 + 4)\) 23
Figure 3-5: Representation of \(f(g(3))\) Using Only Forms (Circles) 25
Figure 3-6: Representation of \(f(g(3))\) Using Forms (Circles) and Nodes (Boxes) 26
Figure 3-7: \(g(3)\) is Replaced by 2 Using Only Forms (Circles) 26
Figure 3-8: \(g(3)\) is Replaced by 2 Using Forms (Circles) and Nodes (Boxes) 27
Figure 3-9: \(g(3)\) is Replaced by 2 in the Expression \(f(2) + f(g(3))\) 28
Figure 3-10: \(g(3)\) is Replaced by 2 in the Expression \(f(2) + f(g(3))\), When \(f(2)\) has Already Been Evaluated to Produce 7 29
Figure 3-11: Subactions Created by Complete and FComplete Actions 43
Figure 3-12: Subexpressions That Must be Complete (CA) or FComplete (FC) Before a Function can be Invoked 43
Figure 4-1: Star and Tree Configurations, with Controllers, Lieutenants, and Satellites 46
Figure 4-2: (a) Partial Time Estimates (PTE) (b) Total Time Estimates (TTE) 54
Figure 4-3: Schedules Produced for \(f(p(h,k),q(h,k))\) (a) by the Standard Scheduler and (b) by ABS 57
Figure 4-4: Actions created when the Controller requests Sat-2 to evaluate \(E\) and Sat-3 to evaluate \(f(E)\) 65
Figure 5-1: Flow of Information in the DSN Algorithm Using Three Microphones 74
Figure 5-2: Summary of DSN Experiments and Theoretical Speedups 77
Figure 5-3: DSN Evaluation Times 78
Figure 5-4: DSN Speedups 78
Figure 5-5: Categories of Processing Costs 79
Figure 5-6: Processor Usage for the Controller in Four DSN Experiments with Six Processors 80
Figure 5-7: Message Statistics for the Controller in Four DSN Experiments with Six Processors 80
Figure 5-8: Messages for the DSN Base Experiment with Six Processors 83
Figure 5-9: Messages for the DSN Base Experiment with Six Processors, Continued from Figure 5-8 84
Figure 5-10: Schedule for the DSN Base Experiment with Six Processors 84
Figure 5-11: Messages for the DSN Optimal Partitioning Experiment with Six Processors 85
Figure 5-12: Schedule for the DSN Optimal Partitioning Experiment with Six Processors 85
Figure 5-43: Overhead per Exported Computation in the Base Version of Three Experiments

Figure 5-44: Recovery from Overeager Evaluation

Figure 5-45: Execution Time for Recursive Fibonacci Program in Pascal and Stardust

Figure 5-46: DSN Schedule for a Six Processor Configuration, Base Experiment with Processor Failure

Figure 5-47: Messages for the DSN Base Experiment with Six Processors, of which Processor Number Four Fails

Figure 6-1: A Data Flow Example (Data Driven)

Figure 6-2: A Reduction Example (Demand Driven)

Figure 6-3: The Conditional in a Data Flow Machine: If x<3 Then f(x) Else g(x)

Figure 6-4: A Possible Representation for f(A[1])

Figure 6-5: A Simple Data Flow Loop Representation

Figure 6-6: Loop Representation in Id

Figure 6-7: Schematic of the Irvine Data Flow Machine: Processors and Routing Network

Figure 7-1: Expression Evaluation Overhead

Figure 7-2: Evaluation of 3 + 4 Using Forms (Circles) and Nodes (Boxes)

Figure 7-3: Evaluation of 3 + 4 Using Forms (Circles), Nodes (Boxes) and Short Forms (Unenclosed)

Figure 7-4: Evaluation of 3 + 4 Using Forms (Circles), Nodes (Boxes) Short Forms, and Short Nodes (Both Unenclosed)

Figure 7-5: Stack Discipline for Call

Figure 7-6: Stack Discipline for Return
Chapter 1
Introduction

Let us go, through certain half-deserted streets,
The muttering retreats
Of restless nights in one-night cheap hotels
And sawdust restaurants with oyster-shells:
Streets that follow like a tedious argument
Of insidious intent
To lead you to an overwhelming question ...

With the advent of local area networks, operating systems have begun to support inter-
processor communication at the user level. This support ranges from controlled access to
the raw device [Unix 82] to sophisticated message passing schemes embedded in the operat-
ing system kernel [Rashid 81].

Distributed applications are difficult to write with the communication primitives alone. Pro-
grammers who do so find themselves dealing with details of data formats, synchronization,
scheduling, and message passing costs that they would usually prefer to avoid. Subtle timing
phenomena can be a never-ending source of errors, particularly for applications attempting to
provide fault tolerance. The development of distributed applications has been somewhat
eased by the introduction of higher level mechanisms for distribution, synchronization, and
fault tolerance.

Programs are usually partitioned into concurrently executable tasks by means of special
language features. These features can be embedded in the programming language, as in
ADA [USDOD 83], *Mod [Cook 80], PL/1TS [Feldman 79], or Brinch Hansen's "distributed
processes" [Brinch-Hansen 78]. Alternatively they can be implemented as a separate process
control language such as DPL [Ericson 82] or PCL [Lesser 79]. Data flow languages such as
VAL [McGraw 82] and ID [Arvind 78] contain no explicit partitioning features but simplify
automatic partitioning by providing side effect free constructs.

Task distribution is the process of assigning tasks to processors, usually with the goals of
balancing the processing load and minimizing communication costs. Distribution
mechanisms exhibit considerable variety since their design depends on the homogeneity of the processors, the homogeneity of communications, the stability of the configuration, the amount of explicit user control desired, and the extent to which distribution decisions depend on partial results of the computation. The Xerox worm programs [Shoch 82] multiply and preserve themselves by duplicating their code on idle processors. Contract nets [Smith 80] provide dynamic load balancing by polling the available processors for idle resources. TASK [Schwans 82] provides automatic scheduling on a multiprocessor with heterogeneous communication by using a user-supplied estimate of proximity between pairs of data objects and between data objects and hardware.

Locking constructs control access to shared data. Constructs such as semaphores [Dijkstra 68] and monitors [Hoare 74], originally designed for single computer multiprocess systems, are also appropriate in distributed environments. Atomic transactions [Gray 80] ensure that collections of updates to a distributed data base either run to completion or abort without altering the data base, forming a basis for fault tolerance in a distributed system with shared data.

The goal of this research is to provide automatic partitioning, scheduling, and fault tolerance for distributed applications on loosely coupled processors. These activities are done dynamically so that the system can respond to changes in the environment, changing processing resources, and changing processing demands. An important part of the goal is to make distribution as automatic as possible ("distribution" will be used in this document as a general term for partitioning, scheduling, and providing fault tolerance). While hand tuning would probably give better performance, automatic distribution increases programming productivity and allows applications to be ported among widely varying hardware configurations. Loosely coupled systems were chosen because of the increasing prevalence of local area networks connecting processors of low cost and high performance.

The product of the research is a language and execution environment called STARDUST\(^1\). STARDUST programs are written in a general-purpose applicative language and marked with an estimate of each function’s execution time. The system partitions the programs by expanding user function calls with high execution time estimates and by breaking up calls to the system’s list operators. The segments are scheduled on the available processors, taking into account both load balancing and the costs of message passing. Tasks are restarted in the event of processor failure by moving them to other processors.

STARDUST provides nearly automatic partitioning, scheduling, and fault tolerance in

\(^1\)STARDUST is not an acronym
return for the restriction that programs be written without using side effects. While other recent attempts to design parallel execution systems for applicative languages [Keller 79, Mago 80, Arvind 82] are targeted for special purpose tightly coupled hardware, **STARDUST** is designed for an "unmassively" large collection of general purpose computers communicating over a local area network. This fact has deeply influenced the design: The long communication latencies brought about by the use of a local area network commit the system to large grain parallelism and carefully planned communication strategies.

Time constraints have forced us to forego implementation in a number of areas. These include mechanisms for greater control of the distribution process by the user; performance measurement tools; performance improvement through compilation, overwriting of structures, and list operator folding; garbage collection; automatic generation of execution time estimates; and flexible execution strategies that can trade off processing time and accuracy.

### 1.1. Summary of Results

This work demonstrates the following claims through argument and experiment:

- An applicative language with a small set of list operators is capable of neatly expressing many numerical computations. An even larger class of computations can be conveniently expressed if lower level routines are coded in an imperative language.

- The applicative programs can be automatically distributed at run time with a simple greedy algorithm. The only information that the user must supply is an execution time estimate for each of his functions; the estimate may depend on the parameters of the function.

- Fault tolerance in such a system can be achieved by simply restarting failed computations.

- The system can be implemented with fair performance on a network of general-purpose computers. Performance could be significantly improved through the use of compilation, microcoding, and specialized communication techniques.

### 1.2. Outline of The Dissertation

Chapter 2 describes an applicative language based on lists, recursion, and a small fixed set of list operators. Several simple programs are given in full.

Chapter 3 describes the implementation strategy on a single processor. Particular attention is given to lazy evaluation and to algorithms for detecting redundant subcomputations at run-time.
Chapter 4 describes the extension of the single processor evaluation strategy to a distributed environment, showing how automatic partitioning, scheduling, and failure recovery are provided.

Chapter 5 contains experimental results. Three test programs were run in a six-processor environment: a quick sort, a molecular modelling program, and a realistic signal processing algorithm.

Chapter 6 compares STARDUST to related work, particularly to data flow machines. The fundamentally different hardware of data flow machines (they are highly parallel, have low communication costs, and contain special purpose routing hardware) lead to a very different implementation strategy.

Chapter 7 describes performance and functionality improvements that would be necessary to bring STARDUST from a prototype to a production quality system. A detailed proposal for a compiler is given.

Chapter 8 presents some directions for future investigation.

Chapter 9 reviews the major components of STARDUST, their strengths and weaknesses, and their applicability to practical distributed processing.
Chapter 2
The Language

I know the voices dying with a dying fall
Beneath the music from a farther room.
So how should I presume?

STARDUST is a simple applicative language that borrows most of its constructs from LISP and APL and most of its syntax from PASCAL. The currently implemented version will be called STARDUST 1.0 when it is necessary to differentiate it from versions that might be written in the future. STARDUST comprises the following elements:

- Intrinsic and user-defined types including lists, records, and blocks. Lists and records have lazy semantics while blocks have strict semantics.
- User functions with typed arguments, call-by-need parameter binding, and lexical scoping.
- Standard arithmetic and boolean operators.
- Conditionals with and without parallel semantics.
- The list operators CAR, CDR, CONS, and CONCAT.
- Constructors for building lists and records from their components and selectors for extracting components from lists and records.
- Conversion between blocks and lists.
- A set of built-in list operators including MAP, which applies a unary function to every element of a list; REDUCE, which uses a left-associative binary operator and its identity to combine the elements of a list; PAIR, which applies a function to corresponding elements from two lists of equal length, creating a new list of the results; CROSS, which applies a function to all pairs of elements from any two lists; and SELECT, which forms a subset of a list from a predicate.
- A CHOICE construct, which expresses different methods of calculating a value, giving the system a chance to perform automatic tuning. The arguments to CHOICE should always evaluate to the same result, but this will not be enforced by the system. In STARDUST 1.0, CHOICE invokes all arguments and returns the first one to complete. This is meant as an interim solution only: all discussion in this
A special value called fail, which is the result of any illegal operation. The standard arithmetic and boolean operators return fail if any of their arguments are fail.

• Escapes to separately compiled code in other languages. These are called user intrinsics to distinguish them from the system intrinsics.

2.1. Description

A special language was designed chiefly as a matter of convenience: it was possible to include exactly those features that were needed to allow the test programs to be expressed. PASCAL syntax was imitated because it is easy to parse and convenient for expressing numerical applications.

The syntax is expressed in an extended Backus-Naur Form using the following special symbols and constructs:

- () Group symbols together
- <> Delimit tokens
- " Overrides special meaning of characters
- x|y Either x or y
- x? x is optional
- x* Zero or more repetitions of x
- {x # y}* Zero or more repetitions of x, separated by y

2.1.1. Commands to the Interpreter

<session> ::= { { <type-declaration> | <fn-definition> | <expression> } ";" }*

STARDUST is an interactive language. The user types a series of expressions, type declarations and function definitions to the interpreter, which responds by printing the values of expressions and updating its global bindings to agree with the declarations and definitions.

2.1.2. Type Declarations

<type-declaration> ::= TYPE <identifier> = <type>

<type> ::= <identifier> | <simple-type> | <structure-type>

<simple-type> ::= INTEGER | BOOLEAN | STRING

<structure-type> ::= <list-type> | <record-type> | <blk-type>

<list-type> ::= LIST OF <type>
<record-type> ::= RECORD { { <identifier> : <type> } # : }* END

<blk-type> ::= BLOCK OF <integer-constant> <type>

Examples:
Type T1 = Integer;
Type T2 = List Of T1;
Type T3 = Block Of 256 T1;
Type Complex = Record r: Integer; i: Integer End;

Three structured types are included in the language: lists, records, and blocks.

Lists and records are represented internally as arrays of pointers to other values (actually arrays of pointers to Nodes, as defined in Chapter 3.), and have lazy semantics: components are not evaluated until they are accessed. List length is associated with the value rather than with the type.

Block types are represented as arrays of values rather than arrays of pointers. They have a compact representation, strict semantics, and a length that is included in the type declaration. The only available operation on block types is conversion to list types and back. In a full implementation of STARDUST, blocks would have the same syntax and built-in functions as lists, only differing in performance and termination semantics.

Block types are included primarily for use by code written in other languages. For example, in the distributed sensor network computations (see Section 5.1), the signal data and cross correlation data are kept in block structures rather than in lists. Cross correlations and peak picking are done by PASCAL subroutines, while STARDUST treats each block as a simple constant to avoid the overhead of pointers, storage allocation and hashing.

2.1.3. Function Definitions

<fn-definition> ::= { LET | DEF } <identifier> <arglist> { : <type> }? = <fn-pragmas> { <expression> | FORWARD }

<fn-pragmas> ::= { { ATOMIC | EXPLODE } # , )* }?

<arglist> ::= { "(" { <identifier> : <type> } # ; )* ")" }?

Examples:
Let Answer = 42;

Let Fib(x:Integer):Integer = [Atomic]
If x<3 Then 1 Else Fib(x-1)+Fib(x-2);
Let \( \text{Sum}(x:\text{List Of Integer}):\text{Integer} = \) Begin
\( \text{Time } 40*\text{Length}(x); \)
Let \( \text{Temp} = \text{Car}(x); \)
\( \text{Temp}^\text{Temp} + \text{Sum}(\text{Cdr}(x)); \)
End;

Let \( \text{f}(x:\text{Integer}):\text{Integer} = \) Forward;

Let \( \text{g}(y:\text{Integer}):\text{Integer} = 2*\text{f}(y); \)

Let \( \text{f}(x:\text{Integer}):\text{Integer} = \text{g}(x); \)

Nested function definitions follow lexical scoping rules. Local variables such as \( \text{Temp} \) in the above example are simply nested functions with no arguments.

If the result type is omitted it will usually be inferred from the type of the expression used to define the function. Recursive definitions must specify the result type.

Function definitions may be augmented with either of the pragmas Atomic or Explode, providing rudimentary control over the partitioning algorithm. Functions declared Atomic are always scheduled as a unit; functions with the Explode pragma are divided up unless scheduled as part of a larger unit.

Function definitions should be augmented with an estimate of their execution time. This time estimate can be any legal STARDUST expression, and may refer to the arguments of the function. The time estimate is evaluated before the function is called, with the result that functions terminate only if their time estimates terminate.

2.1.4. Infix Operators

\[
\langle \text{expression} \rangle :: = \langle \text{infix0} \rangle \\
\langle \text{infix0} \rangle :: = \langle \text{infix1} \rangle | \langle \text{infix0} \rangle \{ \text{AND} \ | \text{OR} \} \langle \text{infix1} \rangle \\
\langle \text{infix1} \rangle :: = \langle \text{infix2} \rangle | \langle \text{infix1} \rangle \{ = \ | \text{<>} \ | \text{>} \ | \text{<} \} \langle \text{infix2} \rangle \\
\langle \text{infix2} \rangle :: = \langle \text{infix3} \rangle | \langle \text{infix2} \rangle \{ + \ | - \} \langle \text{infix3} \rangle \\
\langle \text{infix3} \rangle :: = \langle \text{item} \rangle | \langle \text{infix3} \rangle \{ \text{*} \ | \text{DIV} \} \langle \text{item} \rangle \\
\langle \text{item} \rangle :: = \langle \text{struct} \rangle | \langle \text{listop-call} \rangle | \langle \text{prim} \rangle
\]

Examples:
\( (1+2\times3<4 \And 5>6) = (((1+(2\times3))<4) \And (5>6)) \)

Binary operators of equal precedence are left-associative. No automatic type coercion is provided.
2.1.5. Structure Manipulation Constructs

```
<struct> ::= <access> | <replace> | <list-constructor> | <rec-constructor> | <blk-conversion>
<access> ::= { <item> [ <expression> ] } | { <item> . <identifier> }
<replace> ::= <access> "<" <expression>
<blk-conversion> ::= <type> : <expression>
<list-constructor> ::= <type> : [ { <expression> # , } * ]
<rec-constructor> ::= <type> : [ { <record-entry> # , } * ]
<record-entry> ::= <expression> | <identifier> = <expression>
```

Examples:

```
Type T1 = List Of Integer;
Type T2 = Record r:Integer; i:Integer End;
Type T3 = Block Of 3 Integer;

Let v1 = T1:[3,4,5];
Let v2 = T2:[1,2];
Let w2 = T2:[i=2,r=1];

v2=w2;
v2.r=1;
v1[1]=3;
(v1[1] <- 2) = T1:[2,4,5];
(v2.r <- 9) = T2:[9,2];
BlockToList(T3:v1) = v1;
```

All expressions in the above example evaluate to True. Lists are constructed by specifying their elements in order; records are constructed by specifying their elements in order or by name. The back-arrow construct creates a new structure with the selected element replaced with a new value.

Language support for blocks is minimal since they were designed only for communication with user intrinsics (see Section 3.6). They are converted to lists by simply calling BlockToList, but the special construct <block-type>:<list-value> must be used to convert lists to blocks since structure length is part of block types but not of list types.
2.1.6. List Operators

\[ \langle \text{list-op-call} \rangle ::= \langle \text{iota} \rangle | \langle \text{map} \rangle | \langle \text{reduce} \rangle | \langle \text{pair} \rangle | \langle \text{cross} \rangle | \langle \text{select} \rangle \]

\[ \langle \text{binary-intr} \rangle ::= \text{AND} | \text{OR} | = | "<" | "=" | "+" | "+." | "*" | \text{DIV} \]

\[ \langle \text{binary-op} \rangle ::= \langle \text{identifier} \rangle | \langle \text{binary-intr} \rangle \]

\[ \langle \text{iota} \rangle ::= \text{IOTA} (\langle \text{expression} \rangle, \langle \text{expression} \rangle, \langle \text{expression} \rangle) \]

\[ \langle \text{map} \rangle ::= \text{MAP} (\langle \text{expression} \rangle, \langle \text{identifier} \rangle) \]

\[ \langle \text{reduce} \rangle ::= \text{REDUCE} (\langle \text{expression} \rangle, \langle \text{expression} \rangle, \langle \text{binary-op} \rangle) \]

\[ \langle \text{pair} \rangle ::= \text{PAIR} (\langle \text{expression} \rangle, \langle \text{expression} \rangle, \langle \text{binary-op} \rangle) \]

\[ \langle \text{cross} \rangle ::= \text{CROSS} (\langle \text{expression} \rangle, \langle \text{expression} \rangle, \langle \text{binary-op} \rangle) \]

\[ \langle \text{select} \rangle ::= \text{SELECT} (\langle \text{expression} \rangle, \langle \text{identifier} \rangle) \]

Examples:
Type LOI = List Of Integer;
Let L1 = Iota(1,6,2);
Let L2 = Iota(2,6,2);
Let Add1(x:Integer) = x+1;
Let Big(x:Integer) = x>2;
L1 = LOI:[1,3,5];
L2 = LOI:[2,4,6];
Map(L1,Add1) = L2;
Reduce(L1,0,+)=9;
Pair(L1,L2,+)=LOI:[3,7,11];
Cross(L1,L2,*)=LOI:[2,6,10,4,12,20,6,18,30];
Select(L1,Big)=LOI:[3,5];

Each list operator was designed so that the type of the result would be known at parse time. \text{IOTA}(x,y,z) generates a list of all integers between x and y by step z. \text{MAP}(L,f) applies f to each element of the list. \text{REDUCE}(L,x,f) combines the elements of the list using the binary function f and the starting value x. \text{REDUCE} associates to the left: \text{REDUCE}([e_1,\ldots,e_n], x, f) = f(\ldots f(f(x,e_1),e_2),\ldots,e_n). \text{PAIR}(L1,L2,f) applies f to pairs of corresponding elements from L1 and L2, while \text{CROSS}(L1,L2,f) applies f to all pairs of elements. \text{SELECT}(L,f) returns a list of all elements of L for which the function evaluates to True.

This set of list operators was designed to encourage programmers to express their control structures in stereotypical patterns that the system can recognize and use for distribution purposes. The set of list operators is small enough that each member can be treated as a special case, yet complete enough that the programmers who restrict their data structures to lists should rarely be tempted to use recursion. This goal might also have been met by using
the iteration constructs of a dataflow language such as Val or ID (see Section 6.1). Note that dataflow iteration constructs and the STARDUST list operators are both only partial solutions to the problem of control structures, since they only provide sequencing through lists and arrays. The user is still left to provide the sequencing through his own structures such as trees and graphs.

2.1.7. Other Constructs

\[
\text{<prim> ::= <nofix-call> | <ui-call> | <conditional> | <block> | <constant> | "(" <expression> ")"}
\]

\[
\text{<nofix-call> ::= <identifier> "{" <expression> ","* ")"}
\]

\[
\text{<ui-call> ::= USERINTRINSIC "{" <expression> ","* ")" "<type>}
\]

\[
\text{<conditional> ::= \{ IF SIF \} <expression> THEN <expression> ELSE <expression>}
\]

\[
\text{<block> ::= BEGIN \{ TIME <expression> ; \}; <declaration>; \}* <expression> ;? END}
\]

\[
\text{<constant> ::= <integer-constant> | <boolean-constant> | <string-constant> | FAIL | NIL}
\]

Examples:

Let \( XC(s_1: \text{Signal}; s_2: \text{Signal}) = [\text{Atomic}] \begin{array}{l}
\text{Time 60;}
\text{UserIntrinsic("XC", s_1, s_2): XCData;}
\text{End;}
\end{array}
\]

Let \( \text{ParallelOr}(x: \text{Boolean}; y: \text{Boolean}): \text{Boolean} = \begin{array}{l}
\text{If x Then True Else y;}
\end{array}\]

The conditional with parallel semantics terminates in either the normal manner (the condition terminates and the associated branch terminates) or when both branches terminate and yield the same result. It can be used as a primitive for defining other operations with parallel semantics, such as a multiply that terminates when either argument is found to be zero, or an OR that terminates when either argument is found to be True. The sequential and parallel conditionals should probably have been called "If" and "Pif" rather than "Sif" and "If", thus making sequential semantics the default. The tendency of the parallel operator to increase concurrency is offset by its tendency to swamp the system with unnecessary computations.

User intrinsics, described in Section 3.6, are a mechanism for calling routines written in other languages. The first argument to all UserIntrinsic calls must be of type String, and the call must be followed by the result type. The programmer should generally package his calls
to UserIntrinsic inside of another function (XC was used in the above example) in order to associate a time estimate with it.

### 2.2. Sample Programs

This section presents some simple STARDUST programs. Further examples may be found with the experimental results in Chapter 5.

#### 2.2.1. Prime Numbers

```plaintext
Let Factors(x: Integer) = Begin  | Factors of x
  Time x*2;
  Let Fac(y: Integer) = Begin
    Time 1;
    (x DIV y)*y = x;
    End;
  Select(Iota(1, x, 1), Fac);
  End;

Let Prime(x: Integer) = Begin  | Primality of x
  Time x*2;
  (Length(Factors(x)) = 2);
  End;

Let Primes(x: Integer) = Begin  | Primes from 2 to x
  Time x*x*2;
  Select(Iota(1, x, 1), Prime);
  End;
```

#### 2.2.2. Towers of Hanoi

```plaintext
| Hanoi(n, f, t, u) moves n disks of descending size from post f to post t using post u. A larger disk is never placed on top of a smaller one. Each element of the output list is a pair containing source and destination posts. |
| For example, Hanoi(3, 1, 3, 2) returns |
  [[1,3],[1,2],[3,2],[1,3],[2,1],[2,3],[1,3]]

Type Hlist = List of List of Integer;
```
Let Hanoi(n: Integer; From: Integer; To: Integer; Using: Integer): Hlist =
Begin
  If n=0 Then nil
  Else
    Concat(Hanoi(n-1,From,Using,To),
           Cons(
             List Of Integer:[From,To],
             Hanoi(n-1,Using,To,From)));
End;

2.2.3. Matrix Multiply

Type Row = List Of Integer;
Type Matrix = List Of Row;

Let Multiply(m1:Matrix; m2: Matrix) = Begin
  Let Columns = Length(m2[1]); | Columns in output
  Let Rows = Length(m1); | Rows in output
  Let Mlength = Length(m1[1]); | Common length

  Let Element(r:Integer;c: Integer) = Begin | Output element (r,c)
    Let Elt(x:Integer) = m1[r][x]*m2[x][c];
    Reduce(Map(iota(1,Mlength,1),Elt),0,+);
  End;

  Let OutRow(r:Integer) = Begin | Output row r
    Let Relement(c:Integer) = Element(r,c);
    Map(Iota(1,Columns,1),Relement);
  End;

  If Mlength = Length(m2) Then
    Map(Iota(1,Rows,1),OutRow)
  Else
    Fail;
End;
Chapter 3
The Implementation

In vials of ivory and coloured glass
Unstoppered, lurked her strange synthetic perfumes,
Unguent, powdered, or liquid – troubled, confused
And drowned the sense in odours; ... 

3.1. Overview

STARDUST is implemented as a reduction machine. Expressions are represented as graph structure, and evaluated by a sequence of transformations that reduce them to simpler forms but leave their values unchanged. In STARDUST these transformations fall into three categories: execution of intrinsics (Figure 3-1), expansion (also called unfolding or application) of user functions (Figure 3-2), and elimination of redundant subcomputations (see Section 3.5). Expressions are evaluated in the standard reduction machine's top-down (request-driven, lazy) order, occasionally modified with bottom-up (data-driven, eager) evaluation to improve performance.

![Figure 3-1: Execution in a Graph Reduction Language](image)

Distribution, described in detail in Chapter 4, is accomplished by sending subgraphs to other processors for evaluation. The processor with responsibility for evaluating a top-level expression will generally expand user functions until their projected execution times fall below a threshold, then send them to other processors for completion. Since the language is applicative, fault tolerance can be achieved through simple restarts.
3.2. Background

A brief overview of termination semantics and parameter passing will ensure a common base of definitions.

3.2.1. Overview of Termination Semantics

Much of the effort in building a distributed evaluation system is concerned with maintaining consistent termination semantics: ensuring that certain expressions terminate even when some of their subexpressions diverge. In a deterministic applicative language such as STARDUST, three interpretations are possible for intrinsic functions. The following two expressions will be used as examples:

\[
\begin{align*}
\text{If } E_1 \text{ Then } E_2 \text{ Else } E_3; \\
E_4 \text{ And } E_5;
\end{align*}
\]

1. **Strict Semantics.** If any of the arguments to a function diverge, then the function itself diverges. This is a very unforgiving policy: Divergence of \( E_3 \) will cause the conditional expression to diverge even if \( E_1 \) evaluates to True.

2. **Sequential Non-Strict Semantics (SNSS).** Functional arguments are evaluated in a predetermined order until the value of the function can be determined. For
example, if $E_1$ evaluates to True then $E_3$ may diverge; if $E_4$ evaluates to False then $E_5$ may diverge.

3. **Parallel Non-Strict Semantics (PNSS).** A function call terminates whenever enough of its arguments terminate to fix its value. For example, $E_4$ may diverge if $E_5$ returns False, and $E_1$ may diverge if $E_2$ and $E_3$ return the same value.

The key distinction between SNSS and PNSS is that once evaluation has started on an argument to an SNSS function it never needs to be aborted. Either the argument will terminate or the expression as a whole will legitimately fail to do so. PNSS demands that the arguments be evaluated with real or simulated parallelism, since in general some action must be taken if *any* of them terminate. It is usually considered necessary that arguments whose values turn out not to be needed be tracked down and aborted, although this strictly speaking has no effect on the termination semantics.

### 3.2.2. Overview of Parameter Passing

While the termination semantics of intrinsic functions can be easily broken down into the three categories listed above, the interpretation of user-defined functions is more complicated. Five different mechanisms are commonly used in imperative languages to bind the parameters in the header of a function definition (the *formal* parameters) to the parameters in the invocation (the *actual* parameters). The first two and the last two are semantically equivalent in languages without side effects, while call-by-value-result is not meaningful.

1. **Call-by-value.** The value of the actual parameter is passed to the function. The actual parameter is evaluated before the function is invoked.

2. **Call-by-reference.** A pointer to the actual parameter is passed to the function. This discipline is usually restricted to items that can be modified: variables and structure elements. Call-by-reference is frequently the most efficient parameter passing mode, but generates an alias between two formal parameters when they are passed the same actual parameter.

3. **Call-by-value-result.** The expression is evaluated and passed to the function as in call-by-value. If the function modifies the corresponding formal parameter and the actual is assignable, then the value is stored in the actual on exit from the function. This mechanism allows modifiable parameters without the aliasing problems of call-by-reference.

4. **Call-by-name.** The actual parameters are not evaluated before passing them to the function; instead the code for generating them is bound with the caller's environment to form a *suspension* or *thunk*, which is evaluated every time the argument's value is needed in the function.

Expressions evaluated under call-by-name in a side effect free language yield the same values as under call-by-value when both terminate. If the expression
representing the actual parameter has side effects, then a series of references to
the formal parameter may return different values.

5. **Call-by-need.** As in call-by-name, a suspension is created for the actual
parameter, but the suspension replaces itself with its value after its first invoca-
tion. While this yields semantics identical to call-by-name in a language without
side effects, it often gives important performance improvements.

Languages using call-by-value, call-by-reference, and call-by-value-result parameter
passing have strict semantics for user functions, even though they usually have SNSS seman-
tics for IF, and often have SNSS semantics for AND and OR. This is considered unacceptable
in an applicative language because it violates the otherwise universal principle that the mean-
ing of a function call is not changed when the call is replaced by its definition with actuals
substituted for formals. **STARDUST** uses call-by-need parameter passing.

### 3.3. Representation of Expressions

Expressions are stored as a collection of *Forms* and *Nodes*. The Form/Node Represen-
tation (FNR) is the basis for both expression evaluation (operators transform the expressions
in FNR to other expressions with the same value), and communication (Forms are transmitted
to other processors for evaluation and the results are returned as Forms).

#### 3.3.1. Forms

Forms represent expressions, while Nodes represent sets of Forms known to have the
same value. There are five classes of Forms, each with its own internal representation.

1. **Constant** Forms represent integers, booleans, strings, *Fail*, and Block structures.
   They contain the Type and a field of bits long enough to describe the value.

2. **Remote** Forms are place holders used when no expression is known for a Node.
   This can occur when a processor is waiting for a value from another machine.
   Remote Forms are represented by the null pointer.

3. **Structure** Forms are used to represent Lists and Records. They consist of a Type,
   the number of elements, and an array of pointers to the Nodes that represent their
   components.

4. **Call** Forms represent function calls. They consist of the Result Type, a pointer to
   a *Function Descriptor* (see Section 3.4), the number of arguments, and an array of
   pointers to the Nodes that represent the arguments.

5. **Closure** Forms represent functions that are used as arguments to the list
   operators. They consist of a function descriptor and the implicit actual
   parameters (see Section 3.4), which constitute the calling environment.
3.3.2. Nodes

A Node is a structure representing a set of Forms known to have the same value. Nodes are used for the redundant common subexpression elimination of Section 3.5. They contain backpointers to the Forms, Actions, Handles, and Names\(^2\) that reference them; when two Nodes are merged the pointers to the deleted Node contained in its Forms, Actions, Handles, and Names are redirected to the remaining one. Each Node has a distinct BestForm, which is the newest Form added to the set.

3.3.3. Handles

A Handle is a structure that does nothing but point to a Node. Nodes contain backpointers to their Handles, and modify the Handles when the Node is merged with another. The pointers to Nodes found in Forms, Actions, and Names are also treated this way. Handles are used when the interpreter needs a Node reference that will not be left dangling as the result of redundant subcomputation elimination.

3.3.4. Discussion

The use of the FNR for redundant subcomputation elimination and distribution will be discussed in later sections. We will now discuss the representation of structure Forms and the implications for lazy evaluation.

Lazy evaluation, as defined here, is a combination of call-by-need parameter passing and "lenient" constructors. It was first introduced to the applicative language literature as an enhancement to the traditional evaluation strategy for LISP [Friedman 76, Henderson 76]. It is strictly more powerful than the traditional scheme in the sense that (1) All expressions whose evaluations terminate under traditional evaluation terminate under lazy evaluation and yield the same result, and (2) More expressions terminate under lazy evaluation than under traditional evaluation.

A constructor is a function that puts its arguments into a structure. In LISP 1.5 the only constructor is CONS; in STARDUST there are constructors for lists (homogeneous structures of variable length whose elements are accessed by CAR, CDR, and by index), and records (heterogeneous structures of fixed length whose elements are accessed by name).

Parameter passing is a way of transferring values between parts of a program. Struc-

\(^2\)Actions and Names are defined in later sections
tures have a similar purpose, though they accomplish it by putting the values where other parts of the program can find them rather than by passing them directly. Given this similarity it is not surprising that the parameter passing mechanisms of 3.2.2 should have their analogs in structure creation mechanisms.

Traditional constructors are strict in all arguments, usually giving rise to an analog of call-by-reference semantics. By contrast, lenient constructors create structures with a suspension for each argument. The suspension is not coerced to a value until the structure element is referenced. The suspension may be evaluated by every reference as in call-by-name, but it is more common to replace the suspension with its value after the first reference as in call-by-need.

Lenient arrays (Figure 3-3.a) were chosen as the representation for lists in STARDUST. Note that while this gives every list value a fixed length, a list variable may have a value of any length. This representation was chosen over linked lists (Figure 3-3.b) on the assumption that STARDUST will be used mostly for numerical computations dealing heavily with fixed-length arrays, for which a fixed-length representation has the advantage of random access and compact representation.

A linked representation would have allowed infinite lists to be defined and had the advantage that CONS and CDR could be performed without copying any elements. If the special advantages of linked lists are needed, they may be defined from the record facility in the following manner. The more general case of a CONS cell whose CAR may be of any type is not possible in STARDUST 1.0 since there are no escapes from the strong typing.

```
Type Cell = Record CarCell: Integer; CdrCell: Cell End;

Let LenientCons(x: Integer; y: Cell) = Cell:[x,y];
| The integers from x to infinity
Let From(x:integer):Cell = LenientCons(x,From(x+1));
| The n'th element of a linked list
Let nth(c:Cell; n: Integer): Integer =
  If n=1 Then c.CarCell
  Else nth(c.CdrCell,n-1);
```

STARDUST also supports Block structures, which are implemented as arrays of values (Figure 3-3.c) rather than the arrays of expressions used for lists. This representation uses less storage but does not support lazy semantics.
(a) Fixed-length arrays with suspensions, used to represent Lists.
(b) Cells with suspensions, not used in STARDUST.
(c) Fixed-length arrays with in-line data, used to represent Blocks.

Figure 3-3: Three list storage mechanisms

3.4. Representation of Functions

3.4.1. Function Descriptors

The Function Descriptors contained in Call Forms and Closure Forms contain both parsing information and run-time information. Function Descriptors have the following fields:

- **NumArgs**: The total number of parameters, including implicit parameters (implicit parameters, described in Section 3.6, are used to access free variables)
- **NumExplicit**: The number of explicit parameters
- **Types**: An array of length NumArgs specifying the types of the input parameters
3.4.2. Function Definitions

The Function Definition tells the interpreter how to modify the FNR when the expression containing the function is evaluated. For intrinsics this is just an integer used to dispatch to a set of PASCAL routines that perform the actual computation. For user functions it is a Parse Tree Representation (PTR) of the body of the function definition. Section 3.6 explains how both kinds of function definitions are used to modify the FNR.

3.4.3. Parse Tree Representation

Like the FNR, Parse Trees represent expressions. They differ from FNR in that they can represent variables (formal function parameters), but not remote expressions. Structures are represented as calls to constructor functions. There are four possible classes for a parse tree element p:

1. Constant parse tree elements contain a Handle (p.Handle) that points to a Node whose BestForm represents the constant.

2. Call parse tree elements contain a pointer to the function descriptor (p.Function) and a list of the parse tree elements that describe the arguments (p.Args).

3. Closure parse tree elements, which are used as arguments to list operators, contain a pointer to the function descriptor (p.Function) and a list of the parse tree elements that represent the implicit actual parameters (p.Args). Implicit parameters are described in Section 3.6.

4. Variable parse tree elements represent formal parameters to the function that the parse tree defines. The parameter is represented by p.Index, a number between one and the number of formal parameters.

3.4.4. Discussion

Parse trees closely mimic the FNR. It would be worth investigating the idea of representing function definitions as FNR with a special Form type for formal parameters. This would allow function definitions to be transmitted between processors using the same mechanism as is used for expressions, and would form the basis for "fully lazy" evaluation, as described in [Hughes 82].
3.5. Redundant Subcomputation Elimination

The detection and elimination of redundant subcomputations can be critical to the efficient execution of a program. They are traditionally eliminated in imperative languages by storing intermediate values in variables or structures, and in applicative languages by binding intermediate values to symbols. For example, the elimination of the common $(3 + 4)$ in the expression $2 \times (3 + 4) + (3 + 4)$ could be expressed in an imperative language as

\[
\begin{align*}
\text{Temp} &:= 3 + 4; \\
\text{Result} &:= 2 \times \text{Temp} + \text{Temp};
\end{align*}
\]

and in an applicative language as

\[
2 \times \text{Temp} + \text{Temp} \text{ Where Temp} = 3 + 4;
\]

Redundant subcomputation elimination (RSCE) in STARDUST has the effect of transforming a tree of computations into a Directed Acyclic Graph (DAG) as in Figure 3-4. This is done with a hashing mechanism at run time, in contrast to the two previous examples where it was accomplished by transforming the program before execution. The particular method of RSCE used to transform the tree into the DAG is not crucial to the rest of STARDUST; some advantages of doing it at run time will be presented after the algorithm has been described.

\[
\begin{align*}
\text{Temp} &:= 3 + 4; \\
\text{Result} &:= 2 \times \text{Temp} + \text{Temp};
\end{align*}
\]

\[
2 \times \text{Temp} + \text{Temp} \text{ Where Temp} = 3 + 4;
\]

RSCE is accomplished by performing a hash lookup on every new Form as it is created. This is not limited to Forms that are input by the user, but also includes Forms that are generated by the interpreter as a result of executing or expanding other Forms. Thus, RSCE works on expressions generated at run time in addition to those appearing explicitly in the program.
3.5.1. Implementation of RSCE

When Forms are originally created they are put on the Rehash Queue (RQ), indicating that they have not been entered into the hash table. RehashAll is invoked after each evaluation of an expression, causing these Forms to be entered into the hash table using a key which, for structure Forms and call Forms, depends on the addresses of the Nodes that represent the components or arguments. If Rehash finds no identical Form in the hash table then the new Form is simply entered; if an identical Form is found then their corresponding Nodes (equivalence classes) are merged with MergeNodes, and any Forms whose keys depended on the address of the expunged Node are placed on RQ. An example appears in Section 3.5.3.

Define RehashAll =
   While RQ is not empty
      Remove the first element F;
      Call Rehash(F);

Define Rehash(F:. Form) =
   Search the hash table for a Form F' that is identical to F;
   If there is none Then
      Enter F in the hash table and Exit.
   Else
      Deallocate(F);  | Only F.Node references F
      MergeNodes(F.Node,F'.Node).

Define MergeNodes(N_1,N_2:Node) =
   For each Form F that references N_1:
      Redirect F to point to N_2;
      Remove F from the hash table;
      Put F on RQ.

   For each Action A that references N_1:
      Redirect A to point to N_2;
      call RenewAction(A)
      Move all Handles and Names from N_1 to N_2.
      Deallocate N_1.

