MIT/LCS/TR-323

EFFICIENT IMPLEMENTATION OF APPLICATIVE LANGUAGES

William Beekley Ackerman

This blank page was inserted to preserve pagination.

Efficient Implementation of Applicative Languages

by

William Beckley Ackerman

S. B., Massachusetts Institute of Technology
1967

S. M., Massachusetts Institute of Technology
1977

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

March, 1984

© Massachusetts Institute of Technology, 1984

Signature of Author _____
Department of Electrical Engineering and Computer Science
March 28, 1984

Certified by _____
Jack B. Dennis
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Graduate Committee

Efficient Implementation of Applicative Languages

by

William Beekley Ackerman

Submitted to the Department of Electrical Engineering and Computer Science
on April 4, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The analysis of parallelism in an applicative program is much easier than in a program written in a conventional statement-oriented style. This makes it possible for an optimizing compiler to prepare such a program for extremely efficient execution on a suitable enormously parallel computer. This thesis explores the transformations that must be made to achieve very high performance for numerical programs when executed on a computer that uses data flow principles in its operation.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Keywords: applicative programming, parallel computation, optimizing compilers, data flow computation.

ACKNOWLEDGMENTS

I would like to thank my thesis supervisor, Jack Dennis, for his encouragement and support during the course of this research, and for providing the intellectually exciting environment that is the Computation Structures Group.

My other thesis readers, Bert Halstead and Arvind, have been very helpful in this work, and I have enjoyed many wide-ranging discussions with them on many topics in the field of computer science.

My many friends in the Computation Structures Group have been a continual source of stimulation, ideas, and companionship. I would particularly like to thank Nena Bauman, Andy Boughton, Dean Brock, Gao Guang-Rong, Clement Leung, and Willie Lim for their contributions to this work and their friendship over the long period that it has been my pleasure to know them.

This research was supported by the Department of Energy under grant number DE-AC02-79ER10473, and by the National Science Foundation under grant number MCS-7915255.

Table of Contents

ACKNOWLEDGMENTS 3

1. INTRODUCTION 7

1.1 Background 8
1.2 Data Flow Analysis with Scalars 10
1.3 Applicative Programming 13
1.4 Array Computation in Applicative Systems 17
1.5 Spatial Distribution of the Computation 18
1.6 Spatial Distribution of Arrays 19
1.7 Regularity of Array References 21
1.8 Synopsis 22

2. THE EXECUTION MODEL AND ITS NOTATION 24

2.1 The Source Language 25
2.2 Compile-time Parameters 28
2.3 The Graph Model 30

3. ITERATION UNFOLDING AND ARRAY INTERLACE 33

3.1 Loop Unfolding 35
3.2 Initial Unfolding 36
3.3 Final Unfolding 36
3.4 Unfolding When the Number of Cycles is Not Well-behaved 37
3.5 Loops That Have No Obvious Index Variable 39
3.6 Rescaling the Index Variable 40
3.7 Interlace 42
3.8 Graphical Representation of Unfolding 43
3.9 Nested Loops 46
3.10 Coalescing Control Structures 49
3.11 Interlace of Multidimensional Arrays 53
3.12 Geometrical Representation of Unfolding 55

4. MATCHING INTERLACE AND UNFOLDING 57

4.1 Random Array Access and the "PERMUTE" Operator 59
4.2 Affine Array Operations with Reference Interval Equal to One 61
4.3 Reference Interval Not Equal to One 64
4.4 Reference Interval * Unfolding < Interlace --- Permute 64

4.5 Reference Interval * Unfolding > Interlace --- Multiple Operators 65
4.6 When Things Are Not Powers of Two 67
4.7 Loops with Many Arrays 68
4.8 Choosing the Interlace and Unfolding 69
4.9 The Periodic Cyclic Reduction Algorithm -- Reduction Part, Inner Loop 70
4.10 The Periodic Cyclic Reduction Algorithm -- Reduction Part, Outer Loop 83
4.11 The Periodic Cyclic Reduction Algorithm -- Substitution Part 84

5. UNFOLDING AND INTERLACE IN MULTIPLE DIMENSIONS 87

5.1 Relationship Among Unfolding, Reference Interval, and Interlace 88
5.2 Unfolding and Interlace That do Not Nest Similarly 90
5.3 Array References Depending on More Than One Variable 94

6. COMBINING COMMON SUBEXPRESSIONS 102

7. AUXILIARY LOOP VARIABLES 108

7.1 Boundary Conditions 109
7.2 Combining Subexpressions Over Long Distances 110
7.3 Nested Loops 112
7.4 Periodic Cyclic Reduction Revisited 114

8. ARRAY CREATION OPERATORS 116

8.1 Structure of Loops that Create Arrays 118
8.2 How to Use the Append Operator 121
8.3 Common Subexpressions Using SELECT and APPEND 127

9. ARRAY REFERENCES WITH UNKNOWN REFERENCE INTERVAL 133

9.1 Reference Multiplier Not Equal to One 138
9.2 Using Multi-selects to Provide the Data 139
9.3 Array Appends with Unknown Reference Interval 141

10. DISTRIBUTION AND DECOUPLING OF CONTROL STRUCTURES 144

10.1 Multidimensional Data Dependencies and the "Wavefront" Transformation 144
10.2 Achieving the Decoupling 148

11. FLATTENING OF ARRAYS 151

11.1 Order of Indices in Flattening 151
11.2 Using Flattened Slices in SELECTs 153
11.3 Getting Permuters and Conditionals Out of the Way 153
11.4 Creating Flattened Slices 156
11.5 Flattening the Control Structure 159

11.6 Interlace Not Equal to Unfolding -- Alternators 162

12. BOUNDARY CONDITIONS 164

12.1 Dividing Loops 165
12.2 Dividing Nested Loops with Multiple Array References 166

13. LARGE SCALE PROGRAM STRUCTURE 168

13.1 Example -- the Cyclic Reduction Algorithm 170
13.2 The Fast Fourier Transform 174

14. PIPELINING AND ARRAY REMOVAL 178

14.1 Array Removal 182

15. CONSIDERATIONS OF MACHINE DESIGN 184

15.1 Array Localization and Management 185
15.2 Reference Counts 186
15.3 Pseudo-reference Counts 188
15.4 The "Create" and "Reclaim" Operators 190
15.5 Fully Static Allocation 191
15.6 The "Pack" and "Unpack" Modules 192
15.7 Boolean Pipelines 193

16. CONCLUSION 194

REFERENCES 196

1. INTRODUCTION

There is an enormous literature of numerical algorithms, for such applications as hydrodynamics, image processing, and weather forecasting, that require very high performance computers, much higher than can be attained without the use of parallelism. These algorithms are commonly executed on vector, pipeline, or array "supercomputers", which exploit parallelism in the program to a limited extent. These algorithms can in fact exhibit incredibly more parallelism than conventional supercomputers exploit. (This is especially true since, as faster computers become available, these programs are written to use an ever finer grain in their analysis or simulation of physical phenomena, which increases the parallelism even further.) The reason that conventional supercomputers fail to fully exploit the parallelism is that they can only take advantage of local parallelism. Their design has a "bottleneck" between the control section and the arithmetic processing section. This bottleneck makes it impossible to exploit parallelism except in a local and restricted way. Tremendous overall speed can be achieved if programs are executed on a computer designed to eliminate the bottleneck and take advantage of parallelism in a more general way. Applicative programming makes such computing systems feasible. The goal of this thesis is to show how to break the bottleneck and exploit the parallelism.

The bulk of this thesis is a presentation of transformations that can be made to applicative programs to achieve high performance. These transformations should be made by an optimizing compiler. This thesis can therefore be thought of as a description of the principles on which such an optimizing compiler ought to be based.

There is really only one aspect of computation that makes the program translation problem difficult -- arrays and other "aggregate" structures such as records. For this reason the major focus of the thesis will be array computation. It will be shown that, if the program is written in an applicative language and represented in the form of a data flow graph, the potential parallelism can be exploited in a natural way. A suitable

direct-execution computer for such graphs, that is, a *data flow computer*, should be able to achieve extremely high performance in executing such a program.

1.1 Background

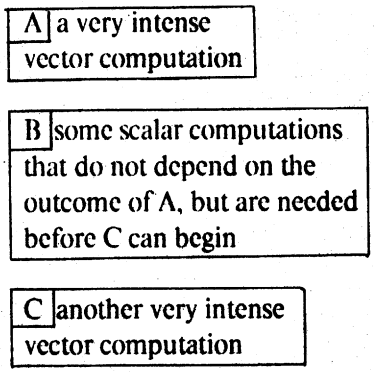
Array processors like the Illiac IV [12] have many processors operating in lock-step. This makes it possible to do repetitive array operations at a speed that is not limited by one's ability to shave nanoseconds on one processor. This in turn makes the bottleneck extremely fast (potentially as fast as we like, for sufficiently large arrays and sufficiently large computers), but it is still a bottleneck. The reason is that the computer is still doing essentially only one thing at a time. Vector and pipeline processors [28] perform repetitive operations on arrays sequentially, but with substantial overlap from one element to the next.

In vector, pipeline, or array processors, the only places where massive speedup can be achieved are repetitive, nearly identical computations on arrays where there is no data dependence among the computations on the various array elements. That is, the only data that are available for the computation of a particular array element are the data in corresponding positions (possibly with shifted indices) in other arrays (including "mask" arrays of control information that can enable or disable certain operations) and global data that affect the computations of all elements.

This amounts to an inflexibility that seriously limits the parallelism exploitable by such machines, in comparison with the parallelism achievable in principle. Among the lost opportunities for parallelism are:

1. Scalar operations. Although high performance computers sometimes have impressive mechanisms to speed up scalar computations through data caches, instruction caches, instruction lookahead, and "on the fly" optimum scheduling of functional units, the vector, pipeline, or array hardware of the big supercomputers is completely ineffective for scalar operations. As supercomputers become more powerful in their ability to process regular arrays, this "scalar bottleneck" becomes proportionally more serious.

Consider the following fragment of a program to be executed on a conventional supercomputer:



Assume all three blocks take about the same amount of real time, but of course blocks A and C contain vastly more computation than B and use the computer's vector mechanism to perform it. Block B could in principle be executed simultaneously with A. Since it actually performs an insignificant amount of computation in comparison to A, it would be very little additional burden to the computer's processing units. It would double the computation speed (assuming a long sequence of these alternating vector and scalar blocks) by letting the intensive computation C commence as soon as A completed, thereby keeping the vector mechanism busy all of the time rather than half of the time. The problem here is that, while the arithmetic units are very powerful, the computer's control mechanism lacks the flexibility to schedule operations in this manner.

2. Irregular array operations. Conventional supercomputers have very little ability to perform different computations on different elements of an array. Their ability is generally limited to use of "mask" arrays that can enable or disable specific instructions on specific array elements. This, too, is a lack of flexibility in the scheduler. The computer's vector mechanism runs at such a high speed that it doesn't know what it is doing in terms of the control structure of the program. It does not have time to stop and think about individual array elements.

3. Sequential data dependencies within iterations. Some operations on arrays have an actual data dependency from the computation involving one element to the computation involving the next. True simultaneous processing is of course impossible here. However, a large amount of parallel processing is usually still possible, because only a small fraction of

the arithmetic operations are in the "critical path". Conventional supercomputers, however, can't take advantage of this because the operations in the critical path are deep inside the pipeline and can't be "fed back" at full speed from one point in the pipeline to an earlier point.

All of these shortcomings arise from one fundamental problem: the control mechanism ("scheduler") of conventional supercomputers lacks the flexibility to take advantage of the parallelism actually present in the algorithm.

1.2 Data Flow Analysis with Scalars

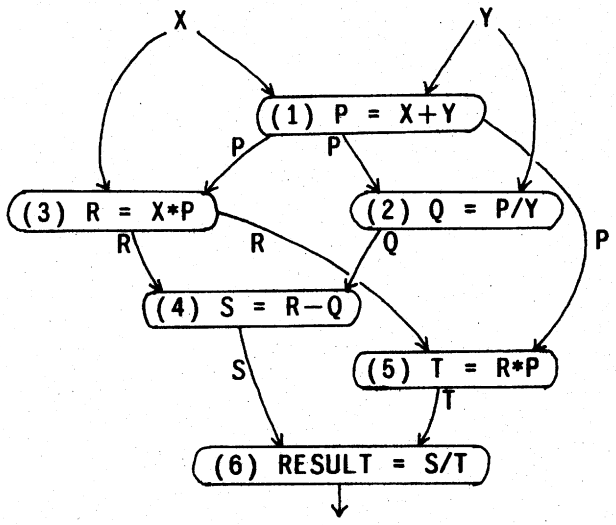
For algorithms not involving arrays, the method for "optimally" scheduling operations to take full advantage of parallelism is well known. It is the data flow model. The instructions of the program are arranged in a "data flow graph", showing their interdependencies.

For example, the FORTRAN program

```
1       P = X+Y
2       Q = P/Y
3       R = X*P
4       S = R-Q
5       T = R*P
6       RESULT = S/T
```

would have the following data flow graph:

Fig. 1.1

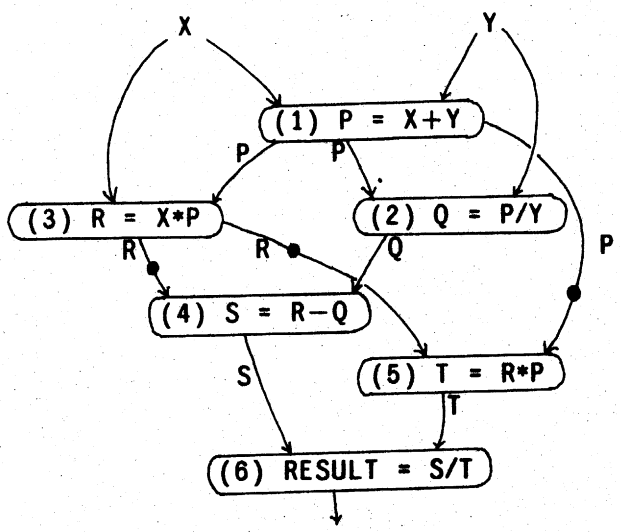


Data items are transmitted in programs in FORTRAN and most other languages through the writing and reading of variables. In a data flow graph this transmission is shown by an arrow from the source operation to the destination operation. For programs that deal only with scalars, the translation from conventional languages is not too difficult [7, 20, 46].

The ideal parallel scheduler will execute each operation as soon as the operands that it requires become available. One can visualize the operation of such a scheduler through the "token" model: Results of operations are created in the form of tokens (shown as black dots) that flow along the arrows. The ideal scheduler will therefore execute each operation when it has tokens present on all of its inputs.

A possible state of the execution of the above graph might be

Fig. 1.2



1 & 3 have completed
2 is undergoing execution

Any scheduler that uses this criterion for instruction execution will find all opportunities for parallelism, without regard for locality. The possible parallel instruction executions will not be limited to a single locus of control, or even a local "window" of control, or even some collection of regular, repetitive operations. Computers capable of direct execution of programs encoded as data flow graphs have been proposed [9, 16, 17, 22, 21, 33, 40], and some prototypes have been built.

The design of practical direct execution data flow computers requires a few enhancements and refinements to this model. Some of these refinements will be described briefly here -- detailed consideration is not crucial to this thesis. A *dynamic* data flow graph can effectively expand and contract during execution. This is typically done when a function is invoked. A function invocation is initially encoded as an "apply" node. When a token reaches that node, it turns into (that is, the computer operates as though it had turned into) the graph representing the function body. A *static* data flow computer can only handle static graphs. In static graphs, function invocations are expanded at compile time, before execution begins. This necessarily precludes general recursion at the hardware level.

Another refinement is the use of special nodes to control conditionals and iterations. In the standard model data flow graph [23, 24] these nodes are called "T", "F", and "MERGE". The "T" gate takes one argument of arbitrary type and one boolean argument. If the second argument is *true*, the first argument is passed on as the result. Otherwise it is simply discarded. The "F" gate does the opposite. "T" and "F" gates control the initiation of conditional computations and the recycling of values during iterations. The "MERGE" gate takes two arguments (labeled T and F) of arbitrary type and a third argument which is boolean. It accepts and passes on as the result whichever of the first two arguments is indicated by the third argument. The unused argument is not absorbed and is not even required for the operator to fire. This operator is used to merge the results of the various arms of a conditional, and to define the initial loop variables for an iteration. See [18, 46] for a detailed description of the use of these gates. In this thesis, data flow graphs will be shown in a level of detail that does not involve these gates -- conditionals and iterations will be shown in terms of the more abstract nodes described in Section 2.3.

1.3 Applicative Programming

For algorithms that deal with arrays, achieving "maximally parallel" execution is a far less tractable task. One reason for this is fundamental and will be discussed in Section 1.4; the other is that array computation is usually not performed applicatively. This second reason is not fundamental -- it is an artificial problem arising from conventional ways of conceptualizing computation. It must be disposed of before the real issues can be addressed.

