Code Generation for Dataflow Software Pipelining

by
Zaharias Paraskevas

A Thesis submitted in to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE
IN
COMPUTER SCIENCE

at the
McGILL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE
August 30, 1989

Abstract

The dataflow model of computation offers an attractive alternative for exploiting program parallelism through overlapped execution. The dataflow concept of computation overcomes the sequential nature of conventional machines, by allowing the availability of data to determine the execution sequence, rather than a sequential instruction counter. This thesis examines the dataflow software pipelining - an efficient code mapping strategy for array operations and iterative constructs on high performance dataflow architectures. It illustrates how the array operations in loop expressions can be effectively exploited on a highly pipelined static dataflow processor architecture based on the *argument fetching* data-driven principle. A code generator has been implemented which automatically generates code for a subset of SISAL The SISAL front-end and a hierarchical data dependence constructor have been used to construct the program graph, where the translation to machine code was applied. By execution of the compiled code on a compiler/architecture testbed, the efficiency of the dataflow software pipelining scheme and certain optimization methods are evaluated. A detailed analysis based on the simulation results is presented, and the impact of some architectural factors is discussed.

Abstract

Le modèle de évaluation *flot de donnees* offre une alternative intéressante de profiter du parallélisme dans les programmes par l'intermédiaire de chevauchement d'exécution La technique de flot de donnees surmonte la nature séquentielle de machines classiques en autorisant l'exécution des instructions selon la disponibilité des données, et non par leur ordonnancement séquentiel. Cette thèse présente une technique flot de données pai pipeline logiciel – une stratégie efficace pour la réalisation de tableaux et boucles sui des machines à flot de données de haute performance Elle explique comment réaliser de façon efficace les opérations répétitives sur tableaux sur des machines flot de données statiques basées sur les principes "apporte-argument" (*argument-fetch*) Un générateur de code pour un sous-ensemble du langage Sisal a été construit Nous construisons le graphe hierarchique du programme avec l'aide du compilateur Sisal et un transformateur de graphes. Ensuite, le code est généré directement du graphe On a exécuter le code sur un simulateur et mesuré l'efficacité de quelques méthodes de optimisation Une analyse des résultats est présentée, et l'impact de divers éléments architecturaux est évalué.

Acknowledgments

The completion of this thesis would not have been possible without the generous cooperation of my advisor, Guang R Gao I am grateful for his enthusiasm, confidence in my ability, and for guidance in the formulation and development of this research I want also to thank professor Jack B. Dennis for the consultations that he offers in our research group, and for his seminal contributions to the field of the static dataflow research

I enjoy the support from the members of our research team Advanced Computer Architecture and Program Structures Group at McGill University In particular, I wish to thank Rene Tio, for his work in the assembler and interpreter and the useful discussions we had during the design and implementation of the instruction set and the set of the performance metrics Many thanks to Robert Yates for his willingness to educate me in SISAL and IF1, for the valuable discussions in code generation and compiler issues, and for his kind offer to read and comment drafts of my work. It is a pleasure to acknowledge my officemate Won Hong, who was always willing to make the appropriate modifications to HDDG Many thanks to Ian Little, who provided a pictorial interactive tool for traversing and viewing graphs, that has been proved a valuable tool during the implementation of the code generator I would like to thank Luc Bouhanne for his invaluable help for the preparation of the final manuscript

To the Vranakis family, I am eternally indebted for their patience and selflessness in standing by my side, both the high and the low moments during the two years of living with them in Montreal. Finally, none of it would have been possible without the endless love and support of my parents. To them I dedicate this work, and I give my greatest thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advancements in technological growth cause the demand for more powerful computing to rise rapidly. Anticipating the continuation of this trend, research in the area of parallel computation seeks to achieve high performance by designing new architectures and structuring programs to exploit the parallelism inherent in many problems. Efficient exploitation of parallelism has been a challenge for designers of von Neumann parallel computers. However, the inflexibility of the program counter based sequential control mechanisms and the physical constraints that are imposed from the electronic technology severely limit the ability to exploit the inherent parallelism in algorithms which demand large quantities of computer power. The sequential nature of von Neumann architectures, where a sequential control mechanism is used to schedule instruction execution, creates the so-called von Neumann bottleneck [11]. Attempts to eliminate this bottleneck by overlapping instruction execution through the use of pipelining, vector processing and von Neumann based MIMD processors have yet to produce satisfiable solutions. In particular, to the two fundamental problems of von Neumann multiprocessing: the unpredictable memory latency and the cost associated with the synchronization among the processors [6,8]

A promising alternative to satisfy the appetite for computational cycles, is offered by the dataflow model of computation The dataflow concept of computation overcomes the sequential nature of conventional machines, by allowing the availability of data to determine the execution sequence, rather than a sequential instruction counter. The side-effect free property eliminates the need for explicit synchronization among the processing elements. An instruction can affect another instruction only if the output of the first instruction

is specified as input to the second instruction. Moreover, dataflow computer exploit the parallelism at the fine-grain level as opposed to von Neumann based multiprocessors.

Dataflow programming languages facilitate the writing of highly concurrent programs. The use of applicative programming paradigm contains implicit concurrency and allows much simpler analysis of programs. The user does not need to use explicit commands to identify concurrent activities. The *single assignment* rule and the *referential transparency* in the functional programming style, allow the parallelism to be fully and naturally expressed in the programs [1,11].

Translators for these types of languages represent the parallelism in a dataflow graph. Consisting of nodes and directed graphs, the data dependencies of operations in a program are explicitly shown. Each node identifies an operation and the arcs indicate the flow of values among the various operations. Given a graph representation of a program, dataflow architectures use a model of execution that exploits the concurrency implied in the graph. The general execution model executes each node when all of its inputs have been received. Such a model can fully support the concurrency expressible in the graph

In the last decade, several ideas and results have been drawn from many different research efforts in dataflow. The first dataflow model being proposed was the static dataflow model [15]. In the static dataflow model, static graphs are used to represent the computation. The major characteristic of this model is that no more than one value can be present on an arc at a time. To ensure this property and to eliminate the possibilities of overwriting, a node becomes enabled only if there are no values on any of its output arcs. Also in a static dataflow computer, all the storage required for program execution is allocated at compile-time.

Since Dennis at MIT first introduced the static dataflow approach, tremendous amounts of research efforts have gone into this area. Another dataflow research effort at MIT under Arvind, introduced the idea of dynamic dataflow model [5]. In the dynamic dataflow model of computation, every value as it moves along the dataflow graph, carries a label. The dynamic tagging approach to dataflow computation associates a unique tag with each instantiation of an instruction within a program. Thus, each data operand for an instruction carries a tag that specifies the particular instantiation for which the data is intended. The tag therefore distinguishes data operands for different instantiations of the same instruction. This allows multiple values to occupy an arc at the same time. This approach is the one realized by the MIT Tagged Token Dataflow Architecture (TTDA) [5].

Although the dynamic approach offers tremendous exploitation of parallelism in loops and in functions calls, it requires a very sophisticated hardware to match the parallelism that the abstract model can explore. Moreover, it does not answer to one of the major criticism against the dataflow model: the unnecessary data movements through the several units of the processing elements. This problem becomes even worse in the dynamic model, where every value must carry a tag which can be several bits long. This creates a considerable overhead in computing and moving the tag through the several units of the processing elements.

As a result of the active research in the static dataflow model, a highly pipelined static dataflow multiprocessor architecture has been proposed recently [18,30]. It is based on the principle of the *argument-fetching dataflow model.* The main characteristic of this model is that data never move through the units of a processing element. In this model data values and control signals are separated, and an instruction fetches its own arguments from a data memory just like in conventional processor architectures Signals which hold the sequencing information among the instructions, are the only entities that are moving around the circular structure of the processing element.

The principle of *dataflow software pipelining* appears as an efficient mapping scheme for exploiting the parallelism in loop constructs, and array operations. Software pipelining is performed on units of program text that define the major structure values involved in a computation. Applying this mapping scheme, the machine code is arranged such that successive computations can follow each other through one copy of the code.

This work focus on the efficiency that is gained by applying the dataflow software pipelining on the argument fetching architecture. Throughout this work, we try to investigate, both theoretically and experimentally, the degree that these two recent developments affect the performance of the executing programs in a static dataflow architecture. For this purpose an experimental compiler has been constructed which generates code for the argument fetching architecture. The principle of dataflow software pipelining has been encapsulated in the loops and conditional expressions, allowing us to identify its effectiveness by running various programs. Three are the main objectives of this thesis:

1. to construct an experimental code generator for the argument fetching architecture, which will be used as the ground work for future developments in the compiler for this architecture;

2. to measure the effectiveness and identify the limitations of the dataflow software pipelining, as this is applied to loops and conditional expressions;

3. to gain a deeper insight of the impact of several key modules of the architecture, and to discover relations between program structure, compiler optimization techniques and architectural characteristics.

Aside from the code generator, the compiler/architecture testbed, which allows us to conduct all the experimental work, consists of a compiler front end, a hierarchical data-dependence graph, an assembler, and an instrumented macrosimulator of the argument fetching architecture.

1.1 Structure of the Thesis

To establish the context of this work, we first describe the concept of the argument fetching model of computation and the principle of dataflow software pipelining. Chapter 2 describes the design of the architecture and the rationale behind the design decisions. It also presents the principle of dataflow software pipelining, using as an example the mapping of a basic code block. A brief comparison with other related work is also described.

Chapter 3 presents an experimental core language based on SISAL single assignment language. It describes in detail all the data types, operations and control constructs that are included in this subset. The main features of this language are the *monolithic* definition of arrays, which are static and rectangular, and the conditional and parallel loop constructs, where the software pipelining mapping scheme is applied.

The fourth chapter presents the mapping schemes for all the operations and constructs of the language that has been defined in the previous chapter. First the program graph is presented as the abstract dataflow graph. In most cases there is an one-to-one correspondence between constructs/operations in the source language and nodes in the program graph. The organization of the machine graph is also presented in this chapter. Machine graph reflects the model of computation of the underline architecture. Finally the translation from the program graph to machine graph is given for all the operations and constructs of the subset of SISAL language, which has been defined in the previous chapter.

Chapter 5 presents the performance analysis on a set of Livermore loops through simulation. The performance gained by applying software pipelining to these loops is evaluated by measuring the speedup and utilization factors. Also the effect of balancing through an in-depth study of the simulation results of one of the Livermore loops is shown. Finally, a detailed analysis based on the simulation results is presented, addressing two key architectural factors that substantially influence the efficiency of the running programs. By experimenting with different architectural configurations the relation between program structure, compiler optimization techniques and the architectural features is addressed

In chapter 6, we concentrate on the technique of software pipelining which has been studied and implemented in von Neumann style architectures. A comparison study of dataflow software pipelining and the software pipelining as applied to Warp systolic array architecture is presented [41]. In this study the effectiveness of software pipelining is investigated in terms of scheduling efficiency, scheduling limitations, space requirements, and compiler complexity. Moreover, a comparison study based on the performance statistics gathered for the same set of Livermore loops on both architectures, is presented. This performance evaluation study is mainly focused on the achieved performance and the scheduling limitations. Also a detail analysis for the statistics gathered for the argument fetching architecture is discussed in relation to the individual loop characteristics.

Lastly , chapter 7 presents the conclusion of this work along with suggested areas for future research.

Chapter 2

The Argument Fetching Dataflow Architecture and the Principle of Dataflow Software Pipelining

In contrast to the sequential von-Newman model of computation, the dataflow model of computation offers a simple and powerful formalism for describing parallel computation. The data flow model of computation is based on the principles of *asynchrony* and *functionality*. The first denotes that all operations are executed when and only when the required tokens are available, while the second implies that any two enabled operations can be executed in either order or concurrently without affecting the result of the computation (determinacy [14]).

The first static dataflow model was proposed by Dennis and Misunas [15]. In this model each instruction is activated by the presence of its operand value; its execution consists of performing the indicated operation and delivering copies of the result value as specified by the destination fields. The presence of acknowledge signals guarantee that an instruction cannot be fired again before its target instructions are ready to receive new data.

One of the major criticism to the traditional static model and to all the other proposed dataflow architectures is the high overhead due to the data traffic. In the traditional data flow model this is due to the unnecessary data movements. The operands of the instructions are "flowing" through all the units in the processing element, although they are used only upon the execution unit. Result values are copied and stored in duplicate whenever there is more than one target instructions.