The actual implementation includes some optimizations to avoid the excess creation and deletion of Nodes and Forms that this simple algorithm causes. For example, it allows the case where one of the Forms, being newly created, does not belong to a Node. Rehash also keeps a usage count for each Node and expunges F' and F'.Node rather than F and F.Node if F.Node is referred to by more objects than F'.Node.

A bucket hashing algorithm with a fixed-length table is used. The hashing key for a constant Form is equal to the sum of the words that make up the constant (only the first ten
words are used if the constant is more than ten words long); the hashing key for a structure is the sum of the addresses of the Nodes that represent the components (again up to a maximum of ten); and the hashing key for a function call or a closure is the sum of the addresses of the argument Nodes and the address of the function descriptor.

3.5.2. Discussion

While expressions could have been represented using only Forms (Figure 3-5), the additional level of indirection provided by Nodes (Figure 3-6) offers important advantages. Suppose, for example, that \( g(3) \) is evaluated to produce 2. If only Forms are used (Figure 3-7), \( g(3) \) must be deleted and the pointer from \( f \) must be redirected to point at the new Form. The information that \( g(3) = 2 \) is lost, and the Form containing \( f \) must be rehashed since it now points to a different destination. (If \( g(3) \) were overwritten by 2, the system would not need to rehash, but overwriting would require that Forms be represented by either fixed-length records or by linked lists, either of which would probably add more overhead than the Node representation.) Figure 3-8 shows the outcome when \( g(3) \) is replaced by 2 using Forms and Nodes. Note that the Form containing \( f \) does not have to be altered, and that the relationship \( g(3) = 2 \) is retained. The old Form \( g(3) \) will be retained in STARDUST 1.0, causing the Node that contains both \( g(3) \) and 2 to act as a cache for the value of \( g(3) \). In principle the Form \( g(3) \) could be deleted to save storage if this caching effect were not desired, though this option is not available in STARDUST 1.0.

![Figure 3-5: Representation of \( f(g(3)) \) Using Only Forms (Circles)](image)
Consider the expression \( f(2) + f(g(3)) \), where \( f \) and \( g \) are intrinsics, \( g(3) \) has the value 2, and \( f(2) \) has the value 7. Figure 3-9 shows the transformations that occur in the FNR when \( g(3) \) is evaluated first, while Figure 3-10 shows the transformations that occur when \( g(3) \) is evaluated after \( f(2) \). Note that in either case \( f(2) \) is evaluated only once.
3.5.4. Importance of Redundant Subcomputation Elimination

RSCE can reduce the asymptotic complexity of certain algorithms by providing automatic dynamic programming (see [Aho 75] for a description of dynamic programming). The recursive definition of Fibonacci numbers, often used as an example of caching and efficiency in applicative languages, has an execution time exponential in \( n \) when evaluated naively, but linear in \( n \) when evaluated with caching as in STARDUST:

\[
\begin{align*}
\text{Fib}(n) &= 1 & \text{for } n = 1 \text{ or } n = 2 \\
\text{Fib}(n) &= \text{Fib}(n-1) + \text{Fib}(n-2) & \text{for } n > 2
\end{align*}
\]

A less trivial example illustrates how automatic RSCE can save the programmer from using arrays to perform explicit caching in scientific calculations. Solutions to Laplace's equation can be approximated using a relaxation scheme, starting with the function \( f \) fixed on some boundary and given some initial values. The values of \( f \) can be estimated from the previous iteration as:

\[
f_{\text{now}}(x,y) = \frac{f_{\text{old}}}{2} + \left( \frac{f_{\text{old}}(x-1,y-1) + f_{\text{old}}(x+1,y-1) + f_{\text{old}}(x-1,y+1) + f_{\text{old}}(x+1,y+1)}{8} \right)
\]

This may be expressed in a purely applicative form as:

\[
\text{Let } f(t,x,y) = \text{Begin} \text{ If } t = 0 \text{ Then Initial}(x,y) \text{ Else If OnBoundary}(x,y) \text{ Then BoundaryValue}(x,y) \text{ Else } f(t-1,x,y)/2 + (f(t-1,x-1,y-1) + f(t-1,x+1,y-1) + f(t-1,x-1,y+1) + f(t-1,x+1,y+1))/8 \text{ End;}
\]
• Step 1: A new Form 2 is created and attached to the Node containing g(3). Since 2 was derived from g(3), it is marked as the "BestForm" of the Node.

• Step 2: The two Nodes that refer to the Form 2 are merged. Pointers are redirected to the new Node.

• Step 3: The two Forms representing f(2) are rehashed and found to be identical. Their Nodes are merged, and the pointers from the Form representing f(2) + f(2) are redirected.

• Step 4: The Form representing f(2) + f(2) is rehashed, but does not match any previous Form.

Figure 3-9: g(3) is Replaced by 2 in the Expression f(2) + f(g(3))

Without some form of RSCE this definition would generate redundant calls to f; in fact the number of such calls is exponential in t if the boundary is large compared to t. (To see this, consider that each call to f(t,x,y) generates five calls to f(t-1,...) if the effects of the boundary are neglected.) With RSCE the number of calls to f is limited to t times the area within the boundary, making the algorithm linear in t but incurring storage costs that are also proportional to t.
• Step 1: $f(2)$ evaluates to 7, and the Form 7 is attached to the Node containing $f(2)$.

• Step 2: $g(3)$ evaluates to 2, and the Form 2 is attached to the Node containing $g(3)$.

• Step 3: The two Nodes containing the Form 2 are merged, and pointers redirected.

• Step 4: The two Forms representing $f(2)$ are rehashed and found to be identical. The Nodes containing these Forms are merged.

• Step 5: The Form representing $7 + 7$ is rehashed, but does not match any previous Form.

Figure 3-10: $g(3)$ is Replaced by 2 in the Expression $f(2) + f(g(3))$, When $f(2)$ has Already Been Evaluated to Produce 7

3.5.5. Comparison With Other Forms of RSCE

Run-time RSCE can be contrasted with the static optimization found in optimizing compilers [Wulf 75], static applicative language optimizations [Bird 80, Burstall 77, Cohen 79, Scherlis 80], and memo functions [Bird 80, Keller 81, Michie 68].

The common subexpression elimination (CSEE) of optimizing compilers is generally
limited to the detection of equivalent expressions in the same static and dynamic environment. It is rare to see CSEE across procedure calls, for example. The expressions eliminated may be explicit:

\[a := b+c; \]
\[d := b+c;\]

or implicit, such as the array access calculations in the following example, where \(a\) and \(d\) have the same dimensions:

\[a[b,c] := d[b,c].\]

An imperative language compiler must use flow analysis to determine whether identical expressions can be merged. For example, the two occurrences of \(b+c\) in the following fragment cannot be merged:

\[a := b+c;\]
\[b := b+1;\]
\[d := b+c;\]

CSEE is trivial in an applicative language since identical expressions in the same environment always have the same value. However, most RSCs cannot be eliminated this way, since they do not originate as identical expressions in the user's program.

Deeper transformations on applicative programs can catch RSCs generated in different environments. For example, Burstall and Darlington have shown how the standard definition of Fibonacci numbers can be mechanically transformed into an efficient applicative program [Burstall 77]. Note that redundant calls to \(f(z)\) are eliminated even though they are generated in the environments of \(f(z+1)\) and \(f(z+2)\). The resulting program follows, expressed in the notation of the paper.

\[
\begin{align*}
f(0) & \leq 1 \\
f(1) & \leq 1 \\
f(x+2) & \leq u+v, \text{ where } <u,v> = g(x) \\
g(0) & \leq <1,1> \\
g(x+1) & \leq <u+v,u>, \text{ where } <u,v> = g(x)
\end{align*}
\]

Memo functions [Michie 68] are a form of run-time RSC. In the version originally proposed by Michie, each memo function keeps a table of the values which it has recently produced. A call to the function initiates a table lookup, and the body of the function executes only if the lookup fails. Each calculated value is placed at the head of the list, on the theory that it is likely to be needed again soon. The more recent work of Keller and Sleep [Keller 81] discusses ways to implement memo functions from within an applicative language.

Aside from the question of whether table or hash lookup is more efficient, memo func-
tions have the disadvantage that RSCs can only be detected when the function is about to be executed. STARDUST must detect them earlier for better automatic distribution.

Furthermore, early RSCE can make CHOICE resolution more effective because it permits accurate estimation of processing requirements to be made earlier. For example, if the user asks for \( f(3,3) \), where \( f(x,y) \) is defined to be \( \text{CHOICE}(g(x) + g(y), h(x + y)) \), the system should be able to estimate the execution time of the first branch knowing that \( g \) will have to be invoked only once.

The STARDUST RSCE algorithm is a form of congruence closure as defined by Nelson and Oppen [Nelson 78]. The algorithm used in STARDUST is similar to the one that Nelson and Oppen use: both algorithms form a key from the function name and the equivalence classes of its arguments; STARDUST uses the key to perform hash table searches while Nelson and Oppen's algorithm sorts all Forms by key and compares adjacent ones for equality. Nelson and Oppen's algorithm is well-suited to batch modifications to the set of known equivalences because a single sort causes all sets of newly equivalent Forms to become adjacent. The STARDUST algorithm is well-suited to incremental modifications (interpreting a program causes incremental modifications) because the system only considers the Forms whose keys actually change.

3.5.6. Why RSCE at Run Time is Sometimes Superior to Static RSCE

Static RSCE has inherent limitations. The existence of RSCs may depend on actual parameters as in the following example, where the calls on \( g \) are redundant only if \( x \) and \( y \) are equal:

Let \( f(x,y) = g(x) + g(y) \)

It would indeed be possible to convert \( f \) statically into a form that eliminated the redundancy, but this would really just be an explicit form of run-time CSEE:

Let \( f(x,y) = \begin{cases} 2g(x) & \text{if } x=y \\ g(x)+g(y) & \text{else} \end{cases} \)

STARDUST may need to run in a multiple-user environment. If two users of a sensor network request a reading on overlapping regions, many of the expensive low-level signal processing computations will be redundant. Static RSCE cannot eliminate these redundant subcomputations, since they are generated completely at run time.
3.5.7. Costs and Limitations

Despite the arguments for detecting redundant subcomputations at run time, it is unquestionably true that the mechanism can consume more cycles than it saves. (This is trivially true if there happen to be no RSCs.) STARDUST 1.0 spends about 20% of its 10 milliseconds per function call doing tasks directly related to RSCE: hashing Forms, comparing them for equality, and merging Nodes. The exact figures depend on the application. Other overhead is added indirectly by the need to maintain Nodes in addition to Forms, and the need to maintain backpointers from Nodes to the Forms, Actions, Handles, and Names that use them. While difficult to measure directly, we estimate that these indirect overheads account for another 20%.

RSCE is more likely to be cost-effective in STARDUST than in other graph reduction schemes, since the system is already committed to a preponderance of large-grain function calls by the other per-expression sources of overhead such as message passing, partitioning, scheduling, and queueing actions. Chapter 7 contains further discussion of performance issues.

STARDUST 1.0 contains no facilities for detecting RSCs across processor boundaries, with the result that if f(1 + 3) and f(2 + 2) are assigned to different processors, f(4) will be computed twice. RSCE across machines would be easy to implement, but due to the large communication overhead it would only make sense to use it for functions with very high execution time estimates or functions which had somehow been recommended by the programmer as likely candidates for matches. A consequence is that expressions with very short projected execution times (such as 1 + 3 and 2 + 2 in the above example) should be evaluated locally before distribution takes place.

3.6. Execution and Expansion

An expression in FNR is evaluated by making a sequence of modifications to the expression graph that leave its value unchanged. This section describes the action of System Intrinsics, User Functions, and User Intrinsics. System intrinsics replace a Form of class call with a value calculated by a set of PASCAL instructions defined by the system implementor. User intrinsics replace it with a value calculated by a PASCAL subroutine defined by the user. User functions replace it with a new expression found by translating the function's parse tree representation (PTR) into FNR, substituting actual parameters for formals. The following code summarizes this process.
Define Evaluate(N: Node) =
Set F to N.BestForm;
If F.Function is a system intrinsic Then
Set F' to Execute(F)
Else
Set F' to Expand(F);
Add F' to N's set of Forms;
Set N.BestForm to F'.

Define Execute(F: Form) =
If F is MAP([N1, ..., Nn],f) Then
Return [f(N1), ..., f(Nn)]
ElseIf F is REDUCE([N1, ..., Nn],s,f) Then
Return f(...f(s,N1),...,Nn)
ElseIf F is PAIR([N1, ..., Nn],[M1, ..., Mn],f) Then
Return [f(N1,M1), ..., f(Nn,Mn)]
ElseIf F is CROSS([N1, ..., Nn],[M1, ..., Mn],f) Then
Return [f(N1,M1), f(N1,M2), ..., f(Nn,Mm)]
ElseIf F is SELECT([N1, ..., Nn],f) Then
Return Select([N1, ..., Nn],MAP([N1, ..., Nn],f))
ElseIf F is Select([N1, ..., Nn],[B1, ..., Bn]) Then
Return a list of all Ni such that Bi is True
ElseIf F is UserIntrinsic(N1, ..., Nn) Then
Return the user-calculated value (see below)
Else
Return the results of applying the standard meaning of the operator to the arguments.

Define Expand(F: Form) =
Translate(F.Function, F.Args)

Define Translate(p: PTR; Actuals: Array Of Node) =
If p is a constant Then
Return the Node that it points to
ElseIf p is a call Then Begin
If p.Function is Construct Then
Return a structure Node made from the results of applying Translate to p.Args
Else
Return a call Node made from p.Function and the results of applying Translate to p.Args
End
ElseIf p is a closure Then
Return a closure Node made from p.Function and the results of applying Translate to p.Args
ElseIf p is an argument Then
Return the corresponding member of Actuals.
3.6.1. Discussion

Evaluate dispatches to Execute or Expand to find the new Form, then patches the new Form into the FNR. Execute creates a new Form using PASCAL code while Expand creates a new Form by placing actual parameters into a copy of the function's parse tree. The actual algorithm used in STARDUST 1.0 is more complex due to some optimizations for reducing the number of Nodes created. A discussion of user intrinsics and free variable resolution follows.

3.6.1.1. User Intrinsics

User intrinsics call PASCAL subroutines. The call to a cross correlator in the distributed sensor network program of Section 5.1 is a typical example. Note that the call is "packaged" inside of the user function XC in order to associate a time estimate with it:

Let XC(s1: Signal; s2: Signal) = [Atomic] Begin
  Time 60;
  UserIntrinsic("XC",s1,s2):XCData;
End;

The first argument to UserIntrinsic must be a string. The system packs the arguments into a buffer which it sends to a PASCAL function called UserInt. This function interprets the first argument as a dispatch (in this case "XC" tells it to do cross-correlations), performs the operation, places the result in the buffer, and returns. STARDUST turns the result into a Node of the return type, which in this example is XCData. Lists, records, blocks, and constants may be used as input or output from user intrinsics. User intrinsics have strict semantics.

The communication buffer could easily be sent in a message to a concurrently executing server process if it were inconvenient to compile and link the user's code with STARDUST. Alternatively, under an operating system such as Unix™, the string could be interpreted as a shell command and the communication could proceed through the standard input and output files of the spawned process.

3.6.1.2. Free Variables

STARDUST allows block structured function definitions with standard lexical scoping: A free variable in a function definition is given the binding of the variable with the same name in the innermost lexically nesting block that contains such a variable. Rather than accessing free variables through an environment, STARDUST passes the values to the inner function as implicit parameters.

Aside from resolving free variable bindings the lexical nesting has no significance, so
function definitions are represented internally as unnested, with function name ambiguities resolved by renaming. Note how \( y \) is passed to the inner function \( f \) through the implicit parameters \( G_1 \) and \( G_2 \) in the following example:

\[
\begin{align*}
\text{Let } f(x,y) &= \text{Begin} \\
\text{Let } g(w) &= \text{Begin} \\
\text{Let } f(z) &= z+y; \\
& \quad f(w+w); \quad | \text{refers to } f \text{ on the previous line} \\
& \quad \text{End}; \\
& \quad g(x+1); \\
& \quad \text{End}; \\
\end{align*}
\]

is represented internally as

\[
\begin{align*}
\text{Let } f_1(x,y) &= g(x+1,y); \quad | \text{corresponds to the outer } f \\
\text{Let } g(w,G_1) &= f_2(w+w,G_1); \\
\text{Let } f_2(z,G_2) &= z+G_2; \quad | \text{corresponds to the inner } f \\
\end{align*}
\]

Closures are represented internally by a function descriptor and its implicit actual parameters. In STARDUST 1.0 closures are created only for the functional arguments of list operators, but the same internal structure could be used for implementing function-valued functions as well. As an example of their current usage, consider the following function and its internal representation:

\[
\begin{align*}
\text{Let } \text{Factors}(x:\text{Integer}) &= \text{Begin} \quad | \text{Factors of } x \\
\text{Let } \text{Fac}(y:\text{Integer}) &= \text{Begin} \\
& \quad (x \div y) \times y = x; \\
& \quad \text{End}; \\
& \quad \text{Select(}\text{Iota}(1,x,\text{I}), \text{Fac}); \\
& \quad \text{End}; \\
\text{Let } \text{Newfac}(y:\text{Integer}; G_1: \text{Integer}) &= \text{Internal representation} \\
& \quad (G_1 \div y) \times y = G_1; \\
\text{Let } \text{Newfactors}(x:\text{Integer}) &= \text{Internal representation} \\
& \quad \text{Select(}\text{Iota}(1,x,1), \text{Newfac}<G_1=x>); \\
\end{align*}
\]

\( \text{Newfac}<G_1=x> \) can be viewed either as \((\lambda y) \text{Newfac}(y,x))\) or as a closure of the original function \( \text{Fac} \) in the \( \text{Newfactors} \) environment.

We will call the explicit and implicit actual parameters of a function \( f \) its references. These are the only values that can influence the outcome of the function, which is an important fact both for redundant subcomputation elimination and for exportation of expressions to other processors.
The redundant subcomputation elimination algorithm needs to know f's references. Simply hashing f and f's explicit actual parameters would cause calls with free variables in different environments to be considered identical, while hashing the entire environment would cause valid identities to be missed. The exportation routine also needs to know f's references, since the slow speed of local area message passing makes it inefficient to send the entire environment, and also inefficient to send values from the environment individually as they are accessed. The interpreter compromises by sending the portion of the environment that is referenced.

This makes it clear that the system should determine the references for each function, and that it is convenient to represent this knowledge by including those references as implicit parameters. Whether it is efficient to do so depends on the style in which STARDUST is used and assumptions about relative costs of argument passing and environment management.

STARDUST will tend to give relatively poor results when non-local variables are passed through many levels of function call before being accessed, and relatively good results when non-local variables are not used or are frequently accessed. Note that in STARDUST 1.0, where every new Form is hashed, parameters are always accessed.

The rest of this section outlines the changes that would be needed to make STARDUST use environments instead of implicit parameters.

The first step is to add an environment to Forms that represent calls to user functions. This environment is a linked list of activation records; the length of the list is equal to the lexical nesting depth of the function being called. An activation record is conceptually an array of values, but due to the call-by-need semantics and redundant subcomputation elimination it must be represented as an array of Handles. The function descriptor, the environment, and the values of the arguments will be all that is needed to determine the value of the function.

The next step is to change variable references in the parse tree from a simple offset to an offset p and a lexical nesting level n. The value of a variable can be found by following the environment pointer back through n activation records, then using the p\textsuperscript{th} element of the activation record.

Environments are used only when user functions are expanded, that is, only when parse tree representation is transformed into FNR. Constant and closure parse tree elements are translated as before. Variable parse tree elements are replaced with the appropriate Node either from the list of actual parameters or from the environment. This process can be
speeded up by following the environment links only once, creating a temporary display. Call parse tree elements are translated to call Nodes with new environment pointers.

The environments added to these new call Nodes are determined by the lexical nesting level of the function being called. If it is at the same level or shallower than the caller, the new environment can be found by removing zero or more activation records from the current one, which can be done by looking it up in the temporary display. If the function being called is lexically deeper, a new activation record must be created for the current function and linked to the current function's environment. Note that if the function being expanded calls no lexically nested functions, this new activation record need not be created.

Closures are represented by a function descriptor and an environment, and closure application is accomplished by expanding the function using the environment in which it was closed rather than the current one. If function definitions were stored as FNR rather than the parse tree representation, it would probably be most efficient to make the substitutions for free variables at the time of the closure, since this would make repeated applications of the function (such as using it as an argument to a list operator) much faster. The environment would still have to be preserved in order to be passed to subfunctions.

As stated earlier, the redundant subcomputation eliminator and the distribution code both need to know the variables referenced by the functions on which they operate. This information could be attached to the function descriptor as an array of parse tree elements; or a separate version of the hashing code and the code that transforms Nodes into messages could be compiled for each function.

Note that in a language with upward funargs the tree of environments must be managed with garbage collection or reference counts. The implicit parameter implementation manages it implicitly as part of the expression graph.

### 3.7. The Evaluator

This section describes the STARDUST evaluation strategy for a single processor. The evaluation strategy supports lazy semantics, parallel conditionals, and the breadth-first traversal of the expression structure needed for distributed evaluation. Chapter 4 describes how this evaluation strategy is extended to a distributed system.
3.7.1. Actions

STARDUST uses call-by-need parameter passing. The standard way to implement sequential non-strict semantics with call-by-need parameter passing on a reduction machine is to use "normal order" evaluation [Turner 79]. In normal order evaluation the top level function call is replaced by its definition until the top level call is a system intrinsic. If the system intrinsic is strict in all of its arguments then they are evaluated sequentially using recursive calls to the normal order evaluator; when they have all completed the system intrinsic is invoked to combine them. If the system intrinsic is sequential non-strict then the arguments are again evaluated sequentially with recursive calls, but the invocation of the intrinsic is intermingled with those calls.

The depth-first nature of normal order evaluation makes it ideal for a single processor, since it uses a minimum of auxiliary storage and can implement the auxiliary storage as a simple stack. This very depth-first nature makes it unsuitable to a parallel execution environment for two reasons. First, the traversal order of a parallel system must contain a breadth-first component in order to spawn concurrent subtasks. Second, the purely depth-first strategy makes it impossible to start work on possibly irrelevant subexpressions, since there is no way to return from them if they fail to terminate.

STARDUST implements breadth-first lazy evaluation using structures called actions. Each action has a type, a Node to which the action applies, and information about the status of the task. There are three action types:

1. Print actions cause the system to print an expression.

2. Complete actions cause the system to reduce an expression until it is either a constant or a structure containing possibly uncompleted components. Complete actions are used to implement lazy evaluation.

3. Fully Complete or FComplete actions cause the system to reduce an expression until it is either a constant or a structure whose components are all fully complete. FComplete actions are used at the top level to evaluate expressions that are to be printed. They are also used as part of the exportation mechanism (see Section 4.6.4).

These actions can be thought of as tiny processes, either active or dormant. Active actions are serviced from a queue by a round-robin scheduler, while dormant actions await the termination of other actions or the expiration of timeouts.

Complete and FComplete actions contain a FIRST flag and a WAITALL flag. The type, Node, FIRST flag, and WAITALL flag of action A will be written as A.Type, A.Node, A.First, and A.WaitAll.
A typical action spawns other actions to perform subtasks, waits for some or all of them to terminate, accomplishes its purpose by modifying a Node or printing a value, and terminates. The system keeps track of the creation history by asserting the relation "A₁ wakes A₂" when action A₂ creates A₁. This relation is used to awaken dormant actions: A₂ may be put on the active action queue when A₁ terminates. This same relation is used to keep track of relevance: If A₁ wakes A₂ then A₁ continues to exist because it is needed by A₂.

If the relation wakes is thought of as a set of backpointers from actions to their creators, it can be seen that the control stack of a normal order evaluator has been replaced by a directed acyclic graph (DAG) of actions. The structure is a DAG rather than a tree because actions have the option of using preexisting actions to perform subtasks rather than spawning new ones. This structure involves more space and time overhead than a stack, but in return the system can jump around in response to evaluation heuristics, and it gives natural mechanisms for implementing distribution and the parallel conditional.

### 3.7.2. Evaluation Algorithm

This section contains a summary of the rules for creating, invoking, and terminating actions. The actual implementation code is more complex due to optimizations and extensions for distributed evaluation.

#### 3.7.2.1. Queue Management

Actions may be on only one queue at a time; insertion on one queue implies removal from any others. The system contains an active action queue (AQ) and a terminated action queue (DQ), which are initially empty. When the user types an expression to the interpreter, the expression is converted to a Node N, a call to CreateAction(Print,nil,N) is performed, and the evaluation loop in EvalLoop is repeated until both queues are empty.

**Define** Awaken(A: Action) =
place A at the end of the AQ.

**Define** DoneAction(A: Action) =
place A at then end of the DQ.
Define EvalLoop =
  While True Do
    CheckInMessages:
      While DQ is not empty
        Dequeue its first element;
        Pass it to TerminateAction.
      If Action is not empty
        Dequeue its first action;
        Pass it to InvokeAction.
      Call RehashAll;
    Define CheckInMessages =
      < defined in Chapter 4 >

3.7.2.2. Action Creation

Type AType = (Complete, FComplete, Print)

Define CreateAction(Type:AType; Parent: Action; Target: Node) =
  If Type = Print Then
    Create a print action A and Call Awaken(A).
  Else
    Search for an action A such that A.Type=Type
    and A.Node = Target. | Don't create duplicates
    If none exists Then
      Create a new action A with A.Type=Type
      and A.Node = Target;
      Set A.First to True;
      Call Awaken(A).
    Assert (A wakes Parent).

3.7.2.3. Action Invocation

Define InvokeAction(A: Action) =
  If A.Type = Print Then
    If A.Node is fully completed Then
      Print A.Node;
      Call DoneAction(A).
    Else
      Call CreateAction(FComplete,A,A.Node)
  Else Begin
    | Exit if the evaluation is complete
    If (A.Type=Complete And A.Node is completed) or
      (A.Type=FComplete And A.Node is fully completed) Then
      Call DoneAction(A) and exit.


| Spawn subactions to evaluate the arguments
| If A.First=True Then
| Set A.First to False;
| Set A.WaitAll to correspond with Figure 3-11;
| Call SubAction(<type>,A,<Node>) with the arguments
| given in Figure 3-11.

| Call the function if the arguments are ready
| If the operation can be applied (Figure 3-12) Then
| Call Evaluate(A.Node);
| Call RenewAction(A).
| End;

Define SubAction(Type: AType; A: Action; Target: Node) =
If Type=Complete And Target is completed Or
Type=FComplete and Target is fully completed Then Exit
Else CreateAction(Type,A,Target)

Define RenewAction(A: Action) =
Set A.First to True;
Call RemoveWakers(A);
Call Awaken(A).

3.7.2.4. Action Termination

Define TerminateAction(A: Action) =
RemoveWakees(A);
RemoveWakers(A);
Deallocate A.

Define RemoveWakees(A: Action) =
For all A' such that A wakes A':
If A'.Type = Print Then Awaken(A')
Else
If A'.WaitAll= False Then Awaken(A')
Else If (a wakes A') => a=A Then Awaken(A');
Deassert (A wakes A').

Define RemoveWakers(A: Action) =
For all A' such that A' wakes A:
If A'.Type <> Print Then
If (A' wakes a) => (a=A) Then
DoneAction(A');
Mark A' as aborted;
Deassert (A' wakes A).
3.7.3. Discussion

InvokeAction may create a Complete action, an FComplete action, or no action on each of the arguments. STARDUST uses FComplete actions whenever this is consistent with termination semantics, since this reduces interaction between actions and subactions, reducing message traffic in the distributed environment. Figure 3-11 describes the subactions created by the actions on the various system intrinsics. Complete and FComplete are interchangeable for expressions that evaluate to constants rather than arrays or records.

Awakened actions always look for the actual state of the Node and its subnodes rather than assuming anything from the mere fact of being awakened. While this is a source of overhead, it proved an indispensable simplification in view of the fact that the redundant subcomputation elimination mechanism can change the representation of Nodes and their completion status at any time. For correct operation we need only ensure that each action is awakened at least as often as is necessary. The First flag is an optimization to avoid repeated calls to CreateAction when an action must be invoked several times, and the WaitAll flag is an optimization to avoid waking to check the completion status of the arguments whenever any one of them completes.

The parallel non-strict conditional can be implemented fairly simply on a single processor. Recall that the PNSS conditional evaluates the subexpressions of "If E₁ Then E₂ Else E₃" concurrently, terminating if either (1) E₁ evaluates to True and E₂ terminates; (2) E₁ evaluates to False and E₃ terminates; or (3) E₂ and E₃ terminate with the same result.

To do this, the action associated with the conditional expression creates FComplete actions for all three of its subexpressions, and sets itself to wake up when any one of them completes. If E₁ completes, then the conditional can "execute" normally: it replaces the conditional expression with E₂ or E₃ depending on the value of E₁. If E₂ and E₃ both terminate, the values are compared. If they are identical, the conditional is replaced by this value; if they are different, the action goes dormant again awaiting the value of E₁.

A possible implementation for the PNSS conditional, not followed in STARDUST, shows the flexibility of the action mechanism. The system could assume that most conditionals terminate in the normal way, with E₁ completing first and E₂ not equal to E₃. An FComplete action could be initiated on E₁ with a timeout, and execution initiated on E₂ and E₃ only when the timeout expired.

Most lazy evaluators do not have a separate FComplete mechanism (an exception is the "greedy" evaluator of [Wadler 84]). Instead the print routine makes repeated Complete re-
<table>
<thead>
<tr>
<th>Function</th>
<th>Subactions from Complete</th>
<th>Subactions from FComplete</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Function:</td>
<td>No subactions</td>
<td>No subactions</td>
<td></td>
</tr>
<tr>
<td>Cons:</td>
<td>CA(2)</td>
<td>FC(1), FC(2)</td>
<td></td>
</tr>
<tr>
<td>Concat:</td>
<td>CA(1), CA(2)</td>
<td>FC(1), FC(2)</td>
<td></td>
</tr>
<tr>
<td>If:</td>
<td>FC(1), FC(2), FC(3) *</td>
<td>FC(1), FC(2), FC(3) *</td>
<td></td>
</tr>
<tr>
<td>Sift:</td>
<td>FC(1)</td>
<td>FC(1)</td>
<td></td>
</tr>
<tr>
<td>Choice:</td>
<td>CA(1), CA(2) *</td>
<td>FC(1), FC(2) *</td>
<td></td>
</tr>
<tr>
<td>Access:</td>
<td>CA(1), FC(2)</td>
<td>CA(1), FC(2)</td>
<td></td>
</tr>
<tr>
<td>Sel2:</td>
<td>CA(1), FC(2)</td>
<td>CA(1), FC(2)</td>
<td></td>
</tr>
<tr>
<td>Replace:</td>
<td>CA(1), FC(2)</td>
<td>CA(1), FC(2), FC(3)</td>
<td></td>
</tr>
<tr>
<td>=, ⊙, +, −, *, div, and, or, &lt;, &gt;, Iota, UserIntrinsic, ListToBlock, BlockToList:</td>
<td>FC all arguments</td>
<td>FC all arguments</td>
<td></td>
</tr>
<tr>
<td>Car, Cdr, Map, Reduce, Pair, Cross, Select:</td>
<td>CA all arguments</td>
<td>CA all arguments</td>
<td></td>
</tr>
<tr>
<td>Structures:</td>
<td>No subactions</td>
<td>FC all arguments</td>
<td></td>
</tr>
</tbody>
</table>

* WaitAll is set to False when an asterisk is present

FC(n) indicates a call to SubAction(FComplete,A,Arg. n)
CA(n) indicates a call to SubAction(Complete,A,Arg. n)

* The choice between FC and CA is arbitrary for unstructured arguments such as integers and booleans. The arguments to If are all given FC actions because this parallel non-strict construct must compare the second and third arguments for equality if they terminate before the first. The algorithm uses FC whenever this is consistent with the desired termination semantics.

**Figure 3-11:** Subactions Created by Complete and FComplete Actions

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Function:</td>
<td>No arguments need be complete</td>
</tr>
<tr>
<td>Cons:</td>
<td>CA(2)</td>
</tr>
<tr>
<td>If:</td>
<td>CA(1) or (FC(2) and FC(3) and Arg2 = Arg3)</td>
</tr>
<tr>
<td>Sift:</td>
<td>CA(1)</td>
</tr>
<tr>
<td>Choice:</td>
<td>(CA(1) or CA(2)) for Complete actions;</td>
</tr>
<tr>
<td></td>
<td>(FC(1) or FC(2)) for FComplete actions</td>
</tr>
<tr>
<td>Sel2:</td>
<td>CA(1) and FC(2)</td>
</tr>
<tr>
<td>Replace:</td>
<td>CA(1) and CA(2)</td>
</tr>
<tr>
<td>ListToBlock, =, ⊙:</td>
<td>All arguments fully complete</td>
</tr>
<tr>
<td>Others:</td>
<td>All Nodes complete</td>
</tr>
</tbody>
</table>

**Figure 3-12:** Subexpressions That Must be Complete (CA) or FComplete (FC)
Before a Function can be Invoked
quests to the lazy evaluator for elements of the output list. FComplete actions improve parallelism by creating subactions for all elements of a structure simultaneously, allowing them to be scheduled on different processors. Furthermore, exported FComplete requests relieve the system from the constant interaction that would be required if the components had to be requested separately.

Actions, elsewhere called tasks, are not new with STARDUST. They may be found, for example, in the AMPS interpreter [Keller 78].
Chapter 4
Distributed Evaluation

I will show you fear in a handful of dust.

This chapter presents the features of STARDUST that support distribution. First we describe the mechanisms for transmitting types, function descriptors, and expressions represented in Form/Node Representation (FNR). We describe partitioning, scheduling, and exportation, then discuss the control structure that drives them. Finally we describe the mechanisms that implement fault tolerance.

4.1. System Organization

Decentralized control for automatic distribution systems has been considered in, for example, [Lesser 80, Smith 80]. Despite the appeal of decentralized control for relieving processing and communication bottlenecks, we decided to centralize the partitioning and scheduling in STARDUST. The processor that receives the expression from the user takes on the role of Controller, dividing the expression into tasks and using the other processors as Satellites. This makes it possible to produce schedules that are closer to optimal when good execution time estimates are available, and reduces communication costs such as polling. All processors run the same interpreter and may alternate between controller and satellite status.

STARDUST 1.0 configures the processors as a one level tree or a star (Figure 4-1.a), with the controller at the center. While we have not run any experiments, we believe that a tree (Figure 4-1.b) can be implemented by exporting relatively large subtasks to secondary controllers, which would use the same algorithm to export smaller tasks to the satellites.
4.2. Low Level Distribution Mechanisms

4.2.1. Operating System Support

STARDUST runs under Accent [Rashid 81], a message-based operating system for the PERQ computer. Separate copies of STARDUST run on each participating PERQ, using point-to-point messages for communication. Accent ensures that the messages are transmitted reliably and that each process is informed if any of the other processes terminate, including the important case where the other processor fails in such a way that it no longer responds to network messages. Since exactly one STARDUST process is running on each processor, there is no important distinction between processes and processors, and we will speak of them interchangeably.

Accent provides primitives for sending messages, receiving messages, and testing the
input message queue for waiting messages. STARDUST polls the message queue between function evaluations, with the result that interrupt handling and explicit locking between message processing and function evaluation are unnecessary. Accent network access is transparent (the same procedures are used to transmit messages between processes on the same processor and processes on separate processors), which was a valuable program development aid because it allowed the system to be debugged on a single processor.

4.2.2. Transmitting Expressions

STARDUST does not use a global address space: each processor has its private collection of expressions which are transferred between processors by copying them. The alternative would be to extend the FNR to a distributed representation by replacing the local pointers with "global pointers," each consisting of a processor identifier and a local pointer. While this idea would be attractive on a multiprocessor, the very different costs of interprocessor and intraprocessor communication in a network environment make copying essential. Pointers are generally inappropriate in a network environment because dereferencing them involves a request and a reply, whereas the request can be avoided whenever the computation is data driven.

Nodes are placed into the output buffer by procedure PackNode, and Forms are output using PackForm. Nodes that have been previously sent are retransmitted using a *global name*, which is a unique bit string formed by concatenating the processor identifier with a sequence number. Global names have no particular significance in STARDUST but can shorten messages considerably.

```plaintext
Define PackNode(n: Node; Dest: Processor) =
  If n has a Name known to Dest Then
    Place a ByName flag in the output buffer;
    Place the Name in the output buffer;
  Else
    Place a ByForm flag in the output buffer;
    PackForm(n.BestForm, Dest);
    If n has no Names Then generate one;
    Place one of n's Names in the output buffer and mark it as being known to Dest.
```

Nodes are placed into the output buffer by procedure PackNode, and Forms are output using PackForm. Nodes that have been previously sent are retransmitted using a *global name*, which is a unique bit string formed by concatenating the processor identifier with a sequence number. Global names have no particular significance in STARDUST but can shorten messages considerably.

```plaintext
Define PackNode(n: Node; Dest: Processor) =
  If n has a Name known to Dest Then
    Place a ByName flag in the output buffer;
    Place the Name in the output buffer;
  Else
    Place a ByForm flag in the output buffer;
    PackForm(n.BestForm, Dest);
    If n has no Names Then generate one;
    Place one of n's Names in the output buffer and mark it as being known to Dest.
```
Define PackForm(f: Form; Dest: Processor) =
    If f is remote Then
        Place nothing in the output buffer
    Else Begin
        PackType(f.Type);
        If f is a constant Then
            Place the bits of the constant in the output buffer
        ElseIf f is a structure Then
            Call PackNode on the Nodes that make up the structure
        ElseIf f is a call Then
            PackFunction(f.Function);
            Call PackNode on the argument Nodes;
        ElseIf f is a closure Then
            PackFunction(f.Function);
            Call PackNode on the implicit actual parameters;
        End

4.2.3. Transmitting Function Definitions and Type Declarations

Function definitions and type declarations are transmitted in a completely ad hoc manner by STARDUST 1.0. The system simply assumes that the same file of function definitions and type declarations has been read by all invocations, and transmits these items by sending their sequence numbers.

Define PackType(t: Type) =
    Put t.Sequence in the output buffer;

Define PackFunction(f: FunctionDescriptor) =
    Put f.Sequence in the output buffer;

Function definitions and types can both be thought of as directed, labeled, possibly cyclic graphs, which could be transmitted using the procedure Transmit.

Define Transmit(q: (Type Or FunctionDescriptor)) =
    If q has been sent before Then
        Send its name
    Else
        Generate a name for q;
        Send the name;
        Send the information stored with q;
        Send the number of descendents;
        Call Transmit on the descendents.

Alternatively, function definitions could be represented in FNR and sent using the PackNode and PackForm procedures defined in the previous section.
4.3. Initialization

Invocations of STARDUST identify themselves with unique Process Identifiers, which provide the information necessary for other invocations to send them messages. They initiate communications with Initialize messages, which contain the following fields.

- **SenderID**: The sender's process identifier;
- **OtherID**: A list of the identifiers of all the other processors that the sender knows about.

A processor that receives an Initialize message compares the list with its internal one, updates the internal one by adding any new identifiers, and sends an Initialize message of its own to the processors represented by the new identifiers. This discipline ensures that the relation "knows about" (call it knows) is an equivalence relation when there are no Initialize messages outstanding: it divides the processors into groups where each member of any group knows about all other members of that group. Thus, in order to fully merge two groups, it is only necessary for one member of either group to learn about one member of the other. Under normal operation each processor will know about all the others. When a new process is started it locates another one through a global name server, polling, or advice from the user, and starts a volley of Initialize messages that acquaint it with the others.

The relation knows is by definition reflexive. Knows is symmetric as well, since the first thing a process A does when it learns of process B is to send B an Initialize message containing A. To prove transitivity we need to show that (A knows B and B knows C) implies A knows C. The symmetry of knows ensures that at some point B knows A will become true. There are three possible orders in which B can learn about A and C, each of which eventually produce the result that A knows C:

1. If B first learns of A, then B will send a message containing A to C when B learns of C; C will send a message to A telling A about C; and A knows C will become true.

2. If B first learns of C, then B will send a message containing C to A when B learns of C; and A knows C will become true.

3. If B learns of A and C from the same initialization message, B will send a message containing C to A (as well as a message containing A to C); and A knows C will become true.
4.4. Partitioning

Expressions to be evaluated are partitioned (divided into subexpressions with appropriate execution time estimates), scheduled (assigned to processors), and exported (sent to the processors). This section and the next two describe these steps.

The system tries to export Nodes with intermediate execution time estimates. If the execution times are too small, communication costs will dominate performance; if they are too large, load balancing and redundant subcomputation elimination will be impaired.