The processing of data in arrays is commonly perceived in terms of the "von Neumann" architecture, which is not compatible with true data flow analysis. In this model of architecture, one conceives of a computer as a machine that manipulates data stored in a memory, by executing commands in some sequence. For programs that deal with scalars, it is not difficult to recast one's thinking in terms of data dependencies instead of sequential command execution, nor is it difficult to translate programs into data flow graphs. The reason is that the "memory slots" of the machine's memory can be thought of as nothing but "pipes" between

arithmetic operations, that is, variables in the program become arcs in the data flow graph. Hence, the von Neumann style of thinking is not a serious burden in this case

The von Neumann style of thinking has a much more profound effect when arrays enter the picture. Instead of being simply "memory slots", the memory consists of "blocks" that are treated as having a very real existence. The need to pass entire arrays from one part of the program to another, and the expense of copying them, means that the memory blocks that arrays occupy have a very real existence in one's conception of what the program is doing, and in the execution model that a programming language presents to the user. Execution of high level languages on von Neumann computers is customarily performed in a way that requires that this execution model be pervasive in the external behavior of the "virtual machine" that defines the language. So pervasive, in fact, that for a great many high level languages, such as FORTRAN, BLISS, and C, the target machine for which the language was designed is clearly discernible from reading the language manual.

This intertwining of the von Neumann conception of the way arrays are manipulated with the way humans write programs makes the bottleneck difficult to break. Vector and pipeline supercomputers are designed to do their vector operations by regular manipulation of successive words in blocks of memory, because that is the way programs are conventionally written. It is nearly impossible to get truly flexible parallel execution without getting away from the notion of memory blocks manipulated exactly the way the programmer specified. But it is nearly impossible for a compiler to translate a program as anything other than such a memory block manipulator, because the actual data dependencies in conventional programs are so obscure.

What is needed, then, is a style of programming that does not enslave the computer to the von Neumann style of execution, and a style of translation and execution that can truly exploit the parallelism. Such a style of programming exists: it is called applicative programming [4]. In applicative programs, all data

dependencies, even those involving arrays, are apparent.[1] (The reason it is easy to deduce the data dependencies of programs that do not use arrays is that such programs are essentially applicative anyway, no matter what language they are written in.)

For the purposes of this thesis, the important characteristic of applicative languages is the following: Whenever any array (or other data aggregate such as a record) is modified by some computational step (say A), the effect of that modification must always be made known, by some mechanism, to other computational steps (say B). If the transmission of that effect takes place because B "knows" where the array is, perhaps having known its location since before A modified it, the language's model is not applicative. Most languages are not applicative -- in most languages those statements that refer to the array X are compiled into code that knows where X is allocated, whether on the stack, in a COMMON block, or in some local or global memory space, and the code manipulates X through that predetermined address.

This type of transmission of effect (called a "side effect") is not compatible with data flow scheduling because the constraint that step A must precede step B does not take the form of a token transmission from A to B.

In an applicative language model the only way computational step A affects step B is through direct transmission of its result value to B. The result of A is a new array (conceptually at least), containing the outcome of A's action. Only steps that receive this result can sense A's action -- no other step can sense it indirectly by reading the array that it knew A would modify. (Of course, in an implementation, step A may very well write into memory. It is the job of the implementation to make sure that the behavior is consistent with the applicative model.)

---

1. We do not consider data dependency removal that might be effected by rearranging the algorithm, or by using mathematical identities beyond the normal purview of a compiler.

This type of transmission is fully compatible with data flow scheduling. One allows tokens to carry array values as well as scalar values. Array operations are performed by functional operators such as the ones to be described shortly.

The use of applicative arrays is a very radical departure from the conventional methods of computation in numerical applications. It has led to widespread and severe criticism regarding its efficiency in such applications. A major goal of this thesis is to answer these criticisms.

A few words about array nomenclature are in order. The differences between applicative arrays and the "arrays" (or "subscripted variables") of conventional languages are profound. It is very easy to use slippery language that confuses the two concepts. In an applicative system, an array is nothing more or less than a series of values, along with the low and high bounds information indicating between what index limits these values lie. The array *is* the information. In von Neumann style thinking, on the other hand, an array is a *place* where values may be stored in sequence.

Strictly speaking, it is therefore incorrect to speak of "writing `X` into array `A` at position `J`." One should only speak of "producing an array with `X` at position `J` and otherwise identical to `A`." In practice, however, the way array computation is performed in numerical programs is not much different in a von Neumann system or an applicative one. In FORTRAN, an array is filled by subscripted assignments to it, typically in `DO` loops. One can quite properly speak of "writing `X` into `A` at position `J`." In an applicative language, one would use an iteration variable which is an array and which is repeatedly rebound. One should speak of "rebinding the iteration variable `A` to an array containing `X` at position `J` and otherwise identical to `A`'s old value." In practice, the two operations are analogous to each other and are used in the same way in algorithms written in conventional and applicative systems, respectively.

In this thesis, the more conventional nomenclature will often be used, even though applicative array operations are being described. The reader should be aware that "writing into an array" is something of an abuse of language in this context.

1.4 Array Computation in Applicative Systems

Having disposed of the fictitious problems with array computation, we can consider briefly, if imprecisely, what it is that really makes array computation difficult to deal with.

The fundamental characteristic of arrays that makes computation with them difficult to analyze is that arrays, in general, contain an amount of data that is incommensurate with the size of the program structures that manipulate them. Programs that manipulate scalars only need to deal with as many data items as there are names to denote them appearing in the text of the program. (Of course, it can be argued that the information content of a small collection of scalars can be increased without limit by increasing the precision, that is, the word length. However, this has no effect on the complexity of the program's behavior as long as the data are treated as indivisible units. If the program looks inside the data words instead of treating them as indivisible units, they become, in effect, arrays.)

Any arrays that are of small fixed size present no problem.[1] The reason is that such an array can be treated simply as that many scalars. Data dependencies can be analyzed for the various array elements individually. Random accesses ("subscripted array references") can be rewritten as conditional program structures that test the index value one case at a time.[2]

---

1. The meaning of "small" is of course subjective. Any number is small if we are willing to make the control structures large enough. Practical experience tells us, however, that 3 is small and 100 is large.
2. It is an aversion to 100-armed conditionals that leads us to say that an array of 100 elements is not "small". It is clearly not efficient to let the control structure of a program grow in proportion to the size of the arrays.

It is when arrays are large, however, or when their sizes are unknown in advance and hence might conceivably be large, that they exhibit their difficult behavior. In this case, program structures such as conditionals can no longer encompass the range of the possible array indices. Then the indexing operations for reading and writing arrays become essential and nontrivial. When a compiler sees just an expression like

```
A[J]
```

it knows nothing, in the most general case, except that J is the result of some integer computation which it perhaps can't analyze, and which it must use to fetch the appropriate element from the array A, whose internal structure it can't discern.

The fact that the manner in which data are written to and read from arrays is often unpredictable makes it very difficult to organize programs for extremely efficient parallel computation. When unpredictable, or "random" access to arrays occurs, it is desirable to store the array in a localized area in physical memory, so that the hardware will be able to make the accesses readily. This is the antithesis of truly bottleneck-free parallel distributed computation.

Arrays that are accessed unpredictably are the bane of conventional supercomputers also. The vector and array mechanisms of these machines can only deal with fairly regular algorithm structures. The various optimizing compilers and program transformers for these systems search the source program for computational structures (DO loops, typically) that can take advantage of the machine's array mechanism.

1.5 Spatial Distribution of the Computation

The way to achieve very high performance is to distribute the computation across many distinct processing units. Since we envision execution on a data flow computer, the first step is to distribute the computation across a data flow graph. The second step is to map that graph onto many processing units in such a way that those units can proceed as nearly independently as possible and with as few constraints as possible. The mapping problem, while extremely important, will not be addressed in this thesis. We will

consider parallelism that is visible in the data flow graph to be parallelism achievable by the computer. In effect, we will consider each instruction to have its own processor, so that "ideal" execution is obtained -- each instruction in the graph can fire when its operands are available, independently of anything else that is happening elsewhere in the computer. This allows us to ignore the mapping problem. In an actual computer, one processing unit would have to serve the needs of many instructions, so the mapping must be chosen to balance the loads on the processors.

The representation of a program as a graph constitutes a distribution of the program over many processors and consequent exposure of its parallelism. This amount of parallelism is not enough. Additional parallelism, enormous amounts of it, will be found among the different cycles of iterations ("loops"). To expose that parallelism in the data flow graph, the cycles will be *unfolded* (or "expanded" or "unrolled"). This is the principal technique whereby huge increases in computation speed are obtained. Of course, this unfolding takes up space in the machine representation of the graph. Very high performance computers will have to have a large "instruction space". The total speed of a system is limited by the product of the instruction space and the processing speed per instruction. (The latter quantity depends on the speed of each processing unit and the number of instruction cells that it must serve, that is, the degree of deviation from ideal execution.) This thesis will be oriented toward extremely large machines executing programs whose loops have been unfolded accordingly.

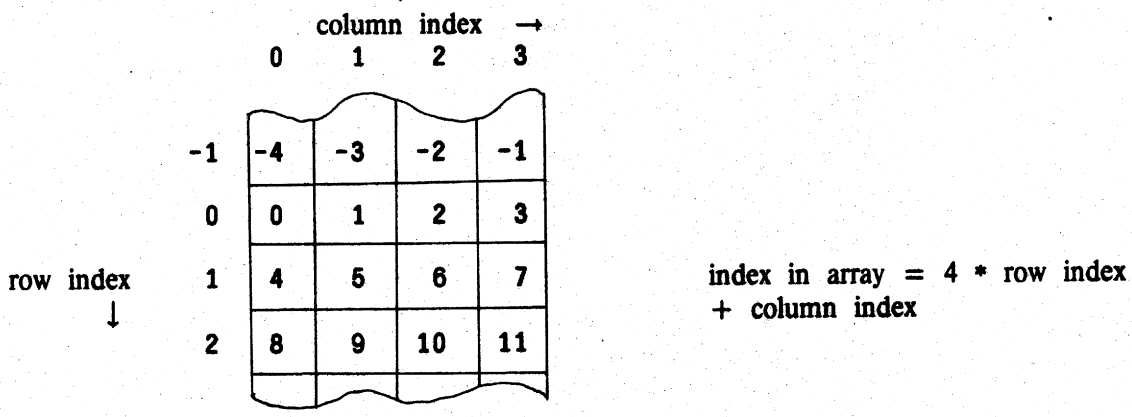
1.6 Spatial Distribution of Arrays

The translation of programs written in an applicative source language into static data flow graphs is not too difficult a task. This is so even if array operations are used, if the execution mechanism for the graphs supports the array operations directly, exactly as they appear in the source program. Such a direct hardware support for arrays means that arrays appear as tokens in the data flow graph, just as scalars do. The operations for reading and writing arrays look like any other operations in the graph -- argument tokens flow into them and results flow out. Implementing such a scheme is rather tricky. Since it is not feasible to carry all of the

array data in an actual token, a method must be worked out whereby the array data are stored in a memory unit and some sort of pointer to it is carried in the token. It is the job of the implementation (hardware, firmware, and compiler) to ensure that the resulting mechanism is equivalent to the idealized mechanism in which the entire array is carried in the token. The details of such a mechanism are beyond the scope of the thesis. The interested reader is referred to [3], in which an implementation is described for general dynamic arrays, or [25] in which it is described for a restricted class of array operations for which efficient execution may be obtained very cheaply. For the purposes of this thesis, it is assumed that some appropriate hardware mechanism exists.

This still leaves a serious bottleneck in the handling of arrays, especially in unfolded loops operating on a very large, high performance computer. To circumvent this, as we expand the loops, we will expand the arrays also. The general representation and translation technique that will be used is *spatial interleaving*. A linear array is rearranged as if it were a two-dimensional strip of fixed width. If the given array's size is known at compilation time, the strip's length is known as well.

Fig. 1.3



Each column is then treated in the hardware as a true array, which is implemented by whatever mechanism the hardware provides. These column arrays are handled independently. High speed is achieved, as on conventional computers, by wide interleaving, that is, by making the strip very wide and having many independently handled arrays. Unlike conventional computers, the separate arrays do not need to pass

through the same bottleneck, so the amount of possible interleaving is virtually unlimited. The interleaving is performed by the compiler and is invisible to the programmer. It is the job of the compiler and the hardware to make the overall computation faithful to the programmer's (uninterleaved) intentions.

When the array is more than one-dimensional, a similar interleaving is used in each dimension.

1.7 Regularity of Array References

The techniques to be shown in this thesis work well only for array operations that take place in regular, repetitive patterns. The bulk of the array references (reading or writing) must be inside iteration loops, with the array index of the datum being accessed related to the cycle number in the loop by an *affine mapping function.* An affine function is a first degree polynomial, that is, a function of the form $f(x)=Ax+B$. A typical iteration that obeys this rule is something like

```
      DO 10 I = 1, 1000, 2
      .......
      ...  A(4*I+35) ...
      .......
10    CONTINUE
```

We can actually relax the restrictions in a number of ways at modest cost in efficiency. These relaxations will be presented later in the thesis. One relaxation is that the function mapping the relative time of data access to its array index can be a *piecewise-affine* function. This means that it can be broken up into a "small" number of pieces, each of which is of the form $f(x)=Ax+B$. The question of how small is "small" is answered in the usual way: it should be small enough that program structures of that size (e.g. a conditional with one arm per piece of the array) are feasible. The number of pieces is typically very much smaller than the size of the array itself.

Common examples of piecewise-affine array mapping can be found in the handling of boundary conditions. All elements of an array except the first and the last are created in a regular sequence by a uniform method, but the first and last elements are computed as special cases. Such an array might be expected to be produced in three pieces.

Perusal of common numerical application programs shows that they generally satisfy the restrictions on regularity of array access. Of course there are exceptions, but those exceptions are relatively rare and do not usually occur in the computationally intensive part of the programs. It follows that, for many numerical calculations, tremendous performance improvement can be obtained through the use of the data flow principles to be presented.

1.8 Synopsis

Chapter 2 will define the conventions and notation to be used in the rest of this thesis. Chapter 3 will describe the two basic operations that we will perform on programs to obtain high performance: *unfolding* of loops and *interlace* of arrays. Chapters 4 and 5 will describe how those operations will be made to work in practical situations. Chapters 6 and 7 will describe additional important transformations. Chapter 8 will discuss the special problems that arise when programs "write" into arrays. Chapter 9 will discuss the types of transformations that must be made when the compiler has less information about array reference indices than we would like. Chapter 10 will describe how parts of a program can be permitted to get "out of step" with each other for improved performance. Chapter 11 will describe a final transformation of arrays just before they are mapped onto hardware structures. Chapter 12 will describe how occasional "irregular" array references are handled. Chapter 13 will show how these methods apply to programs with nontrivial structure, presenting two real example programs: a tridiagonal equation solver and the fast Fourier transform. Chapter 14 will describe some special array optimizations that can sometimes be made. Chapter 15 will present a few thoughts about the design of the actual computer upon which these transformed programs should run efficiently. Chapter 16 will present some conclusions about the manner in which a practical optimizer might

2. THE EXECUTION MODEL AND ITS NOTATION

The execution model that will be used is the static data flow graph. This is an ideal model for the study of parallelism in applicative programs, since the data flow graph of such a program faithfully represents the possible parallelism in its execution. Since there are no side effects, the execution of any operation can take place as soon as its immediate predecessors in the graph have computed their results. It does not need to wait for any other control signal; no other activity in the program could possibly affect it.

The appropriate computer for this model is an idealized "data flow computer" that can "fire" (execute) all operations independently. Such a computer, in the ideal limit of true simultaneous execution of *all* enabled operations, can take ultimate advantage of the parallelism in an applicative program.

In the static data flow model, the data flow graph does not change its structure during program execution. This property assumes that functions are expanded "in line", prior to execution, as if by a macro preprocessor, and makes execution of recursive functions very difficult. The graph consists of nodes which can perform predetermined applicative operations, and directed arcs carrying data "tokens" among the nodes. Except for the special nodes to control conditional and iterative computation, the nodes have a straightforward firing rule: a node can fire whenever its input arcs contain tokens and its output arcs do not. When it fires, it absorbs its input tokens, using the data they contain as arguments to compute its function, and places a copy of the result of that computation on each output arc to be sent to other nodes.

As noted previously, the translation of an applicative program into a data flow graph is not difficult if only scalar values are involved. Likewise, the qualitative analysis of how such a graph would be executed on any reasonable data flow computer is trivial. When arrays are involved, the transformations that must be made become rather involved.

Since all array operations must be performed as applicative operations encoded in the data flow graph, a set of "basic" machine level operations suitable for high speed computation must be chosen. These are the data flow computer's analogue of the indexed load and store operations that are the basic array instructions of conventional computers. The fundamental operations are

```
select      written as: A[J]
append      written as: A[J:V]
```

Select performs the same "subscripted read" operation as in conventional languages. Append returns an array similar to the argument "A", but with value "V" at index "J", replacing whatever "A" has at that index. While different operations can be devised, applicative programming requires that any "subscripted write" operation be somewhat similar to the **append** operator.

There are a number of alternative forms of the **append** operator. Two that are especially useful for the sequential creation of arrays are:

```
addh        written as: addh(A,V)
addl        written as: addl(A,V)
```

These return an array whose upper bound has been increased, or lower bound decreased, respectively, with the given value as the new element. (We assume that the low and high bounds of an array are part of the information associated with the array.)