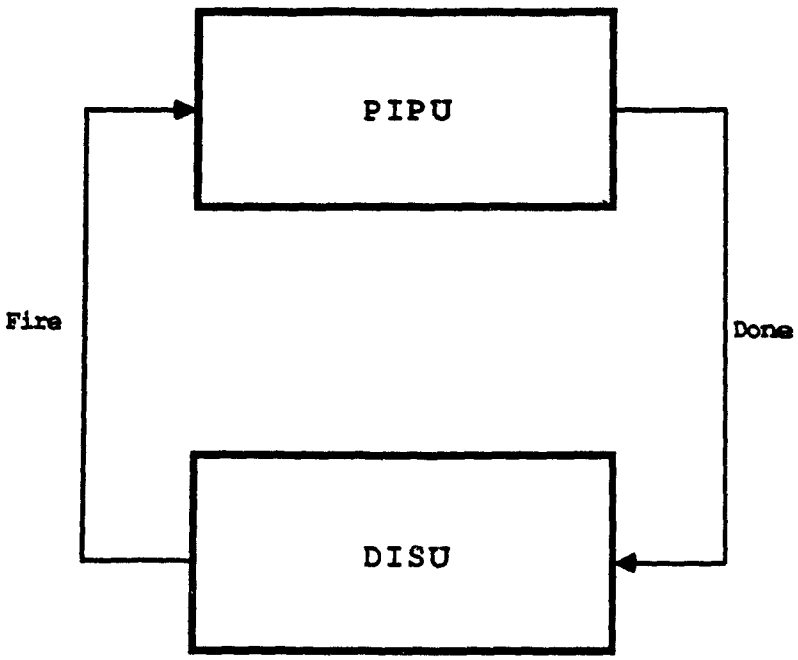

Figure 2.1: An argument-fetching data flow processor

In the traditional static dataflow architecture, known as *argument-flow* architecture, the main reason for the above inefficiencies arises from the decision to keep data and control information bound together in packets as they traverse the circular structure of the processor pipeline [16].

2.1 The Argument Fetching DataFlow Architecture

The main principle that differentiates the new argument-fetching architecture from the traditional argument-flow static data flow architecture, is the separation of data and signals in the information packets(tokens). It becomes evident that it is better for an instruction to fetch its own arguments from a data memory than its predecessor instructions to store result values in the operand fields of several target instructions

The target architecture to be studied in this thesis, is a pipelined dataflow processor architecture based on the *argument fetching* data-driven principle The key feature is that data never "flow", while instruction scheduling remains data-driven. This architecture has been proposed and reported in [18,23,24].

Figure 2.1 shows a block diagram of this architecture. The argument-fetching processor

consists of two major processing modules design to perform the instruction execution and scheduling functions:

- The *data flow instruction scheduling unit (DISU)* holds the *signal graph* of the collection of data flow instructions allocated to the processing element and maintains a record of which instructions are enabled for execution.
- The *pipelined instruction processing unit(PIPU)* is an instruction processor that uses conventional techniques to achieve fast pipelined operation. The PIPU executes enabled instructions and informs the DISU when each instruction finishes execution.

The *fire* link in Figure 2.1 is for transmitting the addresses of enabled instructions from the DISU to the PIPU. The *done* link is for transmitting the addresses of the instructions which have completed execution in the PIPU, together with a *condition code* used by the DISU to control the sending of conditional signals By decoupling data and signals, the argument fetching architecture combines the powerful data-driven instruction scheduling of the data flow model (DISU), with the simplicity of a conventional pipelined processor (PIPU).

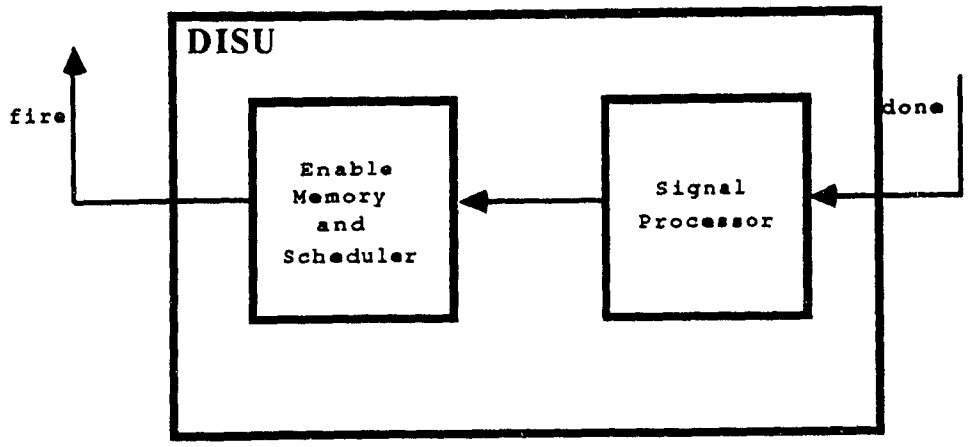

Figure 2.2: The structure of DISU

2.1.1 The Organization of DISU

The DISU in the argument fetching architecture plays the role of the program counter by providing addresses of candidate executable instructions. The difference is that the DISU is a "data-driven program counter" which maintains not one but a pool of instructions that are ready for execution.

The structure of DISU is shown in figure 2.2. It consists of a *a signal processing unit(SP)* and an *enable controller unit(EC)*. The signal graph, which hold the sequencing information among the instruction of a program, is represented in the DISU by the signal lists stored in the *signal list memory* of the SP unit. Each signal list represents a set of signal arcs leaving the associated instruction of the signal graph The *enable count memory* of the enable controller unit holds the *count* and the *reset* status values of each node in the signal graph.

In response to a *done* signal from the PIPU for an instruction, the SP unit retrieves the signal lists for this instruction and sends a *count* signal for each entry in the active list The EC unit receives the count signal and decrements the count value of the indicated node. When this count value becomes zero, an enable flag for this instruction is set and the reset value is copied back into the count to prepare the next firing cycle of the instruction. The

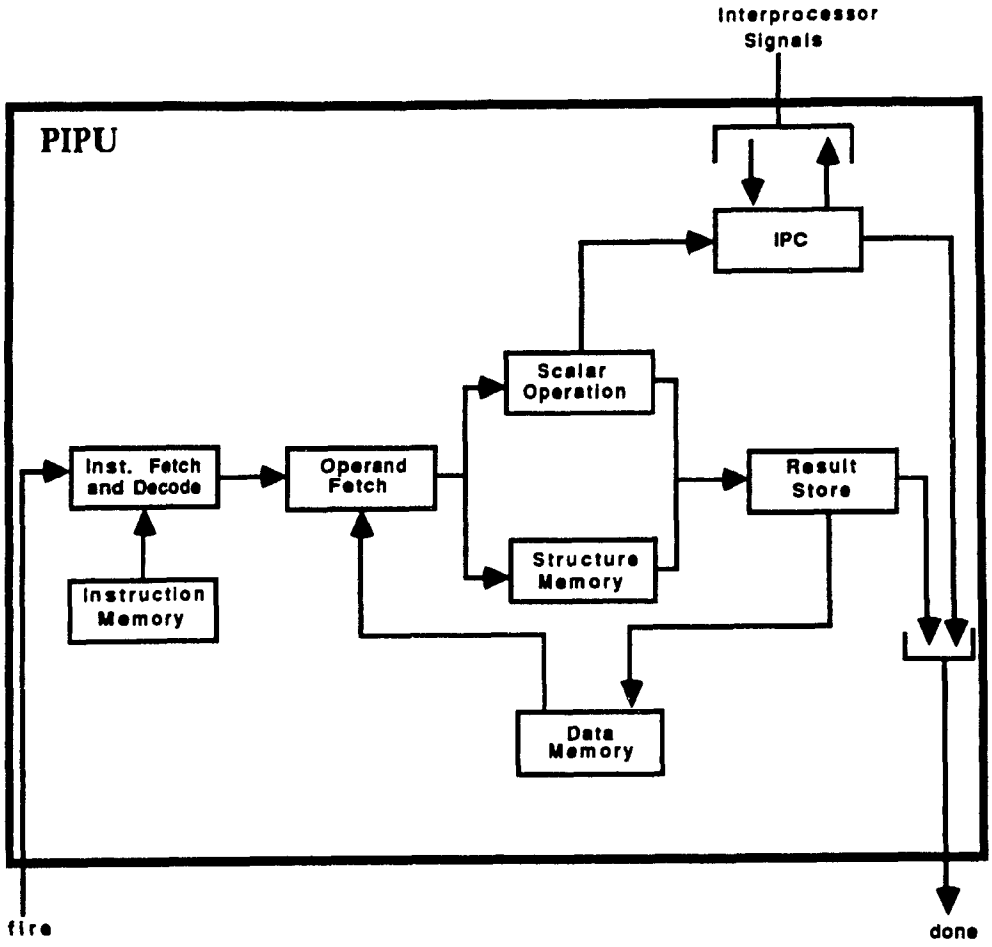

Figure 2.3: The structure of PIPU

EC unit continuously monitors all the enable flags and issues fire signals for the enable nodes.

2.1.2 The Organization of PIPU

The PIPU can be considered as a conventional pipelined processor without a program counter since the DISU is responsible for providing addresses of candidate executable instructions. The block structure organization of the PIPU is shown in figure 2.3.

It consists of six pipeline stages to handle instruction fetch and decode, operand effective address calculation and fetch, instruction execution and store of the result values. The instruction execution consists of a scalar operation unit and a structure operation unit. The scalar operation unit performs arithmetic and logic functions as well as scalar memory operations, while the structure memory unit performs data structure oriented memory operations, such as array accesses. The IPC unit is used for interprocessor communications. The architecture also provides built-in primitives for handling FIFO buffers

2.2 The Concept of Software Pipelining

The static dataflow model of computation derives its simplicity by disallowing more than one instantiation of any instruction simultaneously. It can exploit the parallelism of the programs by *pipelining* data from multiple instantiations of a loop or a procedure body. In the static dataflow model, pipelining means arranging the machine code such that the successive computations can follow each other through one copy of the code. If we present a sequence of values to the inputs of the dataflow graph, these values can flow through the program in a pipelined fashion.

In dataflow computation, program mapping is performed on units of program text that define the major structured values involved in a computation. These program units are compiled into units of code called *code blocks* Code blocks are the units of source programs to be handled by code mapping strategies and will be decomposed and assigned to the processing elements of a dataflow multiprocessor computer. In SISAL, many code blocks can be written as **for-in** expressions. The following SISAL program is an example of such a code block:

```
X := for i in 1,n
          returns array of
          exp(exp(2*A[i], 2) + exp(2*B[i], 2), 2)
     end for
```

This code block take as input two arrays A and B and produces another array X such that:

$$X[i] = ((2 * A[i])^2 + (2 * B[i])^2)^2, \forall i \in (1, n)$$

The main feature of a **for-in** code block is that the array elements of the result array X can be evaluated in parallel because there are no data dependencies among them. In the dataflow graph based on the pipelined mapping of the above example, successive elements of the input array A will be fetched and fed into the dataflow graph. The program is fed sequences of elements A[1], A[2], ..., A[n] and B[1], B[2], ..., B[n]. The computation proceeds in a pipelined fashion.

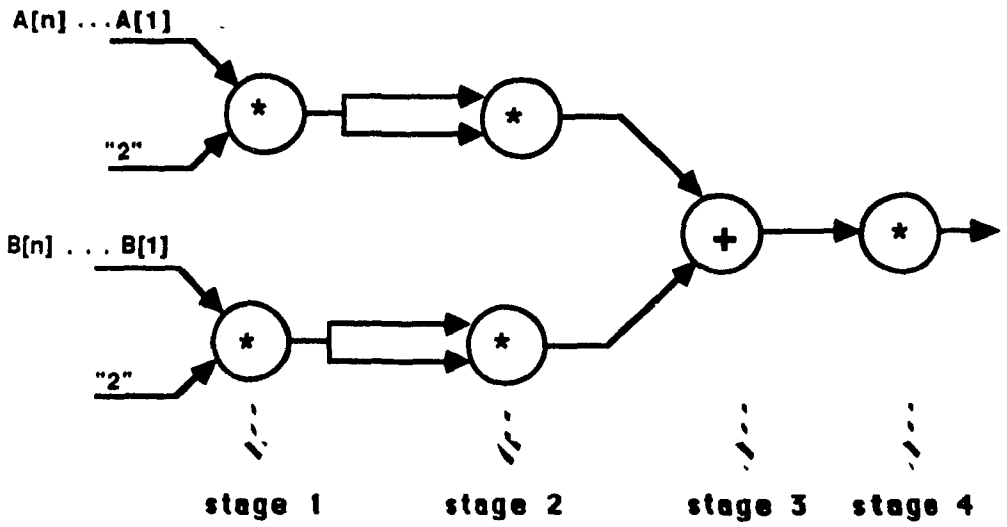

Figure 2.4: Software pipelining of dataflow programs

Figure 2.4 illustrates the fine-grain parallelism that exists in the above SISAL program. In this dataflow graph, nodes represent instructions and arcs represent data values. Instructions that belong to the same stage can be executed in parallel, since there are no data dependencies among them. Moreover, during the pipelined execution of the program, multiple stages can be executed concurrently Stage 1 and 3 are enabled and can be executed in parallel; the same applies to stage 2 and stage 4. The power of fine-grain parallelism can be derived from programs that form a large pipeline in which many instructions in multiple stages can execute concurrently.

However, unlike in vector processors, there is no requirement that the activities of one vector operation must be continuously processed by one or a group of dedicated function units in the processor. The applicative nature of the data flow graph model allows flexible scheduling of the execution of enabled actors in the pipeline In fact, an ideal data flow scheduler (with a sufficiently large data flow computer) will execute each actor as soon as its input data become available. Therefore, massive parallelism of vector operations can be effectively exploited by a data flow computer in a fine-grain manner the scheduling of the physical function units and other resources for sustaining such vector operations are totally transparent to the user This is called *software pipelining* – the arcs drawn between actors correspond to addresses in stored dataflow machine code, and not to the wired connections between logic elements. For more detailed discussion concerning the software pipelining, the readers are referred to [28,21,22].