User functions with large execution times are divided up by passing them to Evaluate, in effect moving to a lower level of abstraction. In addition, certain of the list operators are reorganized to divide them into smaller units. This usually corresponds to partitioning loops in an imperative program, as in Gilbert's approach to breaking up programs for Lawrence Livermore National Laboratory's S1 multi-processor [Gilbert 83]).

4.4.1. The Partitioning Algorithm

A description of the partitioning algorithm follows in which, for brevity, the "execution time of the Nodes being sent to the satellites" will be referred to as the "size of the units." The parameter TTime in the following code is the largest allowable unit size, also called the partitioning granularity parameter. Section 4.7 describes how this and other pseudo-code is merged with the code from Chapter 3.

Define Partition(A:Action) =
    If A.Node.BestForm is a call to a user function Then
        Call UserSplit(A)
    Else If A.Node.BestForm is a call to MAP Then
        Call MapSplit(A)
    Else if < ... other list operators ... >
        Else do nothing.

Define UserSplit(A:Action) =
    If A.Node.Time is not completed Then
        CreateAction(FComplete,a,A.Node.Time);
        Exit(InvokeAction)
    Else If A.Node.Time > TTime Then
        Call Evaluate(A.Node);
        Exit(InvokeAction)

This algorithm tries to enforce a maximum size for the units. This heuristic was chosen (rather than enforcing a minimum size, for example), for three reasons.

First, redundant subcomputation elimination is most effective when as many expressions
as possible reside on a single processor. This fact does not translate into a quantitative statement about the proper size for units, since the importance of RSCE depends heavily on the program being executed, but it leads to the principle that programs should be partitioned as far as possible before the overhead associated with each unit (scheduling, message passing, and interpreter overhead) degrade performance excessively.

Second, the parallelism in some problems exists only at relatively low levels of granularity. For example, a program might consist of a series of large computations that must be executed in sequence, although each computation can be broken down into components that can run in parallel. This fact also leads to the principle that programs should be partitioned as far as possible.

Finally, it is easier and more efficient to implement a maximum size than a minimum size for the units. To enforce a minimum size the system would either have to get information from the programmer about how finely the user functions divide up when expanded, or discover this information itself by provisionally expanding them and restoring the unexpanded version when the resulting expression consisted of excessively small units.

Considerations governing the choice of a value for TTime are given in Section 5.4.

4.4.2. List Operators

STARDUST is designed to contain code for special treatment of the list operators MAP, PAIR, CROSS, and REDUCE. List operators replace most of the standard control structures in imperative languages, and so are well worth the effort of special treatment. Since the considerations for MAP, PAIR, and CROSS are virtually identical and those for REDUCE are quite similar, only MAP was implemented in STARDUST 1.0 and will be used in all our examples.

MapSplit ensures that the time estimates are available, then passes the Node to MapDivide to turn it into the concatenation of several MAPs with shorter lists.

Define \textbf{MapSplit}(A: \textit{Action}) =
\begin{verbatim}
L := A.Node.BestForm.Args[1];  | The list
f := A.Node.BestForm.Args[2];  | The closure
T := f.BestForm.Function.Time; | The time estimate for f
\end{verbatim}

\textbf{If} T \textbf{is} a constant \textbf{Then}
\begin{verbatim}
\textbf{If} L \textbf{is} completed \textbf{Then}
\textbf{MapDivide}
\textbf{Else}
\textbf{CreateAction} (Complete,A,L);
\textbf{Exit}(InvokeAction);
\end{verbatim}
\textbf{Else}
Else If T is a polynomial Then
  If L is fully completed Then
    MapDivide
  Else
    CreateAction(FComplete,A,L);
    Exit(InvokeAction);
  Else
    Perform the Map on the controller.

Define MapDivide =
  <ensure that the time estimate for each unit is
  less that TTime.>

Constant time estimates are treated by first completing the list (it is not valid to fully
complete the list because neither the time estimate nor the function itself are known to need
the values of the list elements). At this point the length of the list and the time estimate for
each element are known, and the call to MAP can be partitioned.

MapSplit treats constant and polynomial (of degree greater than zero) time estimates as
special cases. Polynomials are important because they are common in practice and:

1. The argument of a polynomial is always known to be needed in order to calculate
   the value of the polynomial;

2. The polynomial can be evaluated without creating FNR or calling the interpreter.

Polynomial time estimates are treated by first fully completing the list, which is valid
because the time estimate is known to need the values. (The list elements will evaluate to
integers, so fully completing the list is identical to simply completing each list element.) At this
point the time estimate for each element is known and the call to MAP can be partitioned. In
exchange for having all of the time estimates available when partitioning takes place this
scheme introduces some synchronization, since all list elements are fully completed before
the function is applied to any of them.

If a compiler were available then any combination of strict operators would satisfy both
conditions and could be treated in the same manner as polynomials.

More complicated time estimates simply cause the MAP to be executed on the controller:
the list is completed and a call to the function is created for each element of the list. These
calls are then partitioned and scheduled separately.

While PAIR and CROSS are very similar to MAP, the left-associative list operator
REDUCE(L,I,f) presents some special problems. Without some restrictions on f it is difficult to
do time estimation or partitioning, since the time estimate for any invocation of f can depend
on the results of all previous invocations. However, if the time estimation function $f_T$ is independent of the first argument of $f$, then the estimate for the REDUCE can be calculated exactly as the calls to MAP described above.

REDUCE is inherently serial, since in general $f$ is not an associative operator. As a result, there are no advantages to be gained from doing the work on more than one processor. However, REDUCE applied to an associative operator, perhaps indicated by a special ASSOCIATIVEREDUCE, could be divided up in a manner entirely analogous to MAP. The partial results of an ASSOCIATIVEREDUCE could be combined as a binary tree.

STARDUST 1.0 treats IOTA as a special case, first completing its arguments and then treating it as another representation for a list of integers. MapDivide splits MAP applied to IOTA into the concatenation of several calls to MAP with IOTAs of smaller range, causing less structure to be allocated on the controller and shorter messages to be sent to the satellites than if the calls to IOTA were expanded.

4.4.3. Time Estimates

User functions are annotated with time estimates using the following syntax:

Let MyFunction(x: Integer) = Begin
   Time $f(x)$;
   <function body>;
   End;

The time estimate can be any integer-valued STARDUST expression, and can use the parameters of the function. A number of decisions had to be made concerning the semantics and usage of these time estimates.

Should the time estimate include the time to evaluate the arguments? There is no way for the designer of the function to know how long it will take to evaluate the arguments, so the answer is clearly "no." However, it remains an important question whether the time estimate stored in the FNR at a particular Node should reflect only the evaluation time for the function (making it a partial time estimate, or PTE, as in Figure 4-2.a), or whether it would be more appropriate to include the evaluation time for the arguments (making it a total time estimate, or TTE, as in Figure 4-2.b). The PTE is used to make load balancing decisions, while the TTE is useful for deciding which branch of CHOICE constructs to use. Neither is trivial to calculate from the other, since the FNR allows sharing of common subexpressions. We chose to store PTEs because

- TTEs are not needed in STARDUST 1.0;
• PTEs can be calculated rapidly by taking them directly from the function's time estimate;
• TTE would have to be continuously updated as redundant common subexpressions were eliminated.

Both figures show times estimates for $f(g(h(1),h(2)),g(h(2),h(3)))$, where $f$, $g$, and $h$ were given time estimates of 10 by the user. Note that the total time estimate at the top of Figure (b) is 60 rather than 70, since the contribution from $h(2)$ is only counted once.

**Figure 4-2:** (a) Partial Time Estimates (PTE)  
(b) Total Time Estimates (TTE)

_Can time estimates affect the termination semantics of the language?_ Time estimation functions can alter the termination semantics of the language by either failing to terminate themselves or by using parameters of the function that the function itself would not always cause to be evaluated. There are three possible definitions of termination semantics when the body of the function terminates but the time estimate does not; STARDUST 1.0 uses the first.

1. The function call may or may not terminate depending on whether the interpreter chooses to evaluate the time estimate.

2. The interpreter guarantees termination by providing a time limit for the evaluation of time estimates. When the time limit is exceeded, abort evaluation of the time estimate and proceed as if no time estimate had been given.

3. The interpreter guarantees that the call will not terminate by evaluating time estimates even when their value is not wanted.
Is the time estimate the maximum, the expected value, or some other summary of the actual time? The time estimate is interpreted as an expected value, in the sense that the scheduler works on the assumption that the actual execution time will be equal to the estimate.

Do time estimates specify the time to Complete the expression or to Fully Complete the expression? It is interpreted as the time to fully complete the expression, since this is likely to be the most common situation, but in principle an estimate for both should be allowed.

Should time estimates on user functions take into account the redundant subcomputation elimination (RSCE) that may occur at lower levels of abstraction? The system will perform best if the time estimate reflects the actual execution time, but there is no way for the user to predict what effect RSCE with other expressions will have on the execution time. Note that both sources of unpredictable execution times, RSCE and subexpressions that are never demanded, reduce the execution time. Thus it would be reasonable for the user to provide a maximum execution time if STARDUST were ever used for deadline scheduling.

What should the system do when time estimates are not provided? We are concerned here with cases where the execution time estimates are genuinely unknown. Special syntax can be used for cases when the execution time is trivial, and cases when it is unspecified because the user wished it to be found at a lower level of abstraction by expanding the function. Three approaches are possible; STARDUST 1.0 uses the third.

1. Go to a lower level of abstraction, even though the user did not specifically request it.

2. Assign the tasks of unknown duration equally among the processors. The scheduler should be aware of the uncertainty and avoid scheduling too far in advance, retaining tasks for those processors that finish early.

3. Assign a default constant time estimate to all Nodes that lack one. If this is larger than the partitioning threshold, then this approach reduces to the first one. If the default is large but not above the threshold, then this approach will tend to reduce to the second one, although the system will not be aware that it is scheduling under greater uncertainty.

4.4.4. Deducing Time Estimates from Other Time Estimates

Future implementations may deduce the computational requirements of functions without time estimates from the time estimates of the parent Form. In general it will not be possible to deduce the actual time estimate, but only to place limits on it. For example, if f(1) has a time estimate of 10 time units and expands to g(h(1),k(2)), then it can be deduced that g,
h, and k will each need less than 10 time units. This fact could be conveniently added to the
Nodes at the time when f(1) was expanded.

Deeper deductions could be made if an accurate record could be kept of the actual
processing time spent on each Form. In the preceding example, the system knows that the
sum of g, h, and k is less than 10, and so can infer a smaller maximum execution time on one
when either of the others accumulates actual processing costs.

Collection of accurate processing costs would have additional advantages: it would
make automatic collection of time estimates possible and allow timeouts on the evaluation of
expressions to reflect run time rather than real time. The statistics could also be used for
deciding when to checkpoint values (see Section 4.8) and when to send them to central
repositories for redundant subcomputation elimination across machines (see Section 7.3.2).

Forms would be used as the basis for this accounting. Each Form would contain a
pointer to the list of Forms that were expanded to create it. (This must be a list because
Forms can be merged during RSCE.) Note that this is a different hierarchy from the "part of"
hierarchy currently in existence. When the system spends time on a Form it must sub-
sequently add the amount of time spent to the time usage fields of all ancestors. A mark/scan
algorithm would have to be used in order to avoid multiple increments to Forms that are
ancestors by more than one path.

4.5. Scheduling

The scheduling algorithm, known as ABS, is a simple modification of a standard greedy
scheduling algorithm. The standard algorithm schedules expressions from the bottom up,
scheduling a computation only after all of its subcomputations have been scheduled. It looks
for the processor that can fit the computation into its schedule as soon as possible after all
subcomputations are complete. Depending on the level of sophistication, the algorithm can
look through the schedule of each processor for holes, or only consider putting the new
computation on the end. It can optionally model the communication costs and incorporate
backtracking [Kazar 84]. Sample output from such an algorithm appears in Figure 4-3-a.

Since STARDUST evaluation must occur top-down in order to implement lazy semantics,
the scheduling must occur top-down as well. The standard scheduling algorithm is applied to
the expression using the back pointers instead of the forward pointers (reversing the pointers
on a directed acyclic graph (DAG) yields another DAG). The computations are scheduled
from some hypothetical completion time working back towards the present, rather than from
the present toward the earliest possible completion time. Computations are scheduled at the
The digits refer to the order in which the computations were processed by the scheduler. Time goes up from the bottom of the figure.

Figure 4-3: Schedules Produced for \( f(p(h,k),q(h,k)) \)
(a) by the Standard Scheduler and (b) by ABS

We have chosen to base ABS on a simple version of the standard algorithm with no backtracking and no search for holes in the schedule. Communication costs are modeled by message passing overhead rather than by latency.

The partitioning and scheduling algorithms have been shown to produce reasonable schedules for some test cases (see Chapter 5). Nevertheless, there is room for improvement, both in the system's model of the environment and the system's tactics for partitioning and scheduling the program to obtain good performance in that environment.

4.5.1. Improving the Model of the Environment

Deficiencies in the system's model of the environment fall into two categories: the system's model of the message passing costs and the system's model of its own execution mechanism.
Experiments showed that the system's model of message passing costs is oversimplified in that it makes no distinction between latency (the time between the sending of a message and its receipt) and overhead (the amount of processor time spent by the sender or receiver in passing the message). Overhead is the better measure of the true cost when both the controller and the satellite have tasks to perform while the message is being sent, while latency is better when both must remain idle.

The system models message passing costs as constant, whereas in reality the overhead and the latency per message increase with the density of the message traffic. In moving from a two-machine system to a six-machine system the overhead per message increased by fifty percent and the latency increased even more. While no definitive tests were performed to prove this hypothesis, we believe that this effect was caused by the increase in lost packets and retransmissions. This effect will of course vary among operating systems and hardware configurations.

The system also models message passing costs as constant with respect to message length. This is actually not a bad approximation, since one of our tests showed that the overhead per message only decreased by about a third when the average message size was decreased from about 350 words per message to less than 50. Nevertheless the dependence of overhead on size could be a critical factor under other operating systems, or even under the present one for large data structures. Unfortunately there are two factors that would make it difficult for the scheduler to predict the message sizes. First, STARDUST keeps track of the objects that have been sent to other machines and uses global names for subsequent transmissions (see Section 4.2.2). Second, the size of objects cannot in general be determined from their type, since the length of lists is not part of their type, and since records and lists can be recursive. It would be possible for the scheduler to use the actual size when it was available and a default otherwise; the frequency of available sizes could be increased by making the size an optional part of list type declarations.

In addition to having an inexact model of the message passing mechanism, STARDUST has an inexact model of its own evaluation mechanism in that (1) It assumes that actual processing is bottom-up, whereas at least some of the processing is generally top-down; (2) It assumes that if a remote processor is given a set of expressions that can be evaluated immediately they will be evaluated serially in the order they were dispatched rather than with the concurrency that the action mechanism implies; and (3) It tries to ensure that branches of the computation end together, whereas when the expressions are actually executed the branches of the lowest level of the computation will actually start together.

STARDUST 1.0 partially addresses the second issue by inserting actions with trivial time
estimates into the schedule at the point when their values are needed, rather than always placing them at the bottom. This introduces a small amount of backtracking into the otherwise purely greedy algorithm. For example, in Figure 4-3-b, if a third argument to q had been needed, and this argument had a small time estimate, it would have been inserted between the previously-scheduled computations q and k rather than placed under k. This in turn would allow arguments to this new computation to be scheduled later (higher in the figure), and a tighter schedule would be produced.

4.5.2. Improving the Scheduling Algorithm

In addition to a more exact model of the execution environment, more sophisticated scheduling strategies would probably be beneficial.

One important strategy missing from STARDUST 1.0 is the ability to back up from mistakes in the partitioning process. STARDUST 1.0 examines an expression, decides whether it is a good unit to export (whether its expected execution time is small enough), and breaks it up if not. It assumes that the resulting expression will always be better than the original, while in reality it may be worse. For example, the expression may be broken up into an excessive number of small ones that will have to be evaluated locally to avoid communication overhead, or it may be broken into a tiny expression and one that will have to be broken up again because it is almost as large as the original. It should be possible to compare the original and the result, keeping the better one.

Another missing strategy is the special treatment of combinations of list operators. Consider the following example:

\[
\text{Map(Map(L, f), g)};
\]

STARDUST 1.0 does not recognize that the nested MAPs are equivalent to the expression:

\[
\text{Begin}
\begin{align*}
\text{Let } & h(x) = f(g(x)); \\
\text{Map(L, h);} \\
\text{End;}
\end{align*}
\]

and so partitions and schedules the list twice. (See [Wadler 84] for a system that interprets such combinations efficiently.) The other pattern that arises frequently and should probably be treated as a special case is:

\[
\text{Reduce(Map(L, f), z, g)}
\]
If \( g \) is associative and \( z \) is the identity element for \( g \), then this fragment could be transformed by the partitioning algorithm to the following expression, where \( L_1 \) through \( L_n \) are sublists (not elements) of \( L \).

\[
\text{Reduce}([\text{Reduce}(\text{Map}(L_1, f), z, g), \ldots, \text{Reduce}(\text{Map}(L_n, f), z, g)], z, g)
\]

The scheduling algorithm in STARDUST 1.0 is largely restricted to creating schedules and assuming that they will be carried out as planned. The only direct feedback is a side effect of the mechanisms for ensuring correct termination semantics and fault tolerance: the scheduling processor is eventually notified when expressions given to other processors terminate. Better performance in the face of inaccurate or missing time estimates could be achieved if the satellite processors reported back large anomalies between the actual and projected execution times.

These reports are more likely to be worth their cost in a tightly coupled system. In the limit of very fast communication, such as in a multiprocessor, the predictive scheduler could be done away with entirely and replaced with polling and opportunistic scheduling. In a more loosely coupled system the feedback would have no effect when the time estimates were accurate but would provide the scheduler with important information when the time estimates were inaccurate.

Some problems were caused by the fact that we allowed computations to occur on the same processor that was doing the scheduling. We felt that it was important to allow this, not only because this processor might constitute a sizable proportion of the total processing power in small configurations, but because performing small computations locally is an important optimization. One such problem was purely representational: the partitioning algorithm would divide up a list of expressions into a set of smaller lists of size suitable for exportation, and join them together with CONCAT operators. The CONCAT operators were sometimes scheduled locally and executed immediately, turning the smaller lists back into a single large one. A more fundamental problem stemmed from the fact that scheduling and evaluation are driven by the same action queue mechanism: it is important to avoid the situation where the controller does long numerical computations while the satellites wait for tasks. We partially solved this problem by putting scheduling and evaluation actions on queues of different priority, but this failed to be a complete solution because it was not always possible to decide which queue the action belonged on.

Finally, the communication costs should have been integrated into the schedule in a
more sophisticated manner. The system currently keeps track of the amount of communication overhead that each processor will incur, and simply assumes that it will take place before any other computation can begin (it would put a block of time for communication just below k and h in Figure 4-3-b). When new tasks are added to the beginning of the schedule, the communication time block is moved to be under them. This simple scheme leads to excessive prejudice against scheduling tasks on processors that have high communication loads, since in reality the communication events may occur in otherwise idle slots in the schedule.

4.6. Exportation

Exportation is the process of giving an expression to another processor to evaluate. This section describes the mechanisms needed to support exportation.

4.6.1. Actions

Four new fields are added to Complete and FComplete actions. The Boss field indicates on which processor the action is to be processed, and the RequestedBy field indicates which processor placed it there. Both are initialized to zero, indicating that the action has neither been scheduled nor dispatched. In addition, the SendTo and SendName fields indicate which processor the result should be sent to and the global name that should be sent with the result to identify it. SendTo is initialized to zero, indicating that the result is not needed by another processor.

4.6.2. Messages

PleaseExecute messages ask another processor to evaluate an expression and return the result in a NewNodeValue message. PleaseExecute messages contain the following fields:

- SenderID: The sender's identifier;
- ToEval: The Node to be evaluated, represented in FNR;
- ActionClass: An indication of whether to Complete or Fully Complete the Node;
- SendTo: The processor to which to send the result;
- SendName: A global name to return with the result.

NewNodeValue messages return the value of an expression. These messages are the standard response to PleaseExecute messages, and consist of the following fields:
- **SenderId**: The sender's identifier;
- **SendName**: The global name specified in the PleaseExecute message;
- **Value**: The value, represented in FNR.

*LostInterest* messages are sent when a processor decides that it no longer needs the value of a Node. LostInterest messages consist of the following fields, which contain enough information to match them to the action that they are designed to abort:

- **SenderId**: The sender's identifier;
- **ActionClass**: An indication of whether the Node was to have been Completed or Fully Completed;
- **SendTo**: The processor to which the result would have been sent;
- **SendName**: The global name that would have been returned with the result.

The system was designed so that receiving a message several times in a row will have the same effect as receiving it only once. If LostInterest messages could be eliminated then the messages would also be order independent, which would leave unusually lenient requirements for the transport mechanism. As it is, PleaseExecute and LostInterest messages must not be received out of order, since the LostInterest messages are used to cancel the PleaseExecute requests. This idempotence and order independence are not important in STARDUST 1.0, since Accent ensures that messages arrive in order and are not duplicated.

### 4.6.3. Procedures

The following procedures support exportation.

Define Export(A:Action) =

Send a PleaseExecute message to A.Boss, setting ToEval to A.Node using global names for A.Node's arguments (see below), and taking ActionClass, SendTo, and SendName from the corresponding fields of A.

Call CreateAction(FComplete,A.Node,N), for each Node N that is a subnode of A.Node.

For each of these new actions

* Set the SendTo field to A.Boss;
* Set the SendName field to the global name used in the first step of Export.

Mark A as a backup action.
Define CheckIncomingMessages =
   If there is a message on the queue Then
      Set Msg to be the message:
      If Msg.Type = Initialize Then
         Add new processor identifiers to the internal list;
         Send Initialize messages to new processors.
      ElseIf Msg.Type = NewNodeValue Then
         Set N to the Node corresponding to Msg.SendName;
         Call MergeNodes(N,Msg.Value).
      ElseIf Msg.Type = LostInterest Then
         For all actions A with ActionClass, SendTo, and
         SendName equal to the corresponding fields of Msg,
         If A is a Normal action Then
            Mark A as Aborted;
            Call DoneAction(A).
         ElseIf A is a Backup action Then
            Forward the LostInterest message to A.Boss.
      ElseIf Msg.Type = PleaseExecute Then
         A := CreateAction(Msg.ActionClass, nil, Msg.ToEval);
         A.RequestedBy := Msg.SenderID;
         A.SendTo := Msg.SendTo;
         A.SendName := Msg.SendName;
         For all N' such that N' is a subnode of Msg.ToEval,
         Call CreateAction(FComplete, Msg.ToEval, N');
         Mark the action as a Shadow.
   
Define TerminateAction(A:Action) =
   RemoveWakees(A);
   RemoveMakers(A);
   If A is a Shadow action Then
      If A is marked as Aborted Then
         Send a LostInterest message to A.RequestedBy, filling
         the corresponding fields with A.SendName and A.Node;
   ElseIf A is a Normal action Then
      If (A.SendTo<>0) And (A.SendTo<>MyID) And
         A is not marked as Aborted Then
         Send a NewNodeValue message to A.SendTo, filling
         the corresponding fields with A.SendName and A.Node
      Deallocate A.

Backup and shadow actions are used for aborting actions that have been exported
(Section 4.6.4) and for recovering from processor failures (Section 4.8).

Export only sends the top level call of an expression to the satellite processor. For
example, if Export is called with the Node representing f(g(1),g(2)), the call to f is sent to the
satellite while the calls to g remains on the controller. Since the satellite processor does not
evaluate the arguments, it does not need to know any Form for them, and the controller can
use a special version of PackForm that always uses global names for the arguments. These same global names are placed in the SendName fields of the actions attached to the argument Nodes so that the satellite can recognize the values when they arrive.

Since several requests for remote evaluation may be outstanding at a given time, it is necessary to include with the reply some indication of which expression it is the value of. While this could have been done by sending back the original expression with its value, shorter messages result from using the global name mechanism used to shorten the transmission of FNR. This is purely a performance consideration: while STARDUST 1.0 exchanges the messages

- Controller request: Evaluate Form F, which is attached to a Node that has the global name N. Send the results to processor P.
- Satellite response: The Node with global name N has value V.

A version of the system that does not use global names would exchange the following messages:

- Controller request: Evaluate Form F, and send the result to processor P.
- Satellite response: The Form F has value V.

4.6.4. Overeager Evaluation

Overeager evaluation is a technique for reducing the number of messages passed when expressions are evaluated. It is initiated when procedure Export attaches FComplete actions to the arguments of the functions that it exports, on the temporary assumption that these functions have strict semantics. If this turns out to be incorrect, a clean-up phase preserves lazy semantics and ensures that unnecessary computations are aborted.

Placing an FComplete action on an argument has two separate effects. First, it causes the argument to be evaluated, including all of its components if it is a structure, without waiting for demand messages. This "demand effect" introduces a data-driven component into the otherwise purely demand-driven architecture of STARDUST. If it initiates no unnecessary computations, then it reduces message traffic by eliminating the need for demand messages, and increases concurrency by allowing tasks to be started before their demand messages would have been received. The second effect, called the "report effect," is that the processor reports the results of the computation only when the entire computation has completed, rather than passing the components in individual messages as they complete. This can reduce message traffic by causing structures to be sent in single messages, but can inhibit concurrency by delaying tasks that need some but not all of the components of the structure.
In the following discussion the controller has requested that a satellite processor ("Sat-2") evaluate an expression E in order that a third processor ("Sat-3") may evaluate f(E) (see Figure 4-4).

Before processing:

\[
\begin{align*}
\text{Controller} & \quad \text{Sat-2} & \quad \text{Sat-3} \\
\text{f(E) [normal action]} & & \\
\end{align*}
\]

After the Controller sends f to Sat-3

\[
\begin{align*}
\text{f(E) [backup action]} & \quad \text{f(E) [normal action]} \\
\text{E [normal action]} & \quad \text{E [shadow action]} \\
\end{align*}
\]

After the Controller sends E to Sat-2

\[
\begin{align*}
\text{f(E) [backup action]} & \quad \text{f(E) [normal action]} \\
\text{E [backup action]} & \quad \text{E [normal action]} \\
\text{E [backup action]} & \quad \text{E [shadow action]} \\
\end{align*}
\]

Figure 4-4: Actions created when the Controller requests Sat-2 to evaluate E and Sat-3 to evaluate f(E)

4.6.4.1 Irrelevant Computations

The first problem caused by overeager requests is that extra computations must be detected and aborted. This is done with backup actions, shadow actions and LostInterest Messages. For simplicity we shall not consider the case where the controller schedules computations on itself, since these are done without backup or shadow actions for performance reasons. The algorithm in STARDUST 1.0 therefore differs from the one presented here, and in particular follows a more conservative policy that sometimes causes extra LostInterest messages to be sent by the controller.

Backup actions form a record of the actions that the controller has exported. When a Normal action is passed to procedure Export, it is changed to a Backup action after the PleaseExecute message has been sent.

Shadow actions are created by the recipient of the PleaseExecute message. One shadow action is created for each of the arguments that the processor expects to receive, and terminates normally when the argument's value is received. If the function terminates before all arguments have been received, the shadow actions attached to these arguments abort, causing LostInterest messages to be sent to the controller, which forwards them to the processors that were evaluating the arguments. These messages would be sent directly to
the processors evaluating the arguments but for the fact that satellites are in general ignorant of the source of the values of their arguments due to top-down scheduling (see Section 4.5).

4.6.4.2. The Report Effect

The second problem caused by overeager evaluation is that Sat-2 does not report any results from an FComplete request until the entire expression has been evaluated. If Sat-3 does not use all components of the structure, this can lead to a long or even infinite wait for information that is not needed to achieve the system's goals. We shall describe the current solution to the problem and steps that could be taken to improve it.

When the controller sends the request to evaluate \( E \) to Sat-2, it sets a timer to a predetermined value. If \( f(E) \) is completed before the timer fires, the timer is simply turned off. If the timer fires before \( f(E) \) is completed, the controller reschedules \( E \) on Sat-3. Since \( E \) is now being calculated on the same processor as \( f \), \( f \) can use components of \( E \) as soon as they become available.

There are two difficulties with this scheme. First, instead of a real time clock set to a predetermined value, it would be much better to accurately account for the processing time that has been spent on \( E \) (as proposed in Section 4.4.4), and use an overtime value that was either directly supplied by the user or estimated from the expected execution time of \( E \). Second, it would have been more efficient to change the reporting behavior of Sat-2 rather than giving the responsibility for evaluating the entire structure to Sat-3. After timing out, Sat-2 could eliminate the report effect by sending the structure to Sat-3 with "remote" Nodes for the components that were not yet evaluated (sending global names rather than Forms), and passing these components as separate messages as they were calculated.

4.6.4.3. Optional Overeager Evaluation

STARDUST 1.0 demonstrates the feasibility of overeager evaluation. While STARDUST 1.0 uses it as the default parameter passing mechanism for exported function calls, a more practical system would include it as an option along with call-by-value and call-by-need. Options for controlling the value of the timer and separately enabling the demand and report effects should also be provided. Sample syntax for these pragmas is presented in Section 7.2.1.

The beneficial effects of overeager evaluation can be seen from the absence of special demand messages in the experimental results of Chapter 5. A sample of recovery from a timeout appears in Section 5.5.
4.6.5. Conditionals

Many uses of conditionals, including some of the most difficult to partition and schedule properly, are made unnecessary by the list operators. For example, although suitable extensions to the language would have allowed us to define MAP in the following manner, such a definition would have hidden from the system some of MAP's essential properties, or at least would have made the system discover them from scratch.

Let Map(L: List; f: Function): List =
  If L=nil Then nil
  Else Cons(f(Car(L)), Map(Cdr(L), f));

As with all other functions, the arguments of conditionals are given FComplete actions when the conditional is scheduled on another processor. This is nothing new for the parallel conditional, which initiates FComplete actions on all of its arguments in any case. For the sequential conditional it amounts to initiating both branches of the conditional and aborting the one that turns out not to be needed. While this makes sense in an environment where there are plenty of idle processors, it can flood a busy system with irrelevant computations. Section 7.2.4 presents a priority scheme in which computations initiated through successive layers of unresolved conditionals receive successively lower priorities.

4.7. Control Structure

Partitioning, scheduling, and exportation are enabled by adding the following conditional clause to the definition of InvokeAction (Section 3.7.2) before the test for A.First = True.

If A.Boss=0 Then
  Partition(A);
  Schedule(A);
If (A.Boss <> 0) And (A.Boss <> MyID) Then
  Export(A);
  Exit(InvokeAction);

While this control structure ensures that partitioning, scheduling, exportation, and evaluation happen serially for any particular action, it does not ensure the same for the system as a whole. For example, it is possible for one Node to be partitioned after another Node has been scheduled and exported. Global serialization would have the following advantages:

- The interface between the steps would have been more sharply defined, allowing them to be designed and implemented independently.
- The control structure for each step would be simplified, and the current control constructs for switching between them could be replaced with a simple sequencer.
• Each step would have the entire output of the previous step to work with, resulting in improved performance or functionality. For example, the scheduler could produce arbitrarily good schedules if it could schedule the computations together rather than sequentially. As another example, the invocation could pack all requests to a particular processor into a single message, substantially reducing the message passing overhead.

This division into distinct steps was not possible for three reasons:

• Since execution time estimates can be arbitrary STARDUST expressions, the evaluator must be called before certain expressions are partitioned and scheduled.

• Non-terminating computations can have non-terminating partitioning, scheduling, and invocation phases. Thus, if a non-terminating computation is irrelevant to the final result, this fact must be discovered before the partitioning, scheduling, and exportation phases are allowed to run to completion. In general this can only be done by invoking the evaluator.

• Even terminating computations can have infinite partitioning phases if the time estimates are missing or incorrect.

The action queue (AQ) is divided into a high-priority and a low-priority action queue. The low-priority queue is reserved for long computations that have been scheduled on the controller, helping to ensure that the controller does not perform computations when it should be partitioning and exporting expressions to idle satellites.

4.8. Fault Tolerance

Easy recovery from processor failures is one of the principal advantages of an applicative language. Since function calls produce no side effects, expressions can be evaluated once, several times, or even partially executed and restarted, without affecting the final outcome. Because of this, the system can recover by simply rescheduling expressions from the failed processor onto a surviving one.

We assume throughout that the Accent operating system detects processor failures by the processor’s lack of response to network messages. A similar mechanism should be built into the STARDUST interpreter itself to detect those cases where other copies of STARDUST have stopped responding to messages without the processor or the operating system having failed.

The proper response to failures depends upon their frequency. We assume that processor failure is a relatively rare event, and so allow time-consuming operations such as scanning the list of all expressions in each survivor. On the other hand, one possible application of
STARDUST is to perform very long computations. Sufficiently long computations lead to a high probability that at least one processor will fail before the computation finishes, so it is important to ensure that computations are not restarted from scratch every time a processor fails.

Two schemes were considered for restarting computations: a centralized one and a decentralized one. Although the decentralized version eventually had to be abandoned, it had enough advantages that it is worth examining the reasons for its failure. Neither version recovers from failures in the controller.

4.8.1. Decentralized Recovery

The decentralized scheme was based on two conditions that turned out to be difficult to satisfy:

- Each processor holds the entire subgraph of all expressions that it has responsibility for evaluating. In other words, the procedure Export (see Section 4.6) must send subnodes in full rather than using only their global names.
- Each processor knows which processor has responsibility for evaluating the subexpressions that it needs.

The first condition ensured that each processor had enough information to evaluate the expressions for which it was responsible without outside help. The second condition led to a very simple recovery mechanism: When a processor learned that another processor had failed, it simply searched its list of expressions for those being evaluated on the failed processor and requested them elsewhere.

The surviving processors tried to reschedule the failed processor's expressions on the same survivor (they all rescheduled tasks from processor P on processor P + 1 if they believed that Processor P + 1 was still functional). This tended to put a heavy load on that processor, but ensured that if the failed processor was evaluating an expression for several recipients only one copy would be restarted. This is only an efficiency heuristic; if for some reason the survivors had inconsistent views of which processor should be used, redundant copies might be evaluated but the correct value would still be calculated.

Longer messages would have to be sent in order to satisfy the first condition. The fact that the messages are sent by the controlling processor promises to create a bottleneck as well as increased message traffic. The second condition is also hard to meet because it conflicts with the need for top down scheduling: computations must sometimes be scheduled and exported before it is known where their subexpressions will be evaluated. Passing this information as soon as it became available would entail an extra message for each expor-
tation, and still leave a "window of inconsistency" between the time when the expression is received and the time the sources of its subexpressions are received.

While we eventually abandoned this scheme because of the projected communication costs, we feel that it would be effective in configurations where communication costs are relatively low. If fan-out schemes are used for disseminating the expression, then the bottleneck at the controller is avoided and only the overall bandwidth has to be high. Broadcast messages and shared memory would make decentralized recovery particularly attractive.

4.8.2. Centralized Recovery

The centralized recovery algorithm relies upon neither of the conditions of the decentralized one. When a processor fails, all surviving processors scan their list of actions for actions that they themselves exported. In the simple case where only one expression has been given to the system and the system is arranged in its present "star" configuration, only the controller will find any.

The Boss field of all such actions is set to zero, causing them to go through the normal scheduling process. There is no need to notify the survivors that were waiting for values from the failed processor, since they were not aware of where their subexpressions were being evaluated in any case.

Actions that were going to send results to the failed processor are simply aborted. When the actions from the failed processor are re-exported, new actions are created for their subexpressions. A trick not shown in the pseudo-code below causes these new actions to be scheduled on the same processor from which the corresponding actions were aborted. This allows them to use any partial results that were calculated before the failure.

Define FailedProcess(P: ProcessorIdentifier) =
For all backup actions A,
  If A.Boss = P Then
    Set A.Boss to 0;
    Put A on DQ;
  End If

For all actions A,
  If A.SendTo = P Then
    Mark A as Aborted;
    Call DoneAction(A).
4.8.3. Controller Failure

The top level expression presents a special problem to both of the recovery algorithms. If the controller fails then the action that caused the top level expression to be evaluated is lost and, in the centralized scheduler, all of the scheduling information is lost as well. This scheduling information includes the loading estimates of other processors and the places where subcomputations are being performed.

STARDUST 1.0 does not tolerate controller failures. They could be handled simply by duplicating the top level expression, if it is allowable that the computation restart from scratch when the controller fails. If this is not allowed, then either (1) occasional checkpointing of the controller's backup actions must take place, or (2) after a failure the new controller must search for partially completed subexpressions.

It will frequently be the case that when the controller fails there is no longer any need for the value of the expression. For example, if the computation is being performed to update the controller's display, it will no longer be necessary to finish the calculations.
Chapter 5
Experimental Results

... The boat responded
Gaily, to the hand expert with sail and oar

This chapter presents the experimental results of three distributed applications written in a combination of the STARDUST language and PASCAL user intrinsics. The applications chosen were realistic ones with enough large grain parallelism that the techniques employed by STARDUST could achieve significant speedups. This chapter also describes some simple experiments that demonstrate recovery from overeager evaluation, recovery from process failure, and the benefits of redundant subcomputation elimination.

5.1. The Distributed Sensor Network Example

The object location algorithm from the distributed sensor network (DSN) project [DSN 78, DSN 82] is a simple yet realistic signal processing task. The program locates a sound source using time-difference-of-arrival techniques on the signals gathered from a set of microphones. Figure 5-1 shows the flow of information when three microphones are used. The signal is generated by the hardware, all pairs of signals are cross correlated, the peaks are determined, unlikely peaks are detected, and a locator algorithm uses the remaining peaks to determine the location.

5.1.1. The Program

The program is similarly divided into a data generator, a cross correlator, a peak picker, a false peak detector, and a locator. All are implemented as user intrinsics, while the connections between them are expressed in the STARDUST language. In order to reduce the memory requirements of the system (experiments are nevertheless still complicated by paging), and to allow processing times to be artificially varied, the actual signal processing algorithms were replaced by short loops that consumed a similar number of processor cycles: one second for
Each cross correlation plus one second each for the false peak detector and the locator. The other computations were not modelled because their contribution to overall processing cost was not significant.

The time to find the cross correlation of two signals in the original form of the problem depended on the distance between the corresponding microphones. We have simplified the problem by calculating the full cross correlation for each pair. The time estimates for cross correlations, false peak detection, and object location are set to 60 in the program that follows, since time estimates are expressed in units of 1/60 second.

```pascal
Type PhonePair = Record
  First: Integer;
  Second: Integer;
End;
```
Type Point = Block of 3 Integer;
Type Peak = Block of 4 Integer;
Type Signal = Block Of 502 Integer;
Type XCData = Block Of 253 Integer;

Let Phones(n: Integer) = Begin
    Time 1;
    Iota(1,n,1);
End;

Let MakePair(x: Integer; y: Integer) = Begin
    Time 1;
    PhonePair: [First=x,Second=y];
End;

Let CheckPair(p: PhonePair) = Begin
    Time 1;
    p.First < p.Second;
End;

Let MakePeak(Phone1: Integer; Phone2: Integer; Value: Integer) = Begin
    Time 2;
    Peak: List Of Integer:[Phone1,Phone2,Value];
End;

Let PhonePairs(n: Integer) = Begin
    Time 1;
    Select(Cross(Phones(n),Phones(n),MakePair),CheckPair);
End;

Let Data(x:Integer; t: Integer) = [Atomic] Begin
    Time 1;
    UserIntrinsic("GetSignal",x,t):Signal;
End;

Let XC(s1: Signal; s2: Signal) = [Atomic] Begin
    Time 60;
    UserIntrinsic("XC",s1,s2):XCData;
End;

Let PeakPick(x: XCData) = [Atomic] Begin
    Time 2;
    UserIntrinsic("PeakPick",x):Peak;
End;

Let Locate(L: List Of Peak) = [Atomic] Begin
    Time 60;
    UserIntrinsic("Locate",L):Point;
End;
Let Detect(L: List Of Peak) = [Atomic] Begin
    Time 60;
    UserIntrinsic("Detect",L): List Of Peak;
End;

Let Go(n: Integer; t: Integer) = Begin
    Let XCPair(p: PhonePair) = Begin
        Time 60;
        PeakPick(XC(Data(p.First,t),Data(p.Second,t)));
        Time n*n*60;
        Locate(Detect(Map(PhonePairs(n),XCPair)));
    End;

The parameter t that is passed to Go and Data allows several experiments to be run
without restarting the STARDUST interpreter. Identical runs cannot be repeated since the
redundant subcomputation elimination mechanism causes the results to be returned without
recalculation.

5.1.2. Summary of Experiments and Results

The experiments use eight microphones, giving a total of twenty-eight cross correlations.
Since cross correlations, false peak detection, and object location form the bulk of the com-
putation, a total of thirty seconds of useful computation are performed.