2.1 The Source Language

Programs will be written in a suitable subset of VAL [5]. This language is, for the purposes of this thesis, a rather ordinary and typical language. It is applicative, it has the desired array operations, and it has suitable control structures. The properties of VAL that set it apart from other languages such as Id [9] or pure LISP [42] relate mostly to syntax and type checking, which are irrelevant to our concerns here. VAL is rather

simple-minded in its approach to functions, in that functions are not treated as "first class objects".[1] This makes it suitable for translation into static data flow graphs. Languages that treat functions more generally, such as Id, LISP, FP [11], or KRC [45], may be more difficult to translate for the execution efficiency that we envision for VAL. (This is not to say that they will never be implemented as efficiently as VAL, but we believe that further developments in architecture and compiler design will be required.) The restrictions imposed by using VAL and static data flow graphs do not impose any undue hardship for many application areas.

The principal array operators of VAL are `SELECT` and `APPEND`, described previously. When building arrays with **append**, one often starts with the array "`array_empty`". Because VAL is applicative, this is a *constant*. The semantics of VAL provides detailed rules governing the behavior of `SELECT`, `APPEND`, and `array_empty`, particularly regarding how array bounds "stretch" when an `APPEND` writes into a position out of the existing bounds. These rules will be ignored in this thesis -- they are not important for the kinds of regular array creation we will be dealing with. We will just assume that application of a series of `APPEND` operations to `array_empty` will always do the right thing. We will not even assume that array bounds are part of the data carried by the token. The difference between the assumptions being made and the actual semantics of VAL (or any similar language), and the difference between those assumptions and the actual behavior of a target data flow computer, are details that may be left to the design of a compiler.

Other array operations have been proposed for the VAL language, but they are not important. Most of them perform "housekeeping" operations involving the array bounds.

---

1. This means that function names and definitions may not be passed as parameters, bound to variables or arrays, or otherwise manipulated. The only thing that one may do with a function is invoke it, and the only thing that one may invoke is a function name which is permanently bound at compile time to a function definition that the compiler can find.

The notation for iterations in VAL is shown in the following example:

```
for J, K := N, 1                    % J and K are loop variables
do  if J = 0 then K                 % exit and return this value
    else iter J, K := J-1, J*K enditer
    endif
endfor
```

The body of a **for** block may evaluate (depending on the values of test expressions involving the loop variables) to an **iter** clause. If so, the loop variables are bound to the new values as indicated, and the body is evaluated again. Otherwise, evaluation of the **for** block is complete. As noted in [4], this is typical of the way iterations must be expressed in an applicative language.

The VAL language also has a **forall** block exemplified by

```
forall J in [LO, HI]                % "vector sum" of A and B
construct A[J]+B[J]
endall
```

In this thesis, the **forall** block will be considered as nothing but a "sugaring" (short notation) for the appropriate **for** block. All considerations of iterative program structures and their transformations will be made in terms of **for** blocks, and the extension to **forall** blocks will be implicit. For example, the above **forall** block can be considered a sugaring for[1]

```
for J, A := LO, array_empty
do  if J > HI then A
    else iter J, A := J+1, A[J: A[J]+B[J]] enditer
    endif
endfor
```

---

1. This ignores a minor difference in the way the VAL semantics treats pathological values of the array bounds.

One may wonder why we artificially sequentialize the **forall** block. The object of the game is to uncover parallelism, and the **forall** block is designed to show the compiler where parallelism lies, so destruction of this information seems counterproductive. The reason we do this is that the correct translation of the **forall** block is the same as the correct translation of the equivalent **for** block, as will become clear later when "loop unfolding" is introduced. When a **forall** block is converted into an iteration and a loop unfolding of `N` is performed, the resulting code simultaneously computes `N` consecutive instances of the **forall**, and then moves on to the next `N`, and so on. If `N` is as large as the machine size will support, this is the best way to translate general **forall**'s.

If the **forall** limits are known to the compiler and are reasonably small, converting it to an iteration loop is still the correct thing to do. Complete unfolding of the resulting iteration will generate the "intuitively obvious" parallel **forall** code.

Whether an optimizing compiler actually converts **forall**'s into iterations, or just acts as though it had, is a minor implementation detail.

2.2 Compile-time Parameters

It will soon become apparent that it is very important for an optimizing compiler to know the numerical values of certain parameters such as array sizes. Of course it is not appropriate for such parameters to be specified as manifest constants at the points where they are used. The Fourier transform procedure, for example, should be written with the array size as a symbolic parameter:

```
function DFT(ARG: array[complex] ; N: integer . . .
```

(The reader who is curious about what comes next should refer to Section 13.2.)

In any given run of a program, it will be known, for each invocation of this function and others like it, what the value of N will be. Typically, the programmer will want to specify the numerical value at the outermost level of the program, leaving it as a symbolic value everywhere else. It is for this reason that some languages such as FORTRAN77 [29] and ADA [6] have a "parameter" or "constant declaration" feature. Using "parameters", values that are required to be known prior to execution can be specified numerically only at the outermost level, and left in symbolic form elsewhere.

A nice property of functional languages is that there need be *no difference* between such a "parameter" and an ordinary "variable". There is no need to tell the compiler that, in one case, a parameter will be numerically specified at the start of the program, and in the other case, that a variable should have memory allocated to it. Since a "variable" in a functional language can't be changed once it is defined, there is no need to allocate memory for it.

A definition such as

```
let N := 2048
in  . . .
    . . .
    . . .
endlet
```

is completely equivalent to substituting 2048 for all free occurrences of N in the block. The same is true for definitions made by passing arguments to functions.

Therefore, it is reasonable that "compile-time parameters" will be specified, using normal mechanisms of the language, by a top-level function such as the following:

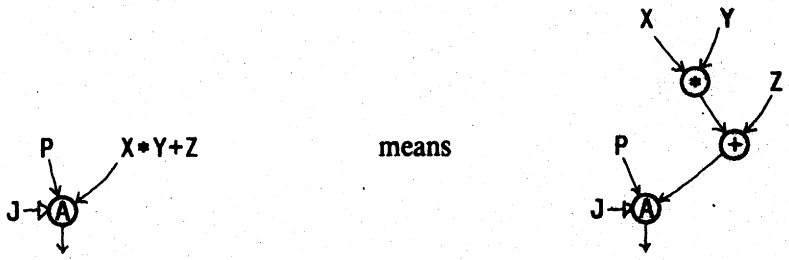
```
function TOP_LEVEL(DATA: array[real] returns array[real])
    SOLVE_NAVIER_STOKES_EQUATION(DATA, 2048)
endfun
```

2.3 The Graph Model

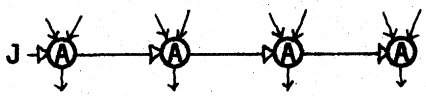
In this thesis data flow graph fragments will be exhibited in a very simplified notation designed to convey just the essential aspects of each graph. Ordinary connections among operators will be shown in the natural way, with arrows. The **select** and **append** nodes will be illustrated by nodes with abbreviated names S and A. The *index* argument to these nodes will be shown with an open arrow:

. Where possible, that arrow will be horizontal. The other argument(s) will be shown with plain arrows. The **append** operator takes two other arguments. The one on the left will be the incoming array; the new value will be on the right.

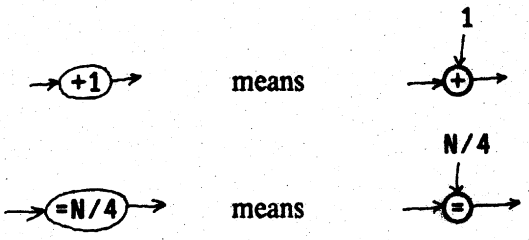
Where the actual graph structure would be more confusing than helpful, ordinary expressions will be written.



Where several nodes receive the same index, an arrow will pass through all of them:

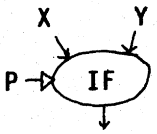


Arguments will occasionally be subsumed into nodes:



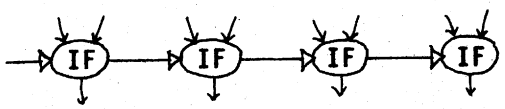
The ubiquitous constant `array_empty` will be written as "Λ".

The notation for conditionals is as follows: An IF node takes a boolean argument on the side, and data at the top. The left data argument is used if the boolean is true; otherwise the right argument is used.

`if P then X else Y endif` is represented as 

The actual representation of a conditional in a static data flow computer is quite different -- a "MERGE" operator plays the part of the IF, but "T" and "F" gates are placed at the top of the subgraphs computing X and Y to allow tokens to enter only the chosen subgraph.

Several IF's in a row can have a single boolean arrow going through them:



Iterations have the most complex notation. (This includes loops made from the VAL **forall** feature -- the latter is treated as an abbreviation for a sequential loop.) The body of the loop is surrounded by a dotted line passing through "LV" nodes at the top and a FOR node at the bottom. The "LV" nodes are sources of the Loop Variables for each cycle. The data shown going into them at the top (outside of the dotted line) are the initial values, computed only once. Just above the final FOR node is an ITERIF node controlling the loop. It is a variation of the IF node of ordinary conditionals.

The body of a VAL iteration loop is an if/then/else, one clause of which is of the form "iter <vars> := <newvalues> enditer", and the other of which defines a tuple of ordinary values.[1] That if/then/else will be an ITERIF node. The boolean control value will enter on the side in the usual way. The "data values" going into it will be "iterating arms" or "returning arms", drawn as boxes with "I" or "R" in them. An iterating arm has as many arrows going into it as there are loop variables. If that arm is selected,

---

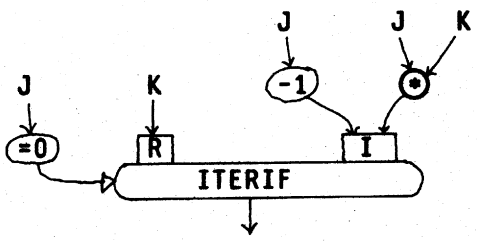
1. The VAL language actually allows many variations of this.

those values are used as the new loop variables, and another iteration cycle takes place. That is, the tokens are considered to be routed back to the respective "LV" nodes and to re-enter the loop body. A returning arm has as many arrows going into it as there are values returned by the entire loop, which is the number of arrows emanating from the FOR node. If a returning arm is selected, those values are returned. That is, the tokens are considered to pass through the FOR node and out of the loop.

If J and K are the loop variables, then

```
if J = 0 then K
else iter J, K := J-1, J*K enditer
endif
```

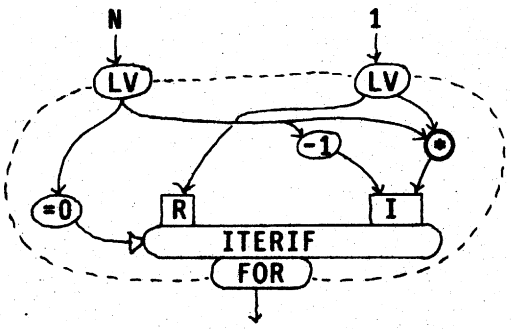
is represented by



and

```
for J, K := N, 1
do  if J = 0 then K
    else iter J, K := J-1, J*K enditer
    endif
endfor
```

is represented by



3. ITERATION UNFOLDING AND ARRAY INTERLACE

The principal program restructuring operation that we will perform is that of *loop unfolding*, also called *loop expansion*. This basically consists of writing out multiple copies of the loop body. In a data flow or other suitable parallel computer, these copies are expected to execute simultaneously, instead of the forced sequential execution implied by the original iteration.

Recall that, in an applicative language, the purpose of an iteration is to apply some transformation to a set of loop variables repeatedly, until some predicate is satisfied. We will be primarily interested in transformations that are performed a certain number of times, known in advance. These are characterized by having one of the loop variables be a counter, and having the predicate be a test of that counter and the transformation involve incrementing it.

The standard iteration might look like

```
%% compute f^N(INITVAL)
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

In a static data flow computer, and most other forms of computer, this suggests only one locus where f is computed, and hence strictly sequential evaluations of f, unless the situation can be improved upon.

A simple form of unfolding would be to rewrite this as

```
%% compute f^N(INITVAL), N is even
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(f(X)), I+2 enditer
    endif
endfor
```

This transformation is the standard model of *loop unfolding*. The function f inside the loop is duplicated (or

replicated some number of times) and the copies strung together in a series. The iteration then needs to cycle only some fraction of the number of times it otherwise would have. The intention is that, when the transformed program is loaded into the computer, the separate copies will execute simultaneously, except where data dependencies constrain them. For example, on a static data flow computer the operations of the different copies would be placed in different instruction cells that would fire independently. In an array processor the copies would be operated in lock-step by the different processors.

This loop unfolding is similar in concept to the "loop unrolling" sometimes used in conventional compilers [8, 14]. In conventional systems, however, the objectives are very different. There the aim is to reduce the number of times the exit test must be performed. There is rarely any assumption that the execution of the loop bodies will overlap at all.

The benefits of making separate copies of the transformation `f` can vary from virtually no overlap to complete overlap. On the surface, it might appear that, if we are computing `f(f(f(f(X))))`, the innermost function must complete before the next can begin, so one might as well leave the function in a loop. This is true for some functions, such as mathematical functions. If we need to compute `log(log(log(log(X))))`, we can't do the `log` operations in parallel. In a typical transformation, however, there is little or no *actual* data dependence among the evaluations of `f` because they refer to different parts of the set of loop variables. In particular, the different evaluations of `f` typically compute independent elements of a result array from different elements of an incoming array.

Any iteration made from a **forall** (and we expect that many iterations will be) necessarily has complete independence of the evaluations of `f`. Other iterations may not have complete independence, but will have near independence, with some sort of "critical path" encompassing only a small part of the computation.

3.1 Loop Unfolding

The standard unfolding is what was described previously, and will be called just *loop unfolding* or *loop expansion.* The *unfolding factor* is the number of copies of f that are expanded.

Here is unfolding factor 4:

```
%% compute f^N(INITVAL), N is multiple of 4
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(f(f(f(X)))), I+4 enditer
    endif
endfor
```

For simplicity, loop unfolding factors will always be powers of 2 in the examples. It is expected that this is the situation that will almost always arise in practice. If the unfolding factor (or the interlace factor, to be discussed later) is not a power of two, things become somewhat more difficult. This will be discussed in Section 4.6.

Since many iteration loops will come from **forall's**, it is useful to examine the loop unfolding of a **forall.**

```
%% compute Z[I] = g(I), 0 ≤ I < N, N is multiple of 4
Z := forall I in [0, N-1]
construct g(I)
endall
```

becomes

```
Z := for X, I := Λ, 0
do  if I=N then X
    else iter X, I := X[I: g(I)], I+1 enditer
    endif
endfor
```

which becomes, with unfolding of 4

```
Z := for X, I := Λ, 0
do  if I=N then X
    else iter X, I := X[I: g(I), g(I+1), g(I+2), g(I+3)], I+4 enditer
    endif
endfor
```

where X[I: α, β, γ, δ] is the VAL notation for an **append** that stores several values at consecutive indices. To get the full benefit of the unfolding, we assume that the 4 storage operations can be performed in parallel. Array interlace, described later, will accomplish this.

3.2 Initial Unfolding

Another transformation is called *initial unfolding*, and consists of taking some number of copies of f out of the loop altogether and doing them first. For example, we might do the first 3 cycles initially.

```
%% compute f^N(INITVAL), N ≥ 3
Z := for X, I := f(f(f(INITVAL))), 3
do  if I=N then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

3.3 Final Unfolding

The third transformation is called *final unfolding*: we execute the loop until some fixed number of cycles remain, and then do those cycles separately.

```
%% compute f^N(INITVAL), N ≥ 3
Z := for X, I := INITVAL, 0
do  if I=N-3 then f(f(f(X)))
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

We can combine all three types of unfolding in a single loop. The following has initial unfolding = 3, loop unfolding = 4, and final unfolding = 2.

```
%% compute f^N(INITVAL), N = 5+some multiple of 4
Z := for X, I := f(f(f(INITVAL))), 3
do  if I=N-2 then f(f(X))
    else iter X, I := f(f(f(f(X)))), I+4 enditer
    endif
endfor
```

If the number of cycles that a loop will undergo is known, and we perform an initial or final unfolding by that number, the loop will disappear completely. This is because the loop that remains after the unfolding will have its end test satisfied immediately. A popular example of this is a **forall** with fixed limits.