Loop unraveling has been proposed for the exploitation of parallelism as another powerful alternative technique in tagged-token dataflow architecture [7]. A major difference is that, during loop unraveling, each iteration is assigned a new tag, which implies new resources are allocated to the activations in the new iteration. For example, in the MIT tagged-token dataflow architecture (or TTDA [7]), this means potential more space in the *token matching store* is required to handle the concurrent activities. In the new MIT Mansoon project [10], this implies that a new copy of memory frame is required for each iteration to hold the tokens for the loop body. In either cases, the overhead becomes large when large number of iterations is to be unraveled. One danger is the possibility of overloading the system with parallelism. The recent work on loop "throttling" technique can partially reduce the overhead in loop unraveling by limiting the number of concurrent iterations [7].

On the other hand, the dataflow software pipelining suggested here will use the same code and data memory space for the entire loop pipelining. Furthermore, no new tags are required for concurrent execution of iterations and the danger of overwhelming the computational resources is prevented.

2.3 Summary

Dataflow computers are based on the concept of *data-driven* computation, i.e., the instruction execution in a conventional von Neumann computer is under program-flow control, where as that in dataflow is driven by data availability. In argument fetching dataflow model of computation, the execution of a program is data-driven without the overhead of transmitting data from one instruction to another.

Dataflow software pipelining is an efficient program mapping strategy for compound constructs and data structure operations, that keep enough instructions enabled for processing. The parallelism in array operations of loop expressions frequently found in scientific computation can be effectively exploited by organizing the dataflow machine program such that array operations can be fully pipelined. Under software pipelining scheme program mapping is performed on units of program text, called *code blocks*, that define the major array values involved in a computation. The static dataflow model of computation can encapsulate the software pipelining very effectively without introducing much complexity to the compiler. Moreover minimal space requirements are needed to exploit this technique in the static argument fetching architecture.

Chapter 3

The Language

It is very important in a parallel computing system to facilitate the exploitation of parallelism of the running programs. Moreover, it is important that application programs are written in high-level programming languages that allow the programmer to abstract away from details of the machine structure. One option is to generate parallel code from a conventional high-level programming language, without any extensions to support parallelism. Due to potential side-effects of program statements, compile-time data dependence analysis is often difficult especially for inter-procedure analysis and handling of aliasing caused by the use of pointers [1,6].

As an alternative, functional programming languages have been proposed. Functional languages are very attractive because their side-effect-free nature means that the order of execution is irrelevant. Recently there has been a lot of effort in implementation of *single-assignment* programming languages. In such languages there is no concept of global storage, and state. When assignment is restricted to occur only once for each variable in a program, the effect is as if assignment statements are definitions of the *value names*.

Instead of reinventing everything from scratch, we decided to start our work with an existing functional language which must satisfy the following two conditions:

1. it has a substantial body of real programming done in large-scale scientific programming, and
2. it has given reasonable consideration to provide array operations.

We have chosen SISAL as our candidate, because it is a language which seems to satisfy the above criteria. Particularly, this work is focus on a subset of SISAL, the main feature of which is the monolithic and static definition of arrays.

3.1 The SISAL Programming Language

SISAL is well-suited to parallel processing applications because of three important features: the language does not have any side effects, it does not require explicit synchronization primitives, and it encourages parallel solutions. Side effects are eliminated by SISAL's single assignment rule. SISAL has no explicit features for synchronization, and programmers need not worry about the difficulty of understanding and programming explicit synchronization primitives

The language is strongly typed using structural type equivalence, and allows only explicit type conversion. A SISAL program is concerned with the definition and use of values. A value may be a constant value or a value name which is associated with the result of an expression evaluation. No value name may be redefined, and so all uses of a given value name in other expressions will always refer to the same value. The value of every expression depends solely upon the values of the inputs to the expression The order of the execution of a SISAL program is thus determined solely from the availability of values for use in the computation of expressions, and does not affect the computed results

An expression may yield more than one result. The number of results produced by an expression is referred to as its *arity*. Expressions may be nested within other expressions, provided that its arity and the types of its results are correct for the context in which it is used. Expressions are entirely free from side-effects. A function call is one kind of expression. A function definition simply encapsulates an expression with proper parameter-passing mechanism. Functions have access only to their arguments and to other functions.

3.2 A SISAL-based Experimental Language

Here we do not attempt to give a complete description of SISAL, as this can be found in [39]. In fact throughout the thesis we will restrict our attention to a subset of SISAL, referred to as the SISAL Kernel. The motivation of defining this language is twofold:

1. to serve as the minimal basis of a functional language which will be developed for the argument fetching static dataflow architecture;

2. to be used by the compiler as a source language from which to generate code for this architecture.

With respect to the above objectives, the SISAL Kernel allows only one *program module* which contains only one function. Since this language has been designed for a static dataflow architecture where function calls are implemented by in-line expansion, this restriction does not affect the generality of the language. By disallowing recursion, a program with several function calls can be translated to a single function body where all the called functions are replaced by their code. The description of SISAL Kernel is given in the following sections.

3.2.1 SISAL Kernel Data Types

The SISAL Kernel supports two data types: *scalars* and *arrays* The remaining set of data types that is supported in SISAL but not in the Kernel that we are concerned with here includes: *streams*, *records*, *unions* and *error values*. Although a large subset of SISAL data types are excluded, this language still can express a large number of kernel programs that consume the most cycles in scientific computation programs which involve only scalar and array operations. Finding efficient mapping schemes for these two data types is, by no doubt, the most crucial part in design and implementation of code generators for high-level languages.

Scalar Types

The SISAL Kernel provides the following scalar types together with the usual operations on and between them:

- *boolean*,
- *integer*,
- *real*, and

- *double real.*

All the SISAL scalar types are provided except the *character* type. Although the SISAL Kernel provides for the declaration of value names, the typing system enables the type of any value name to be directly deducible from its defining expression, so declarations are only used when their inclusion improves program clarity. A number of predefined functions are provided together with explicit type conversion operations The set of predefined functions includes modulus (*mod*), magnitude (*abs*), maximum (*max*), minimum (*min*), and a real and mixed exponentiation (*exp*). No implicit type coercions are performed. The set of all the valid operations for each data type will be listed in the next section.

Arrays

Arrays are the most important data types in scientific numerical computation. Arrays have been a challenge to the researchers of functional languages due to the difficulty of finding an efficient implementation. In a pure functional language which has no notion of computation by effects, an array is treated as a functional aggregate value.

In SISAL, an array has components of arbitrary type and an implicit integer index. Each array's size is determined by evaluation of the expression defining the array value. Multi-dimensional arrays are represented as "arrays of arrays". Moreover an array append operation, generates a new array which agrees everywhere with the old array except in the position that its value has been changed. However, such incremental array update operations also cause large overhead due to copying.

In SISAL Kernel the array definitions meets the following three criteria:

1. have reasonable expressive power to meet with programming requirements in scientific numerical computations;
2. keep space/time efficiency, while providing an implementation scheme which can effectively exploit parallelism;
3. keep clean semantics for parallel processing;

SISAL Kernel adapts the use of **monolithic arrays**, the computation of whose elements are defined all at once when the array is first created [29]. An n-dimensional monolithic array is defined by a mapping:

$$U \rightarrow V$$

where U is the set of the Cartesian coordinates defined on a rectangular n-dimensional Euclidean space, and V is the set of values of type T. In addition to their monolithic nature, arrays in SISAL Kernel have the following three characteristics:

1. Their size is *constant*, known at compile time.
2. They are rectangular, i e. all the elements in the same dimension have the same size.
3. The lower bound is always equal to one.

This class of arrays can be called: **static rectangular arrays**. A large number of numerical computation programs can be written using only static rectangular arrays and thus their mapping to efficient dataflow machine programs is crucial.

3.2.2 SISAL Kernel Operations

Here we specify the sets of operations applicable to each data type of SISAL Kernel. The valid operations for booleans, integers, reals, arrays and type-conversion operations are listed in tables 3.1, 3.2, 3 3, 3.4 and 3.5 respectively. In those tables, symbols have the following meaning: P and Q for boolean, J and K for integers, X and Y for reals, A for arrays, T for arbitrary types, and V for values of arbitrary type

Notice in table 3.4 that the "create by elements" array operation returns an array of the indicated type with low index equal to 1 and high equal to k The k-elements of the array are equal to V_1, ..., V_k respectively. The "type name" is optional, but if present, must conform with the type of V. This "type name" denotes the type of the array operation and therefore must be an array type - not the type of the component Finally the **array_fill** operation creates an array with the given range i.e. from 1 to a high bound Hi. All the elements are equal to the given value V.

Boolean Operations		
operation	*notation*	*functionality*
and	P & Q	bool, bool → bool
or	P ⋆ Q	bool, bool → bool
not	∼P	bool → bool
equal	P − Q	bool, bool → bool
notequal	P ∼= Q	bool, bool → bool

Table 3 1: SISAL Kernel boolean operations

Integer Operations		
operation	*notation*	*functionality*
addition	J + K	int,int→int
subtraction	J - K	int,int→int
multiplication	J * K	int,int→int
division	J / K	int,int→int
modulus	mod(J,K)	int,int→int
exponentiation	exp(J,K)	int,int→int
negation	-J	int→int
magnitude	abs(J)	int→int
maximum	max(J,K)	int,int→int
minimum	min(J,K)	int,int→int
equal	J = K	int,int→int
not equal	J ∼= K	int,int→bool
greater	J > K	int,int→bool
less	J < K	int,int→bool
greater or equal	J >= K	int,int→bool
less or equal	J <= K	int,int→bool

Table 3.2: SISAL Kernel integer operations

Real Operations		
operation	*notation*	*functionality*
addition	X + Y	real,real→real
subtraction	X - Y	real,real→real
multiplication	X * Y	real,real→real
division	X / Y	real,real→real
modulus	mod(X,Y)	real,real→real
exponentiation	exp(X,Y)	real,real→real
exponentiation	exp(X,J)	real,int→real
negation	-X	real→real
magnitude	abs(X)	real→real
maximum	max(X,Y)	real,real→real
minimum	min(X,Y)	real,real→real
equal	X = Y	real,real→real
not equal	X ~= Y	real,real→bool
greater	X > Y	real,real→bool
less	X < Y	real,real→bool
greater/equal	X >= Y	real,real→bool
less or equal	X <= Y	real,real→bool

Table 3.3: SISAL Kernel real operations

Array Operations		
operation	*notation*	*functionality*
create by elements	**array**[type-name][1:V_1...V_k]	int, T→array[T]
create/fill	**array_fill**(1,Hi,V)	int, int, T→array[T]
select	A[J]	array[T], int→T

Table 3.4: SISAL Kernel array operations

Type-Conversion Operations		
operation	*notation*	*functionality*
real-to-integer	**floor**(X)	real→int, double→int
	integer(X)	real→int, double→int
	trunc(X)	real→int, double→int
integer-to-real	**real**(J)	int→real
	double_real(J)	int→double
integer-to-real	**real**(X)	double→real
	double_real(X)	real→double

Table 3.5: SISAL Kernel type-conversion operations

3.2.3 SISAL Kernel Control Constructs

The program structures described in this section are specific forms of expressions legal in the SISAL Kernel. A more detailed description of the these constructs can be found in [39].

The Let Construct

The purpose of the **let** block is to define one or more value names, which are then used in the evaluation of an expression. The result of this expression is the result of the block. Every value name introduced in a "let" block occurs on the left hand side of ":=" exactly once. The scope of each value name introduced in a let block is the entire block following the definition, except any inner constructs that reintroduce the same value name. Value names defined in a block are not available for use outside that block.

The IF Conditional Construct

The conditional construct is used to select one of two alternative expressions for evaluation, depending on the result of the boolean test. The selected expression is evaluated and its results are returned as the results of the whole construct. The **else** branch must always be present to define the results of the conditional when the test yields *false*. Similarly, the expressions given in the two branches must conform in the number and types of results produced.

A single conditional construct can be used, without nesting, to select one of the several expressions for evaluation using one or more *elseif* branches. In this case, the entire construct is an expression whose tuple of values is that of the first arm whose test expression is "true", or the final arm if all test expressions are "false". Finally, the conditional construct does not introduce any value names. All value name scopes pass into the conditional construct.

The FORALL Iterative Construct

In SISAL there are two forms of iterative constructs: (1) the non-product form (**for-initial**), and (2) the product form (**for-in**). The non-product form performs sequential iteration in which one iteration cycle depends on the result of the previous cycles. The product form is a special case of the non-product form that provides a more concise way to specify arrays. It is used to denotes that there are no loop carry dependencies among the cycles of the iteration, so ideally the computations of the cycles can be done in parallel.