All reported execution times are the fastest of five trials, reducing the randomization from
such factors as paging, lost packets, and network traffic from processors not participating in
the experiment. Paging was by far the most serious source of variation, so we developed a
standard testing pattern to minimize it. We ran the first trial, waited for the system to write out
dirty pages, ran the second and third, waited again, and ran the fourth and fifth. The third and
fifth trials consistently had the lowest execution times, and almost always differed from each
other by less than five percent. Trials two and four were typically ten to twenty percent slower
and showed more variation. The first trial always had the longest execution time because it
incurred the overhead of paging in the necessary routines, and because in some of the
experimental variations the call to function PhonePairs was only evaluated once due to re-
dundant subcomputation elimination.

Four variations of the DSN experiment are reported in this section. They are summarized
in Figure 5-2 and described in detail in Sections 5.1.3 through 5.1.6. Execution times and
speedups for the four experiments are presented in Figures 5-3 and 5-4.
Base Experiment: This version is a realistic adaptation of the signal processing routines that were in use before STARDUST was developed, with the exception that actual signal data are not sent between processors. Each cross correlation is scheduled independently.

Optimal Partitioning: Partitioning and communication costs are minimized by causing the cross correlations to be scheduled in groups of optimal size. This experiment shows the best case improvement from treating MAP as a special case in the partitioning algorithm.

Full Messages: The 502-word signals are sent in full to the satellites and the 253-word cross correlations are returned to the controller. This experiment shows performance degradation caused by increased message traffic.

Slow User Intrinsics: The processing time for cross correlations, false peak detection, and object location are multiplied by five, and the resulting execution time measurements are divided by five. Full messages were used as in the previous experiment. This experiment tests the effect of speeding up the interpreter and message passing by a factor of five.

Linear Speedup: Execution time is calculated by dividing the total numerical computation time of thirty seconds (twenty-eight for cross correlating all pairs of eight signals, plus one each for false peak detection and object location) by the number of processors.

Best Possible Speedup: Execution time is calculated by dividing the total cross correlation time of twenty-eight seconds by the number of processors, rounding up, and adding two seconds for false peak detection and object location. The best possible speedup is not linear because the algorithm has some intrinsic granularity (the twenty-eight cross correlations cannot be spread evenly among three, five, or six processors), and some intrinsic serialization (the cross correlations must be performed before the false peak detection, which must be performed before the object location).

*The four experimental variations are described more fully in sections 5.1.3 through 5.1.6.*

**Figure 5-2:** Summary of DSN Experiments and Theoretical Speedups

Figure 5-5 describes the categories used throughout this chapter for reporting processing costs. Figure 5-6 gives a breakdown of the processing costs on the controller for the six-processor version of each experimental variation, and Figure 5-7 gives message passing statistics for the same experiments.

Figures 5-6 and 5-7 summarize the main experimental results of this section. We will discuss the statistics shown in these figures, then give more detailed results for each experiment.
This figure presents evaluation time as a function of the number of processors in four experimental variations. Two theoretical results are added for comparison. The six plots are summarized in Figure 5-2 and explained more fully in the text.

Figure 5-3: DSN Evaluation Times

This figure presents speedup as a function of the number of processors in four experimental variations. Two theoretical results are added for comparison. Speedups are calculated by dividing the execution time into the execution time for the single processor version of the same experiment. The six plots are summarized in Figure 5-2 and explained more fully in the text. The "Optimal Partitioning" and "Slow User Intrinsics" plots overlap to form the middle line on the figure.

Figure 5-4: DSN Speedups
Total Real Time: The elapsed time between starting and finishing the computation. This figure does not include parsing the request or printing the result.

Statistics Gathering: The number of calls to access the process run time clock multiplied by the average real time per call.

Accent Overhead: The difference between the elapsed time and the sum of the following process run times: The message server, the network server, a low priority idle process, and STARDUST itself. This time is mostly accounted for by process swaps and virtual memory management.

Message Server: The run time of the message server process. The message server translates between Accent messages and network packets.

Network Server: The run time of the network server process. The network server handles network communication at the packet level.

Idle Time: The run time of a low priority idle task.

Total Interpreter: The run time of the STARDUST process, adjusted by subtracting the run time incurred by reading the run time clock. This figure is further broken down into the following five categories:

User Intrinsics: The time spent in the User Intrinsics module, which includes the time to unpack the arguments and pack the result.

Message Overhead: The time spent sending, receiving, packing, and unpacking messages. It includes the cost of creating new Forms and Nodes from incoming messages, but does not include the cost of performing the action requested by the message.

Partitioning: The time spent partitioning expressions. Since calls to user functions are partitioned by expanding them, some of the cost attributed to partitioning would have been incurred even if the partitioning step had been eliminated.

Scheduling: The time spent deciding which processor should evaluate each expression.

Other Interpreter: The time spent doing action management, redundant subcomputation elimination, checking for messages, and expression evaluation. Further breakdown of this category was not attempted because the uncertainty introduced by accessing the clock tended to overshadow the small running times of the routines being measured. An experiment with an older operating system (see Section 7.1.1) indicates that about 50% of the "other interpreter" cost is for redundant subcomputation elimination, 20% is for action management, and 10% is for dynamic storage allocation.

Figure 5-5: Categories of Processing Costs
<table>
<thead>
<tr>
<th></th>
<th>Base Experiment</th>
<th>Optimal Partitions</th>
<th>Full Messages</th>
<th>Slow User Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Real Time</td>
<td>13.45 100%</td>
<td>10.74 100%</td>
<td>17.61 100%</td>
<td>8.15 100%</td>
</tr>
<tr>
<td>Statistics Gathering</td>
<td>1.13 8%</td>
<td>0.46 4%</td>
<td>1.48 8%</td>
<td>0.24 3%</td>
</tr>
<tr>
<td>Accent Overhead</td>
<td>3.07 23%</td>
<td>0.82 8%</td>
<td>5.09 29%</td>
<td>0.86 11%</td>
</tr>
<tr>
<td>Message Server</td>
<td>0.46 3%</td>
<td>0.11 1%</td>
<td>1.16 7%</td>
<td>0.25 3%</td>
</tr>
<tr>
<td>Network Server</td>
<td>0.52 4%</td>
<td>0.21 2%</td>
<td>1.18 7%</td>
<td>0.17 2%</td>
</tr>
<tr>
<td>Idle Time</td>
<td>0.11 1%</td>
<td>2.42 23%</td>
<td>0.18 1%</td>
<td>0.02 0%</td>
</tr>
<tr>
<td>Total Interpreter</td>
<td>8.16 61%</td>
<td>6.72 63%</td>
<td>8.52 48%</td>
<td>6.61 81%</td>
</tr>
<tr>
<td>User Intrinsics</td>
<td>5.04 38%</td>
<td>5.01 47%</td>
<td>3.90 22%</td>
<td>5.90 72%</td>
</tr>
<tr>
<td>Message Overhead</td>
<td>0.35 3%</td>
<td>0.41 4%</td>
<td>0.61 3%</td>
<td>0.12 1%</td>
</tr>
<tr>
<td>Partitioning</td>
<td>0.31 2%</td>
<td>0.09 1%</td>
<td>1.06 6%</td>
<td>0.19 2%</td>
</tr>
<tr>
<td>Scheduling</td>
<td>0.24 2%</td>
<td>0.07 1%</td>
<td>0.24 1%</td>
<td>0.04 1%</td>
</tr>
<tr>
<td>Other Interpreter</td>
<td>2.22 17%</td>
<td>1.14 11%</td>
<td>2.71 15%</td>
<td>0.36 4%</td>
</tr>
</tbody>
</table>

This figure presents controller processor costs for four experimental variations with six processors. The variations are summarized in Figure 5-2 and explained more fully in the text. The categories are defined in Figure 5-5. The percentages reported here are of Total Real Time.

**Figure 5-6:** Processor Usage for the Controller in Four DSN Experiments with Six Processors

<table>
<thead>
<tr>
<th></th>
<th>Base Experiment</th>
<th>Optimal Partitions</th>
<th>Full Messages</th>
<th>Slow User Intrinsics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words Sent</td>
<td>1371</td>
<td>385</td>
<td>19062</td>
<td>18452</td>
</tr>
<tr>
<td>Words Received</td>
<td>975</td>
<td>600</td>
<td>7488</td>
<td>6912</td>
</tr>
<tr>
<td>Messages Sent</td>
<td>25</td>
<td>5</td>
<td>26</td>
<td>24</td>
</tr>
<tr>
<td>Messages Received</td>
<td>25</td>
<td>5</td>
<td>26</td>
<td>24</td>
</tr>
<tr>
<td>Data Packets Sent</td>
<td>54</td>
<td>14</td>
<td>90</td>
<td>85</td>
</tr>
<tr>
<td>Data Packets Received</td>
<td>50</td>
<td>10</td>
<td>52</td>
<td>48</td>
</tr>
<tr>
<td>Acknowledgements Sent</td>
<td>50</td>
<td>10</td>
<td>56</td>
<td>48</td>
</tr>
<tr>
<td>Acknowledgements Received</td>
<td>59</td>
<td>19</td>
<td>97</td>
<td>90</td>
</tr>
<tr>
<td>Retransmits Sent</td>
<td>18</td>
<td>10</td>
<td>83</td>
<td>48</td>
</tr>
<tr>
<td>Retransmits Received</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Total Packets Sent</td>
<td>122</td>
<td>34</td>
<td>228</td>
<td>181</td>
</tr>
<tr>
<td>Total Packets Received</td>
<td>109</td>
<td>29</td>
<td>152</td>
<td>138</td>
</tr>
</tbody>
</table>

This figure presents message passing statistics from the point of view of the controller for four experimental variations with six processors. The variations are summarized in Figure 5-2 and explained more fully in the text. "Data packets sent" represents the number of packets successfully sent and acknowledged; the sum of data packets sent, acknowledgement packets sent, and retransmissions sent is the total number of packets sent. The same relation holds for packets received.

**Figure 5-7:** Message Statistics for the Controller in Four DSN Experiments with Six Processors
The most surprising result from the base experiment was that only 4% of the total processing cost was attributed to partitioning and scheduling. A much higher figure of 17% ("Other Interpreter") was used to interpret STARDUST expressions.

While no direct measurements were performed, experience has shown that most of the Accent Overhead cost can be attributed to message passing. Thus the total cost of message passing from Accent Overhead, Message Server, Network Server, and Message Overhead amounts to about 33%.

Figure 5-7 shows that eighteen of the fifty-four data packets sent by the controller had to be retransmitted. The "star" configuration, with a single controller communicating with five satellites, adds to communication overhead by causing acknowledgement packets to be dropped.

Optimal partitioning improved performance considerably. The total real time decreased by 2.71 seconds from the base experiment, or 2.04 seconds if the statistics gathering time is subtracted from both. In addition, the controller had 2.42 seconds of idle time which could have been spent on other tasks. While about half of the reduced cost came from the expected sources (Message Server, Network Server, Partitioning, and Scheduling), the other half came from reduced Accent Overhead and Other Interpreter overhead. The reduced Accent Overhead is due to the fact that most of this overhead is associated with message passing; the reduced Other Interpreter overhead is due to the fact that six lists of results are concatenated together rather than twenty-eight single returns. The array representation of lists makes the twenty-eight calls to CONS a significant expense.

Sending full messages severely degraded performance. The total real time increased by 4.16 seconds over the base experiment, or 3.81 seconds if the statistics gathering time is subtracted from both. Degradation would have been even more severe if an additional cross correlation had not been sent to a satellite (notice that User Intrinsic time dropped from about 5 to about 4 seconds). Most of the degradation was due to increases in Message Server, Network Server, and Accent Overhead. The unexpected increases in Partitioning and Other Interpreter costs are a side effect of the way the algorithm was expressed: more of the interpretive load, in particular the calls to PeakPick, were handled on the controller. The increase in message size also exacerbated the problem of packet retransmission seen in the base experiment: a total of eighty-three retransmissions were needed to send ninety packets.

---

3 For those familiar with Accent, the messages were sent out-of-line, resulting in memory validation and invalidation within the STARDUST program and the message server.
Using a slower version of the signal processing routines had the expected effect of reducing the relative cost of the overhead. The total elapsed real time, when normalized by dividing by five, was 8.15 seconds, just 1.15 seconds over the theoretical minimum. The number of retransmitted packets dropped from eighty-three to forty-eight when compared with the full messages experiment (this experiment also uses full messages), which is probably due to the fact that the messages were spread out over time.

5.1.3. Base Experiment

The actual and estimated execution times for cross correlations, false peak detection, and object location are each set to one second. The time limit above which user functions are partitioned (the constant TTime, used by procedure UserSplit in Section 4.4 and MapSplit in Section 4.4.2) is set to 100 "jiffies" (100/60 or 1.67 seconds), which causes the cross correlations to be scheduled independently since 1.67 seconds is smaller than the execution time for two of them together. The signal length is set to 502 words (500 words of data plus 2 words of identifying information) and the cross correlation length to 253 words (250 words of data plus 3 words of identifying information). The system assumes a message passing overhead of 5 jiffies, or 0.08 seconds.

The Forms sent from the controller to the satellites have the structure \texttt{xcpair([n_1,n_2],t)}, where \texttt{n_1} and \texttt{n_2} are integers indicating which signals are to be cross correlated and \texttt{t} is an integer representing the time period.

The message passing history for the six processor configuration is shown in Figure 5-8. When there were no more scheduling activities to perform and no incoming messages to process, the controller evaluated cross correlations to produce the jumps of just over a second in the interpreter run time from 1.83 to 2.97 seconds and from 3.27 to 4.35 seconds. Three of the four seconds in the jump at the end come from evaluating the remaining cross correlation, detecting false peaks, and calculating the location. The remaining second was mostly used to CONS together the results of the cross correlations, which is an expensive operation given the array representation that STARDUST uses for lists.

The schedule for the six processor configuration is shown in Figure 5-10, using "\texttt{xc(n_1,n_2)}" as a shorthand for the cross correlation between signals \texttt{n_1} and \texttt{n_2}. Note that fewer cross correlations were scheduled on the controller than on other processors, since the scheduler models the fact that the controller will be busy receiving the reply messages.

The processing costs for the controller in the six processor configuration appears in column one of Figure 5-6.
This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute and NNV for NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message.

Figure 5-8: Messages for the DSN Base Experiment with Six Processors
### Figure 5-9: Messages for the DSN Base Experiment with Six Processors, Continued from Figure 5-8

<table>
<thead>
<tr>
<th>Time</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
<th>Processor 5</th>
<th>Processor 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>locate</td>
<td>xc(1,2)</td>
<td>xc(1,3)</td>
<td>xc(1,4)</td>
<td>xc(1,5)</td>
<td>xc(1,6)</td>
<td>xc(1,7)</td>
</tr>
<tr>
<td>detect</td>
<td>xc(3,7)</td>
<td>xc(1,8)</td>
<td>xc(2,3)</td>
<td>xc(2,4)</td>
<td>xc(2,5)</td>
<td>xc(2,6)</td>
</tr>
<tr>
<td></td>
<td>xc(5,6)</td>
<td>xc(2,7)</td>
<td>xc(2,8)</td>
<td>xc(3,4)</td>
<td>xc(3,5)</td>
<td>xc(3,6)</td>
</tr>
<tr>
<td></td>
<td>xc(3,8)</td>
<td>xc(3,8)</td>
<td>xc(4,5)</td>
<td>xc(4,6)</td>
<td>xc(4,7)</td>
<td>xc(4,8)</td>
</tr>
<tr>
<td></td>
<td>xc(5,7)</td>
<td>xc(5,7)</td>
<td>xc(5,8)</td>
<td>xc(6,7)</td>
<td>xc(6,8)</td>
<td>xc(7,8)</td>
</tr>
</tbody>
</table>

### Figure 5-10: Schedule for the DSN Base Experiment with Six Processors

5.1.4. Optimal Partitioning

**TTime** (see Section 4.4) is set by the operator to a value such that the call to Map(PhonePairs(n),XCPair) is divided evenly among the available processors. For example, given two processors the twenty-eight cross correlations are divided into two sets of fourteen, each of which is scheduled as a unit. This experiment shows the best improvement that can be attained by using MapSplit (see Section 4.4.2) to reduce the overhead of scheduling and passing the cross correlation requests as individual messages. The Forms sent from the controller to the satellites have the following structure: Map([[x_1,y_1],...,[x_n,y_n]],xcpair).

Execution time for the six processor configuration dropped from 13.45 to 10.74 seconds. The message passing history and schedule for this configuration are shown in Figures 5-11 and 5-12, and a breakdown of processing costs for the controller appears in column two of Figure 5-6.
Table 1:

<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>&lt;Start&gt;</td>
</tr>
<tr>
<td>0.62</td>
<td>0.32</td>
<td>0.03</td>
<td>0.02</td>
<td>0.07</td>
<td>PE 2 map(List1,xcpair)</td>
</tr>
<tr>
<td>0.83</td>
<td>0.45</td>
<td>0.03</td>
<td>0.02</td>
<td>0.08</td>
<td>PE 3 map(List2,xcpair)</td>
</tr>
<tr>
<td>0.97</td>
<td>0.50</td>
<td>0.03</td>
<td>0.05</td>
<td>0.10</td>
<td>PE 4 map(List3,xcpair)</td>
</tr>
<tr>
<td>1.12</td>
<td>0.55</td>
<td>0.03</td>
<td>0.05</td>
<td>0.12</td>
<td>PE 5 map(List4,xcpair)</td>
</tr>
<tr>
<td>1.25</td>
<td>0.60</td>
<td>0.03</td>
<td>0.07</td>
<td>0.12</td>
<td>PE 6 map(List5,xcpair)</td>
</tr>
<tr>
<td>5.20</td>
<td>4.20</td>
<td>0.07</td>
<td>0.10</td>
<td>0.15</td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>7.72</td>
<td>4.27</td>
<td>2.37</td>
<td>0.10</td>
<td>0.18</td>
<td>NNV 3 list(map(List1,xcpair))</td>
</tr>
<tr>
<td>7.77</td>
<td>4.28</td>
<td>2.37</td>
<td>0.10</td>
<td>0.20</td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>7.93</td>
<td>4.33</td>
<td>2.38</td>
<td>0.10</td>
<td>0.22</td>
<td>NNV 5 list(map(List2,xcpair))</td>
</tr>
<tr>
<td>8.15</td>
<td>4.38</td>
<td>2.40</td>
<td>0.12</td>
<td>0.23</td>
<td>NNV 4 list(map(List3,xcpair))</td>
</tr>
<tr>
<td>8.32</td>
<td>4.47</td>
<td>2.40</td>
<td>0.12</td>
<td>0.23</td>
<td>NNV 2 list(map(List4,xcpair))</td>
</tr>
<tr>
<td>8.48</td>
<td>4.55</td>
<td>2.43</td>
<td>0.12</td>
<td>0.23</td>
<td>NNV 6 list(map(List5,xcpair))</td>
</tr>
<tr>
<td>10.95</td>
<td>6.92</td>
<td>2.47</td>
<td>0.12</td>
<td>0.23</td>
<td>&lt;finish&gt;</td>
</tr>
</tbody>
</table>

Where:

List1 = \([1;6],[1;6],[1;7],[1;8],[2;3]\)  
List2 = \([2;4],[2;5],[2;6],[2;7],[2;8]\)  
List3 = \([3;4],[3;5],[3;6],[3;7],[3;8]\)  
List4 = \([4;5],[4;6],[4;7],[4;8],[5;6]\)  
List5 = \([5;7],[5;8],[6;7],[6;8],[7;8]\)

This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low-priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute and NNV for NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message.

Figure 5.11: Messages for the DSN Optimal Partitioning Experiment with Six Processors

Processor 1: locate; detect; map([1;2],[1;3],[1;4],xcpair)  
Processor 2: map([1;6],[1;6],[1;7],[1;8],[2;3],xcpair)  
Processor 3: map([2;4],[2;5],[2;6],[2;7],[2;8],xcpair)  
Processor 4: map([3;4],[3;5],[3;6],[3;7],[3;8],xcpair)  
Processor 5: map([4;5],[4;6],[4;7],[4;8],[5;6],xcpair)  
Processor 6: map([5;7],[5;8],[6;7],[6;8],[7;8],xcpair)

Figure 5.12: Schedule for the DSN Optimal Partitioning Experiment with Six Processors

5.1.5. Full Messages

The program is modified in such a way that the 502-word signals are sent to the satellites in full. All other parameters are set as in the base case. This experiment was designed to model the situation where the signal data are originally known only to the controller. Instead of the integers sent in previous examples, the Forms sent from the controller to the satellites have the following structure: XC(<signal 1>,<signal 2>).
STARDUST 1.0 has no way of expressing the idea that a certain piece of information exists only at one site and must be sent to the others (this problem is discussed more fully in Section 7.3.3). In order to cause the signal data to be sent to the other machines, the function "Go" was divided into a "Prep" phase and a "Go" phase. Prep calls the data gathering routine "Data", in effect reading the data into the controller, and XCPair uses the Explode pragma to ensure that the data is sent in full rather than encoded into a call to XCPair. The experiments were run by first calling Prep, then enabling statistics gathering while executing Go.

Type PrepStructure = Record
  Entry1: List Of PhonePair;
  Entry2: List Of Signal;
End;

Let Prep(n: Integer; t: Integer) = Begin
  Let DataPrep(n:Integer) = Begin
    Time 1;
    Data(n,t);
    End;
  PrepStructure:[Entry1=PhonePairs(n),
    Entry2=Map(Iota(1,n,1),DataPrep)];
  End;

Let Go(n: Integer; t: Integer) = Begin
  Let XCPair(p: PhonePair) = [Explode] Begin
    Time 60;
    XC(Data(p.First,t),Data(p.Second,t));
    End;

    Time n*n*60;
    Locate(Detect(Map(Map(PhonePairs(n),XCPair),PeakPick)));
  End;

Another effect of this code modification is that the peak picking routine is executed on the controller rather than being sent to the satellites as part of the call to XCPair. This has a small effect on processing times and causes the 252-word cross correlations to be sent back to the controller instead of the 4-word representation of the peak.

The Prep function also calls PhonePairs, with the result that the list of pairs of microphones is generated before any of the five trials are run. In the base experiment and the optimal partitioning experiment this list was calculated during the first trial, and found by the other four through redundant subcomputation elimination. In practice this call to PhonePairs made no difference since the fastest trial, from which the results in this section are taken, was still always the third or the fifth.
Execution time for the six processor configuration increased from 13.45 to 17.61 seconds. The message passing history and schedule are shown in Figures 5-13 and 5-15, and a breakdown of processing costs for the controller appears in column three of figure 5-6.

5.1.6. Slow User Intrinsics

The processing time for cross correlations, false peak detection, and object location are increased from one second to five seconds. All other parameters are set as in the full message experiment. This experiment was designed to test the effect of speeding up the interpreter and message passing by a factor of five, so all reported execution times for this experiment have been divided by five. The Forms sent from the controller to the satellites have the same structure as the full messages of the previous section: \( XC(<\text{signal 1}>,<\text{signal 2}>) \).

The execution time for the six processor configuration, when normalized by dividing by five, was 8.15 seconds compared with 13.45 seconds for the base experiment and 7.0 seconds for the best possible execution time. The message passing history and schedule are shown in Figures 5-16 and 5-18, and a breakdown of processing costs for the controller appears in column four of Figure 5-6. We believe that a factor of five speedup for both the interpreter and the message handler are feasible with relatively straightforward recoding and special purpose protocols.

Note that since communications costs are relatively low compared with processing costs, more cross correlations are scheduled on the controller than in previous examples. This is correct behavior: the controller has more time left over from message passing for doing useful computations.

5.1.7. Summary

This set of experiments has shown that a simple but realistic signal processing algorithm can be expressed using user intrinsics for the numerical computations and STARDUST for the interconnections. The computation is properly partitioned, scheduled, and executed on one to six processors. For the present system the speedups are modest; the speedups would be quite respectable if the speed of the interpreter and message passing system could be improved by a factor of five.

The other important facts demonstrated by these experiments are:

- The optimal partitioning experiment showed that treating the MAP operator as a special case reduces message traffic and total execution time. Not only were communication, partitioning, and scheduling costs greatly reduced, but the
This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute and NNV for NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message.

Figure 5-13: Messages for the DSN Full Messages Experiment with Six Processors
Figure 5-14: Messages for the DSN Full Messages Experiment with Six Processors, Continued from Figure 5-13

<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.92</td>
<td>5.15</td>
<td>0.15</td>
<td>1.13</td>
<td>1.13</td>
<td>NNV</td>
<td>4</td>
<td>xc(3,4)</td>
</tr>
<tr>
<td>12.98</td>
<td>5.15</td>
<td>0.15</td>
<td>1.13</td>
<td>1.13</td>
<td>NNV</td>
<td>2</td>
<td>xc(3,8)</td>
</tr>
<tr>
<td>13.05</td>
<td>5.15</td>
<td>0.15</td>
<td>1.13</td>
<td>1.13</td>
<td>NNV</td>
<td>4</td>
<td>xc(4,6)</td>
</tr>
<tr>
<td>13.13</td>
<td>5.18</td>
<td>0.15</td>
<td>1.13</td>
<td>1.13</td>
<td>NNV</td>
<td>5</td>
<td>xc(2,5)</td>
</tr>
<tr>
<td>13.35</td>
<td>5.18</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>6</td>
<td>xc(3,6)</td>
</tr>
<tr>
<td>13.42</td>
<td>5.18</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>5</td>
<td>xc(3,5)</td>
</tr>
<tr>
<td>13.47</td>
<td>5.18</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>2</td>
<td>xc(5,6)</td>
</tr>
<tr>
<td>13.53</td>
<td>5.20</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>4</td>
<td>xc(5,8)</td>
</tr>
<tr>
<td>13.60</td>
<td>5.25</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>6</td>
<td>xc(4,8)</td>
</tr>
<tr>
<td>13.65</td>
<td>5.25</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>5</td>
<td>xc(4,7)</td>
</tr>
<tr>
<td>13.72</td>
<td>5.28</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>2</td>
<td>xc(7,8)</td>
</tr>
<tr>
<td>13.77</td>
<td>5.30</td>
<td>0.15</td>
<td>1.15</td>
<td>1.17</td>
<td>NNV</td>
<td>3</td>
<td>xc(6,7)</td>
</tr>
<tr>
<td>14.05</td>
<td>5.47</td>
<td>0.15</td>
<td>1.17</td>
<td>1.18</td>
<td>NNV</td>
<td>6</td>
<td>xc(6,8)</td>
</tr>
<tr>
<td>14.33</td>
<td>5.52</td>
<td>0.17</td>
<td>1.18</td>
<td>1.20</td>
<td>NNV</td>
<td>5</td>
<td>xc(6,7)</td>
</tr>
<tr>
<td>17.97</td>
<td>8.90</td>
<td>0.18</td>
<td>1.18</td>
<td>1.22</td>
<td></td>
<td></td>
<td>&lt;finish&gt;</td>
</tr>
</tbody>
</table>

Figure 5-15: Schedule for the DSN Full Messages Experiment with Six Processors

"other interpreter" costs were reduced since fewer partial results had to be combined. In addition the Accent overhead dropped significantly, since it is very nearly proportional to the number of packets sent and received.

- The full messages experiment showed that sending moderately large amounts of data from the controller to the satellites significantly increased execution time.

- The slow user intrinsics experiment showed that the communication cost estimates are properly taken into account by the scheduler. The proportionately lower communication costs allowed more computations to be performed by the controller.
<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>&lt;Start&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.75</td>
<td>1.92</td>
<td>0.08</td>
<td>0.03</td>
<td>0.06</td>
<td>PE</td>
<td>2 xc(1,3)</td>
<td></td>
</tr>
<tr>
<td>2.93</td>
<td>1.92</td>
<td>0.08</td>
<td>0.06</td>
<td>0.06</td>
<td>PE</td>
<td>3 xc(1,4)</td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>1.98</td>
<td>0.06</td>
<td>0.08</td>
<td>0.07</td>
<td>PE</td>
<td>4 xc(1,5)</td>
<td></td>
</tr>
<tr>
<td>3.40</td>
<td>1.98</td>
<td>0.08</td>
<td>0.15</td>
<td>0.10</td>
<td>PE</td>
<td>5 xc(1,6)</td>
<td></td>
</tr>
<tr>
<td>3.58</td>
<td>2.00</td>
<td>0.08</td>
<td>0.20</td>
<td>0.10</td>
<td>PE</td>
<td>6 xc(1,7)</td>
<td></td>
</tr>
<tr>
<td>3.80</td>
<td>2.00</td>
<td>0.08</td>
<td>0.23</td>
<td>0.12</td>
<td>PE</td>
<td>2 xc(1,8)</td>
<td></td>
</tr>
<tr>
<td>3.97</td>
<td>2.05</td>
<td>0.08</td>
<td>0.27</td>
<td>0.13</td>
<td>PE</td>
<td>3 xc(2,3)</td>
<td></td>
</tr>
<tr>
<td>4.22</td>
<td>2.07</td>
<td>0.08</td>
<td>0.33</td>
<td>0.17</td>
<td>PE</td>
<td>4 xc(2,4)</td>
<td></td>
</tr>
<tr>
<td>4.38</td>
<td>2.07</td>
<td>0.08</td>
<td>0.37</td>
<td>0.17</td>
<td>PE</td>
<td>5 xc(2,5)</td>
<td></td>
</tr>
<tr>
<td>4.58</td>
<td>2.08</td>
<td>0.08</td>
<td>0.42</td>
<td>0.18</td>
<td>PE</td>
<td>6 xc(2,6)</td>
<td></td>
</tr>
<tr>
<td>4.82</td>
<td>2.10</td>
<td>0.08</td>
<td>0.47</td>
<td>0.20</td>
<td>PE</td>
<td>2 xc(2,8)</td>
<td></td>
</tr>
<tr>
<td>4.95</td>
<td>2.12</td>
<td>0.08</td>
<td>0.48</td>
<td>0.22</td>
<td>PE</td>
<td>3 xc(2,4)</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>2.13</td>
<td>0.08</td>
<td>0.53</td>
<td>0.27</td>
<td>PE</td>
<td>4 xc(3,5)</td>
<td></td>
</tr>
<tr>
<td>5.28</td>
<td>2.17</td>
<td>0.08</td>
<td>0.55</td>
<td>0.27</td>
<td>PE</td>
<td>5 xc(3,6)</td>
<td></td>
</tr>
<tr>
<td>5.43</td>
<td>2.18</td>
<td>0.08</td>
<td>0.58</td>
<td>0.28</td>
<td>PE</td>
<td>6 xc(3,7)</td>
<td></td>
</tr>
<tr>
<td>5.62</td>
<td>2.20</td>
<td>0.08</td>
<td>0.62</td>
<td>0.28</td>
<td>PE</td>
<td>2 xc(4,5)</td>
<td></td>
</tr>
<tr>
<td>5.83</td>
<td>2.23</td>
<td>0.08</td>
<td>0.65</td>
<td>0.30</td>
<td>PE</td>
<td>3 xc(4,6)</td>
<td></td>
</tr>
<tr>
<td>5.98</td>
<td>2.27</td>
<td>0.08</td>
<td>0.70</td>
<td>0.32</td>
<td>PE</td>
<td>4 xc(4,7)</td>
<td></td>
</tr>
<tr>
<td>6.18</td>
<td>2.30</td>
<td>0.08</td>
<td>0.75</td>
<td>0.35</td>
<td>PE</td>
<td>5 xc(4,8)</td>
<td></td>
</tr>
<tr>
<td>6.40</td>
<td>2.33</td>
<td>0.08</td>
<td>0.78</td>
<td>0.38</td>
<td>PE</td>
<td>6 xc(5,6)</td>
<td></td>
</tr>
<tr>
<td>6.62</td>
<td>2.35</td>
<td>0.08</td>
<td>0.82</td>
<td>0.43</td>
<td>PE</td>
<td>2 xc(5,8)</td>
<td></td>
</tr>
<tr>
<td>6.72</td>
<td>2.37</td>
<td>0.08</td>
<td>0.82</td>
<td>0.45</td>
<td>PE</td>
<td>3 xc(6,7)</td>
<td></td>
</tr>
<tr>
<td>6.87</td>
<td>2.40</td>
<td>0.08</td>
<td>0.83</td>
<td>0.47</td>
<td>PE</td>
<td>4 xc(6,8)</td>
<td></td>
</tr>
<tr>
<td>7.08</td>
<td>2.42</td>
<td>0.08</td>
<td>0.88</td>
<td>0.48</td>
<td>PE</td>
<td>5 xc(7,8)</td>
<td></td>
</tr>
<tr>
<td>12.65</td>
<td>7.43</td>
<td>0.08</td>
<td>1.05</td>
<td>0.63</td>
<td>NNV</td>
<td>2 xc(1,3)</td>
<td></td>
</tr>
<tr>
<td>12.72</td>
<td>7.43</td>
<td>0.08</td>
<td>1.05</td>
<td>0.63</td>
<td>NNV</td>
<td>3 xc(1,4)</td>
<td></td>
</tr>
<tr>
<td>12.78</td>
<td>7.47</td>
<td>0.08</td>
<td>1.05</td>
<td>0.63</td>
<td>NNV</td>
<td>4 xc(1,5)</td>
<td></td>
</tr>
<tr>
<td>12.85</td>
<td>7.52</td>
<td>0.08</td>
<td>1.05</td>
<td>0.63</td>
<td>NNV</td>
<td>5 xc(1,6)</td>
<td></td>
</tr>
<tr>
<td>12.92</td>
<td>7.52</td>
<td>0.08</td>
<td>1.05</td>
<td>0.63</td>
<td>NNV</td>
<td>6 xc(1,7)</td>
<td></td>
</tr>
<tr>
<td>18.50</td>
<td>12.68</td>
<td>0.08</td>
<td>1.15</td>
<td>0.67</td>
<td>NNV</td>
<td>2 xc(1,8)</td>
<td></td>
</tr>
<tr>
<td>18.67</td>
<td>12.70</td>
<td>0.08</td>
<td>1.15</td>
<td>0.67</td>
<td>NNV</td>
<td>3 xc(2,3)</td>
<td></td>
</tr>
<tr>
<td>18.63</td>
<td>12.72</td>
<td>0.08</td>
<td>1.15</td>
<td>0.67</td>
<td>NNV</td>
<td>4 xc(2,4)</td>
<td></td>
</tr>
<tr>
<td>18.70</td>
<td>12.75</td>
<td>0.08</td>
<td>1.15</td>
<td>0.67</td>
<td>NNV</td>
<td>6 xc(2,6)</td>
<td></td>
</tr>
<tr>
<td>18.77</td>
<td>12.78</td>
<td>0.08</td>
<td>1.15</td>
<td>0.67</td>
<td>NNV</td>
<td>5 xc(2,5)</td>
<td></td>
</tr>
<tr>
<td>24.33</td>
<td>18.03</td>
<td>0.08</td>
<td>1.20</td>
<td>0.73</td>
<td>NNV</td>
<td>2 xc(2,8)</td>
<td></td>
</tr>
<tr>
<td>24.40</td>
<td>18.05</td>
<td>0.08</td>
<td>1.20</td>
<td>0.73</td>
<td>NNV</td>
<td>3 xc(3,4)</td>
<td></td>
</tr>
<tr>
<td>24.50</td>
<td>18.07</td>
<td>0.08</td>
<td>1.20</td>
<td>0.73</td>
<td>NNV</td>
<td>4 xc(3,5)</td>
<td></td>
</tr>
<tr>
<td>24.57</td>
<td>18.08</td>
<td>0.08</td>
<td>1.20</td>
<td>0.73</td>
<td>NNV</td>
<td>6 xc(3,7)</td>
<td></td>
</tr>
</tbody>
</table>

< Continued in Figure 5-17 >

This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute and NNV for NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message. The real and elapsed times are shown before normalization.

Figure 5-16: Messages for the DSN Slow User Intrinsics Experiment with Six Processors
5.2. The Quick Sort Example

This section describes the experimental results from a quick sort program. Quick sort algorithms sort an array by estimating a median value; reordering the array with all values greater than the estimated median in the top portion of the array; and recursively quick sorting the two portions of the array, possibly in parallel. Sufficiently small arrays are sorted with a standard bubble sort.

5.2.1. The Program

The program differs from a true quick sort in that the values are split into two new arrays rather than being reordered in place. This causes considerable dynamic storage allocation, particularly because STARDUST block types have a fixed length (see Sections 2.1.2 and 2.1.5). The idea of reordering values in place cannot be expressed in the STARDUST language nor, to our knowledge, in other applicative languages. The program follows.
Type SortData = Block Of 2000 Integer;

Type SortBlock = Record
SortLength: Integer;
SortData: SortData;
End;

Type SortBlockList = List Of SortBlock;

Let GetSortBlock(Len: Integer;
    Incr: Integer;
    Offset: Integer) = [Atomic] Begin
    Time t;
    UserIntrinsic("getsortblock",Len,Incr,Offset):SortBlock;
End;

Let BubbleSort(b: SortBlock) = [Atomic] Begin
    Time t;
    UserIntrinsic("bubblesort",b):SortBlock;
End;

Let Merge(b1: SortBlock; b2: SortBlock) = [Atomic] Begin
    Time t;
    UserIntrinsic("merge",b1,b2):SortBlock;
End;

Let Split(b: SortBlock) = [Atomic] Begin
    Time t;
    UserIntrinsic("split",b): SortBlockList;
End;

Let QSort(b: SortBlock): SortBlock = Begin
    Time t;
    If b.SortLength < 300 Then
        BubbleSort(b)
    Else
        Merge(QSort(Split(b)[1]),
            QSort(Split(b)[2]));
End;

GetSortBlock(Len,Incr,Offset) generates an array of random numbers of length Len. Incr is added to each value and the results are shifted Offset places. Like the parameter t in the DSN example, Incr and Offset were provided to allow several trials to be run without being trivialized by redundant subcomputation elimination.

BubbleSort sorts an array with a simple $O(N^2)$ algorithm. The time estimate was determined empirically. Merge concatenates two sorted arrays to produce a sorted array.

Split uses the central element of the array (b.SortData[b.SortLength div 2]) as an estimate of the median and produces two arrays, one with all values less than the median estimate and one with all values greater than or equal to the median estimate.
Qsort calls BubbleSort if the length of the array is less than 300. This threshold was
determined empirically to minimize sorting time on a single processor. If the size of the input
array is greater than 300 then the array is split, the subarrays are sorted, and the results are
merged. Note that while the array appears to be split twice by the two calls to Split(b),
redundant subcomputation elimination ensures that the code is only executed once. The run
time estimate of b.SortLength is a good approximation when the length of the array is be-
tween 300 and 2000, though a logarithmic term would have made it more accurate.

5.2.2. Summary of Experiments and Results

As in the DSN experiment (see Section 5.1.2) each data point is the fastest of five trials.
The trials consisted of sorting 2000 numbers and differed from one another only in the value
of Incr passed to GetSortBlock. The results were not as stable as in the DSN experiment: 7%
variation was typical among the second, third, and fourth trials. The first trial tended to be
higher due to the cost of paging in the routines, while the fifth was often much higher due to
paging caused by exhaustion of physical memory.

Two variations of the quick sort experiment are reported in this section. They are sum-
marized in Figure 5-19 and described in detail in Sections 5.2.3 and 5.2.4. Execution times
and speedups for the two experiments are presented in Figures 5-20 and 5-21.

Base Experiment: This version is a standard quick sort algorithm with the exception
that subarrays are copied rather than sorted in place.

Slow User Intrinsics: The processing time for BubbleSort is multiplied by five and the
resulting execution time measurements are divided by five. This experiment tests the effect of speeding up the interpreter and mes-

Linear Speedup: Execution time is calculated by dividing the base experiment's
single processor execution time of 33.5 seconds by the number of processors.

The two experimental variations are described more fully in sections 5.2.3 and 5.2.4.

Figure 5-19: Summary of Quick Sort Experiments

Figure 5-5 describes the categories used throughout this chapter for reporting process-
ing costs. Figure 5-22 gives a breakdown of the processing costs on the controller for the
one- and six-processor version of the base experiment and the six-processor version of the
slow user intrinsics experiment, and Figure 5-23 gives their message passing statistics.

The sequence of splits used to sort the array is shown in Figure 5-24. The original array
This figure presents evaluation time as a function of the number of processors in two experimental variations. Linear speedup is plotted for comparison. The three plots are summarized in Figure 5-19 and explained more fully in the text. The evaluation times plotted here are the difference between Total Real Time and Statistics Gathering (see Figures 5-22 and 5-5).