```
Z := forall I in [0, 3]
construct f(I)
endall
```

This loop will have 4 cycles, which is a small enough number that we can perform a complete initial unfolding, obtaining

```
Z:= [0: f(0), f(1), f(2), f(3)]
```

3.4 Unfolding When the Number of Cycles is Not Well-behaved

Each of the transformations, as shown in the comments of the programs above, requires that certain properties of **N** be satisfied. It is expected that, when an optimizing compiler makes any of these transformations, it will know whether the requirements are satisfied. This is because **N** has been specified as a "parameter", in the manner described in Section 2.2. It is still possible, with extra code, to handle the general case. Consider initial unfolding. If we did not know that $N \geq 3$, we could write

```
Z := if N=0 then INITVAL
elseif N=1 then f(INITVAL)
elseif N=2 then f(f(INITVAL))
else
    for X, I := f(f(f(INITVAL))), 3
    do  if I=N then X
        else iter X, I := f(X), I+1 enditer
        endif
    endfor
endif
```

A similar sort of thing could be done for final unfolding. There is considerable waste of space, but the waste of time becomes relatively insignificant for large **N**.

For loop unfolding, if **N** is not the required multiple of the unfolding factor, we do this

```
%% unfolding = 4, but can handle any value of N
Z := for X, I := INITVAL, 0
do  if I ≤ N-4 then iter X, I := f(f(f(f(X)))), I+4 enditer
    else
        if I=N then X
        elseif I=N-1 then f(X)
        elseif I=N-2 then f(f(X))
        else f(f(f(X)))          % I = N-3 here
        endif
    endif
endfor
```

The waste of time in this should be comparatively small for large **N**.

If **N** is not a multiple of the unfolding but its remainder modulo the unfolding is known, we can do better than the above example. If **N mod 4 = 3**, we can use an initial (or final) unfolding of 3. What is left over will be a multiple of 4.

To summarize, it is only when the number of cycles is not known (as opposed to known but not a multiple of the unfolding) that we need to put in extra conditionals. Wherever possible, the compiler should know the number of cycles of the important loops in a program.

3.5 Loops That Have No Obvious Index Variable

The foregoing has been defined in terms of an index variable that counts up to a fixed limit. This was done because most of the loops that are important in numerical programs (including all loops arising from **forall**'s) are of this type. However, loop unfolding, initial unfolding, and final unfolding can be performed on any loop at all.

A typical loop is

```
for X := INITVAL
do  if P(X) then R(X)
    else iter X := S(X) enditer
    endif
endfor
```

where `X` denotes all of the loop variables, `P` denotes the exit predicate, `R` denotes the value(s) to return, and `S` denotes the next-state function.

If we know that this will cycle at least 3 times, we can perform an initial unfolding of 3:

```
for X := S(S(S(INITVAL)))
do  if P(X) then R(X)
    else iter X := S(X) enditer
    endif
endfor
```

If we know that this will cycle a multiple of 4 times, we can perform a loop unfolding of 4:

```
for X := INITVAL
do  if P(X) then R(X)
    else iter X := S(S(S(S(X)))) enditer
    endif
endfor
```

We can similarly perform a final unfolding. These transformations depended on some knowledge of how many cycles would take place, in that they neglected to evaluate P(X) in certain places and just assumed P(X) was false. This is equivalent to the assumptions made previously about N when the loop was controlled by a counter. If we can't make assumptions about when P(X) will be true, then we have to put in extra code, just as in the loops controlled by a counter. For example, a completely safe initial unfolding would look like

```
if P(INITVAL) then R(INITVAL)
elseif P(S(INITVAL)) then R(S(INITVAL))
elseif P(S(S(INITVAL))) then R(S(S(INITVAL)))
else
    for X := S(S(S(INITVAL)))
    do  if P(X) then R(X)
        else iter X := S(X) enditer
        endif
    endfor
endif
```

3.6 Rescaling the Index Variable

When loop unfolding is performed on a loop controlled by an index variable, it is often useful (i.e. it saves excess arithmetic operations) to *rescale* the index variable, usually by dividing it by the amount of the unfolding.

If we have:

```
Z := for X, I := INITVAL, 0
do  if I=N then X
    else iter X, I := f(X), I+1 enditer
    endif
endfor
```

and we perform loop unfolding of 4:

```
Z := for X, I := INITVAL, 0
do   if I=N then X
     else iter X, I := f(f(f(f(X)))), I+4 enditer
     endif
endfor
```

we can introduce a new variable J = I/4 and rewrite this as

```
Z := for X, J := INITVAL, 0
do   if J=N/4 then X
     else iter X, J := f(f(f(f(X)))), J+1 enditer
     endif
endfor
```

This is particularly useful when interlaced arrays are involved.

Rescaling can also involve adding or subtracting a fixed offset. For example

```
Z := for X, I := INITVAL, 1
do   if I=97 then X
     else iter X, I := f(X), I+1 enditer
     endif
endfor
```

could expand to

```
Z := for X, I := INITVAL, 1
do   if I=97 then X
     else iter X, I := f(f(f(f(X)))), I+4 enditer
     endif
endfor
```

which could be rescaled by I = 4J+1, obtaining

```
Z := for X, J := INITVAL, 0
do   if J=24 then X
     else iter X, J := f(f(f(f(X)))), J+1 enditer
     endif
endfor
```

3.7 Interlace

Perhaps the most important transformation one can perform to improve the performance of numerical computations involving arrays is to spatially separate array operations that need to take place at nearly the same time. By having the array operations take place in different hardware units, the bottleneck is removed, and those hardware units can do their tasks in parallel. This is the basis for the memory interlace used in vector and pipeline computers, and the physically distinct processing units of array computers.

Arrays can be interlaced directly in the data flow graph. The amount of interlace for each array can be chosen independently for optimum performance. By interlacing arrays directly in the graph, the issues relating to interlace can be separated from issues of the computer hardware, though the optimum amount of interlace for a given program requires knowing how the machine behaves.

Interlace consists of dividing the array into α *slices*, where the slice to which an element belongs is determined by its index modulo α. α is the *interlace factor* (or the *interlace*). For simplicity it is always a power of 2.

If an array A is given an interlace factor of 4 (or a "4 way interlace") there are 4 slices, called A_0, A_1, A_2, and A_3. Those elements that were originally A[-8], A[-4], A[0], A[4] etc. are stored in A_0. Those elements that were A[-3], A[1], A[5] etc. are stored in A_1, and so on.
The mapping function for interlace factor α is

$$\underset{\text{(in the original array)}}{A[\alpha I+J]} = \underset{\text{(in the interlaced array)}}{A_J[I]}$$

$$\text{where } 0 \leq J < \alpha$$

or equivalently,

$$A[I] \text{ is stored in } A_{I \bmod \alpha}[\, \lfloor I/\alpha \rfloor \,]$$

where mod is defined to yield a nonnegative result and $\lfloor X \rfloor$ truncates toward $-\infty$.

3.8 Graphical Representation of Unfolding

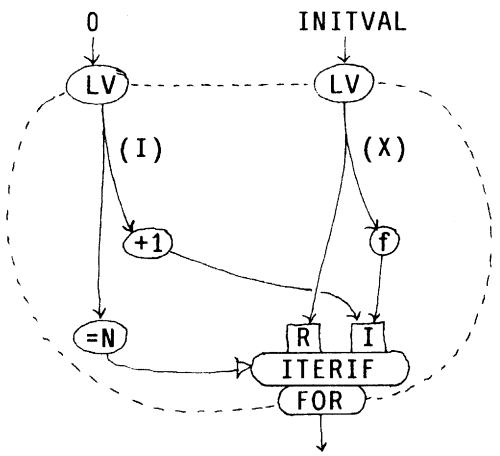
When iterations are drawn in graphical notation, loop unfolding is performed essentially by splicing together multiple copies of the part of the loop body that lies between the "LV" nodes and the values going into the "iter".

Referring back to the original example:

```
Z := for X, I := INITVAL, 0
do   if I=N then X
     else iter X, I := f(X), I+1 enditer
     endif
endfor
```

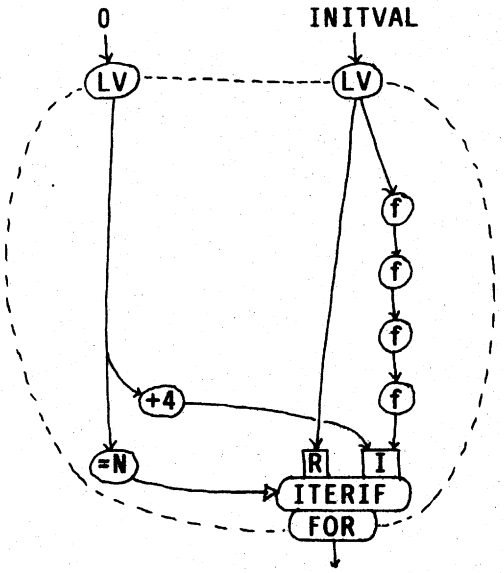
the graph is

Fig. 3.1



with 4-way unfolding, assuming we know that N is a multiple of 4:

Fig. 3.2



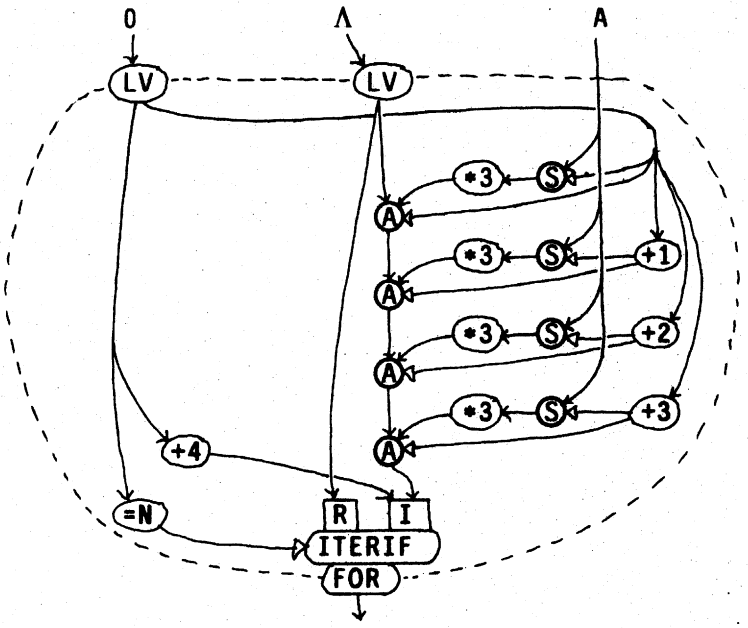
Of course, in a typical loop that arises in practice, the part that gets replicated is more than just the single node "f". It is a complex subgraph which, when replicated, overwhelms the rest of the loop. Furthermore, the replicated copies are often intertwined with each other quite intricately, as subsequent chapters will show.

When unfolding is performed on a loop making references to arrays, and the arrays are interlaced, the compiler can use its information about the array index in each unfolded instance to optimize the array references. Certain references in certain loop instances will, by virtue of the known array index modulo the interlace factor, refer only to certain array slices. The resulting graph typically looks something like this:

```
B := forall I in [0, N-1] construct A[I]*3 endall
```

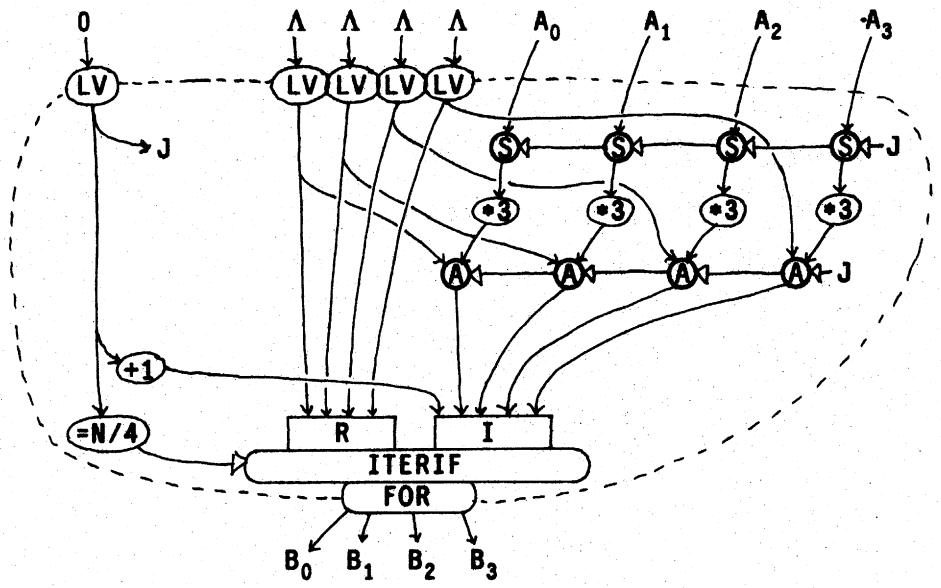
with unfolding 4 only:

Fig. 3.3



with interlace of 4 on both incoming and result arrays, and rescaling:

Fig. 3.4



The potential for parallelism when unfolding and interlace are used can be readily seen.

3.9 Nested Loops

When loops are nested one can choose the unfolding independently at each level.

Consider the following nested loop:

```
%% assume M and N are multiples of 4
for I, X := 0, INITVAL
do  if I = N then X
    else
        let NEWX := for J, Y := 0, X
            do  if J = M then g(Y, I)
                else iter J, Y := J+1, f(X, Y, I, J) enditer
                endif
            endfor ;
        in iter I, X := I+1, NEWX enditer
        endlet
    endif
endfor
```

The functions f and g symbolize the inner workings of the loops. Their dependence on the values of the counter variables is shown explicitly, which will demonstrate that loop unfolding requires a lot of careful bookkeeping.

If we perform a loop unfolding of 4 on the inner loop, we get the following, where some extra assignments have been made to keep the complexity manageable.

```
// this is the inner loop only
for J, Y := 0, X
do  if J = M then g(Y, I)
    else
        let NEWY1 := f(X, Y, I, J) ;
            NEWY2 := f(X, NEWY1, I, J+1) ;
            NEWY3 := f(X, NEWY2, I, J+2) ;
            NEWY4 := f(X, NEWY3, I, J+3) ;
        in iter J, Y := J+4, NEWY4 enditer
    endif
endfor
```

We can also perform a loop unfolding of 2 on the outer loop, obtaining