SISAL Kernel uses only the product form of the SISAL iterative construct. The product form allows inner and/or outer (Cartesian) array index products to be specified. Since our main concern has been focused in the efficient implementation of monolithic arrays, we use the **for-in** construct as one parallel monolithic array constructor. The syntax of this construct is the following:

for *generator*
    body
    returns *return-expression-list*
end for

The **for-in** construct comprises from three parts:

1. a *generator*, which specifies the range of values for which the body will be executed,
2. the *loop body*, and
3. the *return-expression-list*, which specifies the value(s) to be returned.

operation	*comments*
elem **in** A	process all elements of array A.
elem **in** A **at** I	process all elements of array A and bind the corresponding indexes to I.

Table 3.6: Array element **in** expression

SISAL Kernel supports all the expressions that can exist in the *generator* of a **for-in**. These expressions are **in**, **at**, **dot**, and **cross**. Using these four forms of generators, elements of an array may be available inside the loop without explicit subscripting. Table 3.6 shows how this can be accomplished by manipulating an array A. In case that the array A is multidimensional, the **in** expression process the elements across the outermost dimension of the array. Also, multiple ranges may be specified and combined using either **dot** or **cross** products. In a **dot** product, the ranges must have the same size and the i-th indexing element of each range is used to drive the i-th iteration. The **cross** product is short-hand for a nested **for-in**. A more detailed description for the functionality of these four expressions is given in [39].

The result of **for-in** construct is the tuple of values defined by the *return-expression-list*. In the **for-in** construct, since all the elements of the range are independent of each other, ordering of the results is defined by using the range expression's list ordering. Each return expression must contain some expression that describes a result to be produced. The expression list is defined from two parts: a prefix and an expression. In SISAL Kernel there are two possible prefixes:

- **value of**
- **array of**

The **value of** prefix signifies that the following expression produces a single value in one of two ways. If the **value of** is not followed by one of the reduction operators, the result produced by this clause is the last element of the sequence (as described above). If a *reduction operator* is in the clause, it means that all elements of the sequence will be combined using the reduction operator to produce the single value. The *reduction names*, their operation along with the legal set of types, are listed in table 3.7. The reduction operation is always performed within the lowest dimensioned sequences and then successively applied to all higher level dimensioned sequences.

Reduction Operators		
operator name	*legal types*	*operation*
sum	int, real, double, boolean	addition or boolean OR.
product	int, real, double, boolean	multiplication or boolean AND.
least	int, real, double	minimum.
greatest	int, real, double	maximum.

Table 3.7. SISAL Kernel reduction operators

The **array of** produces an array having size equal to that of the sequence of the results as described above and containing exactly those values in the order defined by the sequence. The lower bound of the array is always equal to one. If the **for-in** defines a cross product range, then the resulting array is multi-dimensional. SISAL Kernel applies one restriction in the construction of multi-dimensional arrays it does not allow the exportation of a subarray which is used for the construction of a bigger array This is a reasonable assumption which does not affect the generality of constructing multi-dimensional arrays in a **for-in** construct. The reason behind this assumption will become apparent in chapter 5, where the machine code mapping schemes are presented

For the reduction operators, SISAL Kernel also supports a masking clause modifier. Each result clause may optionally be followed by a boolean expression preceded by a **when** indicator which acts as a filter This filter determines if specific sequence values should be taken out of the sequence prior to final result calculations After each iteration cycle, the masking clauses are evaluated. If the **when** clause is *false*, the corresponding expression value is dropped from the sequence.

3.3 Summary

SISAL Kernel is an experimental core language based on SISAL single-assignment language. Although it is in a way restrictive in the set of data types and the operations that it allows, it is powerful enough to express kernels that capture much of the computation of larger scientific programs It is also suitable for experimenting and measuring the efficiency of the generated code in respect to mapping schemes that are applied from the compiler in a static dataflow architecture.

The main features of the language are: (1) the monolithic definition of arrays, which are

rectangular and their size is known at compile time, and (2) the **for-in** construct, which is used as the main parallel monolithic array constructor. In addition to array data types all the other scalar types are supported, with the exception of character types. The **for-in** and the **if** constructs constitute the major features of code blocks where the software pipelining mapping scheme will be applied.

Chapter 4

Translation to Machine Code

Here we will show how to systematically translate program graphs into machine graphs which contains only instructions suitable for execution on the argument fetching architecture. First the program graph will be presented. The program graph is composed from a collection of nodes, each with an operation code that identifies its function, and some number of *inputs* and *outputs*

Translation to machine code is accomplished by substituting each node of the program graph by groups of tuples, each one containing the machine operation code along with the arguments and the signaling information. Here we present the mapping schemes for all the operations and constructs of the SISAL Kernel. For each simple or compound node that represents an operation or a construct of the SISAL Kernel, we present its machine graph showing pictorially the data and the signal dependencies among the machine instructions.

The categories of simple and the compound nodes that correspond to the operations and constructs of the SISAL Kernel, are the following:

1. *arithmetic* and *boolean* nodes,
2. *type conversion* nodes,
3. *array* manipulation nodes,
4. nodes dealing with *multiple values*.

5. *select* compound node, and

6. *forall* compound node.

As we describe the translation to machine code, we will introduce the machine instructions as needed. For a more detail description of the available machine instructions and their functionalities, the reader should refer in Appendix A.

4.1 A Hierarchical Data Dependence Program Graph

The single-assignment and functional properties of SISAL make it particularly suited to translation into data dependence graph form. The first step in the SISAL compiler, is to produce a machine-independent graph, known as IF1 (Intermediate Form - version 1) [45]. IF1 is very high level, and the structure of an IF1 program follows closely the SISAL program it was derived from. Also a number of machine-independent optimizations had been applied at the IF1 level from the SISAL compiler. Loop-invariant removal, common subexpression elimination, constant folding, dead-code elimination and function in-line expansion are some of the most important optimizations.

IF1 files comprise a number of lines that contain printable ASCII characters The hierarchical data dependence graph (HDDG) that is used here, is based on the IF1 common intermediate form for SISAL. The HDDG data structures are produced from the IF1 program in two steps. In the first step, the IF1 program file is parsed, and the data structure representing that program is constructed. In the second step, the data structure is manipulated to construct a HDDG data structure. At this step, all the implicit data dependencies in IF1 are converted to explicit.

HDDG is based on *acyclic graphs*. The HDDG data structures represents a set of *graphs*, one for each SISAL function. Graphs consists of *nodes* and *edges*. Nodes denote operations and edges represent data dependencies between nodes. Types may also be attached to each edge. Graph boundaries surround groups of nodes and edges.

Nodes represent operations on values. There are two types of nodes: *simple* and *compound* nodes. The difference between these two categories of nodes is that compound nodes contain subgraphs while simple nodes do not. The semantics of simple nodes describe the relation of the inputs of the node to its outputs. Nodes receives their values from a number

of *input ports* and "deposit" the produced values to the *output ports*. Most simple nodes have a fixed number of input and output ports, although there are variable numbers of inputs to nodes that build structured data types.

The *compound* nodes are hierarchically defined, so that substructures are denoted by subgraphs. The semantics of compound nodes describes the ways in which the subgraphs of the compound node interact. Compound nodes gather the values needed by their subgraphs, so they generally have an arbitrary number of input and output ports.

4.2 Machine Graph

A program for the argument fetching architecture is a set of *instruction tuples*. Each *instruction tuple* consists of a *p-instruction*, which defines a three-address instruction that is executed when this instruction becomes enable, along with an *s-instruction* that specifies the sequencing information among instructions of a program and is processed in DISU. Each *s-instruction* contains two fields: the *enable* field indicates how many signals must yet be received for an instruction to become enable and the *reset* field holds the value to be placed in the *enable* field when the instruction is fired. Also, each signal is tagged with a condition code. There are the possible condition codes: true, false, and unconditional. According to the result condition code of the corresponding *p-instruction*, only one of these signal lists becomes active each time an instruction completes its execution.

The assembly form of the program tuples is named A-Code. A-Code supports a reduced set of instructions which can be divided into arithmetic, logic, comparison, type conversion and data transfer operations. The set of the *p-instructions* that are supported is given in Appendix A.

Translations from the intermediate program graphs to machine graphs are most easily expressed pictorially. Figure 4.1 presents the machine graph for the expression (a+b)(c+2). In a machine graph representation, *instructions tuples* are represented by boxes along with all the incoming arcs. Throughout the rest of the chapter, the term *instruction tuples* will be interchangeably used with the term *machine instructions* or simply *instructions*. The operation code of the p-instruction is written inside the box. The arcs are labeled to denote a signal or/and a data dependency between two tuples. All the incoming signals labeled with "d" represent data arcs and correspond to an existence of an argument in the three-address p-instruction of this tuple.

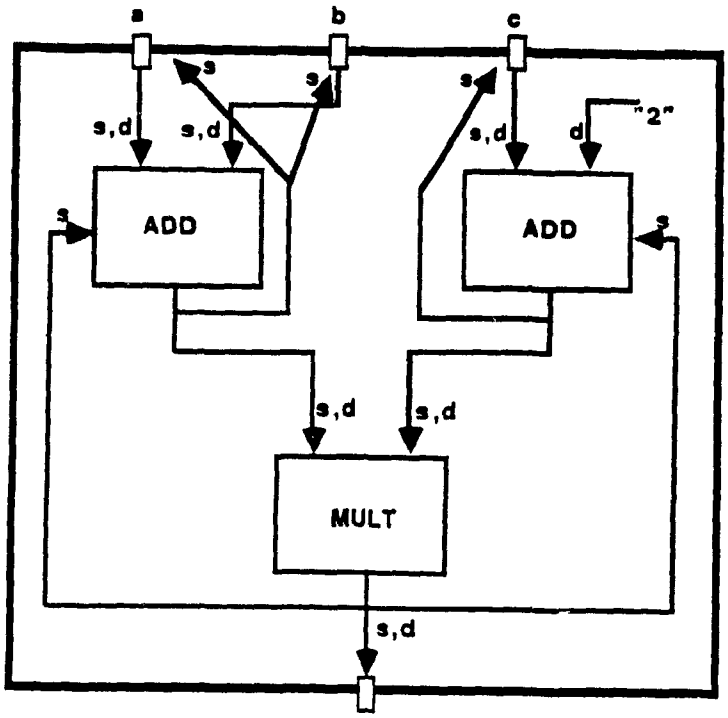

Figure 4.1. Machine graph of the $(a + b)(c + 2)$ expression

Literals are represented by denoting their values inside quotes along with "d" arcs pointed to the instructions that use them. The result register of every tuple is not showed explicitly in the machine graph: it is implied that for every tuple there is unique register that holds the result of the operation. In general, every tuple has its own result register unless if it is stated otherwise.

The arcs labeled with "s" represent signals and correspond to a signal in the s-instruction. There are two types of signals: the *count* and the *no-count* signals These two type of signals are needed because the typical initial values of the *enable* and *reset* fields for each instruction are not equal. For the first firing, an instruction requires signals only from its predecessors (*count* signals). To fire again, an instruction should wait also for acknowledgment signals (*no-count* signals) from the instructions that consume the data produced by this instruction. Therefore, for each instruction the value of the *enable* field is the sum of all the *count* signals, while the value of the *reset* field is the sum of all the signals In the machine graphs presented here, all the forward signal arcs represent *count* signals, while the backward signal arcs represent *no-count* signals. The *enable* and *reset* fields of each instruction is not explicitly shown in the machine graphs, but they are easy derivable by counting the incoming forward and backward signals The conditional code of a signal

HDDG nodes	*Machine op. codes*	*operation*
Plus	**Add**	addition
Times	**Mult**	multiplication
Minus	**Sub**	subtraction
Div	**Div**	division
Mod	**Mod**	modulo
Abs	**Abs**	absolute value
Neg	**Cns**	negation
Not	**Not**	boolean negation
Less	**Lt**	boolean "less"
LessEqual	**Leq**	boolean "less or equal"
Equal	**Eq**	boolean "equal"

Table 4.1. Primitive arithmetic and boolean operations

can be derived from the label inside the box of the source tuple. An "F" denotes a false conditional code, a "T" denotes a true conditional code, while the absence of any label denotes an unconditional code.

A non-deterministic signal merge is defined by using a $\otimes$ symbol in the machine graph. In the current implementation literals are implemented without using signals to indicate that they are available. This eliminates the existence of signals in the s-part of the tuples to notify that a constant is available. Therefore, a literal appears only as an argument in the p-instruction of a tuple.

Finally note that in most of the graphs that will be presented in this chapter, the outermost box always corresponds to the boundary of the HDDG node whose machine graph is presented.