Figure 5-20: Quick Sort Evaluation Times

This figure presents speedup as a function of the number of processors in two experimental variations. Linear speedup is plotted for comparison. Speedups are calculated by dividing the execution time into the execution time for the single processor version of the same experiment. The two plots are summarized in Figure 5-19 and explained more fully in the text.

Figure 5-21: Quick Sort Speedups
### Figure 5-22: Processor Usage for the Controller in Two Quick Sort Experiments

<table>
<thead>
<tr>
<th></th>
<th>Base Experiment (1 machine)</th>
<th>Base Experiment (6 machines)</th>
<th>Slow User Intrinsics (6 machines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Real Time</td>
<td>34.49 100%</td>
<td>28.01 100%</td>
<td>11.02 100%</td>
</tr>
<tr>
<td>Statistics Gathering</td>
<td>1.61 5%</td>
<td>1.88 7%</td>
<td>0.43 4%</td>
</tr>
<tr>
<td>Accent Overhead</td>
<td>4.37 13%</td>
<td>5.25 19%</td>
<td>0.96 9%</td>
</tr>
<tr>
<td>Message Server</td>
<td>0.03 0%</td>
<td>0.23 1%</td>
<td>0.06 1%</td>
</tr>
<tr>
<td>Network Server</td>
<td>0.03 0%</td>
<td>0.11 0%</td>
<td>0.04 0%</td>
</tr>
<tr>
<td>Idle Time</td>
<td>0.03 0%</td>
<td>1.67 6%</td>
<td>3.03 28%</td>
</tr>
<tr>
<td>Total Interpreter</td>
<td>28.42 82%</td>
<td>18.87 67%</td>
<td>6.50 59%</td>
</tr>
<tr>
<td>User Intrinsics</td>
<td>21.00 61%</td>
<td>9.83 35%</td>
<td>4.46 41%</td>
</tr>
<tr>
<td>Message Overhead</td>
<td>0.00 0%</td>
<td>1.23 4%</td>
<td>0.24 2%</td>
</tr>
<tr>
<td>Partitioning</td>
<td>0.00 0%</td>
<td>0.15 1%</td>
<td>0.02 0%</td>
</tr>
<tr>
<td>Scheduling</td>
<td>0.10 0%</td>
<td>0.26 1%</td>
<td>0.07 1%</td>
</tr>
<tr>
<td>Other Interpreter</td>
<td>7.32 21%</td>
<td>7.40 26%</td>
<td>1.71 16%</td>
</tr>
</tbody>
</table>

*This figure presents controller processor costs for two experimental variations with one and six processors. The variations are summarized in Figure 5-19 and explained more fully in the text. The categories are defined in Figure 5-5. The percentages reported here are of Total Real Time.*

### Figure 5-23: Message Statistics for the Controller in Two Quick Sort Experiments

<table>
<thead>
<tr>
<th></th>
<th>Base Experiment (1 machine)</th>
<th>Base Experiment (6 machines)</th>
<th>Slow User Intrinsics (6 machines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words Sent</td>
<td>0</td>
<td>1974</td>
<td>2138</td>
</tr>
<tr>
<td>Words Received</td>
<td>0</td>
<td>1921</td>
<td>2084</td>
</tr>
<tr>
<td>Messages Sent</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Messages Received</td>
<td>0</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Data Packets Sent</td>
<td>0</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>Data Packets Received</td>
<td>0</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Acknowledgements Sent</td>
<td>0</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Acknowledgements Received</td>
<td>0</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>Retransmits Sent</td>
<td>0</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>Retransmits Received</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total Packets Sent</td>
<td>0</td>
<td>49</td>
<td>57</td>
</tr>
<tr>
<td>Total Packets Received</td>
<td>0</td>
<td>45</td>
<td>45</td>
</tr>
</tbody>
</table>

*This figure presents message passing statistics from the point of view of the controller for two experimental variations with one and six processors. The variations are summarized in Figure 5-19 and explained more fully in the text. "Data packets sent" represents the number of packets successfully sent and acknowledged; the sum of data packets sent, acknowledgement packets sent, and retransmissions sent is the total number of packets sent. The same relation holds for packets received.*
of 2000 words was split into a 12-element array and a 1988-element array; the 1988-element array was then split into a 792-element array and a 1196-element array, and so on. Since the lengths are all different, we are able to refer to these subarrays by their length enclosed in brackets: for example we can say that [2000] is split into [12] and [1988]. The sequence of splits was the same for all five trials because the data differed only by the constant Incr that was added to each element.

Figures 5-22 and 5-23 summarize the main experimental results of this section, showing very disappointing speedups for both the base and the slow user intrinsics experiments. Speedup for the base experiment never even reached 1.5, and additional processors after the second did no appreciable good. The slow user intrinsics experiment did little better: speedup was never greater than 2.0, and did not appreciably improve after the third processor was added.

The reason for the poor performance can be seen by comparing the one- and six-processor versions of the base experiment shown in Figure 5-22. The large contributions from Accent Overhead and Other Interpreter are undiminished, while only half of the User Intrinsic time was moved to other processors. In addition, small but significant costs were added by message passing, partitioning, scheduling, and idle time. To understand the poor performance of this experiment we must understand three problems: the origin of the Accent Overhead and why it was not spread among the satellites; the origin of the Other Interpreter overhead and why it was not spread among the satellites; and the poor scheduling choices that lead to overuse of the controller for user intrinsics.

Unlike the DSN experiment, where Accent Overhead was mostly attributable to message passing, the Accent Overhead in the quick sort experiment is mostly a product of the enormous amount of virtual memory that our program allocates: over 500 pages for a single trial. Each of the twenty-seven arrays in Figure 5-24 uses just over 2000 words apiece in its unsorted and sorted form; exporting the bubble sorts to the satellites does not change the fact that each of these arrays must be allocated on the controller.

The Other Interpreter costs are high simply because the STARDUST interpreter evaluates expressions slowly. Little of this load is transferred to the satellites: only the calls to BubbleSort are exported while Merge, Split, and Qsort are interpreted on the controller. Note however that if larger arrays were sorted it would be possible to increase the partitioning granularity parameter TTime (see Section 4.4) to the point where calls to Qsort on relatively large arrays would be exported. This would export both Accent Overhead and Other Interpreter overhead to the satellites.
Figure 5-24: The Series of Splits Used to Sort the 2000-Element Arrays
The Accent Overhead and Other Interpreter overhead are relatively small in the slow user
intrinsics experiment, with the result that speedups are poor rather than virtually non-existent.
The third problem — poor scheduling — will be discussed in Section 5.2.3 when schedules
and message passing histories are given.

5.2.3. Base Experiment

The time limit above which user functions are partitioned (the constant TTime, used by
procedure UserSplit in Section 4.4 and MapSplit in Section 4.4.2) is set to 100 "jiffies"
(100/60 or 1.67 seconds). The system assumes a message passing overhead of 5 jiffies, or
0.08 seconds.

The Forms sent and received by the controller contain the array to be sorted and the
sorted result. The system uses 2000-word blocks for representing even these relatively short
arrays, but a special feature ensures that the entire block is not transmitted in each message:
the user intrinsics Split and Merge fill the unused portion of the blocks they return with zeros,
and the message packing routines encode long terminating strings of zeros in an efficient
format.

The message passing history and schedule for the six processor configuration are
shown in Figure 5.25 and 5.26, and a breakdown of processing costs for the controller appears in column two of Figure 5.22.

The message passing history shows the origin of the poor scheduling: large jumps in the
interpreter run time (the column labelled "SD") early in the run indicate that calls to Bub-
bleSort are being executed before the program is completely partitioned and exported to the
satellites. This leaves the satellites idle at the beginning of the run, and the controller idle at
the end, and also increases the number of computations scheduled on the controller because
it tends to leave all processors idle at the moment when scheduling is performed. Since the
processors are otherwise equal the controller is chosen to receive the next user intrinsic
because it does not incur any message passing overhead. This behavior was caused by a
flaw in the heuristics used to determine whether actions should be placed on the high priority
or low priority action queue (Section 4.7 briefly describes the two action queues).
This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute and NNV for NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message. The bracketed numbers refer to the subarrays found in Figure 5-24.

**Figure 5-25:** Messages for the Quick Sort Base Experiment with Six Processors

<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>(MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>&lt;Start&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.68</td>
<td>3.82</td>
<td>0.15</td>
<td>0.03</td>
<td>0.02</td>
<td>PE 2</td>
<td>bubblesort([125])</td>
<td></td>
</tr>
<tr>
<td>9.20</td>
<td>6.80</td>
<td>0.15</td>
<td>0.05</td>
<td>0.03</td>
<td>NNV 2</td>
<td>[125]</td>
<td></td>
</tr>
<tr>
<td>10.58</td>
<td>7.87</td>
<td>0.15</td>
<td>0.05</td>
<td>0.03</td>
<td>PE 2</td>
<td>bubblesort([150])</td>
<td></td>
</tr>
<tr>
<td>12.33</td>
<td>8.92</td>
<td>0.15</td>
<td>0.05</td>
<td>0.05</td>
<td>NNV 2</td>
<td>[150]</td>
<td></td>
</tr>
<tr>
<td>13.17</td>
<td>9.62</td>
<td>0.15</td>
<td>0.05</td>
<td>0.05</td>
<td>PE 2</td>
<td>bubblesort([101])</td>
<td></td>
</tr>
<tr>
<td>14.63</td>
<td>10.58</td>
<td>0.15</td>
<td>0.07</td>
<td>0.07</td>
<td>NNV 2</td>
<td>[101]</td>
<td></td>
</tr>
<tr>
<td>15.47</td>
<td>11.27</td>
<td>0.15</td>
<td>0.07</td>
<td>0.07</td>
<td>PE 2</td>
<td>bubblesort([277])</td>
<td></td>
</tr>
<tr>
<td>15.67</td>
<td>11.35</td>
<td>0.18</td>
<td>0.08</td>
<td>0.07</td>
<td>PE 3</td>
<td>bubblesort([117])</td>
<td></td>
</tr>
<tr>
<td>15.82</td>
<td>11.42</td>
<td>0.18</td>
<td>0.10</td>
<td>0.07</td>
<td>PE 3</td>
<td>bubblesort([195])</td>
<td></td>
</tr>
<tr>
<td>16.83</td>
<td>12.06</td>
<td>0.18</td>
<td>0.15</td>
<td>0.07</td>
<td>PE 4</td>
<td>qsort([88])</td>
<td></td>
</tr>
<tr>
<td>18.42</td>
<td>13.37</td>
<td>0.18</td>
<td>0.17</td>
<td>0.07</td>
<td>NNV 4</td>
<td>[88]</td>
<td></td>
</tr>
<tr>
<td>19.47</td>
<td>14.02</td>
<td>0.22</td>
<td>0.18</td>
<td>0.07</td>
<td>NNV 3</td>
<td>[117]</td>
<td></td>
</tr>
<tr>
<td>19.72</td>
<td>14.10</td>
<td>0.23</td>
<td>0.18</td>
<td>0.07</td>
<td>NNV 3</td>
<td>[195]</td>
<td></td>
</tr>
<tr>
<td>19.90</td>
<td>14.17</td>
<td>0.23</td>
<td>0.18</td>
<td>0.07</td>
<td>NNV 2</td>
<td>[277]</td>
<td></td>
</tr>
<tr>
<td>21.87</td>
<td>15.57</td>
<td>0.23</td>
<td>0.18</td>
<td>0.07</td>
<td>PE 2</td>
<td>qsort([28])</td>
<td></td>
</tr>
<tr>
<td>22.07</td>
<td>15.67</td>
<td>0.23</td>
<td>0.18</td>
<td>0.08</td>
<td>PE 3</td>
<td>bubblesort([284])</td>
<td></td>
</tr>
<tr>
<td>24.57</td>
<td>17.90</td>
<td>0.23</td>
<td>0.22</td>
<td>0.10</td>
<td>NNV 2</td>
<td>[26]</td>
<td></td>
</tr>
<tr>
<td>24.62</td>
<td>17.92</td>
<td>0.23</td>
<td>0.22</td>
<td>0.10</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26.32</td>
<td>17.98</td>
<td>1.70</td>
<td>0.23</td>
<td>0.12</td>
<td>NNV 3</td>
<td>[284]</td>
<td></td>
</tr>
<tr>
<td>28.57</td>
<td>19.52</td>
<td>1.70</td>
<td>0.23</td>
<td>0.12</td>
<td>&lt;finish&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5-26:** Schedule for the Quick Sort Base Experiment with Six Processors
5.2.4. Slow User Intrinsics

The message passing history and schedule are shown in Figures 5-27 and 5-28, and a breakdown of processing costs for the controller appears in column three of Figure 5-22. Performance improved somewhat but the problem of locally executing calls to BubbleSort before finishing with the partitioning remained.

<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td>&lt;Start&gt;</td>
</tr>
<tr>
<td>5.75</td>
<td>4.22</td>
<td>0.12</td>
<td>0.00</td>
<td>0.08</td>
<td>PE</td>
<td>2</td>
<td>bubblesort([125])</td>
</tr>
<tr>
<td>17.50</td>
<td>15.58</td>
<td>0.12</td>
<td>0.02</td>
<td>0.08</td>
<td>NNV</td>
<td>2</td>
<td>[125]</td>
</tr>
<tr>
<td>19.32</td>
<td>16.93</td>
<td>0.12</td>
<td>0.02</td>
<td>0.08</td>
<td>PE</td>
<td>2</td>
<td>bubblesort([150])</td>
</tr>
<tr>
<td>21.85</td>
<td>18.67</td>
<td>0.12</td>
<td>0.03</td>
<td>0.08</td>
<td>PE</td>
<td>3</td>
<td>bubblesort([157])</td>
</tr>
<tr>
<td>22.03</td>
<td>18.75</td>
<td>0.12</td>
<td>0.05</td>
<td>0.08</td>
<td>PE</td>
<td>4</td>
<td>bubblesort([101])</td>
</tr>
<tr>
<td>24.25</td>
<td>20.30</td>
<td>0.12</td>
<td>0.07</td>
<td>0.10</td>
<td>PE</td>
<td>5</td>
<td>bubblesort([277])</td>
</tr>
<tr>
<td>24.43</td>
<td>20.40</td>
<td>0.13</td>
<td>0.10</td>
<td>0.10</td>
<td>PE</td>
<td>6</td>
<td>bubblesort([117])</td>
</tr>
<tr>
<td>24.80</td>
<td>20.55</td>
<td>0.13</td>
<td>0.13</td>
<td>0.12</td>
<td>NNV</td>
<td>2</td>
<td>[150]</td>
</tr>
<tr>
<td>25.17</td>
<td>20.78</td>
<td>0.13</td>
<td>0.15</td>
<td>0.13</td>
<td>NNV</td>
<td>4</td>
<td>[101]</td>
</tr>
<tr>
<td>26.20</td>
<td>21.58</td>
<td>0.13</td>
<td>0.15</td>
<td>0.13</td>
<td>PE</td>
<td>2</td>
<td>bubblesort([88])</td>
</tr>
<tr>
<td>27.40</td>
<td>22.42</td>
<td>0.13</td>
<td>0.17</td>
<td>0.13</td>
<td>PE</td>
<td>3</td>
<td>bubblesort([227])</td>
</tr>
<tr>
<td>35.38</td>
<td>29.90</td>
<td>0.13</td>
<td>0.22</td>
<td>0.13</td>
<td>NNV</td>
<td>3</td>
<td>[157]</td>
</tr>
<tr>
<td>35.55</td>
<td>29.97</td>
<td>0.13</td>
<td>0.22</td>
<td>0.13</td>
<td>NNV</td>
<td>2</td>
<td>[88]</td>
</tr>
<tr>
<td>35.70</td>
<td>30.07</td>
<td>0.13</td>
<td>0.22</td>
<td>0.13</td>
<td>NNV</td>
<td>6</td>
<td>[117]</td>
</tr>
<tr>
<td>36.67</td>
<td>30.85</td>
<td>0.13</td>
<td>0.22</td>
<td>0.13</td>
<td>PE</td>
<td>2</td>
<td>bubblesort([284])</td>
</tr>
<tr>
<td>37.10</td>
<td>31.12</td>
<td>0.13</td>
<td>0.22</td>
<td>0.17</td>
<td></td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>38.47</td>
<td>31.18</td>
<td>1.28</td>
<td>0.23</td>
<td>0.18</td>
<td>NNV</td>
<td>3</td>
<td>[277]</td>
</tr>
<tr>
<td>38.52</td>
<td>31.20</td>
<td>1.28</td>
<td>0.23</td>
<td>0.18</td>
<td></td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>40.60</td>
<td>31.30</td>
<td>3.12</td>
<td>0.25</td>
<td>0.20</td>
<td>NNV</td>
<td>5</td>
<td>[277]</td>
</tr>
<tr>
<td>41.52</td>
<td>31.90</td>
<td>3.12</td>
<td>0.25</td>
<td>0.20</td>
<td></td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>54.08</td>
<td>31.97</td>
<td>15.47</td>
<td>0.28</td>
<td>0.20</td>
<td>NNV</td>
<td>2</td>
<td>[284]</td>
</tr>
<tr>
<td>56.23</td>
<td>33.45</td>
<td>15.47</td>
<td>0.28</td>
<td>0.20</td>
<td></td>
<td></td>
<td>&lt;finish&gt;</td>
</tr>
</tbody>
</table>

This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute and NNV for NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message. The real and elapsed times are shown before normalization. The bracketed numbers refer to the subarrays found in Figure 5-24.

**Figure 5-27:** Messages for the Quick Sort Slow User Intrinsics Experiment with Six Processors
5.2.5. Summary

This experiment yielded disappointing results for three main reasons.

First the algorithm was not particularly appropriate to an applicative language. A large amount of memory was allocated because the blocks had to be copied rather than sorted in place, and this problem was exacerbated by the fixed length of STARDUST block types. Variable length block types should be implemented as part of the policy recommended in Section 2.1.2 of making block types differ from list types only in their performance characteristics and termination semantics.

Second, the problem divided up in such a manner that most of the virtual memory allocation and STARDUST expression evaluation took place on the controller. Since these accounted for 39% of the total processing costs in the single-processor base experiment (everything but the 61% cost of the User Intrinsics), a maximum speedup of only 2.6 would have been obtained if all of the user intrinsics could have been exported at no cost to the controller.

Finally, an error in the heuristics that control evaluation order of expressions on the controller caused some of the locally scheduled calls to BubbleSort to be evaluated before partitioning and exportation were complete. This led to idle satellites in the beginning of the run and an idle controller at the end.

5.3. The Molecular Modelling Example

This section describes the experimental results from a molecular modelling program that attempts to find the lowest-energy configuration of a collection of atoms. The atoms are given an initial configuration, then repeatedly perturbed by random amounts. After each perturbation the new configuration is kept if its total energy is lower than the total energy of the old one.
This example was taken from [Whiteside 84] with one simplification: the original algorithm contains a certain probability of retaining the higher-energy configuration while the program used for this experiment always keeps the lower-energy configuration. The algorithm differs from general stochastic hill climbing in that the quantity to be minimized is a linear combination of contributions from the pairs of atoms: when an atom is moved only the contribution from that atom needs to be recalculated.

5.3.1. The Program

The random perturbations to the positions of the atoms (ModifyLocation) and the contribution to the total energy from a pair of atoms (Energy2) are calculated by user intrinsics. The rest of the program is written entirely within the STARDUST language.

Type Position = Block Of 3 Integer;
Type PosList = List Of Position;
Let NumAtom = 8;

Let MakePosition(x: Integer; y: Integer; z: Integer) = Position: List Of Integer: [x,y,z];
Let InitialLocations(Offset: Integer) = PosList: [
    MakePosition(Offset,Offset,Offset),
    MakePosition(Offset,Offset,20+Offset),
    MakePosition(Offset,20+Offset,Offset),
    MakePosition(Offset,20+Offset,20+Offset),
    MakePosition(20+Offset,Offset,Offset),
    MakePosition(20+Offset,Offset,20+Offset),
    MakePosition(20+Offset,20+Offset,Offset),
    MakePosition(20+Offset,20+Offset,20+Offset)];

Let Etime2 = 60;
Let Etime = Etime2 * NumAtom;

Let ModifyLocation(p: Position; n: Integer; t: Integer) = [Atomic] Begin
    Time 6;
    UserIntrinsic("modify",p,n,t): Position;
End;

Let Energy2(p1: Position; p2: Position) = [Atomic] Begin
    Time Etime2;
    UserIntrinsic("energy2",p1,p2): Integer;
End;
Let Energy(n: Integer; p: PosList) = Begin
  Time Etime;
  Let ApplyEnergy2(m: Integer) = Begin
    Time Etime;
    Energy2(p[n],p[m]);
  End;
  Reduce(Map(Iota(1,NumAtom,1),ApplyEnergy2),0,+); End;

Let Locations(t:Integer; Offset: Integer): PosList = Begin
  Time t*Etime; I Should be t*Etime*2
  Let n=t-(t div NumAtom)*NumAtom;  | The atom to modify
  Let Elist1 = Begin
    Time t*Etime;
    Locations(t-1,Offset);
  End;
  Let Elist2 = Begin
    Time t*Etime;
    Elist1[n]_ModifyLocation(Elist1[n], t, 0);
  End;
  Sif t<1 Then InitialLocations(Offset)
  Else Sif Energy(n,Elist2) < Energy(n,Elist1) Then
    Elist2
  Else
    Elist1;
  End;

InitialLocations is called only once for each trial. It generates a set of initial locations for the 8 atoms, placing them on the corners of a 20 by 20 by 20 cube. Like the parameter t in the DSN example and the parameter Incr in the quick sort example, Offset is provided so that several trials may be run without being trivialized by redundant subcomputation elimination.

ModifyLocation(p,n,t) makes a random modification to position p by adding a random integer between -7 and 8 to each coordinate. ModifyLocation is informed that p is the position of the n' th atom, and that this modification is being made as part of the t' th iteration. ModifyLocation in effect resets the random number generator, throws away $3*(n + NumAtom*t)$ random numbers, then uses the next 3 to modify the x, y, and z coordinates of the position. The parameters n and t are necessary because of the applicative nature of the STARDUST language: without them ModifyLocation would either have to return the same value whenever a particular position was supplied, or violate the otherwise universal constraint that a function's value depends only on its parameters. Another way to view this is that the usual convenient random number function of imperative languages has the side effect of updating state in the generator and so cannot be directly adapted to an applicative language.
Energy2 calculates the contribution to the total energy from a pair of atoms. The function used in this experiment was the square of the distance between them, and an empty loop was used to raise the running time to one second. This figure of one second is arbitrary: it is probably too high if a simple electrostatic model is used but too low if quantum mechanical effects are accounted for.

Energy calculates the contribution to the total energy from the \( n^{th} \) atom.

Locations\((t,\text{Offset})\) calculates the positions of the atoms after \( t \) perturbations. It recursively calculates the position after \( t-1 \) perturbations, chooses an atom to modify (the \( n^{th} \) atom), compares the energy contribution from the \( n^{th} \) atom in its modified and unmodified positions (Elist2 and Elist1), and returns the list of positions with the \( n^{th} \) atom in the lower energy position. The definition of \( n \) causes Atom number 2 to be perturbed when Locations\((1,\text{Offset})\) is calculated, Atom 3 to be perturbed when Locations\((2,\text{Offset})\) is calculated, and so on.

When the program is executed on a single processor, redundant subcomputation elimination ensures that the call to Energy2 will not be repeated when the positions of two atoms remain unchanged after a set of NumAtom iterations. Since we chose to do 10 iterations with 8 atoms, RSCE only had an effect on the last 2 iterations and so did not make a major contribution to reducing run time. Instead of the 160 calls to Energy2 that the program specifies, 148 were made in the single processor configuration and 152 in the six processor configuration of the base experiment.

5.3.2. Summary of Experiments and Results

Three variations of the Molecular Modelling (MM) experiment are reported in this section. They are summarized in Figure 5-29 and described in detail in Sections 5.3.3 through 5.3.5. Execution times and speedups for the three experiments are presented in Figures 5-30 and 5-31.

Figure 5-5 describes the categories used throughout this chapter for reporting processing costs. Figure 5-32 gives a breakdown of the processing costs on the controller for the six-processor version of both experimental variations, and Figure 5-33 gives message passing statistics for the same experiments.

Figures 5-32 and 5-33 summarize the main experimental results of this section. We will overview the relation between the structure of the MM program and the messages that it passes, discuss the statistics shown in Figures 5-32 and 5-33, and then give more detailed results for each experiment.
Base Experiment: This version is a realistic adaptation of a molecular modelling algorithm.

Coarser Partitioning: The partitioning granularity parameter TTime is increased from 100 to 1000, which reduces the opportunities for parallelism but also reduces partitioning and scheduling costs.

Slow User Intrinsic: The processing time for Energy2 is multiplied by five and the resulting execution time measurements are divided by five. This experiment tests the effect of speeding up the interpreter and message passing by a factor of five.

Linear Speedup: Execution time is calculated by dividing the base experiment's single processor execution time of 179.4 seconds by the number of processors.

The three experimental variations are described more fully in sections 5.3.3 through 5.3.5.

Figure 5.29: Summary of Molecular Model Experiments and Theoretical Speedups

The top-level routine of MM is Locations. For each level of recursion it generates 2 calls to Energy: one to calculate the energy contribution from the unperturbed nth atom and one to calculate the energy contribution when the atom's position is modified. Each call to Energy makes 8 calls to Energy2 giving a total of 16 calls to Energy2 per call to Locations. The top-level call in these experiments was Locations(10,Offset) for some value of Offset, giving a total of 160 potential calls to Energy2.

Locations is first called with t = 10. The calls to Energy2 associated with t = 10 cannot be evaluated before Locations(9,Offset) has been calculated, since the value of Energy2 depends on both of its arguments which in turn depend on Locations(9,Offset). However, the STARDUST interpreter must allow for the possibility that Energy2 does not need both of its arguments to return a value, and so exports the call to Energy2 in a PleaseExecute message with a global name for each argument to be used to identify the argument when it is finally sent. (Sections 4.2.2 and 4.6 describe global names and exportation.)

Recursion of Locations continues until Locations(0,Offset) is evaluated to return the initial locations. These are sent to the invocations of Energy2 generated by Locations(1,Offset), which calculates the energies and return them to the controller. The controller sums the energies for the perturbed and unperturbed versions (either locally or on another processor), compares the two, and decides which should be the value of Locations(1,Offset). These new locations are sent to the invocations of Energy2 generated by Locations(2,Offset), and the process continues. Note that three messages are sent for each exported call to Energy2: one for the call and one each for the arguments.
This figure presents evaluation time as a function of the number of processors in three experimental variations. Linear speedup is plotted for comparison. The four plots are summarized in Figure 5-29 and explained more fully in the text. The evaluation times plotted here are the difference between Total Real Time and Statistics Gathering (see Figures 5-32 and 5-5).

Figure 5-30: Molecular Model Evaluation Times

This figure presents speedup as a function of the number of processors in three experimental variations. Linear speedup is plotted for comparison. Speedups are calculated by dividing the execution time into the execution time for the single processor version of the same experiment. The four plots are summarized in Figure 5-29 and explained more fully in the text.

Figure 5-31: Molecular Model Speedups
<table>
<thead>
<tr>
<th><strong>Base</strong></th>
<th><strong>Base</strong></th>
<th><strong>Coarse</strong></th>
<th><strong>Slow User</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Experiment</strong></td>
<td><strong>Experiment</strong></td>
<td><strong>Partitions</strong></td>
<td><strong>Intrinsics</strong></td>
</tr>
<tr>
<td>(1 machine)</td>
<td>(6 machines)</td>
<td>(6 machines)</td>
<td>(6 machines)</td>
</tr>
<tr>
<td><strong>Total Real Time</strong></td>
<td>182.08 100%</td>
<td>107.73 100%</td>
<td>122.11 100%</td>
</tr>
<tr>
<td><strong>Statistics Gathering</strong></td>
<td>6.14 3%</td>
<td>11.97 11%</td>
<td>1.86 2%</td>
</tr>
<tr>
<td><strong>Accent Overhead</strong></td>
<td>3.70 2%</td>
<td>29.78 28%</td>
<td>1.94 2%</td>
</tr>
<tr>
<td><strong>Message Server</strong></td>
<td>0.08 0%</td>
<td>6.67 6%</td>
<td>0.60 1%</td>
</tr>
<tr>
<td><strong>Network Server</strong></td>
<td>0.13 0%</td>
<td>4.36 4%</td>
<td>0.29 0%</td>
</tr>
<tr>
<td><strong>Idle Time</strong></td>
<td>0.88 1%</td>
<td>16.81 16%</td>
<td>94.35 77%</td>
</tr>
<tr>
<td><strong>Total Interpreter</strong></td>
<td>171.15 94%</td>
<td>38.14 35%</td>
<td>23.07 19%</td>
</tr>
</tbody>
</table>

**User Intrinsics** | 144.58 79% | 10.93 10% | 1.02 1% | 2.07 4% |
**Message Overhead** | 0.00 0% | 1.93 2% | 0.02 0% | 0.41 1% |
**Partitioning** | 0.00 0% | 1.73 2% | 0.28 0% | 0.36 1% |
**Scheduling** | 0.30 0% | 6.13 5% | 4.36 9% | 4.91 10% |
**Other Interpreter** | 26.27 14% | 22.61 21% | 0.08 0% | 0.41 1% |

This figure presents controller processor costs for three experimental variations with one and six processors. The variations are summarized in Figure 5-29 and explained more fully in the text. The categories are defined in Figure 5-5. The percentages reported here are of Total Real Time.

Figure 5-32: Processor Usage for the Controller in Three Molecular Modelling Experiments

<table>
<thead>
<tr>
<th><strong>Base</strong></th>
<th><strong>Base</strong></th>
<th><strong>Coarse</strong></th>
<th><strong>Slow User</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Experiment</strong></td>
<td><strong>Experiment</strong></td>
<td><strong>Partitions</strong></td>
<td><strong>Intrinsics</strong></td>
</tr>
<tr>
<td>(1 machine)</td>
<td>(6 machines)</td>
<td>(6 machines)</td>
<td>(6 machines)</td>
</tr>
<tr>
<td><strong>Words Sent</strong></td>
<td>0</td>
<td>20514</td>
<td>2900</td>
</tr>
<tr>
<td><strong>Words Received</strong></td>
<td>0</td>
<td>6372</td>
<td>667</td>
</tr>
<tr>
<td><strong>Messages Sent</strong></td>
<td>0</td>
<td>452</td>
<td>29</td>
</tr>
<tr>
<td><strong>Messages Received</strong></td>
<td>0</td>
<td>177</td>
<td>16</td>
</tr>
<tr>
<td><strong>Data Packets Sent</strong></td>
<td>0</td>
<td>906</td>
<td>61</td>
</tr>
<tr>
<td><strong>Data Packets Received</strong></td>
<td>0</td>
<td>354</td>
<td>30</td>
</tr>
<tr>
<td><strong>Acknowledgements Sent</strong></td>
<td>0</td>
<td>359</td>
<td>30</td>
</tr>
<tr>
<td><strong>Acknowledgements Received</strong></td>
<td>0</td>
<td>909</td>
<td>70</td>
</tr>
<tr>
<td><strong>Retransmits Sent</strong></td>
<td>0</td>
<td>231</td>
<td>8</td>
</tr>
<tr>
<td><strong>Retransmits Received</strong></td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total Packets Sent</strong></td>
<td>0</td>
<td>1496</td>
<td>99</td>
</tr>
<tr>
<td><strong>Total Packets Received</strong></td>
<td>0</td>
<td>1286</td>
<td>100</td>
</tr>
</tbody>
</table>

This figure presents message passing statistics from the point of view of the controller for three experimental variations with one and six processors. The variations are summarized in Figure 5-29 and explained more fully in the text. "Data packets sent" represents the number of packets successfully sent and acknowledged; the sum of data packets sent, acknowledgement packets sent, and retransmissions sent is the total number of packets sent. The same relation holds for packets received.

Figure 5-33: Controller Message Statistics for Three Molecular Modelling Experiments
The message log of the base experiment (Figure 5-34) contains only messages concerned with the two innermost invocations of Locations: Locations(1,Offset) and Locations(2,Offset). Note that the sums and comparisons act as a synchronization point: 16 calls to Energy2 are evaluated, their results are sent to the controller for comparison, and new positions are sent which allow the next 16 calls to be evaluated.

The base experiment has much in common with the DSN base experiment. The computations exported to the satellites consume one second of processor time and are generated at the lowest level by a call to the relatively efficient MAP operator. A few simple differences explain the fact that the speedup for the MM experiment never reached 2.0 while the speedup for the DSN experiment was nearly 3.0.

MM sends three messages for each call to Energy2 that it exports, compared to DSN's single message for each exported cross correlation. The total message passing overhead is 51.74 seconds, estimated by the sum of Accent Overhead (most of which is accounted for by message passing), Message Server, Network Server, and Message Overhead. Since 159 calls to Energy2 were exported, the total message passing overhead is 0.32 seconds per export, as compared to 0.17 for the DSN experiment. (See Figure 5-43.)

The MM program involves more expensive STARDUST computation per exported call than DSN: Other Interpreter costs are 0.14 seconds per call to Energy2 as compared to 0.05 seconds per call to the cross correlator.

The sum of these two sources of overhead for the MM experiment is 0.46 seconds for each one-second exported computation, which places a limit of 2.2 on the speedup. The actual speedup for six processors was only 1.83 because of two sources of synchronization. First, the program is fully partitioned and scheduled before any of it is evaluated. Second, the results converge on the controller after every 16 calls to Energy2. Since only one of the 160 calls to Energy2 was evaluated on the controller, the 16 calls usually had to be split unevenly among the 5 satellites.

In the coarse partitioning experiment calls to Energy were exported rather than broken down into calls to Energy2. This improved performance for configurations of 2 or 3 processors because of the decrease in message passing costs, but hurt performance for larger collections. Rather than having 16 calls to Energy2 to evaluate in parallel between synchronization points, the coarser partitioning left the system with only 2 calls to Energy, placing a limit of 2.0 on the speedup. Even this modest figure was not attained for reasons explained in Section 5.3.4.
The slow user intrinsics experiment had no unexpected results. The overhead was reduced considerably, causing more calls to be scheduled on the controller and giving much better overall performance.

5.3.3. Base Experiment

The time limit above which user functions are partitioned (the constant TTime, used by procedure UserSplit in Section 4.4 and MapSplit in Section 4.4.2) is set to 100 "jiffies" (100/60 or 1.67 seconds). The system assumes a message passing overhead of 5 jiffies, or 0.08 seconds.

The Forms sent by the controller have one of three formats: calls to REDUCE on an 8-member list, calls to Energy2, and constants representing positions. None of these messages are long: the average message length is 45 words. All incoming message contain a single integer, which gives them a length of exactly 36 words. (The message length includes 11 words of message header and the 8 words needed to point to the out-of-line content of the message and specify its length, as well as information needed by the interpreter.)

The schedule and an excerpt from the message passing history of the six processor configuration are shown in Figure 5-38 and 5-34, and a breakdown of processing costs for the controller appears in column two of Figure 5-32.

Figure 5-34 shows the portion of the message log concerned with calculating the two innermost invocations of Locations. The messages between the real time values of 36.95 and 40.33 export the calls needed to evaluate Locations(2,Offset) (where the position of Atom 3 is modified); the messages between 41.80 and 45.27 export the calls needed to evaluate Locations(1,Offset) (where the position of Atom 2 is modified); the messages between 46.95 and 49.20 send the arguments to the calls associated with Locations(1,Offset); the messages between 49.53 and 53.63 return the values of the Energy2 calls to the controller, where they are put into a list and sent to the REDUCE operator; the messages between 53.83 and 56.13 send the arguments to the calls associated with Locations(2,Offset); and the messages between 56.30 and 60.47 collect the results.

Performance could be greatly improved by sending fewer messages. For example, the 42 messages between 43.15 and 49.20 could be compressed into a single larger message for each satellite. Note that a small amount of compression already takes place: Atom 2 is sent to satellite 5 only once, despite the fact that it is used in two computations there. Atom 2 is sent twice to satellite 6 because at the time the call to Energy2(Atom2,Atom2) was exported it was not known that the two arguments were identical, and different global names were used for the two arguments. A separate message was used to resolve each global name.
<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>36.95</td>
<td>19.05</td>
<td>0.08</td>
<td>1.47</td>
<td>0.75</td>
<td>PE</td>
<td>2</td>
<td>Reduce(List1,0,+)</td>
</tr>
<tr>
<td>37.08</td>
<td>19.12</td>
<td>0.08</td>
<td>1.48</td>
<td>0.75</td>
<td>PE</td>
<td>3</td>
<td>Reduce(List2,0,+)</td>
</tr>
<tr>
<td>38.27</td>
<td>19.98</td>
<td>0.08</td>
<td>1.50</td>
<td>0.77</td>
<td>PE</td>
<td>2</td>
<td>energy2(Atom3?,Atom1)</td>
</tr>
</tbody>
</table>
| 38.40| 20.03 | 0.08 | 1.53 | 0.77 | PE | 3   | energy2(Atom3?,Atom2')|}
| 38.63| 20.08 | 0.08 | 1.57 | 0.77 | PE | 4   | energy2(Atom3?,Atom3?)|}
| 38.67| 20.13 | 0.08 | 1.57 | 0.78 | PE | 5   | energy2(Atom3?,Atom4)|
| 38.80| 20.18 | 0.08 | 1.58 | 0.78 | PE | 6   | energy2(Atom3?,Atom5)|
| 38.93| 20.23 | 0.08 | 1.62 | 0.80 | PE | 2   | energy2(Atom3?,Atom6)|
| 39.07| 20.28 | 0.08 | 1.66 | 0.80 | PE | 3   | energy2(Atom3?,Atom7)|
| 39.20| 20.33 | 0.08 | 1.67 | 0.82 | PE | 4   | energy2(Atom3?,Atom8)|
| 39.32| 20.38 | 0.08 | 1.67 | 0.82 | PE | 5   | energy2(Atom3,Atom1)|
| 39.48| 20.45 | 0.08 | 1.67 | 0.82 | PE | 6   | energy2(Atom3,Atom2')|}
| 39.65| 20.55 | 0.08 | 1.68 | 0.82 | PE | 2   | energy2(Atom3,Atom3)|
| 39.78| 20.62 | 0.08 | 1.68 | 0.82 | PE | 3   | energy2(Atom3,Atom4)|
| 39.92| 20.68 | 0.08 | 1.68 | 0.83 | PE | 4   | energy2(Atom3,Atom5)|
| 40.05| 20.73 | 0.08 | 1.70 | 0.83 | PE | 5   | energy2(Atom3,Atom6)|

< Continued in Figure 5-35 >

This figure gives an excerpt of the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute, RET for sending NewNodeValue, and NNV for receiving NewNodeValue. "Sat" is the satellite processor that is the source or destination of the message.