```
%% assume M and N are multiples of 4
for I, X := 0, INITVAL
do  if I = N then X
    else
        let NEWX1 := for J, Y := 0, X
            do  if J = M then g(Y, I)
                else
                    let NEWY1 := f(X, Y, I, J) ;
                        NEWY2 := f(X, NEWY1, I, J+1) ;
                        NEWY3 := f(X, NEWY2, I, J+2) ;
                        NEWY4 := f(X, NEWY3, I, J+3) ;
                    in iter J, Y := J+4, NEWY4 enditer
                endif
            endfor ;
            NEWX2 := for J, Y := 0, NEWX1
            do  if J = M then g(Y, I+1)
                else
                    let NEWY1 := f(NEWX1, Y, I+1, J) ;
                        NEWY2 := f(NEWX1, NEWY1, I+1, J+1) ;
                        NEWY3 := f(NEWX1, NEWY2, I+1, J+2) ;
                        NEWY4 := f(NEWX1, NEWY3, I+1, J+3) ;
                    in iter J, Y := J+4, NEWY4 enditer
                endif
            endfor ;
        in iter I, X := I+2, NEWX2 enditer
        endlet
    endif
endfor
```

This nested loop has "2*4" unfolding. (We could use different amounts of unfolding in the different instantiations of the inner loop arising from the outer unfolding, but there is rarely any reason to do so.) The total number of instantiations of the innermost loop body is the product of all of the unfoldings, 8 in this case. This product could be considered the "total" unfolding, since it is a measure of the total amount of space in the computer that the nested and unfolded loops will consume.

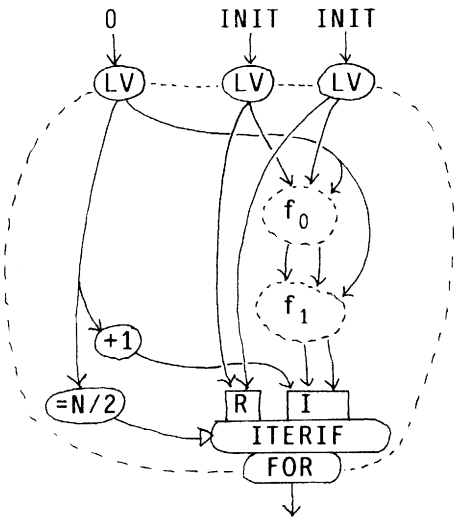
One might ask why we don't use all of the unfolding we have space for on the innermost loop, and none on the outer ones. On conventional computers, one typically optimizes the innermost loop most carefully. For example, variables manipulated by inner loops are given highest priority for using high-speed registers. We do not envision a data flow computer having high-speed registers that various parts of the computation compete for, so this consideration does not apply. When there are data dependencies from one iteration cycle to the next, preferentially unfolding the innermost loop provides a slight advantage in terms of overhead in loop control operators. However, when the loop cycles can be executed simultaneously, the loop cycles are correlated with elements of an array, and there are references to neighboring array elements in each unfolding, it is advantageous to unfold loops at all levels. The *shape* of the unfolding of nested loops refers to the relative amounts of unfolding at the various levels. It is the shape of the rectangular grid of unfolded loop cycles. This will be illustrated by various "2-dimensional" algorithms shown in Chapters 6 and 7.

3.10 Coalescing Control Structures

When there are nested loops, and unfolding is used, it very frequently occurs that the multiple instantiations of the inner loop can be executed simultaneously.

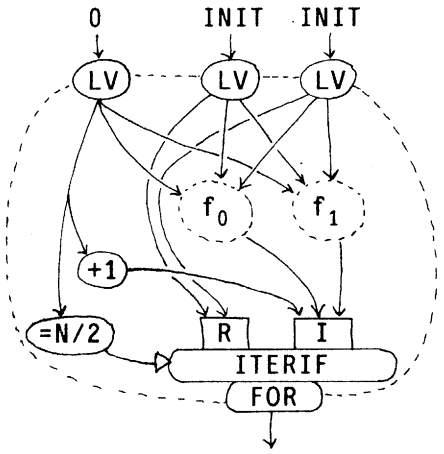
Suppose we use unfolding of two on some outer loop, obtaining something such as the following:

Fig. 3.5



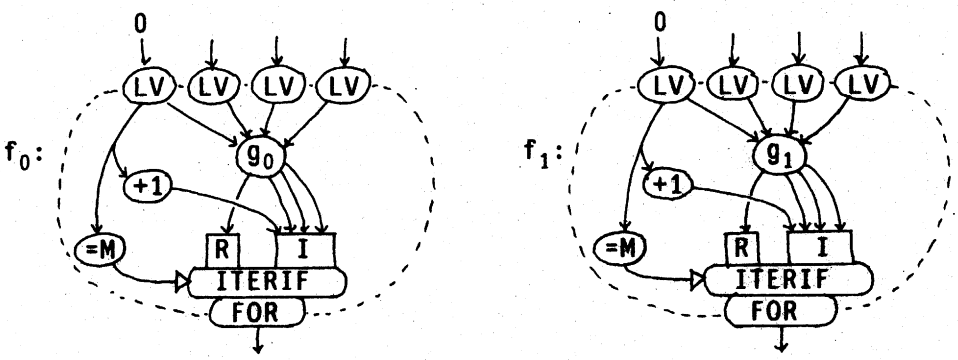
Now it may happen that, when the inner workings of the "f" subgraphs are worked out, there will in fact be no data dependency going from the first unfolding to the second. The graph might look like

Fig. 3.6



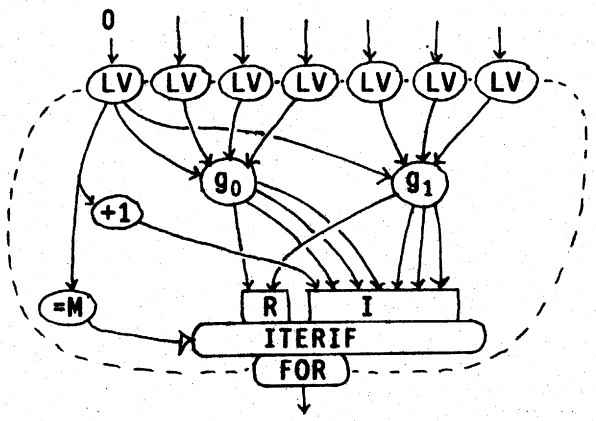
Now, if the occurrences of

Fig. 3.7



and the control structures are congruent, that is, the index variables and end tests are handled similarly, as is highly likely if the inner loops are just different instantiations of the same code arising from unfolding in the outer loop, they can be combined:

Fig. 3.8



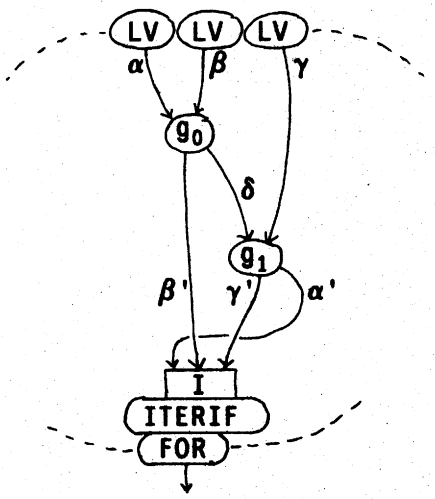
The benefit of this is not the fact that a few nodes have been saved, but that the subgraphs for g_0 and g_1, lying in the same loop, can share common subexpressions with each other. The significance of this will be discussed in Chapter 6.

One might think it unlikely that, when the outer loop is unfolded, the unfoldings would have no data dependencies among themselves, but in fact it occurs frequently. Almost all of the nested loops that will be examined in this thesis will have the control structures of the inner loops coalesced.

This coalescing of control structures is somewhat similar to the transformation known as "loop jamming" in conventional compilers [8, 14]. The transformation in conventional systems is performed in terms of "code motion" from one loop to another, principally for the purpose of economizing on loop control instructions. In the applicative case we have little concern for economy in the control computation, since it can be performed in parallel with the rest of the program. Instead, the objective is to prevent data values from having to be communicated among different loops, and thereby to allow common subexpressions to be combined.

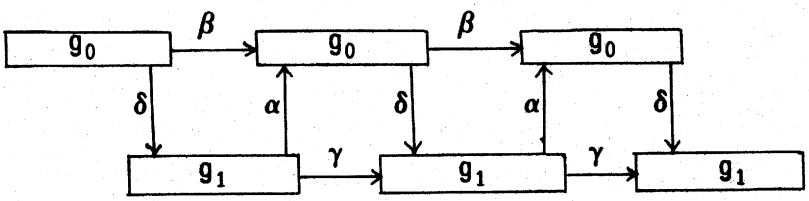
The fact that loop control structures are coalesced does not mean that they operate in lock-step. The recycling of loop variables from the "iterating arm" of an `ITERIF` back to the `LV` nodes should be able to occur at different times. The coalesced inner loop might look, in part, like

Fig. 3.9



Within each cycle, g_0 may have to be somewhat ahead of g_1. It should be possible for g_0's loop variable β to recycle and start the next cycle of g_0 while g_1 is still busy. The timing might look sort of like this, showing when each loop variable gets recycled (except δ, which is just passed from g_0 to g_1).

Fig. 3.10



This "decoupling" phenomenon will show up in the "wavefront transformation" in Section 10.1.

3.11 Interlace of Multidimensional Arrays

The model of multidimensional arrays we use is the "vector of vectors" model. Every genuine array in the language is actually a one-dimensional structure, whose element types can be anything. If we choose arrays of reals as the element type, we get an array of arrays of reals, which is equivalent to a two-dimensional array. If `X` is such an array, `X[I][J]` selects an element from it. `X[I]` selects the Ith one of the "second level" arrays, and `(X[I])[J]` selects the final element. This is the model used by some languages, such as VAL; CLU, and C, and one of the models available (indirectly) in most languages with a general type system, such as PASCAL.

The model used by some other languages, notably FORTRAN and PL/I, is the "flat" model. The lowest level rows are concatenated end-to-end to make one-dimensional arrays. If there are more levels, these results are then concatenated, and so on. The result is a one-dimensional array with a mapping function for references. For example, a [1-10]x[1-100]x[1-20] array (as might be declared in FORTRAN by "`DIMENSION A(10, 100, 20)`") is actually implemented as a single array running from 1 to 20000, and a reference to `A(I, J, K)` is treated as `A(2000*I + 20*J + K-2020)`.

From an implementation standpoint, flattening is a good idea. It makes all array selections possible with one memory reference (and array memory references may be expensive in data flow computers). It also means that only scalar data are actually stored in the array memory -- not array descriptors. This is extremely important in an applicative data flow system, because there is large overhead associated with reference

accounting as array descriptors are manipulated [2, 3]. We want to cut down on manipulation of array descriptors, as will be made clear in Chapter 15, so we refrain from storing them in other arrays.

However, premature flattening of arrays will preclude the use of interlace to the same advantage in the multidimensional case as in the one-dimensional case. Therefore, we will use the "vector of vectors" model. When considering interlace and various program optimizations, arrays will be treated as true hierarchical objects. They will not be flattened. Interlacing will be performed on each dimension independently, yielding a multidimensional collection of slices. After loop unfolding and array interlace are performed, *each slice* will be flattened. The result will be far superior in general to what would be obtained by initially flattening the array and then interlacing the result.

This multidimensional interleaving and flattening of individual slices, with each array considered separately, constitutes a radical restructuring of the arrays as described in the source program. When a complicated program is running on a data flow computer, the array elements will be scattered throughout the machine in a manner that is far from obvious. It is the applicative nature of the programming system that makes it feasible for a compiler to do this, because the compiler knows where every element of every array comes from.

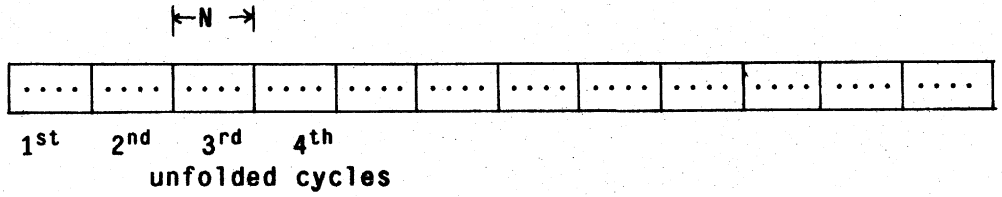
Incidentally, the flattening of arrays in FORTRAN is mandated by the semantics. If `A` is a 10x10 array, it is legal to send its first row to a procedure `F` with the call `F(A(1))`, assuming `F` expects a one-dimensional, ten element argument. The second row may be sent with the call `F(A(11))`, and the Kth with `F(A(10*K+1))`. This works because the flattening mechanism is a fixed part of the language. It is therefore extremely difficult to write a FORTRAN compiler that uses some other array organization. This is a case of overspecification of the semantics of a programming language, an all too common occurrence. Note that, with the "vector of vectors" model, it is equally easy to send the Kth row of a two-dimensional array to some procedure. A data flow computer using the "vector of vectors" model has the freedom to interlace, flatten, or

otherwise manipulate the array in many ways while remaining totally faithful to the language semantics.

3.12 Geometrical Representation of Unfolding

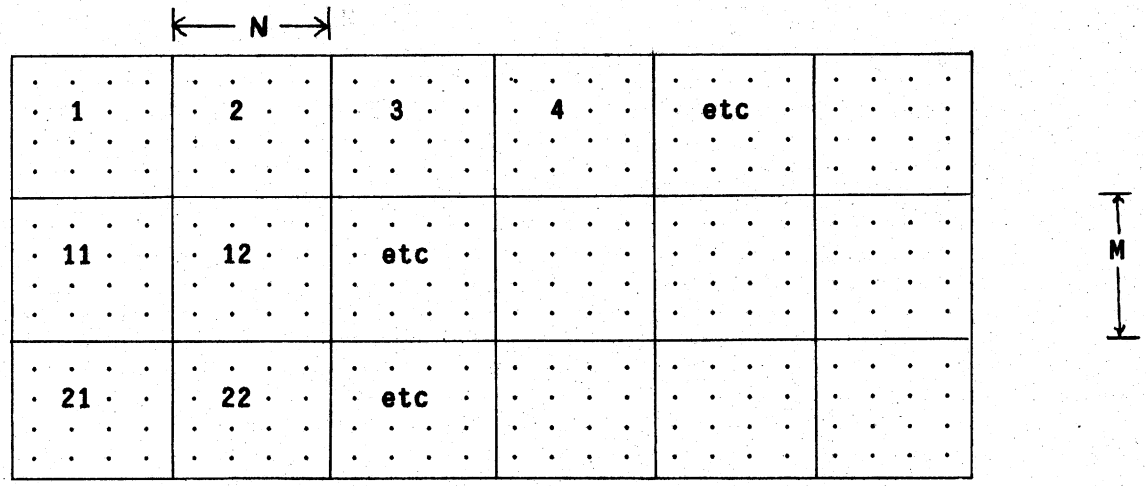
If we consider each cycle of a single loop to be a point, and perform N-way unfolding on that loop, then each unfolded cycle will process a group of N adjacent points. The groups will be processed in sequence.

Fig. 3.11



If we consider a double nested loop to be a plane array of points, the programmer's intention is that the points be processed sequentially in a "raster scan" order. Individual points in each row will be processed from left to right, and the rows will be processed from top to bottom. If we perform an M*N unfolding and coalesce the control structures of the inner loops, we will still have a double nested loop. Each cycle of that loop will process an M*N rectangle of points. The order of the cycles of the unfolded loop is such that the rectangles will be processed once again in a raster scan order, but on a coarser scale.

Fig. 3.12



For K-fold nested loops, each unfolded cycle will process a K-dimensional box in the K-dimensional space, and the boxes will be processed in the usual order.

This is the computational process we envision for a high-performance applicative computer. It will processes the data space in large localized chunks. It will work best on algorithms with locality of data dependence.

4. MATCHING INTERLACE AND UNFOLDING

This chapter will examine the relationship among the loop unfolding factor, the array interlace factor, and the parameters of the program for loops that deal with arrays. As will be seen, a careful match among these quantities is crucial to efficient translation. One of the major tasks of an optimizing compiler is to examine the parameters of the loops and choose the appropriate amounts of interlace for the arrays and unfolding for the loops. When this is done correctly, the resultant code structures will permit extremely rapid, bottleneck-free distributed computation.

The discussion will be primarily aimed at **select**'s. **Append**'s will also be considered, but in a simple-minded way, as though **append** were a slight variation of **select**. This assumption will suffice for the purposes of the present chapter, though there are other aspects of **append** that will be discussed in Chapter 8.

The situation that can be exploited well, in terms of parallel or overlapped accesses to interlaced arrays, is an iteration (or equivalent **forall**) whose array accesses are affine. The following example shows such an iteration. Any iteration in which all array accesses are affine is of this basic form.