4.3 Mapping Arithmetic and Boolean operations

Most of the arithmetic and boolean nodes are supported as primitive machine instructions in the argument fetching architecture. Table 4.1 shows the one-to-one relation between the HDDG nodes and their corresponding machine instruction operation codes. As figure 4.2 shows, this set of nodes is directly translated into one machine instruction. This

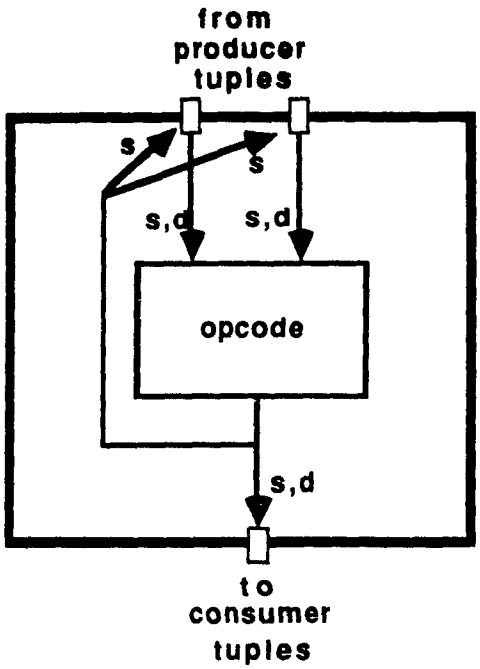

Figure 4.2: Machine graph for primitive arithmetic and boolean operations

instruction is activated whenever the two input values are available. These two input values represent the two arguments in the *p-instruction*, where the corresponding operation will be applied. Upon the end of the execution of this instruction, the "producer" and the "consumer" instructions are notified by receiving signals from the executed instruction. Note that for the "Abs", Chs" and "Not" machine instructions, the translation is the same except that these instructions receive one instead of two input values.

There are four HDDG nodes in this category that are not translated into one machine instruction. These nodes are: (1)*max*, which finds the maximum between two input numbers, (2)*min*, which finds the minimum between two input numbers, (3)*not equal*, which finds if the two input arguments are not equal, and (4)*exp* which implements the exponentiation function. In the following, we present the mapping schemes for each of these four arithmetic nodes.

4.3.1 Translation of Max and Min Operations

These two nodes have similar implementation graphs. Both are expanded in four instructions whose operation codes and signaling information is shown in figures 4.3 and 4.4. The "LT" machine instruction receives two arguments: the comparand and the comparator, which correspond to the left and the right arguments respectively. This instruction returns a "true" conditional code ("T"), if the comparand is strictly less than the comparator; otherwise it returns "false" ("F").

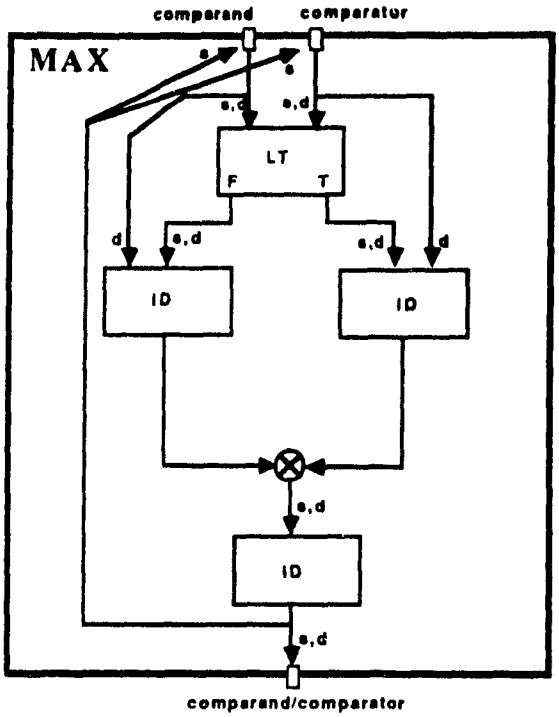

Figure 4.3: Machine graph of Max operation

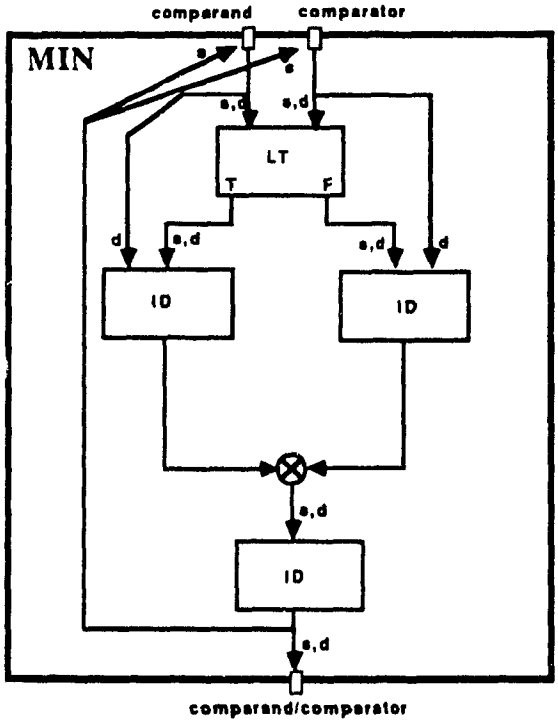

Figure 4.4: Machine graph of Min operation

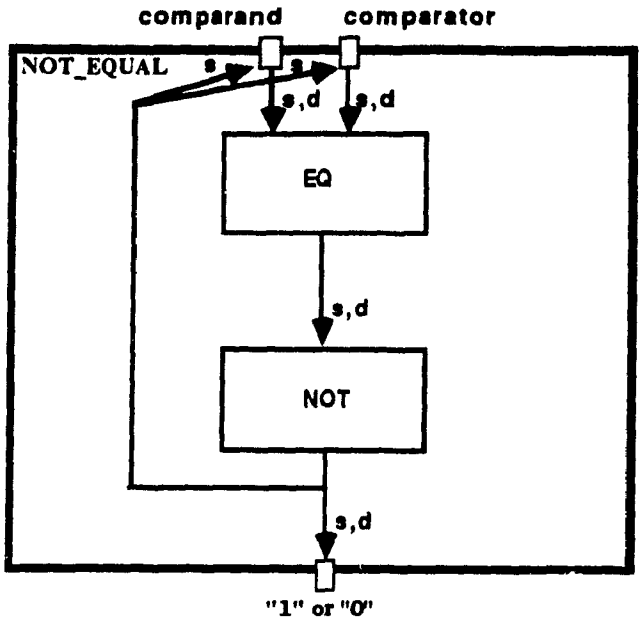

Figure 4.5: Machine graph of the NotEqual operation

The "ID" operations simply transfer the contents of the one input argument they receive, to their result register. Notice that in both figures the two "IDs" which receive signals from the boolean operation "LT", are acting as gates where the minimum or the maximum value is passed accordingly to the executed operation. Each time only one of these two "IDs" will be activated in respect to the boolean condition code that is returned from the "LT" instruction. The last "ID" is used to pass the selected value to the instructions that use the result of this operation. The non-deterministic merge denotes that only one signal is necessary to be send from the two predecessor "IDs" to activate the last "ID" instruction.

4.3.2 Translation of the NotEqual Operation

Figure 4.5 shows the machine graph of the "NotEqual" simple node. This HDDG node takes two operands and returns "true", if these operands are not equal, and "false" otherwise.

In A-Code "true" and "false" are represented by the integer values "1" and "0" respectively. The mapping of this instruction consists of two machine instructions, an "EQ" and a "NOT" instruction. "EQ" receives two arguments which are compared. If they are equal, "1" is produced; otherwise "0". "NOT" is a boolean negation operation. If the input operand is "1", then "0" (i.e."false") is returned as the result of the operation; otherwise a 1 (i.e."true") is returned.

4.3.3 Translation of the Exponential Operation

Figure 4.6 shows the mapping of the exponential function. The exponential function is represented in the HDDG by the simple node *exp*. This function takes the first input to the power of the second input. The result is always a real or double (the latter only occurs if one of the operands is a double). The translation presented here does not support the *error conditions* that can occur based on the sign and the type of the input arguments, it is the programmer's responsibility to use the right sign and type of arguments.

The "IGEN" shown in figure 4.6 is a machine instruction that is used in most of the mapping schemes that will be presented in this chapter. This instruction acts as a generator of a sequence of index values within a specific range. The first argument specifies the lower bound "a", while the second argument specifies the higher bound "b" of the of index range. "IGEN" generates the range: (a+1), ..b. Each time an index is generated, a "true" conditional code is returned; when all the indexes within the specific range have been generated, "IGEN" returns a "false" conditional code. "IGEN" is always accompanied by an "ID" instruction which is used to initialize the result register of "IGEN" to the lower index value. This restriction is enforced by the simulator of the architecture ("AD") and is expected to have a better solution in the future.

Figure 4.6 presents the machine graph of the exponential operation. This implementation follows an iterative algorithm, where the integer exponent (B) defines the number of times that the mantissa (A) should be multiplied by itself. The "IGEN" and the "MULT" machine instructions constitute a loop where the main computation of the operation takes place. Based on the value of B (exponent), "IGEN" counts how many times the first argument A (mantissa) should be multiplied. Every activation of IGEN" is followed by a subsequent activation of the "MULT" instruction, which multiples the result of the previous multiplication with the value of mantissa. When there are no more iterations, "IGEN" returns a "false" conditional code, and signals to the boolean operation "LT" (less than). This instruction checks the sign of the second input argument, which is the exponent, and accordingly activates one of the subsequent two instructions. If the exponent is positive, then the previously calculated result from the "MULT" passes through an "ID" instruction, which acts as a gate for this value. If the exponent is negative, a "DIV" instruction reverses the previously calculated result, by dividing "1" with this value.

The operations denoted by dashed boxes are optionally used according to the data

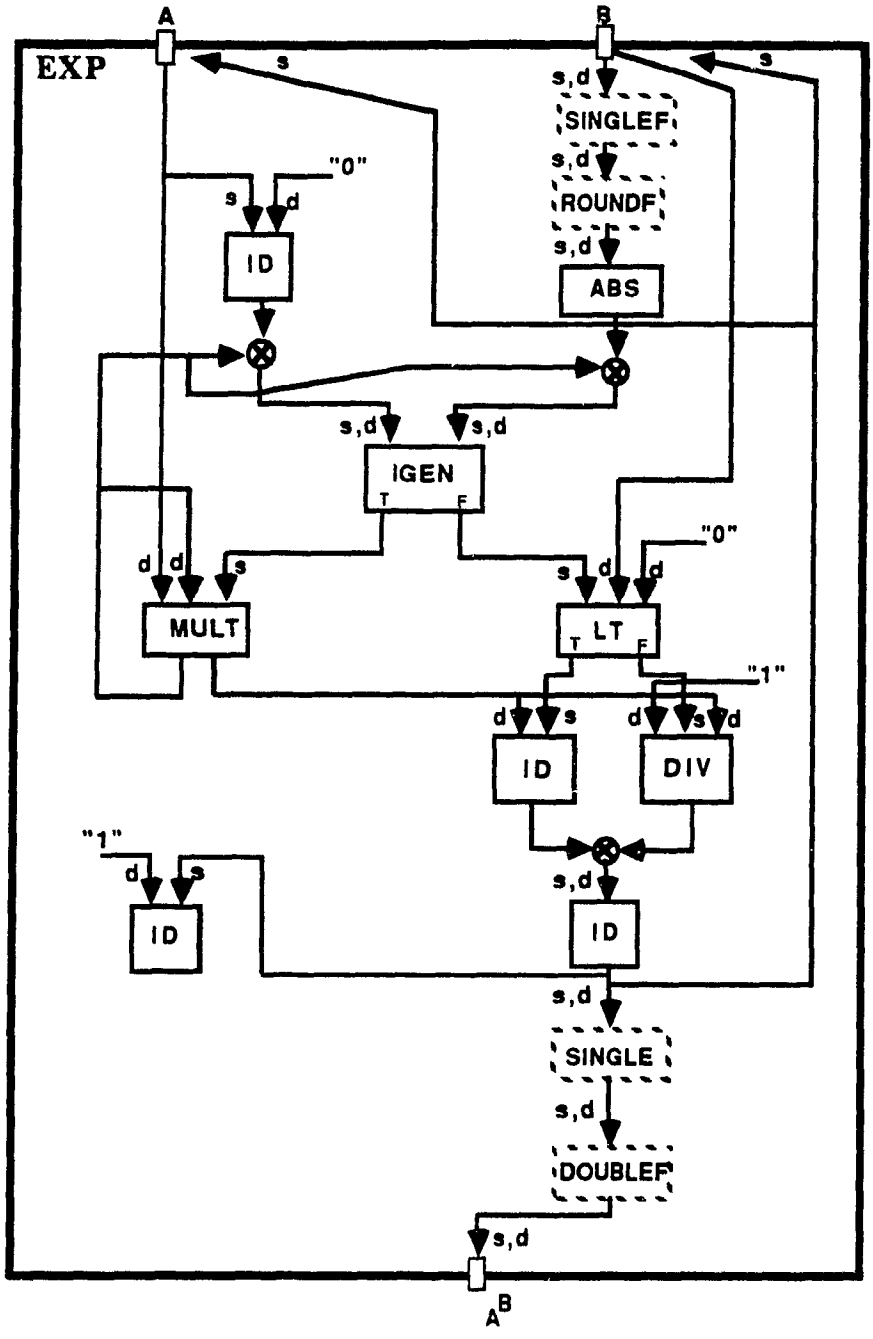

Figure 4.6: Machine graph of the Exp operation

type of the two input arguments. The implementation of type-conversion operations will be presented in the following section. "ABS" returns the absolute value of the exponent, which is used as the higher bound of the generated range. The "ID" instruction at the top of figure 4.6 is used to initialize the result register of "IGEN" to "0", such that it will be ready for the next activation of the "exp" function. Finally the "ID" instruction in the bottom left, is used to initialize the result register of the "MULT" instruction to "1".