The excerpt gives the two iterations in which the positions of atoms 2 and 3 are perturbed. The symbols Atom1 through Atom8 refer to the original positions of the atoms; Atom2? and Atom3? refer to the perturbed positions of atoms 2 and 3; and Atom2' refers to the lower-energy choice between Atom2 and Atom2?. List1 through List4 refer to the following lists:

List1 = [Energy2(Atom3?,Atom1), Energy2(Atom3?,Atom2'),
         Energy2(Atom3?,Atom3?), Energy2(Atom3?,Atom4),
         Energy2(Atom3?,Atom5), Energy2(Atom3?,Atom6),
         Energy2(Atom3?,Atom7), Energy2(Atom3?,Atom8)]

List2 = [Energy2(Atom3,Atom1), Energy2(Atom3,Atom2'),
         Energy2(Atom3,Atom3), Energy2(Atom3,Atom4),
         Energy2(Atom3,Atom5), Energy2(Atom3,Atom6),
         Energy2(Atom3,Atom7), Energy2(Atom3,Atom8)]

List3 = [Energy2(Atom2?,Atom1), Energy2(Atom2?,Atom2?),
         Energy2(Atom2?,Atom3), Energy2(Atom2?,Atom4),
         Energy2(Atom2?,Atom5), Energy2(Atom2?,Atom6),
         Energy2(Atom2?,Atom7), Energy2(Atom2?,Atom8)]

List4 = [Energy2(Atom2,Atom1), Energy2(Atom2,Atom2),
         Energy2(Atom2,Atom3), Energy2(Atom2,Atom4),
         Energy2(Atom2,Atom5), Energy2(Atom2,Atom6),
         Energy2(Atom2,Atom7), Energy2(Atom2,Atom8)]

Figure 5-34: Messages for the Molecular Modelling Base Experiment with Six Processors
<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.20</td>
<td>20.78</td>
<td>0.08</td>
<td>1.70</td>
<td>0.85</td>
<td>PE</td>
<td>6</td>
<td>energy2(Atom3,Atom7)</td>
</tr>
<tr>
<td>40.33</td>
<td>20.87</td>
<td>0.08</td>
<td>1.72</td>
<td>0.86</td>
<td>PE</td>
<td>2</td>
<td>energy2(Atom3,Atom8)</td>
</tr>
<tr>
<td>41.80</td>
<td>21.82</td>
<td>0.08</td>
<td>1.72</td>
<td>0.87</td>
<td>PE</td>
<td>2</td>
<td>Reduce(List3,0,+    )</td>
</tr>
<tr>
<td>41.93</td>
<td>21.90</td>
<td>0.08</td>
<td>1.75</td>
<td>0.87</td>
<td>PE</td>
<td>3</td>
<td>Reduce(List4,0,+    )</td>
</tr>
<tr>
<td>43.15</td>
<td>22.77</td>
<td>0.08</td>
<td>1.75</td>
<td>0.87</td>
<td>PE</td>
<td>2</td>
<td>energy2(Atom2?,Atom1)</td>
</tr>
<tr>
<td>43.28</td>
<td>22.85</td>
<td>0.08</td>
<td>1.75</td>
<td>0.87</td>
<td>PE</td>
<td>3</td>
<td>energy2(Atom2?,Atom2?)</td>
</tr>
<tr>
<td>43.42</td>
<td>22.90</td>
<td>0.08</td>
<td>1.78</td>
<td>0.87</td>
<td>PE</td>
<td>4</td>
<td>energy2(Atom2?,Atom3)</td>
</tr>
<tr>
<td>43.55</td>
<td>22.95</td>
<td>0.08</td>
<td>1.80</td>
<td>0.87</td>
<td>PE</td>
<td>5</td>
<td>energy2(Atom2?,Atom4)</td>
</tr>
<tr>
<td>43.68</td>
<td>23.00</td>
<td>0.08</td>
<td>1.82</td>
<td>0.88</td>
<td>PE</td>
<td>6</td>
<td>energy2(Atom2?,Atom5)</td>
</tr>
<tr>
<td>43.85</td>
<td>23.07</td>
<td>0.08</td>
<td>1.82</td>
<td>0.92</td>
<td>PE</td>
<td>2</td>
<td>energy2(Atom2?,Atom6)</td>
</tr>
<tr>
<td>43.98</td>
<td>23.13</td>
<td>0.08</td>
<td>1.83</td>
<td>0.92</td>
<td>PE</td>
<td>3</td>
<td>energy2(Atom2?,Atom7)</td>
</tr>
<tr>
<td>44.12</td>
<td>23.22</td>
<td>0.08</td>
<td>1.83</td>
<td>0.92</td>
<td>PE</td>
<td>4</td>
<td>energy2(Atom2?,Atom8)</td>
</tr>
<tr>
<td>44.25</td>
<td>23.28</td>
<td>0.08</td>
<td>1.83</td>
<td>0.93</td>
<td>PE</td>
<td>6</td>
<td>energy2(Atom2,Atom1)</td>
</tr>
<tr>
<td>44.42</td>
<td>23.35</td>
<td>0.08</td>
<td>1.83</td>
<td>0.93</td>
<td>PE</td>
<td>6</td>
<td>energy2(Atom2,Atom2)</td>
</tr>
<tr>
<td>44.57</td>
<td>23.43</td>
<td>0.08</td>
<td>1.83</td>
<td>0.93</td>
<td>PE</td>
<td>2</td>
<td>energy2(Atom2,Atom3)</td>
</tr>
<tr>
<td>44.70</td>
<td>23.48</td>
<td>0.08</td>
<td>1.83</td>
<td>0.93</td>
<td>PE</td>
<td>3</td>
<td>energy2(Atom2,Atom4)</td>
</tr>
<tr>
<td>44.83</td>
<td>23.55</td>
<td>0.08</td>
<td>1.87</td>
<td>0.93</td>
<td>PE</td>
<td>4</td>
<td>energy2(Atom2,Atom5)</td>
</tr>
<tr>
<td>44.98</td>
<td>23.60</td>
<td>0.08</td>
<td>1.88</td>
<td>0.93</td>
<td>PE</td>
<td>5</td>
<td>energy2(Atom2,Atom6)</td>
</tr>
<tr>
<td>45.12</td>
<td>23.68</td>
<td>0.08</td>
<td>1.90</td>
<td>0.93</td>
<td>PE</td>
<td>6</td>
<td>energy2(Atom2,Atom7)</td>
</tr>
<tr>
<td>45.27</td>
<td>23.77</td>
<td>0.08</td>
<td>1.90</td>
<td>0.93</td>
<td>PE</td>
<td>2</td>
<td>energy2(Atom2,Atom8)</td>
</tr>
<tr>
<td>46.95</td>
<td>24.97</td>
<td>0.08</td>
<td>1.92</td>
<td>0.93</td>
<td>RET</td>
<td>5</td>
<td>Atom1</td>
</tr>
<tr>
<td>47.03</td>
<td>25.00</td>
<td>0.08</td>
<td>1.92</td>
<td>0.93</td>
<td>RET</td>
<td>2</td>
<td>Atom3</td>
</tr>
<tr>
<td>47.12</td>
<td>25.03</td>
<td>0.08</td>
<td>1.92</td>
<td>0.93</td>
<td>RET</td>
<td>3</td>
<td>Atom4</td>
</tr>
<tr>
<td>47.22</td>
<td>25.06</td>
<td>0.08</td>
<td>1.95</td>
<td>0.93</td>
<td>RET</td>
<td>4</td>
<td>Atom5</td>
</tr>
<tr>
<td>47.30</td>
<td>25.06</td>
<td>0.08</td>
<td>1.95</td>
<td>0.93</td>
<td>RET</td>
<td>5</td>
<td>Atom6</td>
</tr>
<tr>
<td>47.40</td>
<td>25.12</td>
<td>0.08</td>
<td>1.97</td>
<td>0.93</td>
<td>RET</td>
<td>6</td>
<td>Atom7</td>
</tr>
<tr>
<td>47.48</td>
<td>25.15</td>
<td>0.08</td>
<td>1.97</td>
<td>0.93</td>
<td>RET</td>
<td>2</td>
<td>Atom8</td>
</tr>
<tr>
<td>47.58</td>
<td>25.17</td>
<td>0.08</td>
<td>1.97</td>
<td>0.95</td>
<td>RET</td>
<td>6</td>
<td>Atom2</td>
</tr>
<tr>
<td>47.63</td>
<td>25.17</td>
<td>0.08</td>
<td>1.97</td>
<td>0.95</td>
<td>RET</td>
<td>4</td>
<td>Atom2</td>
</tr>
<tr>
<td>47.70</td>
<td>25.17</td>
<td>0.08</td>
<td>1.97</td>
<td>0.97</td>
<td>RET</td>
<td>3</td>
<td>Atom2</td>
</tr>
<tr>
<td>47.78</td>
<td>25.17</td>
<td>0.08</td>
<td>1.98</td>
<td>1.02</td>
<td>RET</td>
<td>2</td>
<td>Atom2</td>
</tr>
<tr>
<td>47.87</td>
<td>25.18</td>
<td>0.08</td>
<td>1.98</td>
<td>1.02</td>
<td>RET</td>
<td>6</td>
<td>Atom2</td>
</tr>
<tr>
<td>47.93</td>
<td>25.18</td>
<td>0.08</td>
<td>2.02</td>
<td>1.02</td>
<td>RET</td>
<td>5</td>
<td>Atom2</td>
</tr>
<tr>
<td>48.05</td>
<td>25.20</td>
<td>0.08</td>
<td>2.05</td>
<td>1.02</td>
<td>RET</td>
<td>2</td>
<td>Atom1</td>
</tr>
<tr>
<td>48.13</td>
<td>25.20</td>
<td>0.08</td>
<td>2.06</td>
<td>1.02</td>
<td>RET</td>
<td>4</td>
<td>Atom3</td>
</tr>
<tr>
<td>48.22</td>
<td>25.20</td>
<td>0.08</td>
<td>2.07</td>
<td>1.03</td>
<td>RET</td>
<td>5</td>
<td>Atom4</td>
</tr>
<tr>
<td>48.30</td>
<td>25.20</td>
<td>0.08</td>
<td>2.10</td>
<td>1.03</td>
<td>RET</td>
<td>6</td>
<td>Atom5</td>
</tr>
<tr>
<td>48.40</td>
<td>25.22</td>
<td>0.08</td>
<td>2.10</td>
<td>1.08</td>
<td>RET</td>
<td>2</td>
<td>Atom6</td>
</tr>
<tr>
<td>48.47</td>
<td>25.23</td>
<td>0.08</td>
<td>2.13</td>
<td>1.08</td>
<td>RET</td>
<td>3</td>
<td>Atom7</td>
</tr>
<tr>
<td>48.53</td>
<td>25.27</td>
<td>0.08</td>
<td>2.17</td>
<td>1.08</td>
<td>RET</td>
<td>4</td>
<td>Atom8</td>
</tr>
<tr>
<td>48.63</td>
<td>25.42</td>
<td>0.08</td>
<td>2.17</td>
<td>1.10</td>
<td>RET</td>
<td>3</td>
<td>Atom2?</td>
</tr>
<tr>
<td>48.90</td>
<td>25.43</td>
<td>0.08</td>
<td>2.18</td>
<td>1.10</td>
<td>RET</td>
<td>6</td>
<td>Atom2?</td>
</tr>
<tr>
<td>48.97</td>
<td>25.43</td>
<td>0.08</td>
<td>2.22</td>
<td>1.10</td>
<td>RET</td>
<td>5</td>
<td>Atom2?</td>
</tr>
<tr>
<td>49.03</td>
<td>25.43</td>
<td>0.08</td>
<td>2.23</td>
<td>1.12</td>
<td>RET</td>
<td>4</td>
<td>Atom2?</td>
</tr>
<tr>
<td>49.12</td>
<td>25.45</td>
<td>0.08</td>
<td>2.25</td>
<td>1.12</td>
<td>RET</td>
<td>3</td>
<td>Atom2?</td>
</tr>
<tr>
<td>49.20</td>
<td>25.48</td>
<td>0.08</td>
<td>2.28</td>
<td>1.12</td>
<td>RET</td>
<td>2</td>
<td>Atom2?</td>
</tr>
<tr>
<td>49.30</td>
<td>25.52</td>
<td>0.08</td>
<td>2.30</td>
<td>1.12</td>
<td>&lt; controller enters idle loop &gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49.53</td>
<td>25.55</td>
<td>0.23</td>
<td>2.30</td>
<td>1.12</td>
<td>NNV</td>
<td>2</td>
<td>energy2(Atom2,Atom3)</td>
</tr>
</tbody>
</table>

*Figure 5-35: Messages for the Molecular Modelling Base Experiment with Six Processors, Continued from Figure 5-34*
<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>49.58</td>
<td>25.57</td>
<td>0.23</td>
<td>2.30</td>
<td>1.12</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>49.63</td>
<td>25.58</td>
<td>0.23</td>
<td>2.30</td>
<td>1.12</td>
<td>6</td>
<td>energy2(Atom2,Atom2)</td>
</tr>
<tr>
<td>49.67</td>
<td>25.60</td>
<td>0.23</td>
<td>2.30</td>
<td>1.12</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>49.75</td>
<td>25.62</td>
<td>0.25</td>
<td>2.30</td>
<td>1.12</td>
<td>4</td>
<td>energy2(Atom2,Atom5)</td>
</tr>
<tr>
<td>49.78</td>
<td>25.65</td>
<td>0.25</td>
<td>2.30</td>
<td>1.12</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>49.87</td>
<td>25.65</td>
<td>0.27</td>
<td>2.30</td>
<td>1.12</td>
<td>5</td>
<td>energy2(Atom2,Atom1)</td>
</tr>
<tr>
<td>49.88</td>
<td>25.65</td>
<td>0.27</td>
<td>2.30</td>
<td>1.12</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>50.00</td>
<td>25.67</td>
<td>0.28</td>
<td>2.32</td>
<td>1.15</td>
<td>3</td>
<td>energy2(Atom2,Atom4)</td>
</tr>
<tr>
<td>50.03</td>
<td>25.68</td>
<td>0.28</td>
<td>2.32</td>
<td>1.15</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>50.72</td>
<td>25.70</td>
<td>0.82</td>
<td>2.35</td>
<td>1.17</td>
<td>4</td>
<td>energy2(Atom2,Atom3)</td>
</tr>
<tr>
<td>50.85</td>
<td>25.73</td>
<td>0.82</td>
<td>2.37</td>
<td>1.18</td>
<td>6</td>
<td>energy2(Atom2,Atom7)</td>
</tr>
<tr>
<td>50.88</td>
<td>25.73</td>
<td>0.82</td>
<td>2.37</td>
<td>1.18</td>
<td>5</td>
<td>energy2(Atom2,Atom6)</td>
</tr>
<tr>
<td>50.93</td>
<td>25.75</td>
<td>0.82</td>
<td>2.37</td>
<td>1.18</td>
<td>2</td>
<td>energy2(Atom2,Atom8)</td>
</tr>
<tr>
<td>50.98</td>
<td>25.80</td>
<td>0.82</td>
<td>2.37</td>
<td>1.18</td>
<td></td>
<td>RET 3 List4</td>
</tr>
<tr>
<td>51.08</td>
<td>25.85</td>
<td>0.82</td>
<td>2.40</td>
<td>1.18</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>51.42</td>
<td>25.88</td>
<td>1.05</td>
<td>2.42</td>
<td>1.18</td>
<td>3</td>
<td>energy2(Atom2,Atom2?)</td>
</tr>
<tr>
<td>51.43</td>
<td>25.88</td>
<td>1.05</td>
<td>2.42</td>
<td>1.18</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>51.78</td>
<td>25.90</td>
<td>1.30</td>
<td>2.42</td>
<td>1.22</td>
<td>4</td>
<td>energy2(Atom2,Atom8)</td>
</tr>
<tr>
<td>51.85</td>
<td>25.92</td>
<td>1.30</td>
<td>2.42</td>
<td>1.22</td>
<td>6</td>
<td>energy2(Atom2,Atom5)</td>
</tr>
<tr>
<td>51.95</td>
<td>25.92</td>
<td>1.30</td>
<td>2.42</td>
<td>1.23</td>
<td>5</td>
<td>energy2(Atom2,Atom4)</td>
</tr>
<tr>
<td>52.00</td>
<td>25.92</td>
<td>1.30</td>
<td>2.42</td>
<td>1.23</td>
<td>2</td>
<td>energy2(Atom2,Atom1)</td>
</tr>
<tr>
<td>52.05</td>
<td>25.95</td>
<td>1.30</td>
<td>2.42</td>
<td>1.23</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>52.37</td>
<td>25.95</td>
<td>1.55</td>
<td>2.42</td>
<td>1.23</td>
<td>3</td>
<td>energy2(Atom2,Atom7)</td>
</tr>
<tr>
<td>52.40</td>
<td>25.97</td>
<td>1.55</td>
<td>2.42</td>
<td>1.23</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>52.63</td>
<td>25.97</td>
<td>1.72</td>
<td>2.43</td>
<td>1.23</td>
<td>3</td>
<td>Reduce(List4,0,+</td>
</tr>
<tr>
<td>52.65</td>
<td>25.98</td>
<td>1.72</td>
<td>2.43</td>
<td>1.23</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>53.03</td>
<td>26.00</td>
<td>2.05</td>
<td>2.43</td>
<td>1.25</td>
<td>5</td>
<td>energy2(Atom2,Atom8)</td>
</tr>
<tr>
<td>53.07</td>
<td>26.02</td>
<td>2.05</td>
<td>2.43</td>
<td>1.25</td>
<td></td>
<td>RET 3 List3</td>
</tr>
<tr>
<td>53.15</td>
<td>26.07</td>
<td>2.05</td>
<td>2.45</td>
<td>1.25</td>
<td></td>
<td>&lt;controller enters idle loop&gt;</td>
</tr>
<tr>
<td>53.63</td>
<td>26.08</td>
<td>2.47</td>
<td>2.45</td>
<td>1.25</td>
<td>2</td>
<td>Reduce(List3,0,+</td>
</tr>
<tr>
<td>53.83</td>
<td>26.28</td>
<td>2.47</td>
<td>2.45</td>
<td>1.25</td>
<td></td>
<td>RET 3 Atom8</td>
</tr>
<tr>
<td>53.95</td>
<td>26.28</td>
<td>2.47</td>
<td>2.45</td>
<td>1.27</td>
<td></td>
<td>6 Atom7</td>
</tr>
<tr>
<td>54.06</td>
<td>26.30</td>
<td>2.47</td>
<td>2.47</td>
<td>1.27</td>
<td></td>
<td>RET 5 Atom6</td>
</tr>
<tr>
<td>54.13</td>
<td>26.30</td>
<td>2.47</td>
<td>2.47</td>
<td>1.28</td>
<td></td>
<td>RET 4 Atom5</td>
</tr>
<tr>
<td>54.22</td>
<td>26.30</td>
<td>2.47</td>
<td>2.47</td>
<td>1.32</td>
<td></td>
<td>3 Atom4</td>
</tr>
<tr>
<td>54.28</td>
<td>26.33</td>
<td>2.47</td>
<td>2.48</td>
<td>1.32</td>
<td></td>
<td>RET 2 Atom3</td>
</tr>
<tr>
<td>54.33</td>
<td>26.35</td>
<td>2.47</td>
<td>2.48</td>
<td>1.32</td>
<td></td>
<td>RET 4 Atom3</td>
</tr>
<tr>
<td>54.42</td>
<td>26.37</td>
<td>2.47</td>
<td>2.48</td>
<td>1.32</td>
<td></td>
<td>3 Atom3</td>
</tr>
<tr>
<td>54.48</td>
<td>26.37</td>
<td>2.47</td>
<td>2.50</td>
<td>1.35</td>
<td></td>
<td>RET 6 Atom3</td>
</tr>
<tr>
<td>54.57</td>
<td>26.40</td>
<td>2.47</td>
<td>2.50</td>
<td>1.35</td>
<td></td>
<td>RET 5 Atom3</td>
</tr>
<tr>
<td>54.67</td>
<td>26.40</td>
<td>2.47</td>
<td>2.50</td>
<td>1.38</td>
<td></td>
<td>RET 6 Atom2'</td>
</tr>
<tr>
<td>54.73</td>
<td>26.40</td>
<td>2.47</td>
<td>2.53</td>
<td>1.38</td>
<td></td>
<td>RET 5 Atom1</td>
</tr>
<tr>
<td>54.92</td>
<td>26.47</td>
<td>2.47</td>
<td>2.57</td>
<td>1.40</td>
<td></td>
<td>RET 2 Atom1</td>
</tr>
<tr>
<td>55.07</td>
<td>26.48</td>
<td>2.47</td>
<td>2.58</td>
<td>1.42</td>
<td></td>
<td>RET 3 Atom2'</td>
</tr>
<tr>
<td>55.10</td>
<td>26.48</td>
<td>2.47</td>
<td>2.60</td>
<td>1.43</td>
<td></td>
<td>RET 5 Atom4</td>
</tr>
<tr>
<td>55.20</td>
<td>26.53</td>
<td>2.47</td>
<td>2.62</td>
<td>1.43</td>
<td></td>
<td>RET 6 Atom6</td>
</tr>
<tr>
<td>55.28</td>
<td>26.55</td>
<td>2.47</td>
<td>2.62</td>
<td>1.46</td>
<td></td>
<td>RET 2 Atom6</td>
</tr>
</tbody>
</table>

<Continued in Figure 5-37>

Figure 5-36: Messages for the Molecular Modelling Base Experiment with Six Processors, Continued from Figure 5-35
The calls to Energy2 were sent before the arguments because the system could not be certain that Energy2 needed both of its arguments to complete. If Energy2 had been defined with strict parameter passing (this option is not available in STARDUST 1.0), then the sending of the call could have been deferred until the arguments were ready and they would all have been sent together.
### Table 1: Processor Scheduling

<table>
<thead>
<tr>
<th>Processor</th>
<th>Scheduled Calls to Reduce</th>
<th>Scheduled Calls to Energy2</th>
<th>Executed Calls to Energy2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>39</td>
<td>35</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>30</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>30</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>30</td>
<td>28</td>
</tr>
</tbody>
</table>

The figures in the fourth column are smaller than the figures in the third column because some of the scheduled calls to Energy2 were not executed due to redundant subcomputation elimination.

**Figure 5-38:** Schedule for the Molecular Modelling Base Experiment with Six Processors

Further compression could have taken place at the message packing level as described in the section on "message sharing" (Section 7.1.2.5). Groups of message to the same processor could be combined into a single message even when they had no logical relationship with one another.

The schedule in Figure 5-38 shows that very few computations were scheduled on the controller. This happened because the system correctly recognized that the controller would be very busy sending and receiving messages.

Figure 5-38 also shows that processor 2 was more heavily scheduled than the rest. The system recognized the synchronization point after every 16 calls to Energy2, which rendered all satellites equally available despite the fact that satellites 3 through 6 were less heavily scheduled. The system then simply scheduled the 16 calls in order:

\[ [2,3,4,5,6,2,3,4,5,6,2,3,4,5,6,2] \]

When the first group of 16 calls was scheduled, the controller did not yet know that it would be heavily loaded with message traffic, and the first call was scheduled on the controller. Again, they were simply scheduled in order:

\[ [1,2,3,4,5,6,2,3,4,5,6,2,3,4,5,6] \]

### 5.3.4. Coarse Partitioning

The partitioning granularity parameter TTime is raised from 100 to 1000 jiffies (from 1.67 to 16.67 seconds). Since they have time estimates of only 480 jiffies (8 seconds), calls to Energy are exported directly rather than broken up into calls to Energy2.

The message passing history and schedule are shown in Figures 5-39 and 5-41, and a breakdown of processing costs for the controller appears in column three of Figure 5-32.
The increase in granularity greatly reduced the message passing overhead but reduced the potential parallelism among the energy calculations from 16 to 2, since synchronization takes place after 2 calls to Energy rather than 16 calls to Energy2. The message log shows that Locations(2,Offset) was scheduled as a single unit, placing 32 calls to Energy2 on satellite 3 while all other processors sat idle. This unfortunate scheduling decision was due to an error in the time estimate for Locations: it should have read $t^*Etime^2$ rather than $t^*Etime$. Thus with $t = 2$ and $Etime = 480$, the call to Locations(2,Offset) was just under the limit of 1000 and not partitioned. If the time estimate had been correct, the calls to Energy associated with Locations(2,Offset) would have been executed in parallel, and only the call to Locations(1,Offset) would have remained unpartitioned.

It is instructive to note that a slight restatement of the program would have eliminated parallelism altogether. If the following lines from Locations

\[ \text{Energy}(n, \text{Elist2}) < \text{Energy}(n, \text{Elist1}) \]

had been grouped into a single call to CompareEnergy, defined as

\[
\text{Let CompareEnergy}(n: \text{Integer}; \ p1: \text{PosList}; \ p2: \text{PosList}): \text{Integer};
\text{Time Etime}^2;
\text{Energy}(n, p1) < \text{Energy}(n, p2);
\text{End;}
\]

then CompareEnergy, with a time estimate of 960, would not have been partitioned further. The available parallelism was at a smaller grain than the 1000 jiffy limit recommended by the user, and only the fact that larger units were not available for exportation allowed the parallelism to be found.

5.3.5. Slow User Intrinsics

The processing time for Energy2 is increased from one second to five seconds. All other parameters are set as in the base message experiment. This experiment was designed to test the effect of speeding up the interpreter and message passing by a factor of five, so all reported execution times for this experiment have been divided by five.

The schedule is shown in Figure 5-42 and a breakdown of processing costs for the controller appears in column four of Figure 5-32. The message passing history is similar to that of the base experiment found in Figure 5-34 and is omitted because of its length.

Note that since communications costs are relatively low compared with processing costs, more calls to Energy2 are scheduled on the controller than in previous examples. This
<table>
<thead>
<tr>
<th>Real</th>
<th>SD</th>
<th>Idle</th>
<th>MsgS</th>
<th>NetS</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td>&lt;start&gt;</td>
</tr>
<tr>
<td>0.45</td>
<td>0.22</td>
<td>0.00</td>
<td>0.03</td>
<td>0.03</td>
<td>PE</td>
<td>2</td>
<td>energy(3,locations(9,3))</td>
</tr>
<tr>
<td>0.95</td>
<td>0.58</td>
<td>0.00</td>
<td>0.06</td>
<td>0.03</td>
<td>PE</td>
<td>2</td>
<td>energy(2,locations(8,3))</td>
</tr>
<tr>
<td>1.47</td>
<td>0.95</td>
<td>0.00</td>
<td>0.07</td>
<td>0.03</td>
<td>PE</td>
<td>2</td>
<td>energy(1,locations(7,3))</td>
</tr>
<tr>
<td>1.52</td>
<td>0.98</td>
<td>0.00</td>
<td>0.08</td>
<td>0.03</td>
<td>PE</td>
<td>3</td>
<td>energy(1,locations(7,3))</td>
</tr>
<tr>
<td>2.25</td>
<td>1.46</td>
<td>0.00</td>
<td>0.10</td>
<td>0.05</td>
<td>PE</td>
<td>2</td>
<td>energy(8,locations(6,3))</td>
</tr>
<tr>
<td>2.32</td>
<td>1.48</td>
<td>0.00</td>
<td>0.10</td>
<td>0.05</td>
<td>PE</td>
<td>3</td>
<td>energy(8,locations(6,3))</td>
</tr>
<tr>
<td>2.92</td>
<td>1.88</td>
<td>0.00</td>
<td>0.13</td>
<td>0.05</td>
<td>PE</td>
<td>2</td>
<td>energy(7,locations(5,3))</td>
</tr>
<tr>
<td>3.02</td>
<td>1.90</td>
<td>0.00</td>
<td>0.17</td>
<td>0.07</td>
<td>PE</td>
<td>3</td>
<td>energy(7,locations(5,3))</td>
</tr>
<tr>
<td>3.82</td>
<td>2.47</td>
<td>0.00</td>
<td>0.17</td>
<td>0.07</td>
<td>PE</td>
<td>2</td>
<td>energy(6,locations(4,3))</td>
</tr>
<tr>
<td>3.92</td>
<td>2.52</td>
<td>0.00</td>
<td>0.18</td>
<td>0.08</td>
<td>PE</td>
<td>3</td>
<td>energy(6,locations(4,3))</td>
</tr>
<tr>
<td>4.65</td>
<td>3.02</td>
<td>0.00</td>
<td>0.20</td>
<td>0.08</td>
<td>PE</td>
<td>2</td>
<td>energy(5,locations(3,3))</td>
</tr>
<tr>
<td>4.73</td>
<td>3.07</td>
<td>0.00</td>
<td>0.22</td>
<td>0.08</td>
<td>PE</td>
<td>3</td>
<td>energy(5,locations(3,3))</td>
</tr>
<tr>
<td>5.38</td>
<td>3.56</td>
<td>0.00</td>
<td>0.22</td>
<td>0.10</td>
<td>PE</td>
<td>2</td>
<td>energy(4,locations(2,3))</td>
</tr>
<tr>
<td>5.47</td>
<td>3.58</td>
<td>0.00</td>
<td>0.22</td>
<td>0.12</td>
<td>PE</td>
<td>3</td>
<td>energy(4,locations(2,3))</td>
</tr>
<tr>
<td>5.70</td>
<td>3.72</td>
<td>0.00</td>
<td>0.27</td>
<td>0.13</td>
<td>PE</td>
<td>3</td>
<td>locations(2,3)</td>
</tr>
<tr>
<td>5.80</td>
<td>3.77</td>
<td>0.00</td>
<td>0.28</td>
<td>0.13</td>
<td>PC</td>
<td>3</td>
<td>locations(2,3)</td>
</tr>
<tr>
<td>7.38</td>
<td>4.97</td>
<td>0.00</td>
<td>0.28</td>
<td>0.15</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51.10</td>
<td>5.13</td>
<td>43.58</td>
<td>0.30</td>
<td>0.17</td>
<td>NNV</td>
<td>3</td>
<td>locations(2,3)</td>
</tr>
<tr>
<td>51.28</td>
<td>5.22</td>
<td>43.58</td>
<td>0.30</td>
<td>0.17</td>
<td>RET</td>
<td>2</td>
<td>locations(2,3)</td>
</tr>
<tr>
<td>51.37</td>
<td>5.27</td>
<td>43.58</td>
<td>0.30</td>
<td>0.17</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>59.52</td>
<td>5.28</td>
<td>51.63</td>
<td>0.32</td>
<td>0.17</td>
<td>NNV</td>
<td>3</td>
<td>energy(4,locations(2,3))</td>
</tr>
<tr>
<td>59.55</td>
<td>5.30</td>
<td>51.63</td>
<td>0.32</td>
<td>0.17</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60.25</td>
<td>5.32</td>
<td>52.27</td>
<td>0.32</td>
<td>0.18</td>
<td>NNV</td>
<td>2</td>
<td>energy(4,locations(2,3))</td>
</tr>
<tr>
<td>60.35</td>
<td>5.42</td>
<td>52.27</td>
<td>0.32</td>
<td>0.18</td>
<td>RET</td>
<td>3</td>
<td>locations(3,3)</td>
</tr>
<tr>
<td>60.52</td>
<td>5.53</td>
<td>52.27</td>
<td>0.33</td>
<td>0.18</td>
<td>RET</td>
<td>2</td>
<td>locations(3,3)</td>
</tr>
<tr>
<td>60.57</td>
<td>5.57</td>
<td>52.27</td>
<td>0.36</td>
<td>0.18</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>69.32</td>
<td>5.58</td>
<td>60.90</td>
<td>0.37</td>
<td>0.20</td>
<td>NNV</td>
<td>3</td>
<td>energy(5,locations(3,3))</td>
</tr>
<tr>
<td>69.35</td>
<td>5.62</td>
<td>60.90</td>
<td>0.37</td>
<td>0.20</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>69.45</td>
<td>5.63</td>
<td>60.95</td>
<td>0.37</td>
<td>0.22</td>
<td>NNV</td>
<td>2</td>
<td>energy(5,locations(3,3))</td>
</tr>
<tr>
<td>69.55</td>
<td>5.72</td>
<td>60.95</td>
<td>0.37</td>
<td>0.22</td>
<td>RET</td>
<td>3</td>
<td>locations(4,3)</td>
</tr>
<tr>
<td>69.75</td>
<td>5.82</td>
<td>60.95</td>
<td>0.38</td>
<td>0.23</td>
<td>RET</td>
<td>2</td>
<td>locations(4,3)</td>
</tr>
<tr>
<td>69.80</td>
<td>5.85</td>
<td>60.95</td>
<td>0.38</td>
<td>0.23</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78.52</td>
<td>5.87</td>
<td>69.53</td>
<td>0.40</td>
<td>0.25</td>
<td>NNV</td>
<td>3</td>
<td>energy(6,locations(4,3))</td>
</tr>
<tr>
<td>78.53</td>
<td>5.88</td>
<td>69.53</td>
<td>0.40</td>
<td>0.25</td>
<td>&lt;controller enters idle loop&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>78.68</td>
<td>5.90</td>
<td>69.62</td>
<td>0.42</td>
<td>0.27</td>
<td>NNV</td>
<td>2</td>
<td>energy(6,locations(4,3))</td>
</tr>
</tbody>
</table>

This figure gives the message passing history from the point of view of the controller. "Real" is the elapsed real time from the start of the experiment in seconds. "SD", "Idle", "MsgS", and "NetS" are the run times in seconds of the interpreter, a low priority background task, the message server, and the network server. "Msg" is the type of message: PE for PleaseExecute (fully complete), PC for PleaseExecute (only complete), RET for sending a NewNodeValue message, and NNV for receiving a NewNodeValue message. "Sat" is the satellite processor that is the source or destination of the message.

"Locations(n,Offset)" refers to the locations after n iterations, while "Locations(n,Offset)?" refers to these same locations with a trial perturbation.

Figure 5-39: Messages for the Molecular Modelling Coarse Partitioning Experiment with Six Processors
### Processor Schedules

<table>
<thead>
<tr>
<th>Processor</th>
<th>Scheduled Calls to Reduce</th>
<th>Scheduled Calls to Energy2</th>
<th>Executed Calls to Energy2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The figures in the fourth column are equal to the figures in the third column, showing that no calls to Energy2 were removed by redundant subcomputation elimination.

### Schedule for the Molecular Modelling Coarse Partitioning Experiment with Six Processors

The controller has more time left over from message passing for doing useful computations.

**Figure 5-40:** Messages for the Molecular Modelling Coarse Partitioning Experiment with Six Processors, Continued from Figure 5-39
The figures in the fourth column are smaller than the figures in the third column because some of the scheduled calls to Energy2 were not executed due to redundant subcomputation elimination.

**Figure 5-42**: Schedule for the Molecular Modelling Slow User Intrinsics Experiment with Six Processors

5.3.6. Summary

This set of experiments gave moderately encouraging results. They show that speedups of over 3.0 could be attained if the interpreter and message passing speed could be improved by a factor of five and point to some important possibilities for reducing message traffic.

5.4. Partitioning Granularity

The following simple calculation can be used to estimate a reasonable value for TTime (TTime, the partitioning granularity parameter, is discussed in Section 4.4). Let N and M be the number of processors and the average overhead incurred on the controller for each exported computation. To keep the N-1 satellites from becoming idle, the average execution time of the exported computations must be at least M(N-1).

The maximum number of processors used in any experiment was 6, which will be used for the value of N. The total overhead was estimated by the sum of all time components except Statistics Collection and User Intrinsics, and the value of M was the quotient of this sum and the number of exported calls. The values of M for the base versions of the DSN, Quick Sort, and Molecular Modelling experiments were 0.27, 1.57, and 0.48 (see Figure 5-43), which gives values for M(N-1) of 1.35, 7.85, and 2.40 seconds.

The standard value of TTime, 100 "jiffies" or 1.67 seconds, is thus on the low side, particularly since it acts as a maximum and the actual average execution times were lower. The Optimal Partitioning version of the DSN experiment shows that performance could indeed be improved by raising it. Some improvement was also shown for the Coarse Partitioning version of the Molecular Modelling experiment when only two or three processors were used, but for larger collections of processors performance was degraded by the fact that the coar-
These statistics are taken from Figures 5-6, 5-22, and 5-32. "Communication Costs" is the sum of Accent Overhead, Message Server, Network Server, and Message Overhead, although a part of the Accent Overhead should be included under Interpretive Overhead since it represents the cost of allocating memory. The top row gives the number of exported computations; all other figures are in units of seconds per exported computation.

Figure 5-43: Overhead per Exported Computation in the Base Version of Three Experiments

ser partitioning hid some lower level parallelism. No coarser partitioning experiment was performed for the quick sort experiment.

5.5. Recovery from Overeager Evaluation

This example shows how STARDUST recovers from an erroneous assumption that the arguments to a function could be evaluated as though the function had strict semantics. The rationale and mechanisms for this "overeager evaluation" are described in Section 4.6.4. The following STARDUST code was used.

```
Let Omega(x:Integer):Integer = Omega(x+1);

Let g(L: List Of Integer) = [Atomic] Begin
    Time 50;
    1+L[2];
End;

Let h(n: Integer) = [Atomic] Begin
    Time 50;
    List Of Integer:[Omega(0),n+1];
End;
```

Figure 5-44 presents the messages sent and received by the controller when $g(h(100))$ was given to the system to evaluate. The function $g$ is exported to satellite 2, while its argument $h(100)$ is exported to satellite 3. Since $h(100)$ is given an FComplete action, the computation would have diverged but for the five-second timeout initiated by the controller. (As mentioned in Section 4.6.4 this fixed-length timeout shows the validity of the recovery
mechanisms but is inadequate for a practical system.) When the timeout occurs, \( h(100) \) is rescheduled on satellite 2, allowing its components to be used as soon as they are calculated.
The function \( g \) soon terminates, causing the shadow action on \( h(100) \) to abort, in turn causing a LostInterest message to be sent to the controller. The controller then forwards this message to the two satellites that were attempting to calculate \( h(100) \). Finally the results of applying \( g \) to \( h(100) \) are received by the controller and the computation terminates.

<table>
<thead>
<tr>
<th>Time</th>
<th>Msg</th>
<th>Sat</th>
<th>Form</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>&lt;start&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.18</td>
<td>PE 2 ( g(h(100)) )</td>
<td>2</td>
<td>h(100) is sent by global name</td>
<td></td>
</tr>
<tr>
<td>0.23</td>
<td>PE 3 ( h(100) )</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.30</td>
<td>&lt;controller waits&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.30</td>
<td>PE 2 ( h(100) )</td>
<td>2</td>
<td></td>
<td>Five-second timeout</td>
</tr>
<tr>
<td>5.35</td>
<td>&lt;controller waits&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.70</td>
<td>RLI 2 ( h(100) )</td>
<td>2</td>
<td>Receive LostInterest from Satellite 2</td>
<td></td>
</tr>
<tr>
<td>5.70</td>
<td>SLI 2 ( h(100) )</td>
<td>2</td>
<td>Could be avoided with a special test</td>
<td></td>
</tr>
<tr>
<td>5.70</td>
<td>SLI 3 ( h(100) )</td>
<td>3</td>
<td>Send LostInterest to Satellite 3</td>
<td></td>
</tr>
<tr>
<td>5.78</td>
<td>&lt;controller waits&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.92</td>
<td>NV 2 102</td>
<td>2</td>
<td>Satellite 2 returns value of ( g(h(100)) )</td>
<td></td>
</tr>
<tr>
<td>5.95</td>
<td>&lt;finish&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This figure gives the message passing history from the point of view of the controller. "Time" is the elapsed real time from the start of the experiment. "Msg" is the message type: PE for PleaseExecute, NV for NewNodeValue, SLI for Send LostInterest, and RLI for Receive LostInterest. "Sat" is the satellite processors that is the source or destination of the message.

Figure 5-44: Recovery from Overeager Evaluation

5.6. An Example of RSCE

A simple recursive Fibonacci number generator shows the effectiveness of the redundant subcomputation elimination mechanism. The execution time for the following program grows approximately linearly in STARDUST, while the execution time for the analogous program in PASCAL grows exponentially. Figure 5-45 presents execution times for PASCAL and STARDUST on a single processor.

Let \( \text{fib}(n: \text{Integer}) = \)  
\( \text{if } n < 3 \text{ Then 1 Else fib}(n-1) + \text{fib}(n-2); \)
The plots give the time in seconds taken to calculate the $n^{th}$ Fibonacci number. The solid line gives execution times for Pascal (actual data points are at all integers between 1 and 24) and the dashed line gives execution times for Stardust (actual data points are at multiples of five between 5 and 50).

Figure 5-45: Execution Time for Recursive Fibonacci Program in Pascal and Stardust

5.7. Recovery from Process Failure

Process number four in the six processor configuration for the base experiment of the DSN example was killed about seven seconds from the start of the run. At this point only one of the five cross correlations that had been scheduled on this processor had returned a value, so the remaining four were scheduled on the surviving satellite processors. The total processing time was 13.5 seconds, only 1.4 seconds longer than the uninterrupted six processor configuration and nearly identical to the total processing time for the five processor configuration.\(^4\) The schedule and the message passing history are given in Figures 5-46 and 5-47.

The performance figures are probably better than those that would typically result from a processor failure for several reasons.

\(^4\)The interpreter was not collecting statistics so its execution time is compared with those presented in Figure 5-3 rather than with those in later sections.
• The rescheduled computations divided evenly over the surviving satellites. If a single extra cross correlation had needed to be rescheduled, one of the satellites would have had one more cross correlation to perform than the rest and delayed the entire computation by over a second.

• There were no actions waiting to send results to the actions on the failed processor. Aborting and restarting such actions would have incurred additional overhead.

• The process rather than the processor was killed, causing a message to be sent immediately to the survivors. If it had been necessary to discover the process death by timeouts, a much longer delay would have resulted.

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
<th>Processor 5</th>
<th>Processor 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>xc(1,2)</td>
<td>xc(1,3)</td>
<td>xc(1,4)</td>
<td>xc(1,5)</td>
<td>xc(1,6)</td>
<td>xc(1,7)</td>
</tr>
<tr>
<td>xc(3,6)</td>
<td>xc(1,8)</td>
<td>xc(2,3)</td>
<td>xc(2,4)*</td>
<td>xc(2,5)</td>
<td>xc(2,6)</td>
</tr>
<tr>
<td>xc(3,8)</td>
<td>xc(2,7)</td>
<td>xc(2,8)</td>
<td>xc(3,4)*</td>
<td>xc(3,5)</td>
<td>xc(3,7)</td>
</tr>
<tr>
<td>xc(4,5)</td>
<td>xc(4,6)</td>
<td>xc(4,7)*</td>
<td>xc(4,8)</td>
<td>xc(5,6)</td>
<td></td>
</tr>
<tr>
<td>xc(5,7)</td>
<td>xc(5,8)</td>
<td>xc(6,7)*</td>
<td>xc(6,7)</td>
<td>xc(4,7)</td>
<td></td>
</tr>
<tr>
<td>xc(3,4)</td>
<td>xc(2,4)</td>
<td>xc(6,8)</td>
<td>xc(7,8)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The cross correlations marked with an asterisk were rescheduled.