```
for T, U := init_arrayT, init_arrayU ;
    <other initializations>
    J := init_index ;
do  if J=exit_index then T, U, <other values>
    else
        let V1 := ... expression ... A[J*factorA+offsetA] ... ;
            V2 := ... expression ... B[J*factorB+offsetB] ... ;
            ...
            new_Tvalue := ... expression ...
            new_Uvalue := ... expression ...
        in  iter
                T := T[J*factorT+offsetT: new_Tvalue] ;
                U := U[J*factorU+offsetU: new_Uvalue] ;
                <other reassignments>
                J := J+increment ;
            enditer
        endlet
    endif
endfor
```

Assume for now that all of the relevant parameters ("`init_index`", "`factorA`", "`offsetA`", "`increment`", etc.) are known to the compiler. If any are not known at compile time, it may still be possible to achieve extremely high performance, but it may require more complex code and more work on the part of the compiler. This will be discussed below.

The interesting activities in this loop are the array reads (of `A` and `B`) and writes (to `T` and `U`). These occur at regular intervals across the respective arrays. By examining the "factor" and "offset" of each array reference, and the initial index and increment, we can determine at just what index the array references occur. The *reference interval* is the distance along the array of the references in consecutive loop cycles. For each array it is just the "factor" for that array reference multiplied by the "increment" for the loop index variable. For example, the first reference to array `A` occurs at (`factorA*init_index+offsetA`), and the reference interval is (`factorA*increment`). It is possible for the same array to be referred to in more than

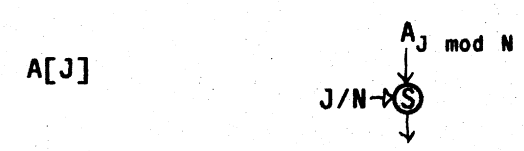
one place in the same loop with different reference intervals.

Assume for simplicity that all reference intervals (as well as all unfolding factors and interlace factors) are powers of 2.

4.1 Random Array Access and the "PERMUTE" Operator

If a completely random access needs to be made to an interlaced array, that is, we have no information in advance about the index, the **select** or **append** node must be able to operate on any of the slices. If the array has an **N**-way interlace, the following is needed:

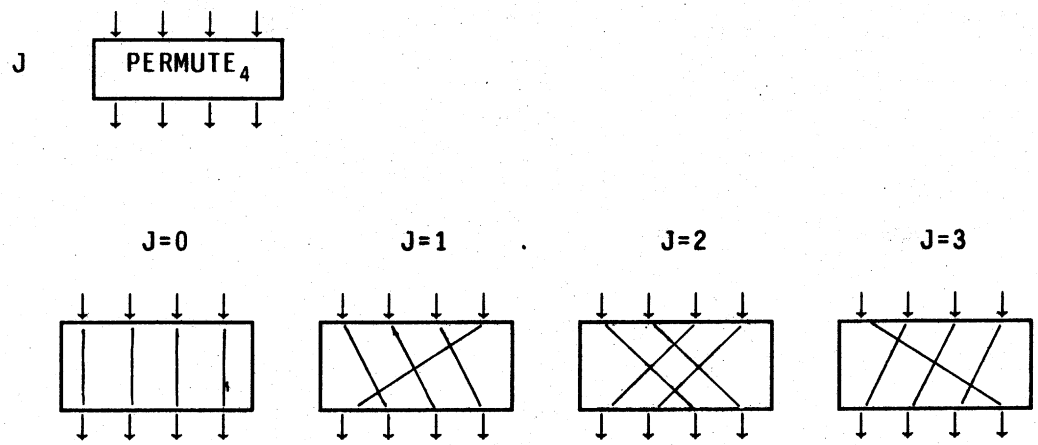
Fig. 4.1



where the division operator is assumed to truncate toward minus infinity.

To get the slice "$A_{J \bmod N}$", a **permute** operator will be used. There is a **permute** operator for every value of **N** that is a power of 2. It permutes **N** incoming tokens by "rotating" them right the amount of its index value.

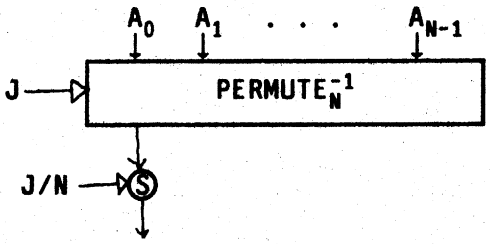
Fig. 4.2



The **permute**$^{-1}$ operator rotates left. We assume that the hardware has the necessary instruction types to do this efficiently, and that there will not be reference counting overhead if array tokens are permuted. (The reference counting problem for array tokens will be discussed in Chapter 15.) **Permute** operators are expensive, however. The large number of arguments indicates that, on any reasonable computer, a **permute** would have to be made out of a large number of actual instructions. One way to build it would be to use a network of 2*2 exchange instructions, built into a network such as an N-cube [43]. Each exchange instruction would be controlled by one bit of the binary representation of the index.

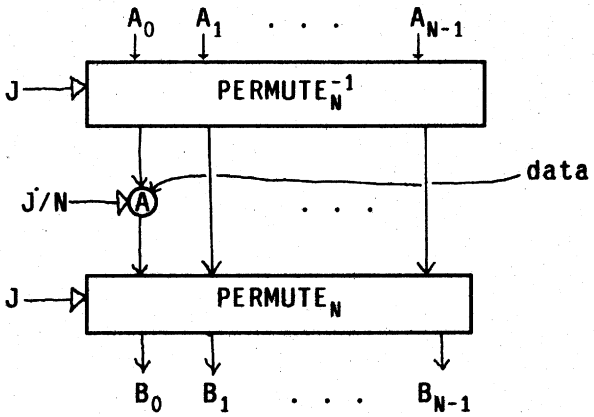
The **permute** operators will be assumed to use their index value modulo `N`. To get `A[J]`, use

Fig. 4.3



An **append** requires that the result be put back into the correct array slice, so it requires

Fig. 4.4



4.2 Affine Array Operations with Reference Interval Equal to One

It is clearly desirable to avoid the full "random" access, with its **permute** operator, presented in the previous section. In Chapter 3 it was shown that, if the desired index is the loop counter variable, the initial value the the counter variable is zero, and the interlace is equal to the unfolding, the situation is very nice. Each **select** or **append** operation in the expanded loop is known to refer to a specific slice, so no **permute** operations are ever needed. The index value going into the **select** or **append** is just the scaled counter variable.

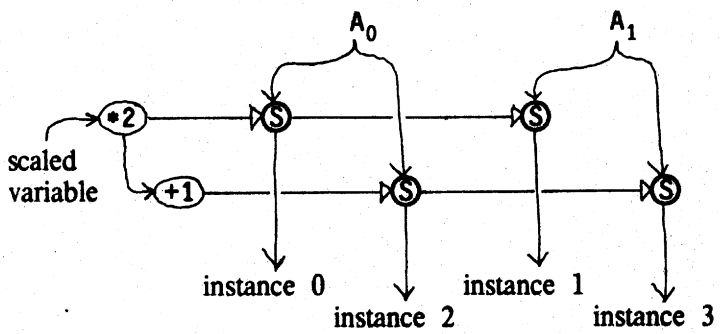
The first array reference does not necessarily take place at index zero, but at "`factor*init_index+offset`". The effect of this is to change the "phase" of the `N` array slices going into the `N` **select** or **append** operators. If this value is known at compilation time (or is known modulo `N`), a simple rearrangement of the incoming and outgoing array slices will take care of things. If the value is not known, some **permute** operators are needed to effect the rearrangement. These operators have to do their work only once, when the array slices enter the loop (and, in case of **append**, once again when they leave), so this is not too great a hardship. Handling the initial value of the loop counter variable and the array reference offset is therefore a relatively minor matter.

The important things to deal with are the reference interval, interlace, and unfolding. For now, let the reference interval be fixed at one.

If the unfolding is greater than the interlace, say twice as great, then two instantiations will have to share the same slice. One will use the slice at index equal to the scaled loop variable times two, and the other at that plus one.

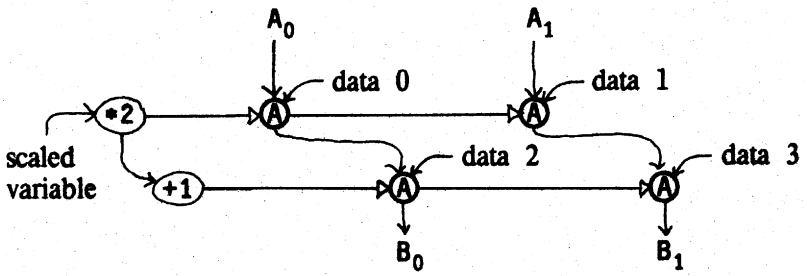
For unfolding = 4 and interlace = 2, we have

Fig. 4.5



Each slice goes into 2 **select** or **append** nodes (or more generally, into $\frac{\texttt{unfolding}}{\texttt{interlace}}$ of them). In the case of **append** nodes, the slice goes into them in series, as in

Fig. 4.6



The general rule for `unfolding > interlace` is that no **permute** operators are ever used, and that each array slice goes to $\frac{\texttt{unfolding}}{\texttt{interlace}}$ operators.[1] This situation isn't too bad, though the use of several **select**

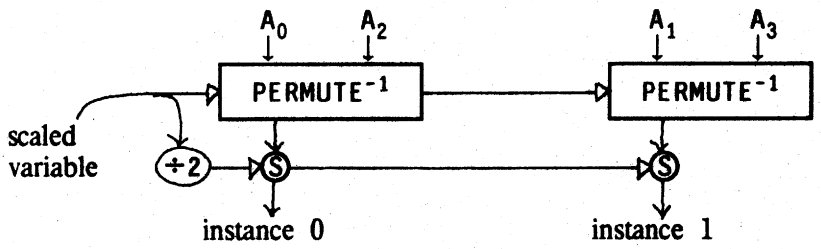
1. $\frac{\texttt{unfolding}}{\texttt{interlace}}$ is, of course, a power of 2.

or **append** operators on the same array slice may tend to create a bottleneck.

Now if the interlace is greater than the unfolding, say twice as great, then each instantiation will have to use two slices alternately, depending on whether the scaled variable is even or odd. The array index that it will present to the slice will be the scaled variable divided by two (perhaps with some offset added).

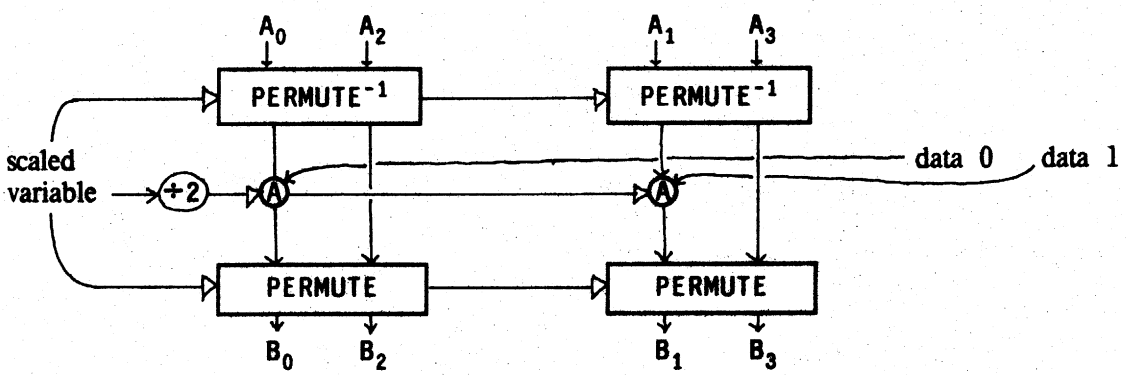
For unfolding = 2 and interlace = 4, we have

Fig. 4.7



Each pair of slices (or more generally, group of $\frac{\texttt{interlace}}{\texttt{unfolding}}$ of them) goes through an inverse permuter into one **select** or **append** node. In the case of **append** nodes, the result has to be permuted back. All the slices that do not go through the **append** node are passed directly from the **permute**$^{-1}$ to the **permute**.

Fig. 4.8



The general rule for `unfolding < interlace` is that each **select** or **append** operator in an instantiation needs a **permute**$^{-1}$ of size $\frac{\texttt{interlace}}{\texttt{unfolding}}$ to find the correct slice. The use of PERMUTE operators looks like something to be avoided, though it might be possible to build a computer in which they are not too expensive.

4.3 Reference Interval Not Equal to One

Suppose that the reference interval is not equal to one, but is some other number (a power of 2, of course) known at compilation time. The important criterion determining what will happen is the comparison of the following two values:

```
reference interval * unfolding  :  interlace
```

If the reference interval times the unfolding is smaller than the interlace, **permute** operators will be needed, with $\frac{\texttt{interlace}}{\texttt{reference interval*unfolding}}$ inputs. If larger, each array slice will go to, approximately, $\frac{\texttt{reference interval*unfolding}}{\texttt{interlace}}$ **select** or **append** nodes. (A correct version of this number will be given later.) Note that the previous results for reference interval equal to one agrees with this.

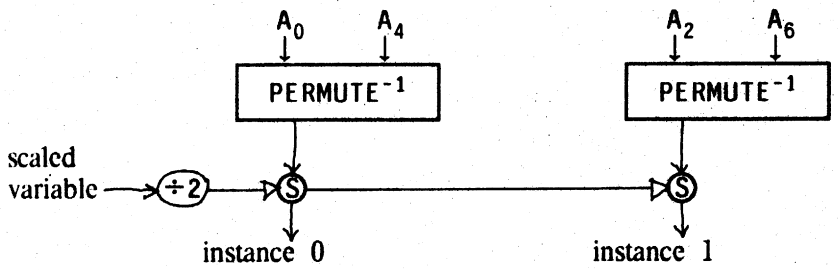
To analyze these cases, consider first the effect of a reference interval less than the interlace. Suppose the array A uses an interlace of 8, so that it is divided into slices A_0, A_1 ... A_7. If an iteration refers to it with a reference interval of two, say only even indices, then only A_0, A_2, A_4, and A_6 are actually used. Inside the loop, the array appears to have only four slices, that is, only a 4-way interlace. Note further that *every* element of each slice is referred to, instead of every other element. The array appears inside the loop to have a reference interval of one, not two. In general, the array appears to have an "effective interlace" equal to $\frac{\texttt{true interlace}}{\texttt{reference interval}}$. Since the loop appears to have a reference interval of one, the preceding section applies.

4.4 Reference Interval * Unfolding < Interlace --- Permute

In this case, the reference interval is clearly less than the interlace. The loop behaves as though the "effective interlace" were $\frac{\texttt{interlace}}{\texttt{reference interval}}$. This is greater than the unfolding, so there are **permute** operators with $\frac{\texttt{interlace}}{\texttt{reference interval*unfolding}}$ inputs.

Example: reference interval = 2, unfolding = 2, interlace = 8

Fig. 4.9

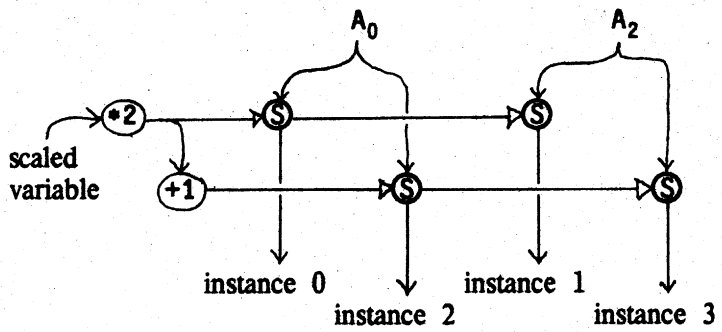


4.5 Reference Interval * Unfolding > Interlace --- Multiple Operators

In this case, we must compare the reference interval and the interlace. Suppose the reference interval is smaller. Then the loop behaves as though the reference interval were one and the "effective interlace" were $\frac{\texttt{interlace}}{\texttt{reference interval}}$. Since the latter quantity is less than the unfolding, each array slice that is used goes to $\frac{\texttt{reference interval*unfolding}}{\texttt{interlace}}$ **select** or **append** operators.

Example: reference interval = 2, unfolding = 4, interlace = 4

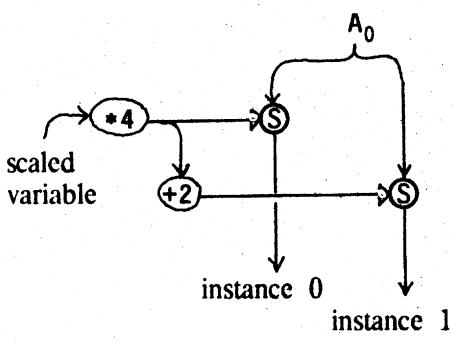
Fig. 4.10



Now suppose the reference interval is greater than the interlace. Then only one slice is actually used, and the "effective reference interval" for the index values going into the array operators is $\frac{\texttt{reference interval}}{\texttt{interlace}}$. Since there is only one slice, there are no **permute** operations. That slice goes to every **select** or **append** operator -- one per instantiation, so each slice that is used is used in a number of operators equal to the unfolding.

Example: reference interval = 4, unfolding = 2, interlace = 2

Fig. 4.11



To summarize:

reference interval * unfolding < interlace :

$\frac{\text{interlace}}{\text{reference interval}}$ slices are used; there is a permuter of $\frac{\text{interlace}}{\text{reference interval}*\text{unfolding}}$ inputs.

reference interval * unfolding > interlace :

$\left\lceil \frac{\text{interlace}}{\text{reference interval}} \right\rceil$ slices are used;

each one that is used goes to $\left\lceil \frac{\text{unfolding}}{\frac{\text{interlace}}{\text{reference interval}}} \right\rceil$ places.

The general rule of thumb is that the reference interval times the unfolding should be equal to the interlace, so that there are neither permutes nor array slices passing through many array operators. This rule may not always be exactly right in practice. Various factors involving the machine speed and the nature of the computation may dictate a ratio other than unity. However, the best ratio will not be grossly different from unity, and will stay reasonably constant as the machine size scales upward (that is, it will not go asymptotically to zero or infinity). If a unit ratio is not obtainable, it is better to have the reference interval times the unfolding be greater than the interlace, to avoid the necessity for permuters.

What happens if the reference interval is not known to the compiler? All is not lost. It simply becomes more difficult to take advantage of the interlace. At worst, all references become "random accesses". Of course, we want to avoid brute-force random accesses if at all possible. There are several useful cases in which they can be avoided. If the reference interval is known to lie within a small bound (typically it is either zero or one, depending on run-time circumstances), a small amount of messy object code can run quite efficiently. This is discussed in Chapter 9. Another case of interest is the one in which the reference interval, while unknown, is known to be a multiple of the interlace factor. In this case the interlace does not cause any inconvenience at all. An example of this is the inner loop of the "cyclic reduction algorithm", discussed at length later in this chapter. More generally, whenever the reference interval and the interlace factor have a known common divisor greater than one, we can do better than brute-force access.

4.6 When Things Are Not Powers of Two

The preceding treatment depended on the fact that if X and Y are both powers of the same prime, either X divides Y or Y divides X, depending on which number is smaller. This made it possible to say that perhaps permuters would be needed, or perhaps array slices would be used in multiple loop instances, but not both. If the interlace, unfolding, or reference interval are not powers of two (or some other prime) we might have the worst of both worlds -- slices used in multiple loop instances and permuters required in each instance. In particular, it is not possible to remove the need for permuters simply by increasing the unfolding, unless the factorization of the unfolding number is taken into consideration.

The "effective interlace", that is, the number of array slices that are actually used, is given by

$$\text{USE} = \frac{\text{interlace}}{\text{GCD(interlace, reference interval)}}$$

This is in all cases consistent with the formulas given previously, because of the equation

$$\frac{X}{\text{GCD}(X, Y)} = \left\lceil \frac{X}{Y} \right\rceil \quad \text{if } X \text{ and } Y \text{ are both powers of the same prime}$$

Each slice goes to $\frac{\texttt{unfolding}}{\texttt{GCD(unfolding, USE)}}$ loop instances

Each loop instance requires a permuter of $\frac{\texttt{USE}}{\texttt{GCD(USE, unfolding)}}$ inputs

It was shown previously that, given the interlace and the reference interval, one could remove the need for permuters simply by making the unfolding large enough -- it had to be at least $\frac{\texttt{interlace}}{\texttt{reference interval}}$. If the numbers are not all powers of two, things are slightly more difficult. We need to make `USE = GCD(USE, unfolding)`, which requires that the unfolding be an integral multiple of `USE`, that is,

$$\texttt{unfolding} = \text{a multiple of } \frac{\texttt{interlace}}{\texttt{GCD(interlace, reference interval)}}$$

4.7 Loops with Many Arrays

Loops may well have several arrays coming in or going out, and many aspects of the program interact when we choose the interlace for each array and the unfolding for each loop. In making these choices, a few obvious truths should be noted.

1. The unfolding in any loop must be one value, not different values in different parts of the loop. If a loop's use of one array suggests that an unfolding of four is optimal and its use of another array suggests eight, a consistent decision must be made. Typically the larger value will be used, since that will avoid the use of **permute** operators.

2. The interlace of any one array must be one value that is consistent for the array's producer and all of its consumers. When matching the interlace of an array to the various loops in which it is used, it is generally best to choose the smallest of the "optimal" values that the loops suggest, once again to avoid **permute** operators. (Actually, one could run an array through an "interlace converter" if a mismatch threatened to be serious.)

3. The reference interval for any array reference in a loop is a property of the program and not under the optimizer's control. The same array could be used in multiple places in the same loop with different reference intervals, that is, the reference interval is a property of each *reference*, not each array.

4.8 Choosing the Interlace and Unfolding

The method for making the various choices of array interlace and loop unfolding might be roughly as follows. It is proposed that, until really sophisticated techniques become available, a computer program to perform these transformations use human-generated "advice", either by querying the user when running or by reading a file.

We first determine, for each loop, what amount of loop unfolding to use. This depends on a compromise between the amount of space available in the machine for instruction cells (large unfolding in large loops consumes an enormous amount of space) and the benefit to be derived from the unfolding. Many factors interact in the unfolding decisions for the various loops in the program. We will discuss some of the issues later, but a truly "scientific" analysis of the problem has not been developed.

Having determined unfolding, we determine the interlace for each array roughly as follows. For each loop that creates or consumes an array, we know the reference interval, and the "ideal" interlace is the product of the reference interval and the unfolding. In many cases, these "ideal" interlaces won't all be the same. If this happens, we generally use the smallest figure. This avoids **permute** nodes but means that we may have array slices going to more than one **select** or **append** node.

The interaction between the loop unfolding and the array interlace does not go in one direction only: We may choose the unfolding of a loop to match the interlace of an array, instead of the other way around. This will happen when two or more loops are linked to each other through an array, and the speed of one of the loops makes it pointless to use as much unfolding in the other as we would like.

An optimizer that automatically chooses interlace and unfolding without human advice will have to be able to handle all of the "local" factors considered here, and "global" factors as well. When an array is created in one loop and used in one or more other loops, the ideal amounts of unfolding in the various loops interact through the array interlace. The individual choices must therefore be made in the presence of many constraints, some of which are likely to be inconsistent. The job of the optimizer is to make these choices, making compromises where necessary, in a way that is compatible with the overall size of the computer and will yield the best performance.

4.9 The Periodic Cyclic Reduction Algorithm -- Reduction Part, Inner Loop

The following program illustrates the considerations that go into choosing unfolding and interlace. It is the inner loop of the "reduction" part of the "periodic cyclic reduction algorithm" (also known as "even-odd reduction") for solving certain second order differential equations in one dimension with periodic boundary conditions. The equation is

$$X'' + AX = Q$$

That is, given the array Q and the scalar A, it finds the array X that solves the difference equation

$$X[J+1] + X[J-1] + (A-2)*X[J] = Q[J] \qquad 2 \leq J \leq SIZE+1$$

subject to the boundary conditions X[1] = X[SIZE+1] and X[2] = X[SIZE+2].

The complete algorithm is:

```
%%%
```

```
function PERI1(Q: areal; HX, C: real; SIZE: integer returns areal)
type areal=array[real] ;

let
    %% Scale incoming array for mesh size.
    FAC := HX*HX ;
    QA: areal := forall i in [2, SIZE+1] construct Q[i]*FAC endall ;

    %%%%%%%%%%%%%%
    %% REDUCTION %%
    %%%%%%%%%%%%%%

    LOGSIZE, FINA, FINQ: areal, FINB: areal :=
    for IH := 1 ;                       % IH runs from 1 through SIZE/2
        COUNT := 1 ;
        B: areal := array_empty[real] ;
        A := 2.0-C*FAC ;
        Q: areal := QA ;                % QA is the scaled incoming array
    do
        let
            %% Produce an array T with
            %% T[K] = Q[K-IH] + A*Q[K] + Q[mod(K, SIZE)+IH]
            %% for all K = 1 + a multiple of 2*IH

            NEWQ :=
            for K := 2*IH+1 ;   % IH is a power of 2
                T := Q ;        % Q, T bounds are 2 to SIZE+1 inclusive
                QK := Q[IH+1] ;
            do  if K > SIZE+1 then T
                else
                    let NQK := Q[mod(K, SIZE)+IH]
                    in  iter K, T, QK :=    % happens SIZE/(2*IH) times
                                K+2*IH,
                                T[K: QK + A*Q[K] + NQK],
                                NQK
                        enditer
                    endlet
                endif
            endfor ;

            NEWB: areal := array_addh(B, A) ;
        in
            if IH = SIZE/2 then
                COUNT, A*A-4.0, NEWQ, NEWB
                %% the A*A-4.0 makes the wrap-around work,
                %%    it will lose if C=0
            else
                iter IH, COUNT, B, A, Q :=
                    IH*2, COUNT+1, NEWB, A*A-2.0, NEWQ enditer
            endif
        endlet
    endfor ;

    %% Repair last element with scale factor magically computed above.
```

```
    QB: areal := FINQ[SIZE+1: FINQ[SIZE+1]/FINA] ;

in

    %%%%%%%%%%%%%%%%%
    %% SUBSTITUTION %%
    %%%%%%%%%%%%%%%%%

    for IH := SIZE/2 ;                  % IH runs from SIZE/2 to 1
        LOGI := LOGSIZE ;
        Q: areal := QB ;
    do  let A := FINB[LOGI] ;
            NEWQ :=
            for K := IH+1 ;
                QK := Q[SIZE+1] ;
                T := Q ;
            do  if K > SIZE+1 then T
                else
                    let NQK := Q[K+IH]
                    in
                        iter K, T, QK :=
                            K+2*IH,
                            T[K: (QK + Q[K] + NQK)/A],
                            NQK
                        enditer
                    endlet
                endif
            endfor

        in  if IH = 1 then NEWQ
            else iter IH, LOGI, Q := IH/2, LOGI-1, NEWQ enditer
            endif
        endlet
    endfor
endlet
endfun
```

For now, we will just look at the inner loop of the reduction part (the first half of the algorithm). Because there are two nested loops, there are, in effect, several nearly identical inner loops, that differ in the value of IH, which is a loop variable of the outer loop. IH is always a power of two, varying from 1 up to SIZE/2. SIZE (also a power of 2) is the size of the matrix being solved, and is very large (perhaps thousands of elements). Since the reference interval of the inner loop is IH*2, the analysis varies from one cycle of the outer loop to the next. Hence we consider each value of IH as defining a different inner loop. The array comes in as Q. The result is built as the loop variable T, and is yielded as the final result of the inner loop. The result of one inner loop goes into the next one as Q, so the interlace of Q and the result should be compatible from one inner loop to the next. Since the reference interval for the select references to Q and for

the **append** references to the result are both the same, the "ideal" interlaces are the same, so we might as well use the same interlace for everything. The bounds of Q and T are [2, SIZE+1] inclusive.

The inner loops are as follows:

```
for K := 2*IH+1 ;              % IH is a power of 2
    T := Q ;                   % Q, T bounds are 2 to SIZE+1 inclusive
    QK := Q[IH+1] ;
do  if K > SIZE+1 then T
    else
        let NQK := Q[mod(K, SIZE)+IH]
        in  iter K, T, QK :=       % happens SIZE/(2*IH) times
                K+2*IH,
                T[K: QK + A*Q[K] + NQK],
                NQK
            enditer
        endlet
    endif
endfor
```

The above code is not the way it really ought to be written. The loop variable QK was manually introduced to reduce the number of array references. This should be done by an optimizer, which will be discussed in Section 7.4.

There are two **select** references to Q within the loop, and one **append** reference to T. They all have reference interval 2*IH,[1] which is a variable with respect to the outer loop, but is, fortunately, a constant within any one of the inner loops. The number of inner loops is $\log_2(\text{SIZE})$.

---

1. One of the references has a **mod** operator in its index. The reference interval is still 2*IH, but an optimizer will have to work hard to deduce that. We propose that optimizers in fact be able to do so. The computation could be expressed with an if/then/else instead of **mod**, which would also require that an optimizer work hard.

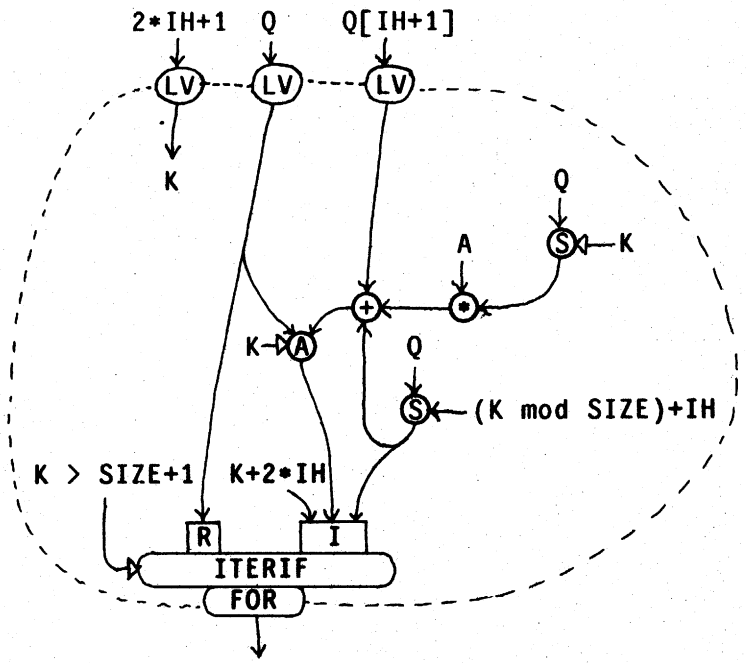
Suppose that, on the basis of some obscure analysis, we choose an interlace of 16. The "ideal" unfolding will vary from one loop to another because of the changing reference interval, as follows:

IH	reference interval	"ideal" unfolding	number of virtual cycles	number of expanded cycles
1	2	8	SIZE/2	SIZE/16
2	4	4	SIZE/4	SIZE/16
4	8	2	SIZE/8	SIZE/16
8	16	1	SIZE/16	SIZE/16
16	32	1	SIZE/32	SIZE/32
32	64	1	SIZE/64	SIZE/64
.	.	.	.	.
.	.	.	.	.
IH	2*IH	1	SIZE/(2*IH)	SIZE/(2*IH)
.	.	.	.	.
.	.	.	.	.
SIZE/2	SIZE	1	1	1

For the first 4 loops, the criterion `reference interval*unfolding = interlace` is met exactly. No **permute** operators are needed in any of the loops.

This is the basic graph:

Fig. 4.12



Those loops for which IH is 8 or more have the reference interval a multiple of the interlace factor. As noted previously, this means that the fact that we don't know the reference interval is no problem. Because of this, along with the fact that these loops have the same unfolding, they can all be treated as one loop with different parameters. The cases IH=1, 2, and 4 must be removed and treated as special cases. This is not surprising -- since they have different unfoldings they will have very different structures and will have to be compiled separately. The case IH=8 should also be treated differently if **permute**'s are to be avoided. The **select** required in "Q[mod(K, SIZE)+IH]" has reference interval a multiple of 16 whenever IH $\geq$ 8, but the offset (the "IH") is only 8 when IH is 8, which would require a **permute**. The initial value for QK, which is Q[IH+1], has a similar problem.

So, to avoid any permutes, only the cases with IH $\geq$ 16 will be made into a common loop. The arrays Q and T are interlaced 16 ways.

The counter variable K is rescaled by

$$K_{old} = 16 * K_{new} + 1$$

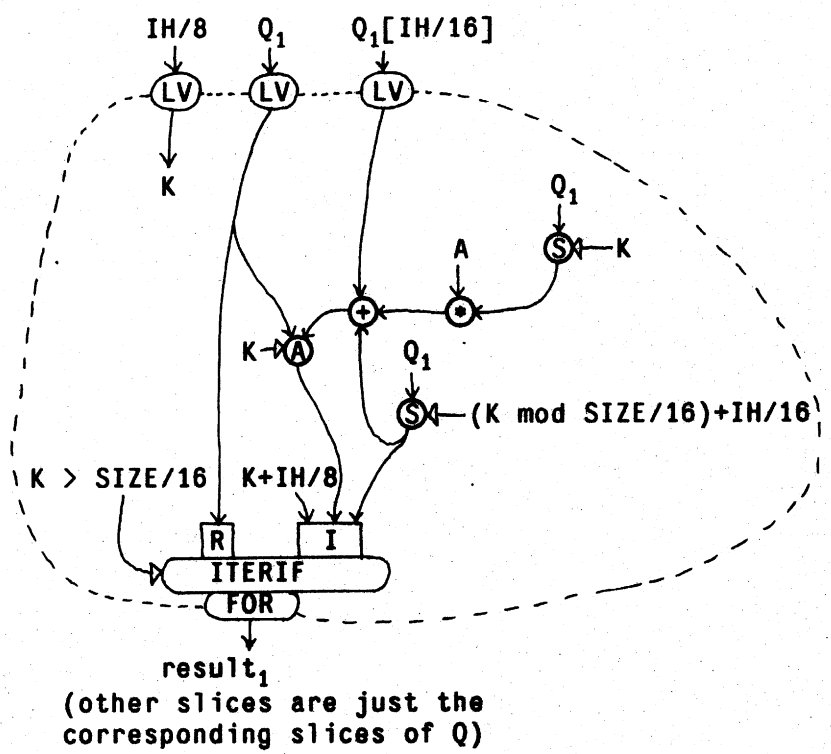
To properly compile this, the identity

$$mod(16*K+1,\ SIZE) = 16*mod(K,\ \frac{SIZE}{16})+1$$

is used. (Yes, optimizing compilers will need a fair amount of virtuosity in dealing with modular arithmetic.)

The resulting graph, for IH≥16, is

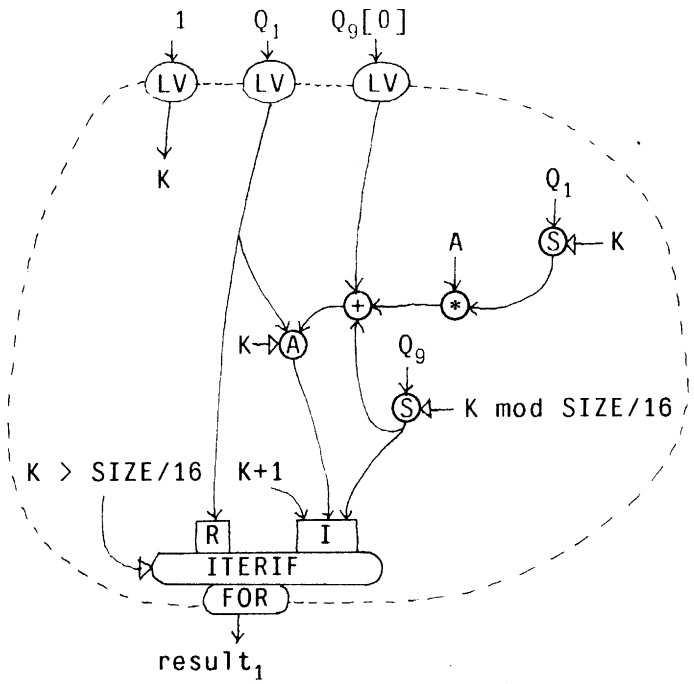
Fig. 4.13



In the remaining graphs we also use a few arithmetical tricks.

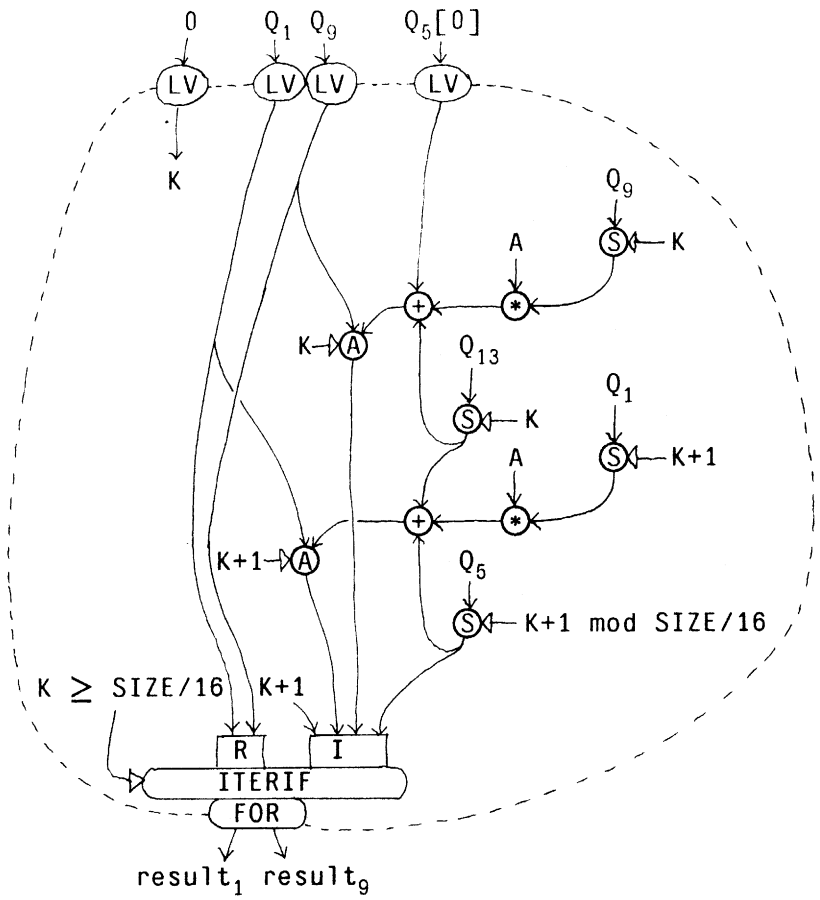
The graph for IH=8 is

Fig. 4.14



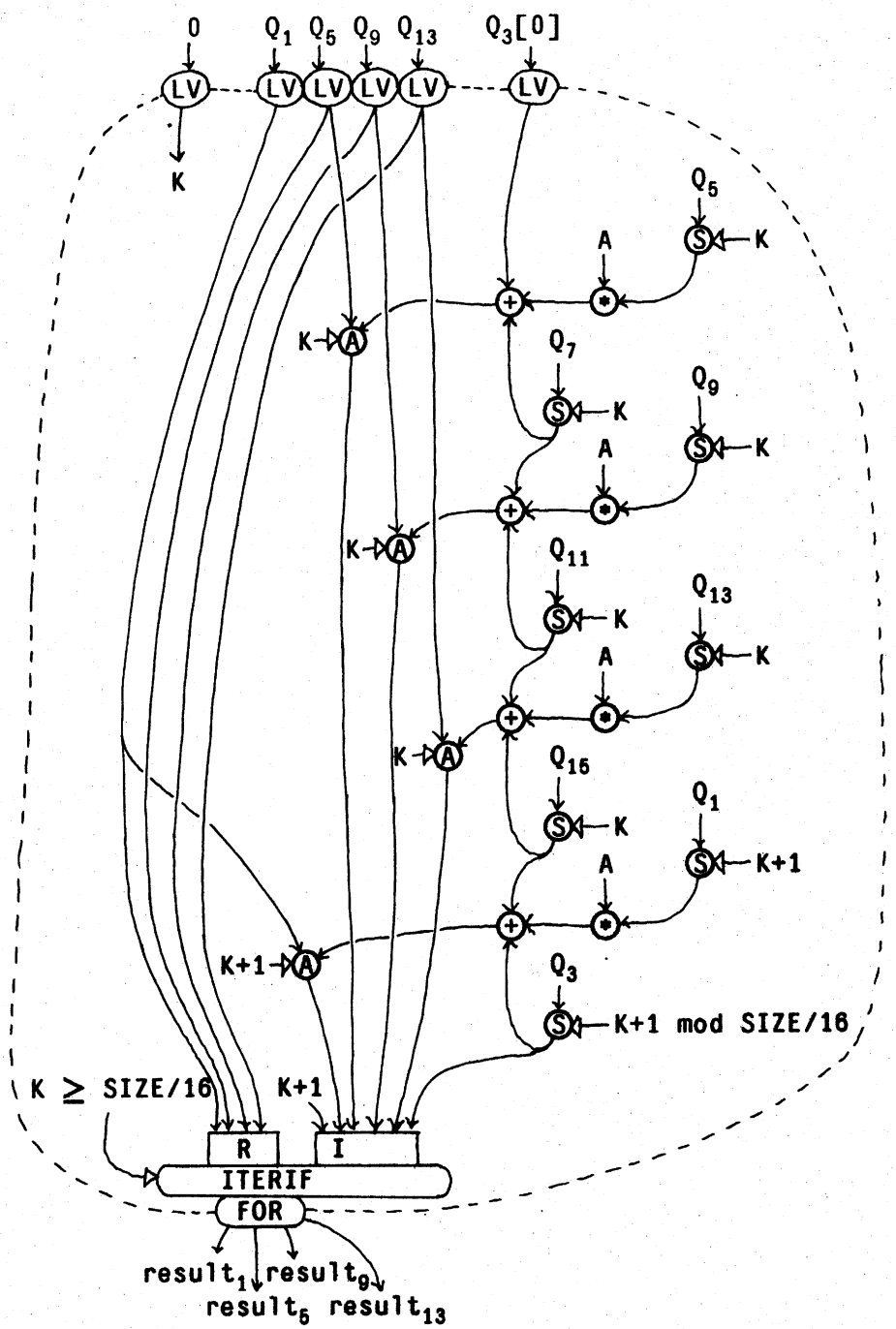
The graph for IH = 4 (unfolding = 2) is

Fig. 4.15



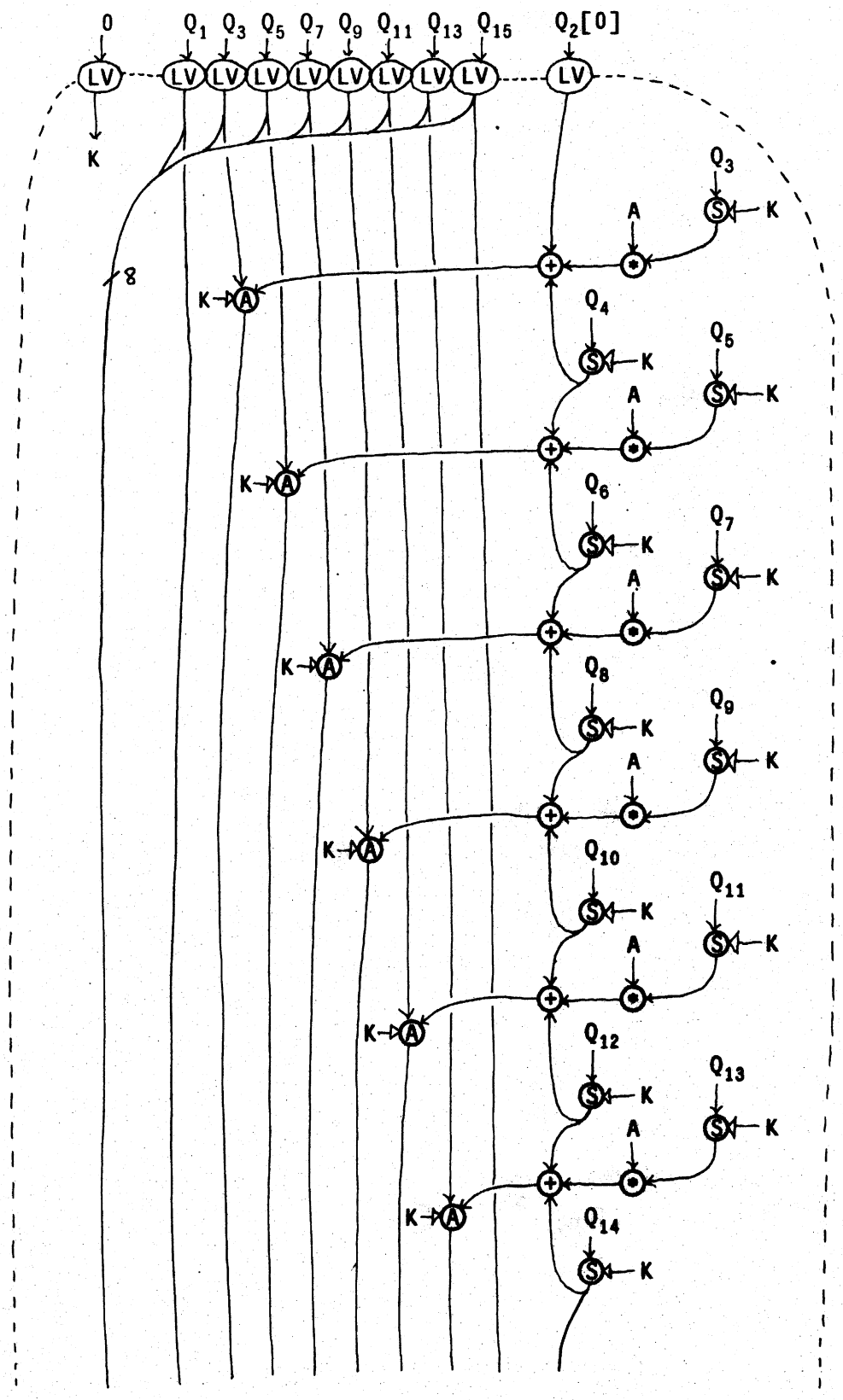
The graph for IH = 2 (unfolding = 4) is

Fig. 4.16



The graph for IH = 1 (unfolding = 8) is

Fig. 4.17

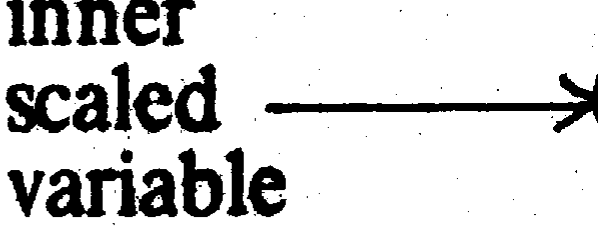


4.11 The Peri

The "subst

principal differen

to 1. The inner l


inner
scaled
variable

....

for

endf

endfor

An arrow is draw

The loop u

arrays themselve

unfolding, 2*2 in