4.4 Mapping Type-Conversion Operations

As mentioned in the previous chapter SISAL doesn't allow coercion, therefore type conversion should be explicitly done by the programmer. With the exception of the *floor* operation, the HDDG nodes that corresponds to the type conversion functions can be divided into two categories (1) the primitive type conversion nodes, where there is a one-to-one correspondence between the HDDG nodes and the machine instructions, and (2) the non-primitive type conversion nodes, that are expanded into two machine instructions. The functionality of these two categories of type conversion nodes is presented in table 4.2 and table 4.3 respectively; their mapping schemes is presented in figures 4.7 and 4.8.

4.4.1 Translation of the Floor Operation

Figure 4.9 shows the machine graph for the *floor* type-conversion operation. This operation converts the real or the double input value to an integer, whose value is the greatest integer less than or equal to the input value. First, the "SUB" instruction subtracts "0.5" from the input value, and then a "ROUND" instruction is used to generate the integer that is closer to the value returned by the "SUB" instruction. If the integral part is less than .5, it returns the largest integer not greater than the input real number, otherwise, it returns the smallest integer that is greater than the input real number. The "SINGLEF" instruction exists only if the input value is a double number, and it is used to convert the double to the corresponding real number.

HDDG nodes	*Machine op. codes*	*functionality*
Bool	**Id**	integer→bool
Single	**Single**	integer→real
	Singlef	double→real
Double	**Doublef**	real→double
Trunc	**Truncf**	real→integer
INT	**Id**	boolean→integer
	Roundf	real→integer

Table 4.2: Primitive type-conversion operations

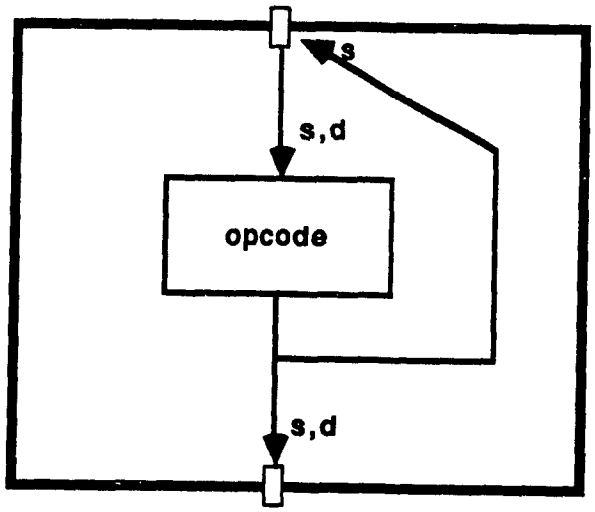

Figure 4.7: Machine graph of the primitive type-conversion operations

HDDG nodes	*Machine instructions*		*functionality*
	Operation code 1	*Operation code 2*	
Double	**Single**	**Doublef**	integer→double
Trunc	**Singlef**	**Truncf**	integer→double
Int	**Singlef**	**Roundf**	double→integer

Table 4.3: Non-primitive type-conversion operations

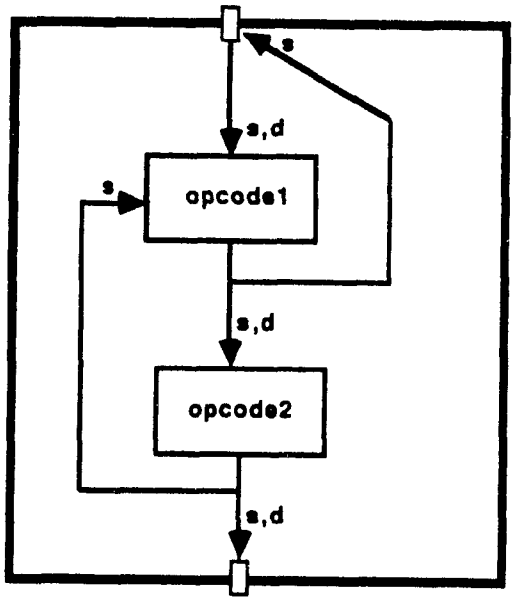

Figure 4.8: Machine graph of the non-primitive type-conversion operations

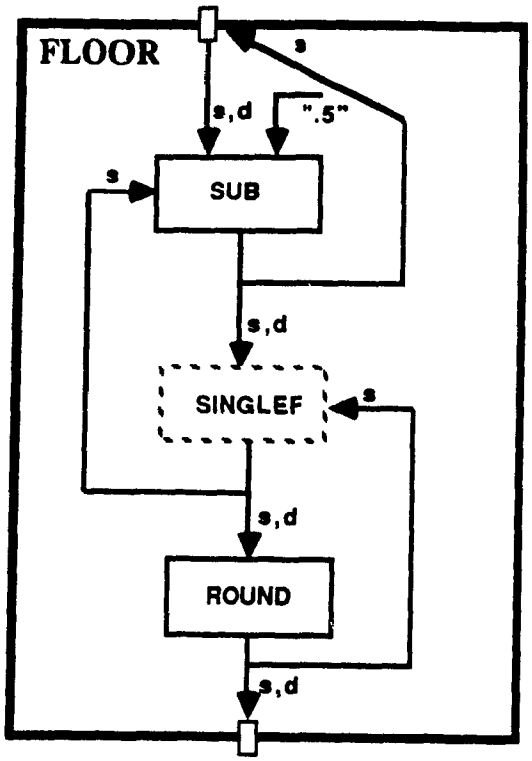

Figure 4.9: Machine graph of the Floor type-conversion operation

4.5 Mapping Array Operations

As mentioned earlier, a research in the dataflow area which is concerned with scientific computing, has been concentrated on the efficient implementation of arrays and other data structures. The most popular scheme is to hold each structure in a special-purpose *structure store* and represent arrays by their *descriptors*. Elements of a structure can be put into or obtained from the store using the descriptor together with an index to construct an array or read the elements of an array respectively.

Here we are concentrated to the class of arrays that has been defined in the previous chapter i.e. the *static rectangular arrays* with a constant lower bound of 1. In this class of arrays, the descriptor is a single pointer to the base of the structure memory, where the elements of the array are stored. Also the implementation of arrays presented here considers arrays as *strict* data structures, i.e. subsequent operations can use an array only if all the elements of that array have been stored. We are currently investigating efficient implementation schemes for structure memory operation able to support *eager* evaluation of array operations. *Eager* implementations schemes for the argument fetching architecture has been proposed in [29].

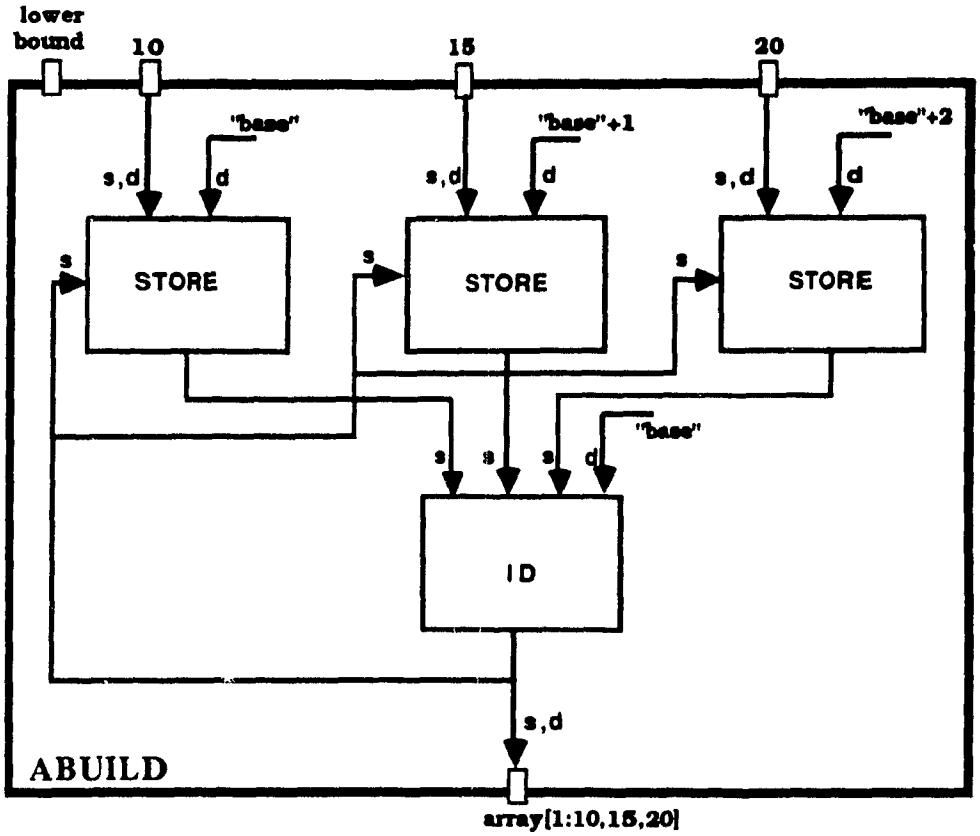

Figure 4.10: Machine graph of one-dimensional ABuild operation

In this section we will present the machine graph for three array operations. These array operations includes two array construction operations and an array *select* operation. In a latter section, two other operations for scattering and generating an array inside a loop, will be presented.

4.5.1 Translation of the Array Build Operation

The array build operation constructs an array by providing a lower bound along with the values of the elements of the array that will be constructed. The SISAL compiler translates the array build expression into a simple node named *ABuild*. The first input port of this node provides the lower bound of the array, while the rest of the input ports provides the elements of the array. Note that for this node, although the number of input ports varies, it is known at compile time.

Translating a one-dimensional Array Build operation

Consider the expression *array*[1: 10, 15, 20] which constructs an one dimension array with lower bound of one and size three. Figure 4.10 presents the machine graph of the above SISAL expression. In the above example, the elements of the array are: 10, 15 20. As mentioned earlier, the number of the input ports and the type of the input data for each *ABuild* node is known at compile time. Therefore the compiler knows the size of the array and can calculate the address in the *structure memory*, where each element will be stored.

The "STORE" machine instructions are used to store the input data into the structure memory location, specified by the statically calculated from the compiler address. The "ID" is used as a synchronization instruction; it is executed when all the elements of the array have been stored. Also the "ID" node is used as a gate that passes the "base-address" of the new array, to successor tuples that uses the elements of the array.

Translating a multi-dimensional Array Build operation

In the case where the data in the input ports of an *ABuild* operation are arrays, a multi-dimensional array is constructed whose dimension is greater by 1, compare to the dimension of the input arrays. Note that the input arrays should have the same dimension and the same size, and their elements should have the same data type.

Figure 4.11 presents the machine graph of the expression *array*[1: A, B], where A and B are arrays of the same size. Their base address in the structure memory is denoted by "A" and "B" respectively. Their size is referred as "input array size" In this mapping scheme, for every input array, an "IGEN" operation is associated, which generates the indexes of every element of the input array. The "MULT" instruction is generated only if the elements of the input arrays are doubles; it is used to calculate the relative address of the input elements. An array read operation is translated into a pair of instructions: (1) an "ADD", which computes the actual element address from the base address of the input array and the generated index, and (2) a "LOAD", which uses this address to perform the actual memory transaction. Similarly, a pair of "ADD" and "STORE" operations are used to store the elements of each input array to the right location reserved for the new array. Note that the elements are always stored in a *row-major-order*.