Figure 5-46: DSN Schedule for a Six Processor Configuration, Base Experiment with Processor Failure
This figure gives the message passing history from the point of view of the controller. "Time" is the elapsed real time from the start of the experiment. "Msg" is the message type: PE for PleaseExecute and NV for NewNodeValue. "Sat" is the satellite processors that is the source or destination of the message. The cross correlations marked with an asterisk were rescheduled.

Figure 5.47: Messages for the DSN Base Experiment with Six Processors, of which Processor Number Four Falls
Chapter 6
Comparison with Other Work

When I count, there are only you and I together
But when I look ahead up the white road
There is always another one walking beside you

This chapter compares STARDUST with the data flow architecture Id; the reduction architecture AMPS; and the task monitoring and invocation system called Contract Nets. Each of the three systems is discussed in turn, and a final section compares them with STARDUST.

6.1. Data Flow and Reduction Architectures

STARDUST is related to the work that has been done in the last decade on highly parallel general purpose architectures. These architectures fall into two broad categories: data flow machines, which use token passing mechanisms to implement data-driven computations; and reduction machines, which implement data or demand-driven computations by manipulating expressions that are represented as strings or graphs. This section compares STARDUST with a representative example of both types of architecture: AMPS and the Irvine Data Flow Computer. A summary of these and other machines may be found in [Treleaven 82].

Both data flow and reduction machines can be viewed as operating on trees of constant applicative forms (caf-trees, after [Hughes 82]), where each node of the tree represents an operation that is to be applied to its descendants. In data flow machines the caf-tree is represented implicitly by the program and tokens, while in a reduction machine the caf-tree is explicit.

Sample data flow and reduction computations appear in Figures 6-1 and 6-2. To understand the design of the machines described in this chapter it is useful to note the features that address the following four problems:

1. Computer languages contain non-strict operators, such as the conditional, which cannot wait for all arguments to be evaluated before executing. Data flow architectures solve
this problem by including some form of special *distribute* node that sends its input token to only one of its two outputs (see Figure 6-3). Irrelevant expressions are not evaluated because they never receive input tokens, but note how the usual bottom-up flow control is reversed (tokens flow through the conditional operator before its operands), and that this scheme can lead to a proliferation of *distribute* gates if the branches of the conditionals use many inputs. Reduction architectures implement non-strict operators with a demand-driven evaluation scheme: subexpressions are not evaluated until they are requested.

(2) The compiler cannot lay out the entire program beforehand as an explicit caf-tree. This would imply a separate node in the tree for every activation of every function that took place during the entire execution of the program, leading to space requirements and compiler execution time proportional to the program execution time. This problem is solved in reduction architectures by copying function definitions at run time when they are activated, and in data flow architectures by generating tags at run time for tokens that represent the state of the computation in different activations of the function. Function copying and tag allocation correspond to the allocation of activation frames on the stack in conventional architectures. STARDUST, like other reduction architectures, copies functions when they are activated.

(3) Structures can be an important source of overhead, since the semantics of applicative languages imply a full structure copy whenever an element is changed. (Applicative languages cannot include constructs for replacing elements of structures, since that would constitute a side effect.) Three classes of solutions are found:

- Structure avoidance mechanisms, such as "listless" evaluation [Wadler 84] or the use of redundant subcomputation elimination in STARDUST to replace structures with function calls;
- Devices for creating structures "all at once" such as the ForAll construct in VAL [McGraw 82];
- Implementation of replacement using destructive operators when it is clear from program logic or reference counts that the old value of the structure element has not been initialized or can no longer be referenced [Arvind 80a].

(4) Structures can act as concurrency inhibitors. In simple models, \( f(A[1]) \) is represented as in Figure 6-4, and the "access" operator is delayed until \( A \) is completely evaluated. However, the actual data dependencies demand only that the first element of \( A \) be evaluated before calling \( f \), allowing the evaluation of \( f \) to proceed concurrently with the completion of the other elements. The standard solution is to use lenient constructors, which represent structures as collections of suspensions that can be selected and grouped together without being evaluated.
Figure 6-1: A Data Flow Example (Data Driven)

Figure 6-2: A Reduction Example (Demand Driven)

Figure 6-3: The Conditional in a Data Flow Machine:
If \( x < 3 \) Then \( f(x) \) Else \( g(x) \)
6.1.1. AMPS

AMPS (the Utah Applicative Multiprocessing System) is the most similar to STARDUST in the literature. The system is summarized in [Treleaven 82], and described in full in [Keller 78, Keller 79].

The proposed machine consists of a collection of microprocessor-sized processing elements, each with its own memory, that communicate through a tree of store-and-forward switching devices. The processors are considered to be loosely coupled, in that communication is a relatively expensive operation. Considerable effort is given to maintaining locality of reference by dynamically allocating subtasks to nearby processors whenever this is consistent with load balancing. A uniform address space is maintained among the processors.

The internal program representation (a graph structure called FGL, for "Flow-Graph Lisp") is capable of directly representing an applicative dialect of LISP. The program is stored as a collection of blocks: code blocks for function definitions and data blocks for the expressions being reduced. An entry in either kind of block can be either a constant or an operator followed by a list of arguments and the entries waiting for its completion. The operators may be intrinsic LISP operators such as CAR, COND, and PLUS; or pointers to user functions in other code blocks. Evaluation proceeds in the usual reduction machine fashion by replacing expressions with their values until a constant form is reached. A user function is invoked by allocating a new data block (possibly on another processor), copying the information from the code block to the new data block, and setting up dummy operators that pass the arguments to the function and return the results.

The AMPS machine supports lenient CONS and call-by-need parameter passing by setting "demand" bits on subexpressions only when they have been found to be relevant to the computation. When an expression's demand bit is set it enters the "demand list" of the local
processor. When removed from this holding list, it is immediately evaluated if its arguments are ready, and deferred pending completion of the necessary subexpressions if not. Parallel execution is initiated when demand bits from strict operators fan out to subexpressions.

The AMPS machine also supports closures and function-valued functions through the use of special pseudo operators, dynamic and static environment pointers, and an APPLY operator that checks argument counts and sets up fetches for the implicit and explicit arguments.

Load balancing is handled by the binary communications tree. Each communications node maintains an estimate of the combined load, or total demand list length, of the processors beneath it. Invoke requests are normally passed back to the branch from which they came. If the requesting branch is heavily loaded but the other is not, then the request is routed down the other branch. If both are heavily loaded, the request is forwarded up the tree.

6.1.2. The Irvine Data Flow Machine

The data flow implementation for the language Id was originally proposed at the University of California at Irvine, and is now under development at the Massachusetts Institute of Technology. A summary may be found in [Treleaven 82]; more complete descriptions of the work may be found in [Arvind 78, Arvind 80a, Arvind 80b, Arvind 81, Arvind 82]. We have chosen to discuss this machine because it is particularly well developed, well documented, and influential.

The basic scheme is the token passing model (see Figure 6-1). Nodes fire only when all inputs have tokens on them; they consume the tokens and produce one new token on each output arc. Conditionals are implemented with a special switch (see Figure 6-3), which sends the input token down exactly one of its output arcs. Figure 6-5 gives an example of how the switch could be used to form a loop to find the first value of x such that f(x)>0. Actual loops are implemented in a more complex fashion to increase concurrency, as discussed below.

The Id machine maintains a list of active tokens and a graph representation of the program. The program graph is not modified during execution. Tokens consist of a tag and a value, where the tag indicates the token's destination. The tags of new tokens are compared with those of the tokens already in the list; when all tokens destined for a particular node have been collected the tags are used to find the entry in the program graph. The program graph entry provides the operation to be performed on the arguments, and is used to determine the tag of the result.
Figure 6-5: A Simple Data Flow Loop Representation

Tags in the Id machine contain additional information that allows multiple invocations of code blocks, such as simultaneous invocations of a function or trips through a loop, to be evaluated in parallel. This facility can be thought of either as providing virtual copies of the program graph or as the ability to maintain several tokens on an arc of the graph simultaneously (these are reduction machine and data flow concepts respectively).

A tag consists of the tuple \(<u,c,s,i>\), where \(u\) specifies the context, \(c\) specifies the code block, \(s\) specifies the statement within the code block, and \(i\) is an iteration count to differentiate trips through a loop.⁵ Virtual copying is accomplished by special operators that modify the tag of a token without changing its value. For example, a typical Id loop is represented by the structure in Figure 6-6. The \(L\) operator creates an entire new context for the loop invocation, which the \(L^{-1}\) operator pops back to the previous context. The \(D\) operator increments the loop index part of the tag, and \(D^{-1}\) sets it back to 1.

The Id machine consists of a collection of processing elements connected by a routing network (see Figure 6-7). The active token list is divided among the processors in such a way that the processor on which a token is stored may easily be calculated from its tag. Tokens with matching tags are routed to the same processor and paired together with a hashing algorithm. Token transmission, token matching, and operator application proceed in parallel.

⁵The \(u\) field is not actually kept with each token [Arvind 81]
6.2. Contract Nets

The contract net formulation [Smith 78, Smith 80] is a framework for writing programs in a loosely coupled distributed environment. Like STARDUST, it utilizes the available processors by generating hierarchies of computation that spread from processor to processor. Both systems schedule the tasks at run time rather than at compile time. Yet the contract net formulation presents the user with a very different interface: a task distribution protocol rather than a programming language.

Contract Nets (CNET) is a language for communication among the nodes of a distributed problem solver such as a sensor network [DSN 78, DSN 82]. The nodes communicate by initiating tasks on one another and receiving the results asynchronously. Processes are not restricted to being stateless. The system design deals mostly with the language of task assignment, and with special protocols that avoid the expense of the full protocol when its power is not needed.

The key concept behind CNET is that of negotiation. Processes with subtasks to dis-
tribute issue *task announcements*; nodes which find the announcement attractive submit *bids*; and the announcer selects one of the bids for a contract *award*. At this point the bidder becomes the task *contractor* and assumes responsibility for executing the task, possibly spawning other subtasks to form a set of concurrent processes logically structured as a tree. The announcer becomes the task *manager*, with responsibility for monitoring the task, processing the results, and possibly recovering from failure of the bidder.

The user is responsible for formulating the policies for deciding which task announcements and bids a node finds attractive. The information transmitted in a bid or an announcement is conveyed in a general object/attribute/value language with a few standard keywords.

### 6.3. Comparison

#### 6.3.1. Communication

AMPS and Id are designed for customized hardware, while *STARDUST* and CNET are designed to run on a network of conventional processors. The computation granularity is correspondingly higher for *STARDUST* and CNET, and the systems must take extra precautions to minimize communication.
STARDUST is designed for a network environment and strives to minimize communication costs. Each STARDUST invocation maintains its own address space, and copies parts of the expression graph to and from other processors. This leads to the concept of Nodes on different processors having the same value rather than to the AMPS concept of a unified address space shared among processors with "global" pointers that carry both machine number and address. A unified address system would have led to a simpler design with more unified local and remote evaluation requests. It was avoided because it would almost certainly increase message traffic, particularly if the scheme were carried to its logical extreme and constant Nodes resided only on one machine. Messages would have to be sent to retrieve the values of subnodes, and to inform superior Nodes whenever two Nodes were merged.

STARDUST has an "overeager" evaluation strategy in that some expressions are requested for evaluation before it is certain that they will be needed (see Section 4.6.4). This strategy was adopted for STARDUST because it decreases the number of messages that need to be passed to export user functions that use all components of structured arguments. Overeager evaluation increases parallelism at the risk of evaluating unnecessary expressions.

STARDUST does not support function-valued functions. This together with the lexical scoping mechanism made it possible to do away with environments and the problems they cause for hashing and exportation (see Section 3.6).

6.3.2. Partitioning

Only STARDUST has a mechanism for automatically dividing the program into units of intermediate granularity. In CNET the partitioning is defined by the way in which the user groups his code into processes; in AMPS it is defined by the way he groups his code into functions (each user function acts as a unit, though the unit does not include the functions that it in turn calls, as it would in STARDUST); in Id the expression is broken down to basic operations. Partitioning is needed in all but the most tightly coupled machine.

6.3.3. Scheduling

Scheduling in CNET is almost completely under the control of the application programmer. Scheduling in AMPS is done by local load balancing, while Id schedules computations on a processing element that is determined by an arbitrary function applied to the tag.

Distribution in STARDUST is centrally planned to eliminate the costs of polling: the planning processor can estimate loads rather than continuously asking for them. The tree-
structured load balancing scheme used by AMPS might also have been effective even on a local area network since it typically uses only a few point-to-point messages to schedule a request.

6.3.4. Fault Tolerance

**STARDUST** is the only one of these four systems that deals explicitly with fault tolerance. The simple restart mechanism cannot be used by CNET because CNET does not use stateless processes. AMPS and Id cannot use this mechanism because they use custom message routing hardware. They cannot be expected to exhibit the typical network failure mode where one or more machines fail but the communication mechanism remains intact.

6.3.5. Redundant Computations

CNET programs can easily handle explicit caching of redundant computations because the processes are allowed to have state. **STARDUST**, AMPS, and Id programs can cache redundant computations by putting their values into structures, but this is not efficient in an applicative language without a special mechanism to avoid repeated copies of the entire structure. Id addresses this problem by using I-structures, while **STARDUST** uses redundant subcomputation elimination.

6.3.6. Lazy Evaluation

**STARDUST** cannot use the traditional reduction machine approach to lazy evaluation that demands the value of each component as it is needed, since this would lead to an excessive number of demand messages. AMPS uses the traditional approach to lazy evaluation, while CNET does not support it.

The Id machine is data driven rather than demand driven, making it impossible to support the simple lazy evaluation scheme of reduction machines. Id supports streams and I-structures instead; neither is quite as general as lazy evaluation, but both can be implemented particularly efficiently for the cases in which they apply.

I-structures [Arvind 80a] are intended to handle the common case where a structure is used only for caching values. The components of an I-structure may be *defined* or *undefined*; an I-structure can only be updated by providing a value for an undefined component or by updating a component that is in turn an I-structure. This makes it possible to represent an I-structure as an array of values with *presence* bits, and to update the structure by modifying
the array without copying it. References to undefined elements are deferred until the value appears. Special hardware is suggested in [Arvind 80b] to support I-structures.

While the termination semantics and the parallelism gains of I-structures can be imitated with lazy lists, the random access nature of I-structures makes them more efficient and natural to program with for numerical computations. Lists are most effective when a linearly ordered set of values is to be accessed only in order or reverse order of creation.

STARDUST arrays have the lazy semantics and random access properties of I-structures. Arrays may be efficiently created by specifying the components: [1,f(99),g(10)], or by using list operators: MAP(IOTA(1,100,1),f); but attempts to generate them element by element lead to repeated copies.

While the STARDUST redundant subcomputation elimination mechanism (see Section 3.5, and note the relaxation example) does not have the compact and efficient storage representation of I-structures, it provides a different kind of flexibility. The hashing mechanism can generally replace an I-structure by using a function call instead of an array access. The advantage of the hash lookup is that it can be used for a wider variety of structures than simple arrays: Sparse arrays can be used as caches by simply failing to access the unneeded locations, and binary trees can be used as caches by defining the function on sequences of boolean values.
Chapter 7
A Practical System

Time for you and time for me,
And time yet for a hundred indecisions,
And for a hundred visions and revisions,
Before the taking of a toast and tea.

While STARDUST 1.0 can achieve significant speedups for certain applications, it is still a prototype. This chapter investigates improvements that would bring it closer to being a production quality system.

STARDUST 1.0 can only evaluate about 100 expressions per second, which is far too slow to allow applications to be written entirely within the language. Recoding the interpreter with macros, microcode, and more efficient data structures would improve execution speed by an order of magnitude, and automatic program transformations could be used to remove many of the algorithmic inefficiencies common to applicative programs. Compiling programs into machine language or microcode might give execution speeds comparable to PASCAL.

Distribution could be improved by giving the system more responsibility for estimating execution times while giving the programmer more control over distribution decisions. The scheduling algorithm should take advantage of feedback from the satellite processors, use data type lengths for better estimates of communication costs, and provide special treatment for common combinations of list operators.

Imperative constructs could be added to improve both the performance of STARDUST and its range of applicability. They can be added "below" the applicative level, as are the current user intrinsics, or added "above" the applicative level by embedding STARDUST in an imperative system.

Redundant subcomputation elimination could be improved by implementing a data management scheme to decide which Forms to retain over long periods of time for their caching effect. Important equalities (those that are difficult to calculate and likely to be
needed again) should also be sent to other processors to improve redundant subcomputation elimination and speed failure recovery.

7.1. Performance

This section addresses the performance of the parts of the interpreter that evaluate expressions: the "other interpreter" costs of Chapter 5. The current system's heavy reliance on user intrinsics can only be eliminated if the interpretive overhead is greatly reduced. The experimental results of Chapter 5 show that these costs only account for about 15% of the total execution time, but this figure would be greatly increased if the applications did not rely on user intrinsics to perform most of the actual computations.

7.1.1. Current Performance

STARDUST runs under Accent, a message-based operating system for the PERQ computer (see Section 4.2.1 for more details). Performance measurement on the Perq is a clumsy enterprise, since the only available tools are a real time and a job time clock, both accurate to at best 1/60 second. Measurements are complicated by the fact that the Accent microcode caches only a few virtual address translations (the Perq has no virtual memory hardware). The cost of a sequence of memory references increases greatly if the locations are in different pages, and the cost of identical function calls can depend on where the current stack frame is allocated, since this determines which local variables are in the same page. The cost of the following simple empty loop with a million repetitions ranges from 11 to 35 seconds, though actual performance is usually a good deal more stable:

\[
\text{For } j := 1 \text{ To } 1000 \text{ Do } \\
\text{For } k := 1 \text{ To } 1000 \text{ Do};
\]

Figure 7-1 presents timing results for prime number generation on a single processor STARDUST system running under POS, a relatively primative operating system with some useful timing facilities. The single processor configuration has no scheduling, partitioning, or message passing overhead, causing it to be a fair approximation of the part of the interpreter addressed in this section.

Since these figures depend greatly on details of the implementation such as the amount of procedure call overhead caused by the use of structured programming techniques, they do not tell us very much about how time would be spent in an optimized version. Sources of overhead are obscured by the fact that much of it comes from maintaining structures that are used for several purposes. Nevertheless, a few observations can be made:
A. 22% Maintain action queues and the wakeup links between them.
B. 9% Decide if the expression has finished evaluating; spawn actions to evaluate needed arguments.
C. 4% Reduce expressions
D. 17% Create new Forms
E. 9% Attach new Forms to old Nodes, merge Nodes that have become equal.
F. 16% Hashing and equality tests: the direct cost of redundant subcomputation elimination.
G. 8% Dynamic storage allocation
H. 12% "Usefulness" calculations (see Section 7.2.4).

Figure 7-1: Expression Evaluation Overhead

• Categories A, D, and H, which account for 51% of the execution time, are composed of trivial routines for operations such as queue management and data structure initialization. Much of the execution time from the remaining categories, particularly E, F, and G, is also spent in simple routines. This shows that some simple microcoding would have a considerable effect on execution speed.

• The cost of redundant subcomputation elimination is mainly reflected in categories D, E, and F. Category F is devoted exclusively to RSCE, while much of the overhead of categories D and E is due to the added data structure complexity that RSCE causes. Thus RSCE accounts for somewhere around half of the overhead in STARDUST 1.0.

7.1.2. Improvements to the Interpreter

The performance of STARDUST 1.0 could probably be improved by an order of magnitude by using macros, microcode, and more efficient data structures.

7.1.2.1. Actions

The action mechanism introduced in Section 3.7.1 was chosen for the flexibility it gave to the developing system. For performance reasons actions could be largely incorporated into the Nodes that they reference. This would not only save storage and time by allocating and initializing fewer structures, but would also save some of the time currently spent detecting identical actions and performing redundant computations when the detection fails.

In the single processor implementation of Chapter 3 there are at most three actions (one each of Print, Complete, and FComplete) per Node. Furthermore, the "A₁ wakes A₂" relation is true only if A₁.Node is a subnode or a time estimate for A₂.Node. Thus, the presence of the
three action types on a Node could be represented by three bits, and the wakes relation could be represented by four bits connected with the pointers to subnodes and their back-pointers. (Only four bits are needed because Print Actions never wake other actions, and are only awakened by FComplete actions on their own Node.) The Node itself would be in the active queue if any of its actions were active.

In the distributed system there can be several actions with the same Type and Node. This is allowed for two reasons. First, the actions may have different Processor/Name pairs to report the results to. This function could be taken over by a list of Processor/Name pairs attached directly to the Node. Second, the actions may play different roles: they may be normal actions, or the backup and shadow actions needed for overeager evaluation (see Section 4.6.4). The function of backup and shadow actions could be taken over by a more specialized mechanism, allowing actions to be completely eliminated as separate data structures.

7.1.2.2. Short Forms and Short Nodes

Fast implementations of powerful languages typically have special "immediate" representations for common simple data types such as integers, booleans, characters, and low precision floats. Examples are the "Immediate Signed Integers" of Smalltalk-80 [Goldberg 83] and the "fixnums" of Common Lisp [Steele 84]. The pointers to Forms in the STARDUST implementation are currently 32-bit quantities; if Forms were forced to lie on double-word boundaries then the low-order bit of the Form pointer could be used to signal immediate data. The type would be stored in two or three of the remaining bits, and the data in the rest.

Short Forms would leave the program logic virtually unchanged, saving storage allocation and initialization at the expense of frequent tests of the low-order bit of Form pointers. Figures 7-2 and 7-3 contrast the evaluation of the expression "3 + 4" with and without short Forms.

Even more storage allocation could be avoided by providing short Nodes as well. The same trick with the low-order bit could be used to encode them. Figure 7-4 shows the evaluation of "3 + 4" using short Nodes and short Forms; the short Forms remain necessary for the caching mechanism.

Short Nodes make it possible to do simple numerical computations without allocating storage. All that is necessary is for the Form constructing routine to recognize simple operators and perform them immediately when both arguments are ready. (STARDUST 1.0
Figure 7-2: Evaluation of 3 + 4 Using Forms (Circles) and Nodes (Boxes)

Figure 7-3: Evaluation of 3 + 4 Using Forms (Circles), Nodes (Boxes) and Short Forms (Unenclosed)

Figure 7-4: Evaluation of 3 + 4 Using Forms (Circles), Nodes (Boxes), Short Forms, and Short Nodes (Both Unenclosed)

actually does this, saving a certain amount of Form allocation and Action manipulation, but the amount of time saved is fairly small since Forms and Nodes must be created for all of the intermediate values.) Evaluation without dynamic storage allocation would be achieved, for
example, when the definition of a function consisting entirely of simple operators was instan-
tiated with evaluated arguments. Short Nodes are critical to the compilation described in
Section 7.1.4.

7.1.2.3. Macros

Procedure calls in the Accent environment take over 50 microseconds. STARDUST 1.0
suffers for its structured programming style: a typical evaluation cycle calls about 100 proce-
dure for a total overhead of about 5 milliseconds. Much of this cost could be eliminated by
rewriting simple functions as macros. For example, the following routine for dequeueing an
action takes 125 microseconds when called as a procedure, but only 70 microseconds when
written in line:

\[
\begin{align*}
n & := \texttt{at}.\texttt{Next}; \\
p & := \texttt{at}.\texttt{Prev}; \\
\texttt{n}.\texttt{Prev} & := p; \texttt{p}.\texttt{Next} := n; \\
\texttt{at}.\texttt{Prev} & := \texttt{nil}; \texttt{at}.\texttt{Next} := \texttt{nil};
\end{align*}
\]

7.1.2.4. Microcode

Quite a few small routines in STARDUST each account for over 2% of the total execution
time. Microcoding them would not only improve their own execution speed, but would
eliminate the overhead of calling them as procedures. The de queueeing routine shown above
takes about 35 microseconds as a microcode subroutine, which is an improvement by a factor
of 3.7 over the PASCAL subroutine version and a factor of 2.0 over the in-line code.

There are two reasons why the factor of 2.0 is less than what is typically seen for
microcode performance improvement. First, the sequence for calling a microcode subroutine
under Accent takes about 6 microseconds, plus over a microsecond for each argument
passed or value returned. This suggests that the effect of microcoding further subroutines
will be synergistic, since the routines will call each other directly rather than through the byte
code interpreter. Second, memory references account for a significant portion of the over-
head in both versions, since the de queueueing example is a heavy user of memory and memory
accesses are relatively slow operations due to the Perq's lack of hardware support for virtual
address translation.

7.1.2.5. Message Sharing

Like most message passing systems, Accent can send the same amount of information
much faster as a large message than as several small ones. To take advantage of this fact,
outgoing messages could be marked with the maximum allowable delay before sending, ac-
cumulated until one of the delays was exceeded, and sent as a single message. All messages
would be flushed when the system went into its idle loop. In some cases this delay could be
fairly long: for example PleaseExecute messages could be substantially delayed when sent to
busy processors.

Message sharing could be used as an alternative to the full overeager evaluation strategy
of Section 4.6.4, where the arguments to remote function calls are reduced with FComplete
actions without waiting for the function to request them. Recall that overeager evaluation
generates a "demand effect" and a "report effect": structure components are evaluated
without the need for separate demand messages and structure components are reported in a
single message. A version of overeager evaluation with only the demand effect would be
appropriate in a system with message sharing, since message sharing would eliminate the
overhead from sending structure components separately when they were completed at
roughly the same time, yet would not have the concurrency inhibition problems of overeager
evaluation when the time between completion of the components was large.

7.1.3. Program Transformation

Program transformations can be used to remove some of the inefficiencies that often
arise from writing simple elegant programs in applicative languages. They can eliminate
redundant subcomputations, as shown in the example from [Burstable 77] in Section 3.5, and
eliminate intermediate list structures and multiple passes over lists [Wadler 84]. A survey of
program transformation techniques can be found in [Scherlis 80].

Wadler's work has particular potential for improving the performance of STARDUST be-
cause his listless transformer not only removes intermediate lists and multiple passes over the
input, but generates code that has the same termination semantics under lazy and eager
evaluation. Eager evaluation is generally more efficient than lazy evaluation because it is not
necessary to create suspensions and because subcomputations can be sent to other proces-
sors without mechanisms to abort them when they turn out to be irrelevant (see Section 4.6.4).
The transformer is not always applicable, so it would not free STARDUST from all of the
problems and inefficiencies that lazy evaluation causes.

7.1.4. Compilation

This section presents a simple compiler for STARDUST. The compiler's structure and the
type of code it produces are determined mainly by the parallel semantics used to implement
the parallel conditional and overeager evaluation. We show how redundant subcomputation
elimination and partitioning can be integrated into the compiler, and how better code can be
produced for user functions that are known to terminate or use call-by-value parameter passing. This compiler has not been implemented.

The major design decisions of STARDUST 1.0 are retained: computation is performed by reducing a graph representation of the expression, sequencing is controlled by the Action mechanism, and free variables are passed to inner functions and bound to closures as implicit parameters.

7.1.4.1. The Target Machine

The target machine has a stack architecture with registers for temporary storage. The stack is controlled by two pointers: Top Of Stack (TOS) points to the last valid word on the stack, while Current points to the arguments of the current call. The number of arguments to routine R is designated R.NumArgs, while NumArgs alone refers to the number of arguments to the current routine. The symbol "+" indicates pointer dereference. The following stack operations are used by the compiled code:

- **Push(Value)** increments TOS, then sets +TOS to Value.
- **Pop(Value)** sets Value to cTOS, then decrements TOS.
- **Call(Routine)** performs the following steps (see Figure 7-5):
  
  Push(Return Address);
  Push(Current);
  Current := TOS-Routine.NumArgs-1;
  Jump to Routine;

- **Return()** performs the following steps (see Figure 7-6):
  
  Pop(TempValue);
  Pop(Currrent);
  Pop(TempReturn);
  TOS := TOS-NumArgs;
  Push(TempValue);
  Jump to +TempReturn;

*Call* assumes that the arguments to the called function have been pushed on the stack. It saves the Current pointer and the Return address and jumps to the new routine. *Return* assumes that the return value is on the top of the stack. It pops the calling arguments, restores the Current pointer, and leaves the return value on the new stack top. The same call sequence is used whether the compiled code is being called from the Expand routine of the interpreter or from other compiled code.

Some additional special-purpose operations shorten the sample code:
Stack-op, for any binary operator op, is equivalent to:

\[
\text{Pop}(\text{Temp}2); \\
\text{Pop}(\text{Temp}1); \\
\text{Push}(\text{Temp}1 \text{ op } \text{Temp}2);
\]

\text{PushCall(n: Integer; f: FunctionDescriptor)} \text{ is equivalent to:}

\[
\text{Pop n Nodes from the stack}; \\
\text{Create and push a call Node containing f and the n Nodes.}
\]

Nodes are either allocated from free storage or expressed in short form as defined in Section 7.1.2.2. Short Nodes are placed directly on the stack while full Nodes are referred to by a pointer.

In the examples that follow, code generated by the compiler is italicized and surrounded by braces. This code is immediately sent to an output stream, making it unnecessary to \textbf{Return} it or concatenate it with previous compiler output. Thus, if it were necessary for the compiler to increment its own internal variable Temp and then emit object code to decrement the stack pointer, the following two lines would appear in the definition of the compiler:

\[
\text{Temp} := \text{Temp}+1; \\
\{\text{TOS} := \text{TOS} - 1\}
\]
7.1.4.2. Elimination of Translate

The current interpreter (see Section 3.6) uses the procedure Translate to turn a function definition in Parse Tree Representation (PTR) and a set of actual parameters in Form/Node Representation (FNR) into an FNR graph that can be evaluated by the interpreter to produce a final result. This section shows how the PTR of a STARDUST function can be compiled into code that produces output identical to that of Translate; later sections show that much of the processing can be done immediately by the compiled code rather than by creating FNR that must be interpreted.

The compiled function receives the actual parameters represented as Nodes on the stack and produces an output Node on the stack. It can be called from the interpreter or, as shown in later sections, from other compiled code. The compiler has the following structure:

```
Define Compile-Function(f: FunctionDescriptor) =
    Compile(f,Definition);
    {Return();
    Set f.CompiledCode to point to the new object code.
```

Figure 7-6: Stack Discipline for Return
Define Compile(p: PTR) =
If p is a Constant Then
  (Dereference p.Handle);
  (Push the resulting Node on the stack)
ElseIf p is a Call Then
  Compile(p.Args[j]), for j from 1 to p.Function.NumArgs
  (PushCall(p.Function.NumArgs,p.Func));
ElseIf p is a Closure Then
  Compile(p.Args[j]), for j from p.Function.NumExplicit+1
  to p.Function.NumArgs;
  (Pop the argument Nodes from the stack);
  (Push a closure Node made from p.Function and the arguments);
ElseIf p is a Variable Then
  (Push(*{Current + p.Index-1}));

This simple compiler avoids the interpretive overhead of examining the PTR and performing recursive calls to Translate. Calls to the routines that create FNR (PushCall, in particular) can be expanded in line to save execution time at the expense of code space. The following is a sample of the output of this compiler:

Uncompiled:
  Let f(x:Integer; y:Integer) = (x+1)*y;

Compiled:
  Push(+Current);
  Push(1);
  PushCall(2,+);
  Push(*{Current+1});
  PushCall(2,*);
  Return();

7.1.4.3. Immediate Evaluation

The compiled code can execute most intrinsic functions immediately. We define the set of fast simple operators to be {*, DIV, +, -, =, =>, >, <, AND, OR, Car, Access, Choice}, the set of slow simple operators to be {Cdr, Cons, Concat, ListToBlock, BlockToList, Iota, Replace, Sel2, Replace, UserIntrinsic}, and the non-simple operators to be the rest: user functions and the operators {If, Sif, Map, Reduce, Pair, Cross, Select}. Simple operators may be evaluated in immediate mode by adding the following lines to the compiler before the block starting with "Else If p is a call Then ...":
Else If p is a Call And p.Function is simple Then Begin
    Compile(p.Args[j]), for j from 1 to p.Function.NumArgs
    {If p.Function is applicable [see below] Then}
    {Perform the simple function}
    {Else}
End;

The phrase "If p.Function is applicable" refers back to the conditions in Figure 3-12. Since p.Function is known at compile time, the test can be compiled. For example, strict binary operators are only applicable if both operands have been reduced to constants, and the following code can be generated.

{If TOS is a constant And (TOS-1) is a constant Then}

Fast simple operators should always be applied immediately. There is no need to suspend their evaluation, since it would take longer to delay them than to actually perform them. Since the slow simple operators either allocate storage or call user code, it is not clear that they should always be evaluated immediately. We will, however, treat them that way on the understanding that immediate evaluation might be made optional in a more fully developed system.

Note that the immediate operations do nothing to "force" the evaluation of the arguments. Since the arguments are general STARDUST expressions, such forcing could generate infinite loops in the compiled code. This is not allowed since overeager evaluation and the parallel conditional can initiate evaluation of subexpressions that are not known to be relevant to the computation, and recovery from this can only happen when the compiled code returns to the interpreter.

The performance gain from immediate evaluation depends heavily on the use of "short Nodes," since without them storage must be allocated whenever an expression is evaluated.

7.1.4.4. Conditionals

The sequential conditional (SIF) is compiled as follows:

Else If p is a call And p.Function is SIF Then Begin
    Compile(p.Args[1])
    {If TOS is a constant Then Begin}
    {If TOS = True Then}
    Compile(p.Args[2])
    {Else}
    Compile(p.Args[3])
    {End}
7.1.4.5. Immediate User Functions

Compilation of calls to user functions is heavily influenced by the parallel evaluation semantics needed to implement the parallel conditional and overeager evaluation. In the uncompiled system, the interpreter guarantees with its "fair" scheduling algorithm that any terminating computation will run to completion despite the presence of non-terminating irrelevant computations. Two possible options could maintain this condition for compiled code: the compiled code could be periodically interrupted by the interpreter, which would preserve the state of the compiled code to continue it later; or the compiled code could be generated in such a way that it eventually either produces a result or saves its state and returns to the interpreter. We have chosen the latter option due to its machine independence (the compiled code preserves its state by returning new FNR rather than having its state preserved by an external agent) and its lack of synchronization problems (the compiled code is never interrupted in critical regions, such as calls to the hash table insertion routine).

It should be clear that code generated by the simple compiler presented above eventually returns to the interpreter, since the compiled code contains no loops or recursion. This section shows how the interpretive burden of action management and FNR generation can be eased for calls to user functions by replacing PushCall with Call:

Deferred User call:
PushCall(f.NumArgs, f)

Immediate User Call:
Call(fCompiledCode)

Non-terminating computations in STARDUST arise only from recursion. For this discussion a recursive call will be any call that a function makes to itself or to any other function that might, through a sequence of calls, invoke the caller. Non-recursive calls can always be compiled in immediate mode since this generates no loops in the compiled code.

Recursive calls may always execute in immediate mode to a bounded depth. This can be implemented by unraveling the function to the given depth or by adding a parameter that represents the depth at run time. The following versions of the factorial function illustrate these possibilities given a bounding depth of two.
Original version:
Def fac(x:Integer) =
  Sif x<2 Then 1 Else x*fac(x-1);

Unraveled version:
Def fac(x:Integer) =
  Sif x<2 Then 1 Else
    x*(Sif (x-1)<2 Then 1 Else (x-1)*fac((x-1)-1));

Parameterized version:
Def fac(x:Integer) = fac2(x,2);

Def fac2(x:Integer;n:Integer) =
  Sif x<2 Then 1 Else x*(
    Sif n>1 Then fac2(x-1,n-1) { immediate call to fac2 } 
  Else fac2(x-1,2)); { generate FNR for call to fac2 }

Unbounded recursion is allowed when the computation is known to be relevant to the final result. Unfortunately this fact can only be determined at run time by passing a flag from the top level call through functions that have strict semantics.

Unbounded recursion is also allowed for functions that are known to terminate either from analysis or on advice from the user. The recursive calls must still be known to be relevant, since recursive functions have infinite expansions even when they have finite execution times. Relevance can frequently be determined at compile time, leading to compiled code that avoids FNR creation and Action manipulation yet contains no run-time checks for relevance.

This version of unbounded recursion is the only one used throughout the remainder of the chapter, though it is compatible with the others. As an example of its effectiveness, the factorial function compiles to the following code which, though long-winded, evaluates in immediate mode when called with a ready argument. Note how the recursive call is immediate when it is known to be relevant but deferred when it is not.

Uncompiled:
Let fac(n:Integer) =
  SIf n<2 Then 1 Else n*fac(n-1);

Compiled:
Push(tCurrent);
Push(2);
If tTOS and +(TOS+1) are constants Then
  Stack-LessThan; | Compare tTOS and +(TOS-1), push result
Else
  PushCall(2,<);
If \( \top \text{TOS} \) is a constant Then Begin

\[ \text{Pop(\text{Temp});} \]

If Temp Then  \| Recursive call is not relevant

\[ \text{Push(1)} \]

Else Begin  \| Recursive call is relevant

\[ \text{Push(\top \text{Current});} \]
\[ \text{Push(\top \text{Current});} \]
\[ \text{Push(1);} \]

If \( \top \text{TOS} \) and \( \top (\text{TOS}+1) \) are constants Then

\[ \text{Stack-Subtract} \]

Else

\[ \text{PushCall(2,-);} \]

| Recursion is OK because relevant

\[ \text{Call(fac compilingcode);} \]

If \( \top \text{TOS} \) and \( \top (\text{TOS}-1) \) are constants Then

\[ \text{Stack-Multiply} \]

Else

\[ \text{PushCall(2,*);} \]

End

End

Else Begin  \| Relevance of recursive call is unknown

\[ \text{Push(1);} \]
\[ \text{Push(\top \text{Current});} \]
\[ \text{Push(\top \text{Current});} \]
\[ \text{Push(1);} \]

If \( \top \text{TOS} \) and \( \top (\text{TOS}+1) \) are constants Then

\[ \text{Stack-Subtract} \]

Else

\[ \text{PushCall(2,-);} \]

\[ \text{PushCall(1,fac);} \]  \| Recursion is not OK

If \( \top \text{TOS} \) and \( \top (\text{TOS}-1) \) are constants Then

\[ \text{Stack-Multiply} \]

Else

\[ \text{PushCall(2,*);} \]
\[ \text{PushCall(3,Sif);} \]

End;

User functions resemble slow simple functions in that, while the rules ensure that termination semantics are preserved, further analysis or advice must be used to determine whether performance is improved.

The compiler procedure Compile-User-Call will encapsulate the decision whether to use immediate mode for user functions.
Define Compile-User-Call(f: FunctionDescriptor) =
If the caller always terminates And
  this call is relevant Then
  {Call(f.CompiledCode)}
Else
  {PushCall(f.NumArgs,f)}

The following code is added to the compiler to handle calls to user functions from compiled code.

If p is a call And p.Function is a user function Then Begin
  Compile p.Args[j], for j from 1 to p.Function.NumArgs;
  Compile-User-Call(p.Function)
End

7.1.4.6. An Example of Call-by-Name Parameter Passing

To illustrate the operation of call-by-name parameter passing, consider the following example, where Ω is a non-terminating user function. The compiler is designed so that Ω.CompliedCode will eventually return FNR rather than looping forever within the compiled code. If x<5, the FNR is subsequently ignored by g.CompliedCode; if not, the interpreter will be invoked on the FNR returned by Ω.CompliedCode and evaluation will properly fail to terminate.