```
forall J
....
fora

enda
endall
```

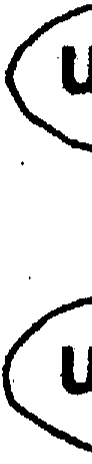
5.3 Array Ref

If two or m

on two or more l

inputs.

Example:



$$I+J -$$

$$\frac{I+J}{2} \_\_$$

If all of the (

u
u

```
forall I

.......

.... A[

.......
```

outside.

---

1. Remember fr

are flattened.

If the com

grid point requir

direction:

The initial values

be true during th

If the source prog

```
construc
      A[I]
      if I
      if I
endall
```

A2
S

A
S

Then, remember

unchanged, we n

$$W' =$$
$$X' =$$
$$Y' =$$
$$Z' =$$

time that would

recomputing then

The technic

shape. Compare

might be useful

consecutively, eve

1. If the machin
not known when
efficient in such
features the hard

end of

and analogously

Now this l

appears that the

descriptors into o

1. VAL does all
thing with them a
in any case.

Incidentall

operators so usef

If the index does

How to do this v

array slices back i

If we perform a

the resultant grap

looks very simple

nested loops of th

Now if we

No interlace less t

for huge permute

dependency inve

In the example ju

$J_1 \neq J_2$, we have

({12
({12

If this is so, the

Combining

computing eleme

previous one or n

```
B[I]
```

important transfo

rather than in the

---

1. This *particula*
interested in gene

endfor

There are, of cou

compiler would n

of the increment

We begin

"reference interv

flow graph for ac

interlace is greate

$$\frac{J+2}{8}$$

Because the refe

A[J+1], or A[J

require any of A

For example, if

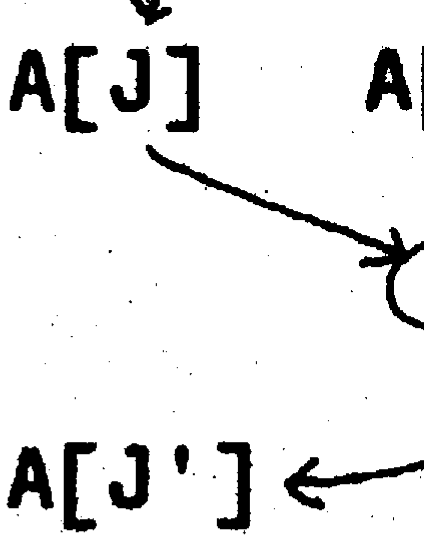
```
inte
refe
refe
unfo
```

```
inte
refe
refe
unfo
```

We can rescale th

Then the situati

interlace = 8 and


A[J]
A[J']

original contents

to substitute the

array. The desig

that inserts the n

interval. It, too,

```
endi
endfor
```

Of course, a data

dependencies are

If the unfolded l

previous cycle, th

corner. Looking

unprocessed part

The diagonal pa

computation. Th

conventional sup

wavefront will fo

To allow th

times, we can pla

11.1 Order of

Knowing t

flatten the array.

formed when tha

Knowledge

defined in terms

through only sele

making permuter

problems that co

values from one

in Chapter 15.

K-dimensional ar

flattened.

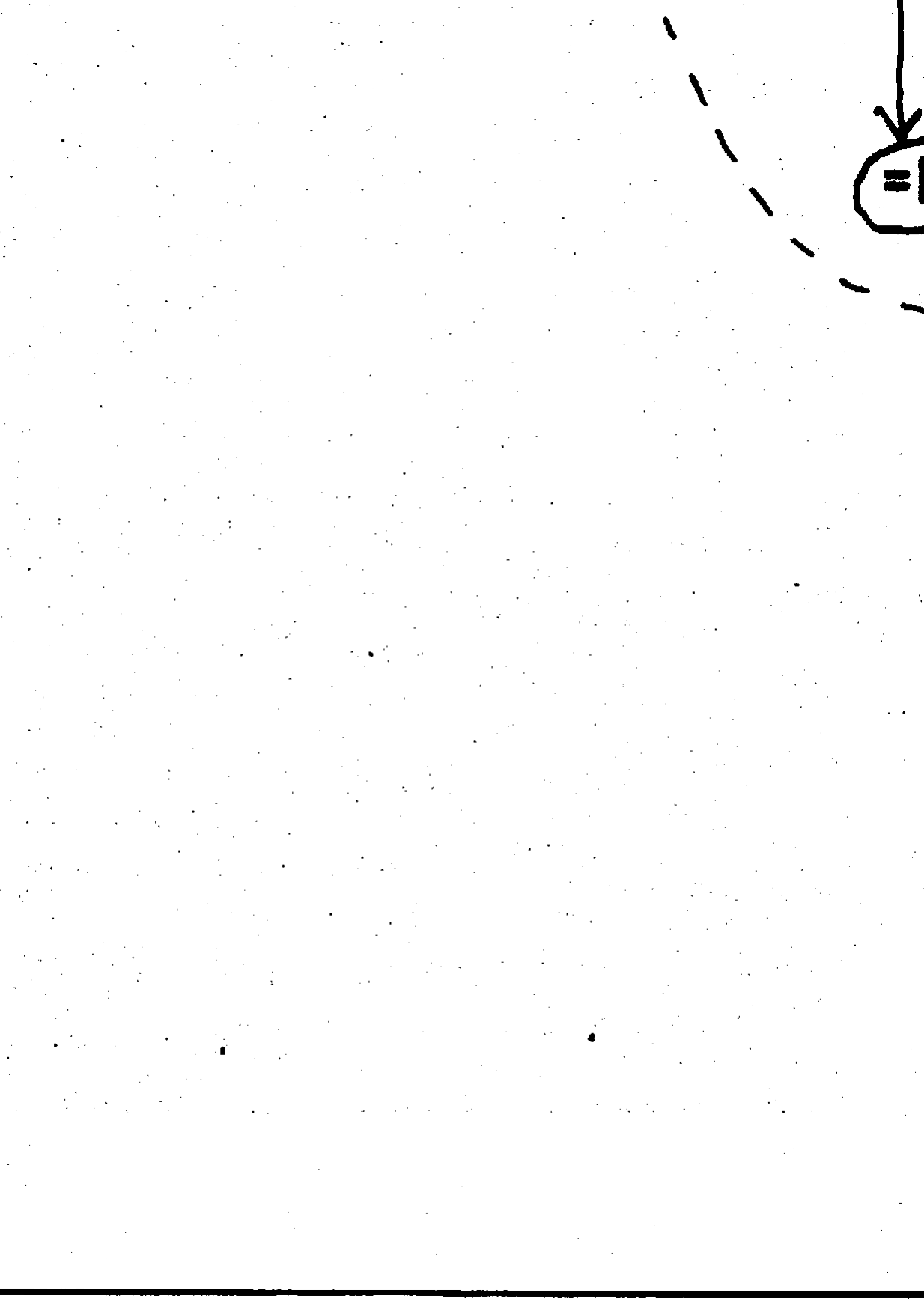
```
DO 99 L
. . . .
A(K, L,
```

That is, a FORT

level/array level

the language and

These graphs are



appending the sc

"flattened" into a

The graph on the

handle efficiently

correct state.

conditionals to se

```
%% I
J :=
.  .
.  .
.  .
```

the amount of co

of this. If we hav

12.2 Dividing

When seve

affine pieces, it i

reference require

for the reference

the loop index va

be badly behaved

will exist in those

be successfully u

Not all par

they are boundar

Section 4.9. The

extra computation

algorithm manipu

arithmetic -- the c

amounts of unfol

```
                    con

                    end
                in
                    ite
            endlet
        endif
    endfor
    endfun
```

Later outer cycles

A interlace

AL, AR inte

Unfolding

Unfolding

AL and AR

The significance

ahead of the rest

bulk of the graph

loop, in the same

hold the boolean

computation.

If these tw

unpacker goes th

run many cycles a

of the unpacker,

yet consumed.

that we pay for)

Chapter 11. The

speeds of the v

arithmetic excep

computer with m

Under wha

the final result of

first element is

dependence, the

control structures

made to run in ste

been formulated

technology. We

interest, and will

minimize the pass

It is reason

dynamically. Tha

is created (by the

mutual reference

is the method of

In a data fl

can be stored in

that arrays are fla

as well use the s

situ, making it

non-applicative a

condition

Assuming that th

end of a loop wh

care of reclaiming

operator does no

the same array to

a technique that i

probably the m

operators, are fa

in both time a

Burkowski's inst

does not take the

the constraint of

searched for near

be calculated fro

in various loops.

computer. If the

conventional syst

into future optim

Algebra of

[12] Barnes, G.
746-757.

[13] Burkowski,
of the Nin
101-111.

[14] Aho, A. V.,

[26] Dennis, J. ...
Second An...
Also, Comp...
Massachuse...

[27] Dijkstra, E...

[28] Flynn, M. J...
Computers

[41] Lawrie, D.
Computers

[42] McCarthy,
Communic

[43] Siegel, H, J

[44] Stone, H. S
4(Decembe

D

(A)

(B)

Scanning Agent Sig

Date Received:

Scanning Agent Sig