As with the previously presented mapping scheme, an "ID" instruction is used as a synchronization tuple, which fires when all the elements of the array have been stored. At

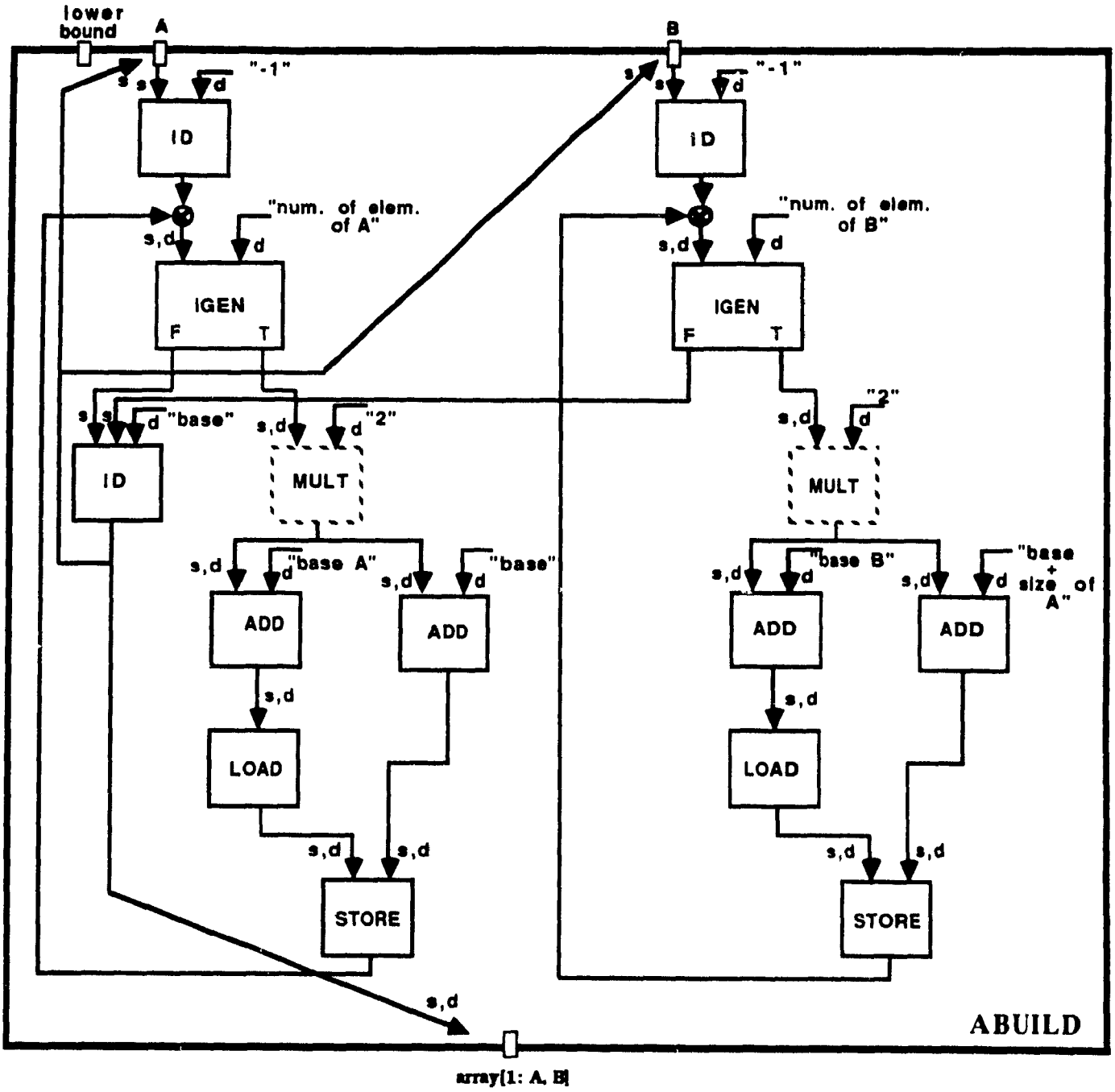

Figure 4.11: Machine graph of multi-dimensional ABuild operation

that point of time, all indexes have been generated, and therefore all the "IGEN" tuples return a "false" conditional code, which activates the execution of the "ID" instruction. The "ID" is also used to pass the base address of the generated array to the rest of the tuples that use this array.

4.5.2 Translation of the Array Fill Operation

The array fill operation creates an array with a given range and all elements equal to a given value. The SISAL compiler translates this operation to a simple node named *AFill*. *AFill* has always three input ports It creates an array filled with the value given on the third port. The integer on port one gives the lower bound, and the integer on port two gives the upper bound of the array being build Since we are concern with static arrays, the values of the first two ports should be known at compile time. The outermost range of the constructed array is equal to the range defined from the first two ports, multiplied by the size of the input element

The element on the third port can be either a scalar or an array Similarly with *ABuild*, two mapping schemes are used for this operation depending on the data type of the input element. The mapping schemes for both cases are presented in figures 4.12 and 4.13. In figure 4.12, the input scalar is stored so many times as this is defined from the range generated by the "IGEN" Therefore, the input range in this case also defines the number of elements of the array

In case of the construction of a multi-dimensional array as this is presented in figure 4.13, two index generator instructions are used The "IGEN" in the top of this figure generates the indexes that are used to calculate the base addresses of the subarrays, where the elements of the input array will be stored. A subarray in this operation always corresponds to the input array. The input array is stored in contiguous space in the structure memory, as many times as this is defined from the input range. The "IGEN" in the bottom of figure 4 13, repeatedly generates indexes that are used to access the elements of the input array. This "IGEN" generates the same set of indexes as many times as this is defined from the range in the first two input ports. Finally note that like in the previous mapping, the elements of a multi-dimensional array are always stored in a *row-major-order*.

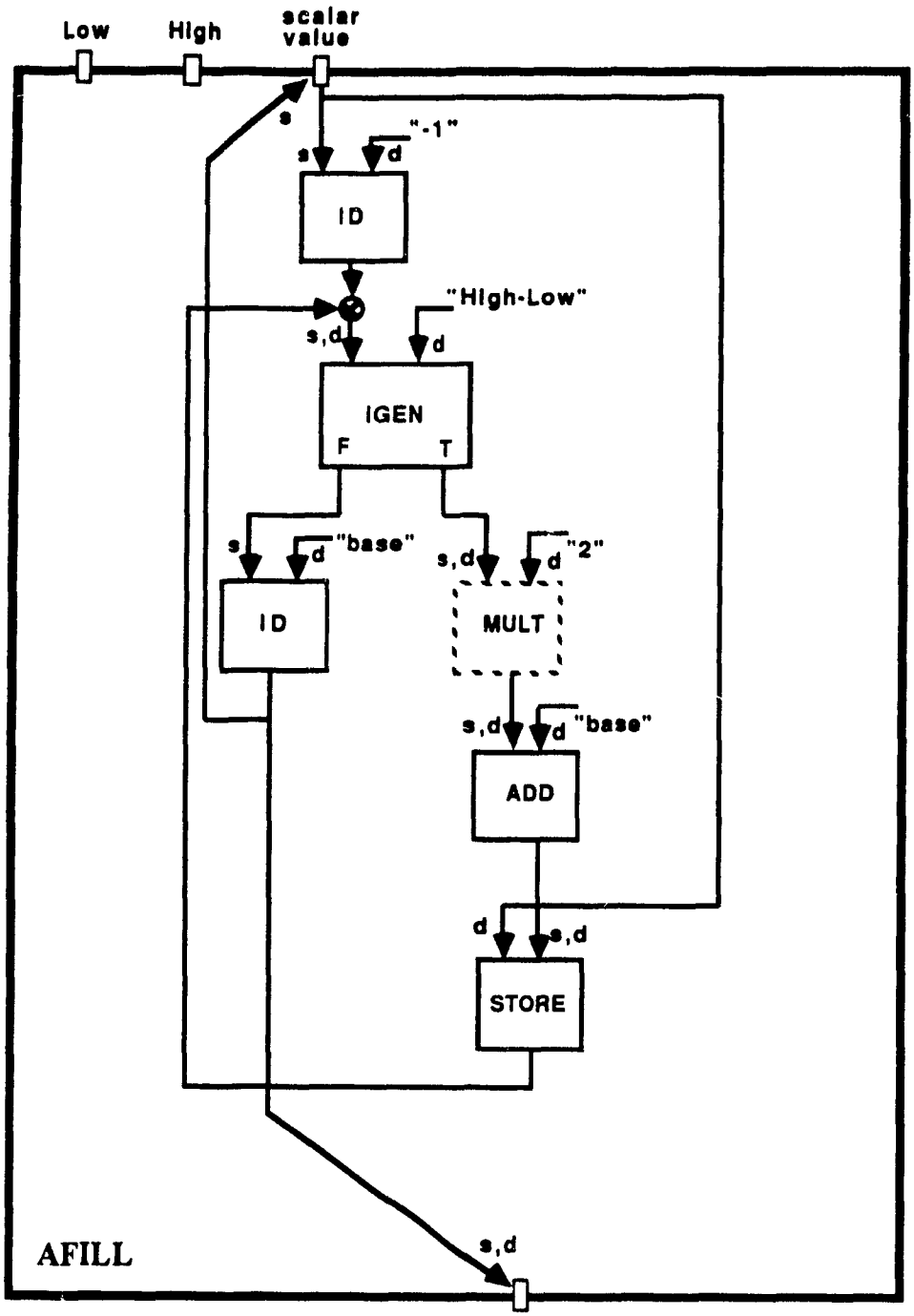

Figure 4.12: Machine graph of one-dimensional AFill operation

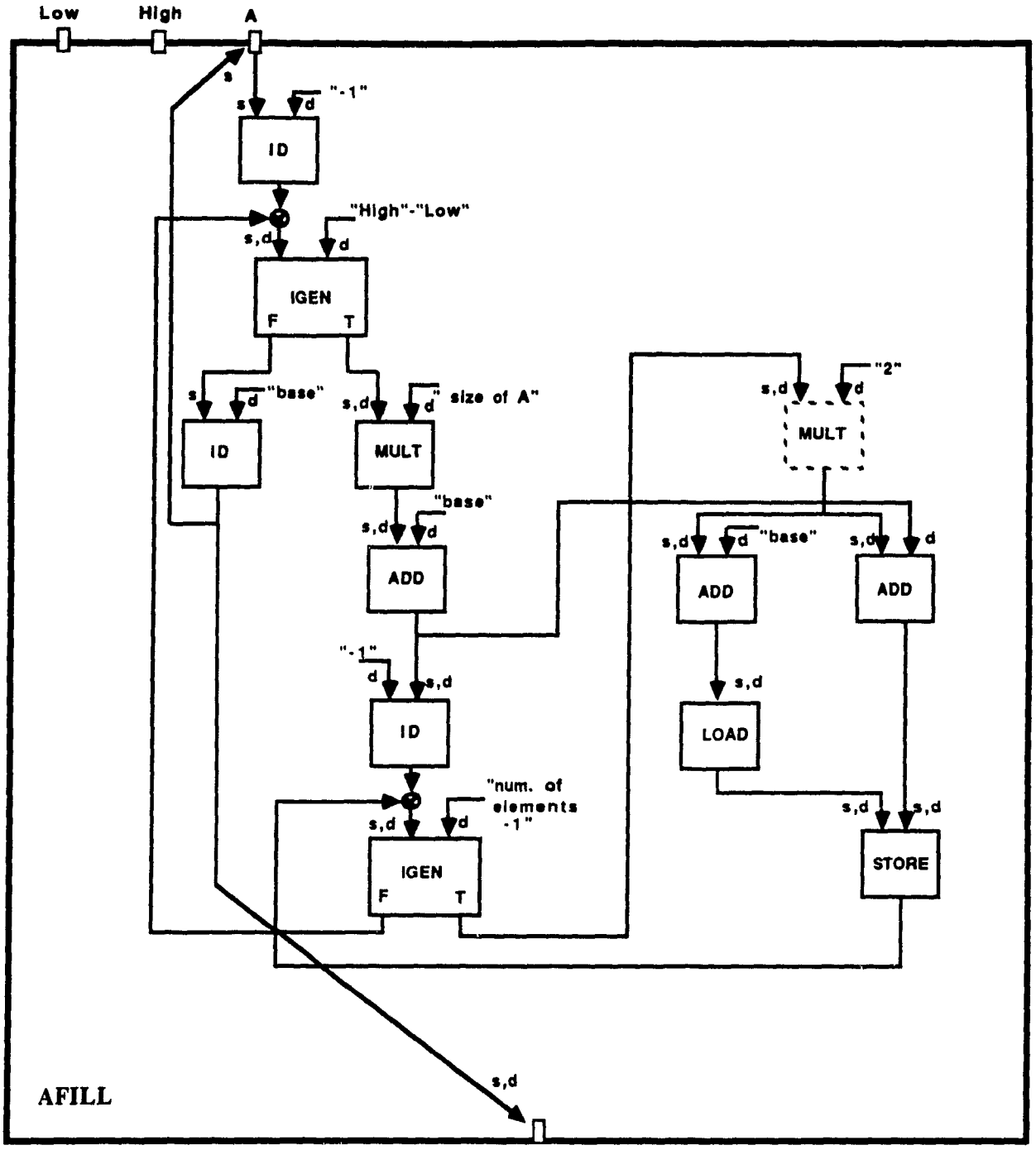

Figure 4.13: Machine graph of multi-dimensional AFill operation

4.5.3 Translation of the Array Selection Operation

The array selection operation is represented in the HDDG by the *AElement* simple node. This node extracts the element of an array at a given index position. Note that only one level of subscripting is done by this operation.

The mapping of the *AElement* node appears in figure 4.14. Notice that the base address of the input array is given in the first input port, while the index value is given in the second input port. Depending on the size and the dimension of the array, the actual index value is calculated from the input index. The first "MULT" instruction will be present only if the elements of the input array are "doubles". The second "MULT" instruction will be generated only if the input is a multi-dimensional array, this instruction produces the relative base address of the selected subarray. The actual address is computed from the "ADD" operation. If the input array is one-dimensional, the "LOAD" operation retrieves the scalar element from the structure memory; otherwise the "LOAD" operation is omitted, and the calculated address represent the base address where the selected subarray is stored.

4.6 Mapping Conditional Expressions

In HDDG the conditional expressions are represented by the *Select* compound node. The *Select* node implements the "*if-then-else*" two way selection, and it can recursively contain other *Select* nodes. Every *Select* compound node consists of three subgraphs: the *Selector*, the *False* and the *True* subgraph. The *Selector* subgraph yields a boolean that selects the appropriate branch subgraph. The *True* and *False* branch subgraphs contain the body of the expressions that should be executed when the boolean value is "true" and "false" respectively. The arity of the conditional expressions varies according to the number of results that should be returned.