Uncompiled:
Let f(x:Integer) = g(x,Ω(x))
Let g(y,z:Integer) = Sif y<5 Then y Else z;

Compiled code for f:
Push(±Current);
Push(±Current);
Call(Ω.CompliedCode);
Call(g.CompliedCode);

Compiled code for g:
Push(±Current);
Push(5);
If ±TOS and ±(TOS-1) are constants Then
  Stack-LessThan; | Compare ±TOS and ±(TOS-1), push result
Else
  PushCall(2,<);
If \( \uparrow \text{TOS} \) is a constant Then Begin
   Pop(Temp);
   If Temp Then
      Push(\( \uparrow \text{Current} \))
   Else
      Push(\( \uparrow \text{Current} + 1 \));
   End
Else
   Push(\( \uparrow \text{Current} \));
   Push(\( \uparrow \text{Current} + 1 \));
   PushCall(3, Sif);

7.1.4.7. Optimizations

Some simple flow analysis of the factorial program reveals that several of the completion checks on the argument are redundant. Completion checks on constants and the results of arithmetic operations on constants can be removed since they always return True, while completion checks on Nodes generated by PushCall can be removed since they always return False. Furthermore, flow analysis can show that fac will produce a completed result when given a completed argument. The optimized code is as follows:

Factorial without redundant tests:
   Push(\( \uparrow \text{Current} \));
   If \( \uparrow \text{TOS} \) is a constant Then Begin
      Push(2);
      Stack-LessThan;  \|\ Compare \( \uparrow \text{TOS} \) and \( \uparrow (\text{TOS-1}) \), push result
      Pop(Temp);
      If Temp Then
         Push(1)
      Else Begin
         Push(\( \uparrow \text{Current} \));
         Push(\( \uparrow \text{Current} \));
         Push(1);
         Stack-Subtract;
         Call(fac.CompliedCode);
         Stack-Multiply
      End
   End
Else Begin
  Push(1);
  PushCall(2,<);
  Push(1);
  Push(*Current);
  Push(*Current);
  Push(1);
  PushCall(2,-);
  PushCall(1,fac);
  PushCall(2,*);
  PushCall(3,Sif);
End;

7.1.4.8. Call-by-Value Parameters

When the compiled code for a user function is invoked before the arguments have been evaluated, it is usually limited to producing FNR in the manner of the old interpreter. Thus, call-by-value arguments are important for good performance.

Call-by-value arguments could be included as a new parameter passing option. It is also sometimes possible to automatically detect the fact that call-by-value produces the same semantics as the standard call-by-name: for example, fac cannot evaluate without its argument, since the conditional needs the value of (x<2), and this in turn needs the value of x. The interpreter can be instructed to treat functions with call-by-value parameters the same way as it treats intrinsic strict operators such as the arithmetics:

- They will only be called in immediate mode if the call-by-value parameters are ready. The test for this could be either in the calling routine or in the routine being called.
- The interpreter will attach Complete Actions to the call-by-value parameters, reducing them before calling the compiled code.

Given this optimization, the compiled code would look much like the output from a PASCAL compiler, since it is no longer necessary to even check whether the arguments are ready.

Factorial with call-by-value optimization:
  Push(*Current);
  Push(2);
  Stack-LessThan;   | Compare +TOS and +(TOS-1), push result
  Pop(Temp);
If Temp Then
    Push(1)
Else Begin
    Push(*Current);
    Push(*Current);
    Push(1);
    Stack-Subtract
    Call(fac.CompiledCode);
    Stack-Multiply
End

7.1.4.9. List Operators

List operators can be compiled in line. For example, the MAP operator, which applies a function to each element of a list, can be compiled with the following code. Note how the implicit parameters are passed.

Else If p is a call And p.Function is MAP Then Begin
    Compile(p.Args[1])  | The list
    {If *TOS is an explicit list Then Begin}
    {For j := 1 To (*TOS).Length Do Begin}
        {Push((*TOS).Components[j])};
        Compile((p.Args[2]).Args[k]),  | Implicit parameters
            for k from 2 to (p.Args[2]).NumArgs;
        Compile-User-Call((p.Args[2]).Function);
    {End}
    {Pop the components, create and push a structure Node}
    {End}
    {Else Begin}
    Compile(p.Args[2]);  | Creates a closure
    {PushCall(2,Map)};
    {End}
End

7.1.4.10. Structures

Structures are formed by a special pseudo function called Construct. Construct is represented as a normal function call in PTR, but the interpreter creates a structure Node from the arguments rather than a call Node. Construct compiles into the following code:

If p is a call and p.Function = CONSTRUCT Then Begin
    Compile(p.Args[j]), for j from 1 to p.Function.NumArgs
    {Pop the argument Nodes from the stack};
    {Push a structures Node made from the arguments};
End
7.1.4.11. In-Line Expansion

Since the semantics of STARDUST are transparent with respect to function expansion, user functions can be expanded in line whenever execution time is more important than code space. While this usually only saves the cost of a function call on the stack machine, it can save FNR generation when the function being called does not always use its arguments. If \( g \) is expanded in line in the example of call-by-need parameter passing in Section 7.1.4.6, the following code would be generated, which does not create FNR for the call to \( \Omega \) when \( x \leq 5 \).

Uncompiled:

\[
\text{Let } \text{Expanded-}f(x: \text{Integer}) = \\
\quad \text{Sif } x \leq 5 \text{ Then } x \text{ Else } \Omega(x);
\]

Compiled:

\[
\begin{align*}
\text{Push}(\ast \text{Current}) ; \\
\text{Push}(5) ; \\
\text{If } \ast \text{TOS and } \ast (\text{TOS}-1) \text{ are constants Then} \\
\quad \text{Stack-LessThan; } | \text{Compare } \ast \text{TOS and } \ast (\text{TOS}-1), \text{ push result} \\
\text{Else} \\
\quad \text{PushCall}(2, <) ; \\
\text{If } \ast \text{TOS is a constant Then Begin} \\
\quad \text{Pop}(\text{Temp}) ; \\
\quad \text{If Temp Then} \\
\quad \quad \text{Push}(\ast \text{Current}) \\
\quad \text{Else} \\
\quad \quad \text{Push}(\ast \text{Current}) ; \\
\quad \quad \text{Push}(\Omega.\text{CompiledCode}) ; \\
\quad \text{End} \\
\text{Else} \\
\quad \text{Push}(\ast \text{Current}) ; \\
\quad \text{Push}(\ast \text{Current}) ; \\
\quad \text{PushCall}(1, \Omega) ; \\
\quad \text{PushCall}(3, \text{Sif}) ;
\end{align*}
\]

In-line expansion of the function calls in list operators is particularly important, since these functions will typically be called frequently from within a loop, and the syntax of the language does not allow them to be expanded in line "by hand."

7.1.4.12. RSCE in Compiled Code

Calls to user functions in the compiled code may be augmented to perform redundant subcomputation elimination. While the following modification of Compile-User-Call shows the RSCE computations being performed in line, they could also be handled by subroutine calls if the space saved was more important than the time lost. The lines marked as optional are invoked when the expression is not found in the hash table. They create a Node with a Form
for the expression and a Form for the result, so that future attempts to evaluate this expression will find it.

Define Compile-User-Call-With-RSCE(f: FunctionDescriptor) =
{Calculate hash value, search};
{If found Then}
{Pop arguments, push result}
{Else Begin}
   If the caller always terminates And this call is relevant Then Begin
      {Temp := Call Node created from f, args} [optional]
      {Enter Temp in the hash table} [optional]
      {Call(f,CompiledCode)}
      {MergeNodes((TOS),Temp)} [optional]
   End Else
   {PushCall(f.NumArgs,f)}
End

7.1.4.13. Partitioning Compiled Code

Time estimates are compiled as ordinary STARDUST expressions. The following compiler code is used to create a preamble to functions with time estimates:

Compile(f.Time);
{Pop(Temp)};
{If Temp is not a constant Then}
   {Attach an FComplete Action to Temp};
   {PushCall(f.NumArgs,f)};
   {Inform the interpreter that the current Node should wait for the time Node to complete}
{Else Begin ... }

The code should distinguish between calls from the Expand routine of the interpreter and calls from other compiled code. When called from the Expand routine it should simply set a flag indicating that no changes were made to the Node, rather than constructing a new one identical to the old one.

The time estimate, when it has finally been determined, can be compared with the limits set for local evaluation (small time estimate), remote evaluation (medium time estimate), and further expansion (large time estimate). The compiled code for these three cases will have the following form:

- **Local Evaluation** causes f.CompiledCode to proceed as usual. Performance might be improved by setting a flag to inhibit the evaluation of time estimates on subexpressions.
• Further Expansion also causes f.CompiledCode to proceed as usual. Time estimates on subexpressions will be evaluated, possibly causing subexpressions to be evaluated remotely (exported).

• Remote Evaluation first determines whether the routine was called from Expand or from other compiled code. If called from Expand, a Node and Action already exist to represent the computation being attempted, and it is only necessary to return a flag or special value to Expand, which passes it up through Evaluate to InvokeAction, which schedules the Action and calls Export to send it to another processor. If the routine was called from other compiled code, it is necessary to create a Node for it with PushCall. It could be exported immediately, or it could be exported as soon as the interpreter got to the new Node through the regular Action mechanism.

The partitioning code can be included directly into the compiled code for list operators. As in the interpretive version (see Section 4.4.2), constant time estimates are treated as an important special case. Once again, consider the MAP operator. For functions with constant time estimates the compiler generates code to divide the MAP into shorter calls or evaluate it immediately, depending on the length of the list. For functions with non-constant time estimates, the code must iterate through the list, attempting to evaluate the time estimates. When this attempt succeeds (returns a constant), the value can be used to decide whether to group the list element with the previous ones or create another group. When the attempt fails (returns FNR that is not a constant), a Node is generated that applies the function to the list element, and the Node will later be partitioned and executed like an ordinary function call.

7.2. Distribution Improvements

7.2.1. Evaluation and Distribution Control

STARDUST contains a number of features, including redundant subcomputation elimination, overeager evaluation, lazy evaluation, and partitioning, that sometimes do more harm than good. STARDUST 1.0 has built into it some simple decisions about when they should be used and when they should be avoided. While it is important to have these defaults, a practical system must give the programmer the ability to override them.

Much of the information can be given to the system in the form of annotations to user functions. STARDUST 1.0 provides a limited means of attaching such pragmas:

Let f(x:Integer) = [Atomic] Begin ...

A more general means of specifying pragmas would copy the syntax currently used for time estimates:
Let \( f(x:\text{Integer}) = \text{Begin} \)
\[
\begin{align*}
\text{Time} & \ 50^x^x; \\
\text{Atomic} & \ x<25;
\end{align*}
\]

If the identity of the caller is relevant to the decision then the programmer can either write several versions of the function that produce the same results but have different pragmas, or the programmer can introduce function parameters that affect the pragma expressions without affecting the result.

Some of the most useful pragmas that can be expressed in this form are:

- **Atomic** <boolean expression>. If the expression evaluates to True, then the function will never be partitioned.
- **Explode** <boolean expression>. The function will always be partitioned.
- **RSCE** <boolean expression>. Redundant subcomputation elimination will be performed for call Nodes using this function.
- **RemoteRSCE** <boolean expression>. Redundant subcomputations will be sought on other machines.
- **StashRSCE** <boolean expression>. The result will be stored on at least one other machine so that it will be available for RemoteRSCE even if this processor fails.
- **PreserveCache** <integer expression>. If the value of the function is preserved for RSCE then it may be eliminated after this amount of time has passed.
- **Priority** <integer expression>. This is the relative priority at which this function should evaluate.
- **AlwaysLocal** <boolean expression>. This function is always scheduled on the controller.
- **AlwaysTogether** <boolean expression>. Calls to this function are always scheduled on the same processor. This would typically be done to improve RSCE.

Other pragmas apply to the function's arguments rather than to the function as a whole. A possible syntax, where the expressions can depend on previous parameters, is:

\[
\text{Let } f(x:\text{Integer}[P_1 = \text{expression}; P_2 = \text{expression}; \ldots ]) \ldots
\]

Where "\( P_j = \text{True} \)" could be abbreviated as

\[
\text{Let } f(P_j \ x:\text{Integer}) = \ldots
\]

These pragmas could be applied to system intrinsics (they would be particularly relevant to the non-strict functions \( \text{If} \) and \( \text{Sif} \)) by encapsulating the system intrinsics inside of user functions. Useful pragmas that take this form are
• Usefulness <integer expression>. This expression controls priority inheritance, which is a measure of the probability that the argument will be needed (see Section 7.2.4).

• Eager <boolean expression>. The argument will be completed before the function is invoked. An expression is completed if it is either a constant or a structure with possibly uncompleted components.

• Greedy <boolean expression>. The argument will be fully completed before the function is invoked. An expression is fully completed if it is either a constant or a structure with fully completed components.

• Overeager <boolean expression>. Call-by-need semantics will be preserved but the system will attempt greedy evaluation and abort it if necessary. This option is currently the default for functions being executed remotely.

Pragmas for controlling scheduling could follow [Schwans 82]. Schwans presents a system in which a desired distance may be specified between pairs of objects and between objects and resources, and the scheduler attempts to satisfy these distance constraints while balancing the processing load. The system was designed for a heterogeneous communication network such as Cm*; on the typical local area network the only possible distances would be same-processor and different-processor.

In STARDUST the distances between classes of calls would be specified statically, while membership in classes could either be static (such as "all calls to g are in class 3") or dynamic:

```
Let g(x:Integer) = Begin
   Time 33*x;
   Class If x<10 Then 1 Else 2;
```

Finally, there are pragmas that apply to the system as a whole rather than to any particular function:

• Timeouts on message sharing

• Timeouts for overeager evaluation

• Defaults for pragmas that are not overridden

We anticipate that in a practical situation the programmer will often spend as much tuning pragmas as producing correct results. Performance measurement capabilities will be crucial in such situations.
7.2.2. Scheduling

There is room for improvement in both the model of the execution environment and the algorithm used to produce schedules that give good performance when that model is valid. These improvements were discussed in Section 4.5. Another class of scheduling improvements could take advantage of the fact that some tasks such as region monitoring execute functions repeatedly with slowly varying parameters, resulting in a series of computations with similar structure. For example, the same branches of conditionals may be taken, the same timing information may hold, and call-by-need arguments that were relevant may be relevant again. STARDUST may be able to take advantage of this information to perform its planning task more quickly and reliably.

Certain applications of this principle are virtually cost-free. For example, if CHOICE resolution is done with the A* algorithm [Nilsson 71], then the best solution from the previous execution of the function may be used with the current arguments to produce an upper bound on the optimization function. If this is close to the actual maximum of the function, then substantial savings in search time can be realized.

Other applications can incur penalties if they turn out to involve wrong guesses. For example, if the system predicts that the test in a conditional will evaluate to true, then it can go ahead and evaluate the test and the true branch in parallel. If the test turns out to be false, then the true computation is wasted.

Since the use of this principle may be expensive or misleading, it is important to decide when it should be applied. Possible solutions are

- Apply it to functions which the user has specially designated.
- Apply it to expensive functions.
- Apply it to all functions for a time, and determine empirically which functions benefit from it.

7.2.3. Automatic Time Estimates

In STARDUST 1.0 the user is responsible for providing functions for the expected processing costs of each function. It would be desirable if the system could estimate these functions automatically.

The problem is difficult since the time estimation functions can be non-linear and depend on summaries of complex structures such as list length and tree depth. A good compromise would be to let the user provide a function with unknown constants, and let the system determine the constants empirically. In particular, if the function had the form
then the constants could be estimated by linear regression.

In any case, the system could compare actual processor usage to the estimates. High variability and strong trends in the residue (estimation error) could be reported to the programmer or used to modify the estimation function.

7.2.4. Priorities

Computations initiated by overeager evaluation (evaluation of call-by-need arguments and structure components before they have been shown to be needed) are currently evaluated at the same priority as other expressions. This gives the system only two options: it can initiate overeager evaluation and have the arguments evaluate at equal priority with other expressions in the system, or it can dispense with overeager evaluation and in effect run the arguments with no priority. Thus the system must make an all-or-nothing choice between evaluating the arguments immediately at the risk of wasting resources on unneeded computations, and waiting for the values to be demanded at the risk of introducing serialization and excessive message passing.

Despite the fact that any such scheme is certain to depend on either shaky assumptions or voluminous advice from the programmer, we decided to experiment with a simple mechanism to estimate the probability that a given expression would actually turn out to be needed. The routines to calculate this probability estimate have been implemented, but we have not had the opportunity to determine the gains from translating it into a priority.

The estimate is called "usefulness", which is an integer equal to Round(-ln(1-p) x W), where p is the estimate of the probability that the Node will be actually needed, and W causes the proper number of significant figures to be preserved (it is currently set to 15.0). STARDUST never converts usefulness back to a probability, but preserves probability semantics by carefully controlling the way in which usefulness measures are combined. The following discussion shows that this measure has the advantages of fairly easy weighting and simple combination, combined with incremental and reversible inclusion of new factors.

The model followed by usefulness calculations is that the probability of a Node's being needed is equal to the probability of the enclosing function call being needed multiplied by a constant depending on the current Node's relation to the enclosing function. For example, we have set this weighting factor to 98% for the first argument of conditionals, and to 50% for both of the other arguments. Weighting (multiplying the probability by a constant) is not a convenient operation when the usefulness representation is used. Luckily there are only a
small number of weights to be considered, and the usefulness values can be restricted by
deciding that values above a certain limit all denote certainty. Thus the multiplication can be
accomplished by a table lookup rather than by transforming the usefulness values to
probabilities and back.

If a Node is used by several expressions as a result of redundant common subexpression
elimination, we assume that its probability of being needed by each of these ancestors is
independent, with the result that $p$, its probability of being needed, is equal to $1 - \prod (1 - p_i)$,
where $p_i$ is the probability that it will be needed by the $i^{th}$ ancestor. This calculation is
convenient in the usefulness representation: it is simply the sum of the usefulness contribu-
tions from the parents.

Sums are not only cheap to calculate, they are incremental and reversible. As new
ancestors are added to a Node their usefulness contributions may simply be added on; when
they are removed their usefulness may be subtracted. This can be important if a Node is
contained in many expressions since, if the usefulness had to be completely recalculated
every time the Node was made part of another expression, the time to calculate it would be
proportional to the square of the number of containing expressions. Unfortunately some
nonlinear behavior remains, since changes to the usefulness of any Node propagate down the
expression structure.

Certain actions such as print actions and actions to complete a Node at the request of
another processor contribute a large usefulness component to the Node. This contribution is
simply summed in with the rest of them.

The primary potential for these usefulness figures is as priorities on the action queue.
We cannot interpret these priorities as absolute, withholding computational resources com-
pletely from Nodes when Nodes of higher priority exit, because there is no guarantee that a
function's arguments will receive a lower priority than the function itself.

7.3. Execution Environment

7.3.1. Imperative Constructs

Future versions of STARDUST may be extended beyond pure expression evaluators to
include some notion of state. There are five main approaches; the first is part of
STARDUST 1.0, the next three are candidates for implementation, and the fifth is included for
completeness.
1. Functions written in other languages ("user intrinsics") may be local applicative: they may use state internally as long as the value of the function depends only on the arguments. Such a user intrinsic implements a pure applicative function using imperative constructs for simplicity or efficiency. Note that this strategy differs from the others in that it allows imperative constructs to be used by the applicative ones rather than the other way around.

2. Programs could be formulated as functions between an input list and an output list. Elements of the output list would be generated as soon as enough of the input list had been received to determine them. Such a program is technically a pure function between two lazy lists (also called "streams" [Landin 65]), but can also be viewed as an interactive program that alternately consumes input and produces output.

3. STARDUST could be used as a set of function calls from a traditional imperative language. The imperative language would be able to assign goals to STARDUST, check for completion, and read back the results.

4. Assignment could be added to the STARDUST language. Parallel evaluation mode would be initiated only for purely applicative subexpressions, or for sets of expressions that the programmer declares to be safe to execute in parallel despite their use of imperative constructs. The language contains enough high-level constructs that a careful programmer will be able to write most of his program without using assignment. ML [Gordon 79] is an example of such an "almost-applicative" language. The authors have written a large theorem proving program in ML with few excursions from the applicative subset.

5. Directly realized state variables could be used as a model. In this formulation a single value called the state is repeatedly computed from its previous value and new inputs. This has been proposed as a model for a text editor [Feldman 82].

7.3.2. Data Management

Redundant subcomputation elimination in STARDUST 1.0 is strictly a single processor feature: it does not find identical expressions that are created on different processors. A matching step to relieve this problem could be added using the present features for hashing and transmitting Nodes.

Before evaluating any expression, a processor could first determine whether any other processor had already evaluated it. If so, the result would be returned immediately. If the other processor was currently working on it, it would send back a message to that effect and return the answer when it finished.

This technique must be applied sparingly because of the communication costs. Since successful matches only serve to eliminate redundant computation, the decision of when to use it can be made solely on the basis of the expected improvement in performance.
This in turn has three components: the cost of looking for a match, the probability of finding one, and the reward for doing so. The cost could be assumed constant, though it should perhaps depend on the complexity of the expression. The reward could be estimated from the execution time estimate of the expression being matched. Unfortunately there is no good way to estimate the probability of finding a match, since in general this depends on the details of the algorithm and the history of the computation. These probabilities could be estimated from experience or taken as advice from the programmer.

**STARDUST 1.0** has several default decisions about the hash table that should be under control of the programmer. These decisions and their implications are presented below, and constructs for controlling the decisions appear in Section 7.2.1. Keller and Sleep [Keller 81] have discussed these problems in the context of applicatively implemented memo functions.

- When the system preserves a Form and its value it saves it only in the local hash table. Consequences are:
  - Saving a Form is relatively fast, in that it involves no message passing. If messages of this class were gathered together and sent as a group the message passing cost could be spread over several Forms.
  - Other processors must query this machine in order to use the results of the computation.
  - The results of the computation are lost if this machine fails before another machine queries it. The computation can, of course, be performed again elsewhere.

- When it saves a Form in the hash table it saves it forever. Consequences are:
  - Once a Form has been saved it can always be found again.
  - The hash table expands to fill virtual memory backup.

- When it looks for a Form it looks for it only in the local hash table. Consequences are:
  - Lookups are relatively fast: no messages need to be sent.
  - Results produced on other processors cannot be used unless they are sent to this machine.

The desirability of saving results on other processors, looking for results on other processors, and keeping results over a long period of time depends on the amount of time it takes to calculate them, which can either be taken from the time estimates or remembered as part of the history of the computation. This is only an approximation for the amount of time it will take to recalculate the results, since the state of the system, and in particular the contents of the hash table, may change. The probability of keeping results over a long period of time should also depend on frequency and recency of use.
On most hardware implementations it will be expensive to broadcast Forms to all available processors and poll all other processors for a Form. It would be much more efficient to send all Forms of a particular type to a designated processor, and look for them only there. Some solutions are:

- Designate a particular processor as the recipient of all results that are important enough to be saved. This solution is simple but may create a bottleneck.

- Allow the programmer to divide call Forms into classes, as suggested in Section 7.2.1. All Forms of the same class would be sent to the same machine.

- Use the hash value of the Form to decide which processor to send it to. This is currently not feasible because call Forms receive different hash values on different machines, since they employ the address of the argument Nodes in the hash function. It would be possible to calculate a "global hash value" by using the full expression that the Form represents rather than just the top level, as is currently the case. If a memo system such as one of those proposed in [Keller 81] were used, where common subexpression elimination takes place only after the arguments have been evaluated, this problem would also be eliminated.

7.3.3. Local Data

The STARDUST scheduler assumes that any function with its arguments can be evaluated on any processor at any time. Since one of the important projected uses of STARDUST is in real-time signal processing, this assumption will have to be relaxed.

Consider the function that receives input from the signal gathering hardware. The signal data from each sensor can either be implemented (as in the DSN example of Section 5.1) as a function of time or as a lazy list (stream) of values. In either case the signal gathering hardware will be sending data to a single computer at intervals.

This fact has implications for both scheduling and fault tolerance.

The scheduler must know that this function can be executed only on a particular processor. Other processors must realize this as well, since the processor that does the scheduling will not see it if it is hidden in the definition of some other function. Furthermore, the scheduler must model the delay before the function returns in order to maintain an accurate model of the time varying load on the other processors. This is only possible if a default is assumed (such as that new data will not be ready until the system has finished processing the results of the last set), or if the delay can be expressed as a new form of function annotation.

Fault tolerance is affected by the existence of data that is only available from one machine. As soon as it is received from the hardware, steps must be taken to place it on stable storage or duplicate it on several machines.
Chapter 8
Future Work

(And I Tiresias have foresuffered all
Enacted on this same divan or bed;
I who have sat by Thebes below the wall
And walked among the lowest of the dead.)

While the previous chapter described improvements that would make the system faster without altering its character, the extensions proposed here involve some substantial changes in the way that functions in the language should be viewed. The two fundamental shifts are:

• Function invocations behave more like goals proposed to an AI programming language. The system has comparative freedom to decide how to attain these goals.

• Function invocations behave more like processes. They receive their arguments as messages and send the result as a message.

8.1. Accuracy

In many instances the user of an adaptable system will wish to trade off processing time and some measure of accuracy or certainty. For example, a region monitoring system which must cut down on its processor usage might employ a less accurate algorithm for determining the location of the objects or a less reliable method for determining their identity.

Functions may be augmented with an estimate of their accuracy defined in terms of the values and accuracies of the arguments. When processing is complete they may report revised versions of the accuracy based on observations made in the course of the calculation.

"Accuracy" is really a substitute for a more general term like "goodness" since it refers to any user-defined quantity that is desirable and generally increases with increased resource expenditure or "better" arguments. The applications programmer may define arbitrary measures of accuracy with a construct in the header of his function definition such as
Goals may be given a desired accuracy, perhaps under several different measures of accuracy. For example, if the goal is a list of the positions of all vehicles in a certain region, then three accuracy measures might be the 95% confidence interval on the positions, the probability that no vehicles were left off the list, and the probability that all vehicles on the list really exist.

The system can search for a combination of subgoals and operators that will produce the desired accuracy with minimal processing costs. If a function alters its accuracy estimate after it has run to completion then the system will have to replan on the basis of these results.

In the course of such replanning, a function trying to achieve a goal might demand higher accuracy of its subgoals than before. It would be ideal if the subgoal function were allowed to use the results and intermediate results of its previous computation to do this, since this information could sometimes result in substantial time savings. Similarly, the higher-level function should be able to incorporate the new information without starting from scratch.

An accuracy scheme implies some subtle but important modifications on the basic demand-driven evaluation strategy. First, functions do not simply request that their arguments be evaluated; they must also decide how accurately they should be evaluated. Second, they may later come back and ask for even more accuracy, making it desirable to calculate this from the less accurate version and other information rather than from scratch. Finally it implies a departure from the notion that functions always yield the same results when given the same inputs: rather, they yield results that are in some way "close."

8.2. Extended Reduce

The ordinary REDUCE construct is used to compress a list, given a binary operator and a starting value. The extended REDUCE allows the reduction to stop before all elements have evaluated. There are two possible forms that this feature could take.

The first would contain a user-defined halting condition. For example, if the user wished to average outcomes of an experiment, his list might be a List Of Integer, his starting value [0,0], and his combining function

\[
\text{Function}(\text{Sum}: \text{List of Integer}; \text{u}: \text{Integer}): \text{List of Integer};
\text{List of Integer} : [n+\text{Sum}[1], 1+\text{Sum}[2]].
\]
This function accumulates the sum of the results in the first member of a list and the number of results in the second member. If the user wanted the accumulation to stop after ten results and didn’t care which they were, he could include the halting condition

\[
\text{Sum}[2] \geq 10
\]

Reduction would cease as soon as either the halting condition was met or all list elements had been successfully evaluated.

The second halting method is related to the accuracy mechanisms of Section 8.1. The REDUCE construct would have a desired accuracy either specified directly or inherited from a containing expression. Processing of list elements would cease when the accuracy measure was met.

The extended REDUCE would be a generalization of the CHOICE mechanism. CHOICE with strict function calls forms an And-Or tree of subgoals, while the extended REDUCE allows more sophisticated goal trees, including at-least-n-of and enough-of Nodes.
Chapter 9
Conclusions

What are the roots that clutch, what branches grow
Out of this stony rubbish? Son of man,
You cannot say, or guess, for you know only
A heap of broken images ...

The research presented in this dissertation has resulted in a system that automatically partitions and schedules programs written in a simple applicative language. It has been shown to perform successfully on some realistic test cases. Many of its deficiencies can be addressed within the existing framework, including the slow execution speed, the lack of user control over partitioning and distribution, and the lack of imperative constructs.

9.1. History and Goals

The research was inspired by an attempt to write a distributed application (the Distributed Sensor Network example of Section 5.1) using only PASCAL and a message passing facility similar to that of Accent. The system took two months to develop, excluding the time needed to write the signal processing algorithms. By automating the partitioning, scheduling, and fault tolerance mechanisms we hoped to create a framework in which such distributed applications could be developed in a matter of hours.

Previous systems have used a special process control language or special language features to describe the top level or task structure of a program while the lower level algorithms were defined in a standard algorithmic language. STARDUST uses a single set of constructs for both, doing away with the distinction between tasks and functions by making every function a potential task. Any function that calls several others may serve the role of a task group generator in addition to its role in the algorithm.

We have experimented with the idea that the system could partition and schedule the program with a minimum of information: execution time estimates for all functions. These
early decisions — a uniform language and minimal user intervention — were responsible for most of the other design and implementation features.

9.2. Development and Evaluation

An applicative language was chosen because of its well-known benefits to distributed computing: easy extraction of parallelism, uncomplicated restarts, and freedom from locking needs. This was a sound decision, though limited imperative constructs such as those described in Section 7.3.1 would have been useful for implementing fast algorithms. Unrestricted use of imperative constructs would have forced us to leave extraction of parallelism to the user and to either use transaction logs for restarts or disallow restarts altogether.

It was clear that a standard depth-first evaluation mechanism using a stack would not be sufficient, since some breadth-first traversal of the expression graph is needed to discover and distribute subcomputations. An action queue was chosen as the most flexible mechanism, allowing breadth-first traversal, depth-first traversal, and combinations of the two. It is also easily extended to priority schemes. While we used purely breadth-first evaluation throughout our implementation, a depth-first option for less space-consuming evaluation would be a useful addition, and an improved notion of priority would probably improve performance.

Run time redundant subcomputation elimination (RSCE) is largely independent of the other features, except to the extent that high run time costs of other features make the cost of RSCE more acceptable. Its cost is large but not prohibitive: it accounts for between a quarter and a half of the execution time in STARDUST 1.0. If there are no redundant subcomputations, then this time is completely lost, but it can greatly improve the execution speed of certain algorithms by providing a form of automatic dynamic programming.

The initial partitioning method — expanding function calls to their definitions when they had excessively high execution time estimates — worked well when the definition contained several functions with similar execution time estimates. Unfortunately one of the most frequent uses of recursion, stepping through lists, usually expands to a relatively small call that processes the first element of the list and a relatively large one that processes the rest of it. Rather than burden the programmer with the responsibility for splitting the list into sublists of equal size, we chose to provide a small set of list operators that would satisfy most list processing needs, and to treat these operators as special cases for partitioning and scheduling. While this worked moderately well for simple programs, it cannot be viewed as a general solution since users are still forced to use recursion to operate on their own data structures.
Furthermore, the simple partitioning heuristics do not always produce good results when combinations of list operators are used. This problem could be lessened by recognizing common combinations or by providing even higher level list operators. A possibly fruitful area of investigation is a facility for allowing the user to define new high level operators and their partitioning heuristics.

The technique of overeager evaluation was designed to increase concurrency and decrease message traffic. The system assumes that function calls that are moved to other processors use all of their arguments and all elements of structured arguments. This technique reduces the number of messages needed to invoke computations (requests for the arguments need not be sent) and sometimes increases concurrency (the function can be expanded in parallel with argument evaluation). However, the cost of spawning and aborting irrelevant tasks can be high when the assumption that all arguments are needed turns out to be false.

Two disciplines for reporting the values of structured arguments were considered: the system could either return each element as it completes or wait for all elements to complete and return them in a single message. Since the former produces excessive message traffic and the latter demands complex and expensive measures to preserve lazy semantics, a compromise was considered, though not implemented. Elements would be evaluated and collected until the oldest one had passed a certain age threshold, then all completed elements would be sent in a single message.

With the mechanisms in place for aborting unneeded tasks, the parallel conditional was easy to implement. This construct gives interesting and possibly useful semantics, and tends to increase concurrency by evaluating its arguments in parallel. Unfortunately it also tends to swamp the system with irrelevant computations.

The irrelevant computations caused by overeager evaluation and the parallel conditional complicate error handling since errors from irrelevant branches must not be allowed to stop the system or even be reported. This should be accomplished by generating Fail tokens that describe the type of error and the place where it occurred. The system would report them only when they were passed back to the top level (STARDUST 1.0 uses a single uninformative Fail token). Irrelevant computations can also hurt performance by allocating storage, spawning subtasks, and simply consuming processor cycles. The extent to which resources are given to computations of unknown relevance should be controlled by advice from the user and the availability of such resources; the system should allow unlimited use of these resources only to branches of the computation that can be guaranteed relevant.
Contrary to expectations the scheduler proved to be the most troublesome component of the system. This was mainly due to the presence of non-strict functions, which make it necessary to do the partitioning and scheduling "top down" in order to ensure that these functions do not wait indefinitely for irrelevant arguments.

The current scheduling algorithm traverses the expression in breadth-first order. Each function call that it scans is either scheduled immediately or partitioned into a combination of calls and placed on the end of the action queue to be scanned again later. This produces proper termination semantics but restricts the scheduler to accepting its arguments in top-down order and scheduling each of them immediately, which in turn complicates the algorithm and sometimes results in poor schedules.

One approach to easing the top down restriction would have been to make use of the "usually strict" heuristic that applies to partitioning and scheduling. The system could attempt to partition the user's call completely before scheduling it, only stopping when the program was completely partitioned or the partitioning algorithm reached a certain depth. The partitioned program could then be scheduled bottom-up.

The scheduler does not consider the size of the objects that must be passed in messages. This was a good decision because determining the size of objects in advance is not in general possible, and because in the current environment the speed of message passing does not depend heavily on message size for messages less than a few hundred words.

The scheduler models the order in which tasks will be executed on other processors and attempts to minimize the synchronization delays as well as balancing the loads. Since simple load balancing would have been sufficient for most test programs examined, and since the model of other processors' evaluation order was too unrealistic to do much good when it was needed, this feature gave little or no benefit to STARDUST 1.0. More realistic models of the evaluation order are certainly possible.

We have described, but not implemented, a straight-forward compiler for STARDUST. The compiler produces relatively inefficient code when the parallel non-strict semantics of the full STARDUST language are allowed, but produces code similar to the output of a compiler for a standard block-structured language when all parameters are passed by value.
9.3. Final Words

We have presented the major decisions that went into the design of STARDUST, the ways in which they followed from one another, and an indication of their strengths and weaknesses. The final question to address is whether in sum they met the original goals of the project: automatic partitioning and distribution with a minimum of user intervention.

The goal of a uniform language was met as well as could be expected. The old dichotomy between process control language and algorithmic language has been replaced by a new one between the STARDUST language and the implementation language of user intrinsics. Note however that this is for reasons of efficiency alone; STARDUST functions and user intrinsics are equally powerful. User intrinsics can always be changed to STARDUST code when the performance of the interpreter or compiler improves to the extent that the unity and clarity of the higher level language outweigh the performance costs. It is unlikely, however, that user intrinsics will ever be done away with completely. The user of a distributed system is usually looking for speed, and will usually be able to find it in the imperative constructs and microcoding that user intrinsics make possible.

The automatic partitioning and scheduling have been shown to work for some simple but realistic test cases. They often produce unsatisfactory results on programs with unknown time estimates, programs that use recursion as a major top level control construct, and programs that have complicated top level control structures. The applicability of STARDUST will continue to increase as these problems are addressed by implementing facilities to correct for faulty load balancing and adding more sophisticated list operators or recognizing standard combinations of the basic ones.
References

[Aho 75] Aho, A.V., Hopcroft, J.E., and Ullman, J.D.
The Design and Analysis of Computer Algorithms.

An Asynchronous Programming Language and Computing Machine.
# 114A, Department of Information and Computer Science, University of

[Arvind 80a] Arvind, and Thomas, R.E.
I-Structures: An Efficient Data Type for Functional Languages.
MIT/LCS/TM-178, Massachusetts Institute of Technology, Cambridge,
Massachusetts, September, 1980.

[Arvind 80b] Arvind, Kathail, V., and Pingali, K.
A Dataflow Architecture with Tagged Tokens.
MIT/LCS/TM-174, Massachusetts Institute of Technology, Cambridge,
Massachusetts, September, 1980.

[Arvind 81] Arvind, and Kathail, V.
A Multiple Processor Data Flow Machine that Supports Generalized
Procedures.
In Eighth Annual Symposium on Computer Architectures. IEEE, 1981.

[Arvind 82] Arvind, and Gostelow, K.P.
The U-Interpreter.

[Bird 80] Bird, R.S.
Tabulation Techniques for Recursive Programs.

[Brinch-Hansen 78] Brinch-Hansen, P.
Distributed Processes: A Concurrent Programming Concept.

[Burstall 77] Burstall, R.M., and Darlington, J.
A Transformation System for Developing Recursive Programs.

[Cohen 79] Cohen, N.H.
Characterization and Elimination of Redundancy in Recursive Programs.
In Sixth Annual ACM Symposium on Principles of Programming
Languages, pages 143-157. Association for Computing Machinery Spe-
cial Interest Group on Automata and Computability and Special Interest
Group on Programming Languages, San Antonio, Texas, January,
1979.
[Cook 80] Cook, R.P.
* MOD - a language for distributed programming.

[Dijkstra 68] Dijkstra, E.W.
Cooperating Sequential Processes.


(DSN 82) Defense Advanced Research Projects Agency.
Distributed Sensor Networks Workshop January 6-7, 1982, Massachusetts Institute of Technology Lincoln Laboratory, 1982.

[Ericson 82] Ericson, L.W.
DPL-82: A language for distributed processing.

[Feldman 79] Feldman, J.A.
High Level Programming for Distributed Computing.

[Feldman 82] Feldman, G.
Functional Specifications of a Text Editor.

[Friedman 76] Friedman, D.P., and Wise, D.S.
Cons Should Not Evaluate its Arguments.

[Gilbert 83] Gilbert, E.J.
Algorithm Partitioning Tools for a High-Performance Multiprocessor.
Report No. STAN-CS-83-946.

Smalltalk-80: The Language and its Implementation.

Edinburgh LCF.

[Gray 80] Gray, J.
A Transaction Model.
IBM Research Report RJ2895, IBM Research Laboratory, San Jose, Ca., August, 1980.
[Henderson 76] Henderson, P., and Morris, J.H.
A Lazy Evaluator.

[Hoare 74] Hoare, C.A.R.
Monitors: An Operating System Structuring Concept.

[Hughes 82] Hughes, R.J.M.
Super-Combinators: A New Implementation Method for Applicative Languages.

[Kazar 84] Kazar, M.L.
Automatic Distribution of Programs and Data in a Distributed Environment.

An Architecture for a Loosely-Coupled Parallel Processor.
UUCS-78-105, Department of Computer Science, University of Utah, Salt Lake City, Utah, 1978.

A loosely-coupled applicative multi-processing system.
In AFIPS. 1979.

Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages.

[Landin 65] Landin, P.J.
A Correspondence Between Algol 60 and Church's Lambda-Notation: Part I.

[Lesser 79] Lesser, V., Serrain, D., and Bonar, J.
PCL: A Process-Oriented Job Control Language.

[Lesser 80] Lesser, V.R., and Erman, L.D.
Distributed Interpretation: A Model and Experiment.

[Mago 80] Mago, G.A.
McGraw, J.R.  
The VAL Language: Description and Analysis.  

Michie, D.  
"Memo" Functions and Machine Learning.  

Nelson, G., and Oppen, D.C.  
*Fast Decision Algorithms Based on Congruence Closure*.  

Nilsson, N. J.  
*Problem Solving Methods in Artificial Intelligence*.  

Rashid, R.F., and Robertson, G.  

Scherlis, W.L.  
*Expression Procedures and Program Derivation*.  
Stanford Artificial Intelligence Laboratory Memo AIM-341, Department of Computer Science Report No. STAN-CS-80-818.

Schwans, K.  
*Tailoring Software for Multiple Processor Systems*.  

Shoch, J.F., and Hupp, J.A.  
The "Worm" Programs -- Early Experience with Distributed Computation.  

Smith, R.G.  
*A Framework for Problem Solving in a Distributed Processing Environment*.  
STAN-CS-78-700 (HPP-78-28), Stanford University, Stanford, California, 1978.

Smith, R.G.  

Steele, G.L.  
*Common Lisp: The Language*.  
Data-Driven and Demand-Driven Computer Architectures.

[Turner 79] Turner, D.A.
A New Implementation Technique for Applicative Languages.

[Unix 82] Accetta, M.J.
ENET(4).
1982.

Western Digital Corporation Systems Technology Center, Pittsburgh, Pa, 1983.

[Wadler 84] Wadler, P.
*Listlessness is Better than Laziness.*

[Whiteside 84] Whiteside, R.A., Hibbard, P.G., and Ostlund, N.S.
"Conventional" and Systolic Parallel Algorithms for Monte Carlo Simulations of Molecular Motions.
1984.
Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA, 15213. Submitted to ACM Transactions on Computer Systems.

*The Design of an Optimizing Compiler.*