Consider a typical conditional expression such as:

if b(x) **then** f(y) **else** g(y).

A naive implementation could allow only one activation of the conditional to be in progress at a time. In such a scheme, the conditional expression can be reactivated only if its previous activation has computed all the expected results. Here we present a *pipelined*

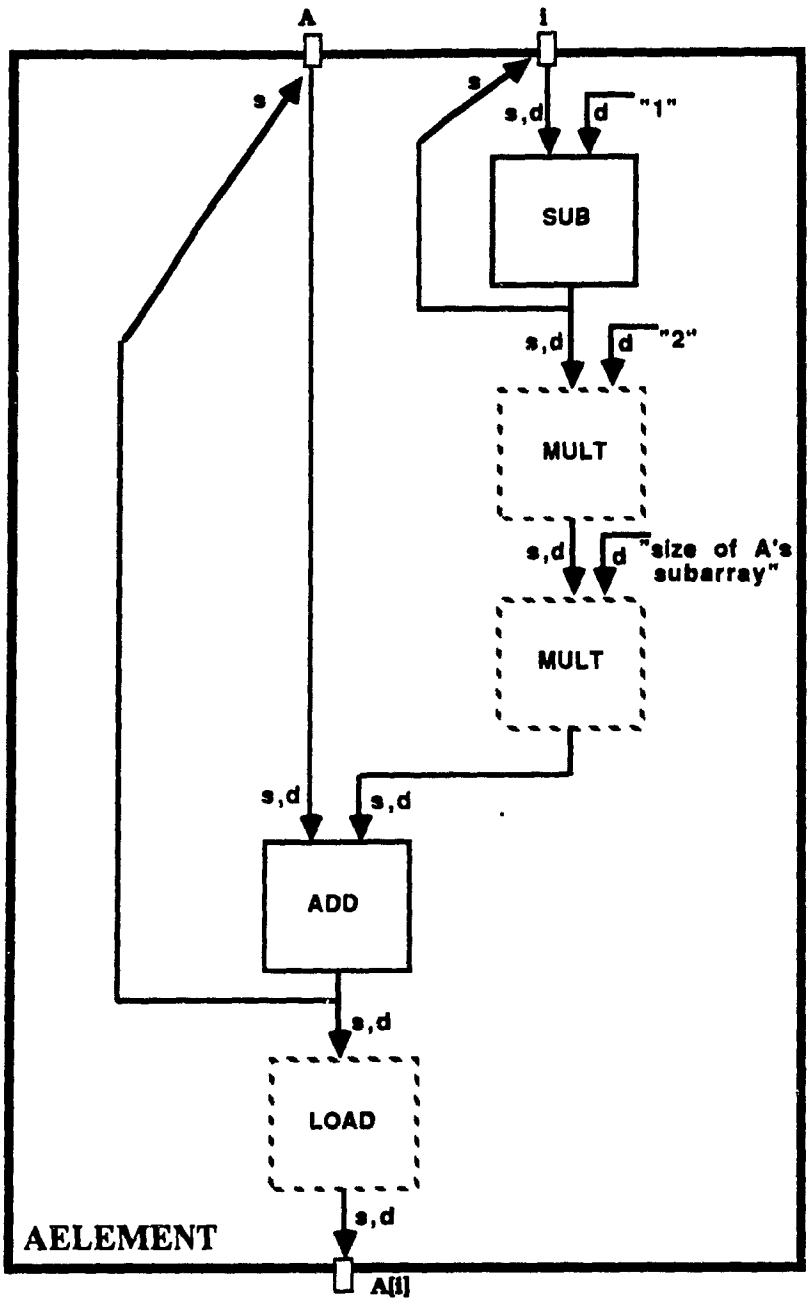

Figure 4.14: Machine graph of the AElement operation

mapping scheme for the conditional expressions This mapping scheme allows more than one activation of the same conditional expression to be active at the same time [31]

Figure 4 15 shows the machine graph for the above conditional expression as this has been derived by translating the *Select* compound node following the pipelined mapping scheme. For demonstration purposes, this figure presents the *Selector* subgraph to be empty; in general, part or the whole boolean expression could be inside this subgraph Notice that since the arity of the test expression b(x) is always equal to one, it is possible to identify the last tuple of b(x) and labeled it as $Last_b$ in the figure Tuple y is the only tuple whose result value is imported in both arms of the conditional expression Although its value is used inside these arms, y signals to $Last_b$ tuple According to the value that is produced from the boolean expression b(x), $Last_b$ signals to either *False* or *True* subgraph.

The "ID" instruction at the top of each branch subgraph, have been introduced by the code generator such that the arity of each value that is imported in the two branch subgraphs of a conditional expression will be always one. In general, for every value that is imported and used inside these two subgraphs, the code generator associates one "ID" tuple in each subgraph These "ID" tuples serve as gates that passes all the imported values to the rest tuples of the corresponding conditional arm.

The pipelined execution of the conditional expressions couldn't be possible without the existence of a FIFO, which stores the values as they are produced from the boolean expression of the conditional Therefore the FIFO buffer is needed to hold the results of tests while the corresponding computations are performed by the conditional arms Depending on boolean values stored in the FIFO, the tuples inside the appropriate branch that produce the final result values are signaled Without lost of generality, in figure 4 15 we assume both f(y) and g(y) return one result value The two last tuples of f and g that produce the final result value, are labeled by $Last_f$ and $Last_g$ respectively Note that, $Last_f$ and $Last_g$ are ordinary tuples in the graph, and must have the same result register

Finally for every value that is returned from the conditional expression, the code generator associates one "ID" tuple that export the result value to the tuples outside the conditional construct Each "ID" tuple receives one signal from each pair of "last" tuples that produce the corresponding final value in the *True* and the *False* subgraph respectively Since only one set of "last" tuples will be activated at a time in either the *True* or the *False* subgraph, a non-deterministic merge is needed to send a signal from each pair to the associated "ID" tuple In figure 4.15, where only one result is returned, the pair of "last" tuples is denoted by $Last_f$ and $Last_g$ and the non-deterministic merge by $\odot$

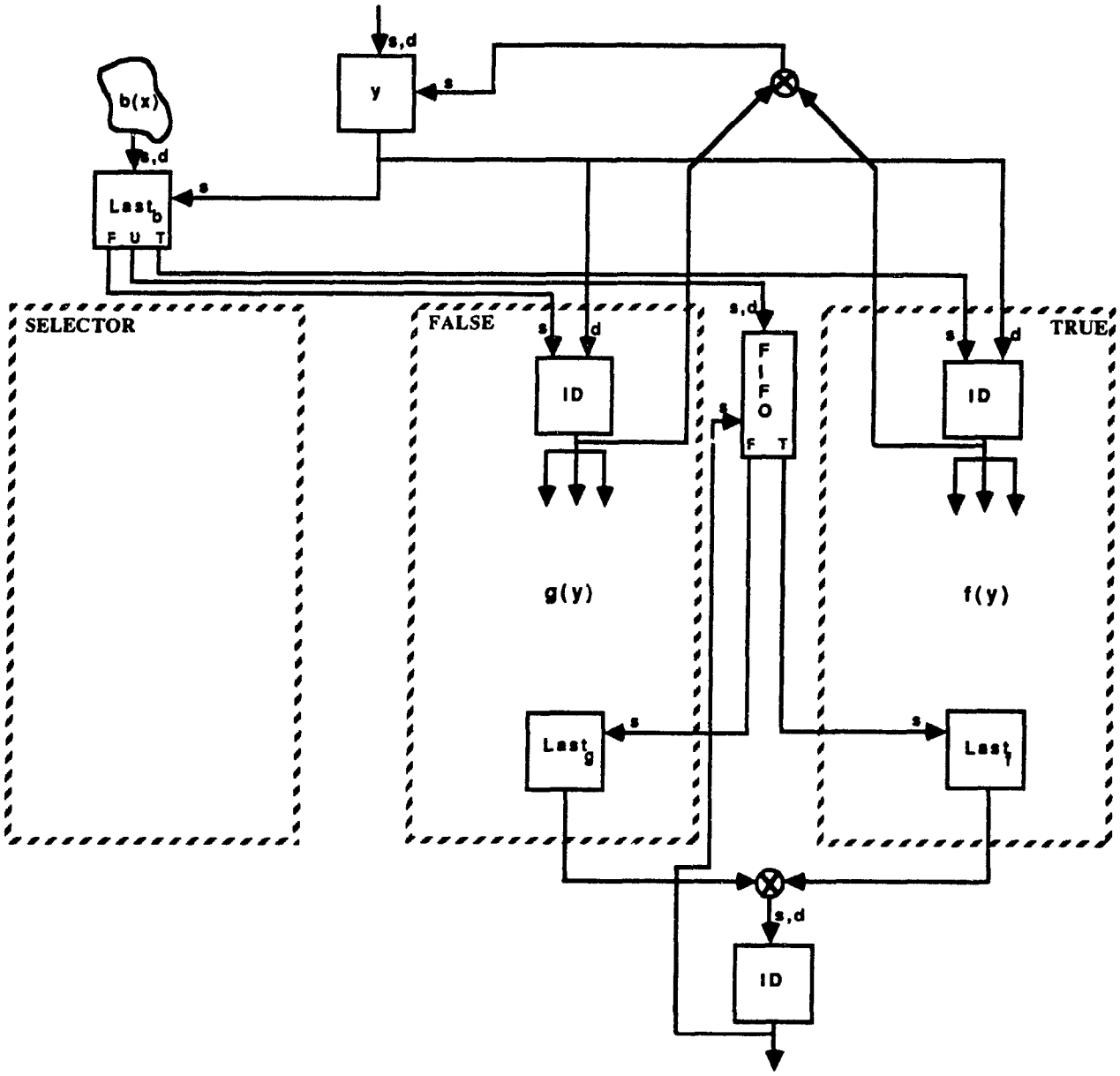

Figure 4.15: Machine graph of the Select compound node

Notice that for the pipelined mapping of the conditional expressions, the compiler generates code for a number of "ID" instructions that does not serve any computational purpose. The overhead of introducing these "dummy" tuples is tolerated from the simplicity, and the modularity of the implemented mapping scheme, as this has been presented here.

4.7 Mapping Forall Expressions

Recall that SISAL *for-in* is a loop construct which states explicitly that there are no data dependencies among the iteration of the loop. SISAL compiler translates this construct to a compound node named *Forall* The *Forall* node in HDDG is used to denote independent execution of multiple instances of an expression. It has three subgraphs.

1. the *Generator*, that produces values for each instance of the loop body;
2. the *Body*, which contains the expression to be evaluated;
3. the *Returns*, that gathers all the results computed from the distinct body instances.

Here we apply the *software pipelining* scheme as this has been described in chapter 2. According to this scheme, instead of providing multiple copies of the body, only one copy is used and the parallelism is exploited by means of pipelining. The realization of the software pipelining mapping scheme is presented in the rest of the chapter, where the translation for each operation in the *Generator* and *Returns* subgraphs is discussed.

4.7.1 Translating Operations Dealing with Multiple Values

In HDDG, the class of nodes dealing with multiple values belongs to one of following two categories:

1. nodes that for a given input generate a sequence of values;
2. nodes that operate on a sequence of values.

The first category of nodes appears only in the *Generator* subgraph of a *Forall* compound node. The only nodes that can appear in this subgraph are two (1) *RangeGenerate*, which generates a sequence of indexes, and (2) *AScatter*, which scatters the elements of an input array upon its first dimension. The machine graphs of these two nodes are presented latter in this section.

In a *Forall* compound node, the second category of nodes appears only in the *Returns* subgraph. Recall from the previous chapter, that the returns expressions that allowed in SISAL Kernel are the **array of**" and "**value of**" expressions. SISAL compiler translates these two expressions to *AGather* and *Reduce* simple nodes respectively. The mapping of these two nodes is also presented latter in this section.

Translation of the Range Generate operation

The *RangeGenerate* node produces a sequence of integers in the (inclusive) range that is specified in the two input ports. The low value of the range is specified in the first port and the high in the second port. If the value at port one is greater than the value at port two, then the output value is null.

The machine graph for this operation is given in figure 4.16. An "IGEN" operation is used to generate the specified range of integers. As mentioned before, for a given range with low value of "a" and high value of "b", "IGEN" generates the range from "a+1" up to "b" integer values. Therefore, in order to include the low index value, a "SUB" operation is used to subtract by 1 the value which is provided in the first port. Also the three "ID" tuples are used to implement correctly the signaling scheme of the "IGEN" instruction.

Two instructions in the *Generator* subgraph always consist the pair of tuples that "drives" the execution of the *Forall* loop. These tuples are called *driving tuples* and are specified by the "SUB" and the "ID" instructions at the top of figure 4.16. For every new activation of the *Forall*, these two tuples hold the values that are used from the reduction operators in the *Returns* subgraph to specify the range of the accumulation.

All the inputs of the *Forall* loop whose values are used in the *Body* and the *Returns* subgraphs of the loop, signal their availability to one of the *driving tuples*. The *Generator* starts to produce the specified sequence of values, only when all signals from the inputs have been arrived. Also a new activation of a loop is allowed only when all the tuples

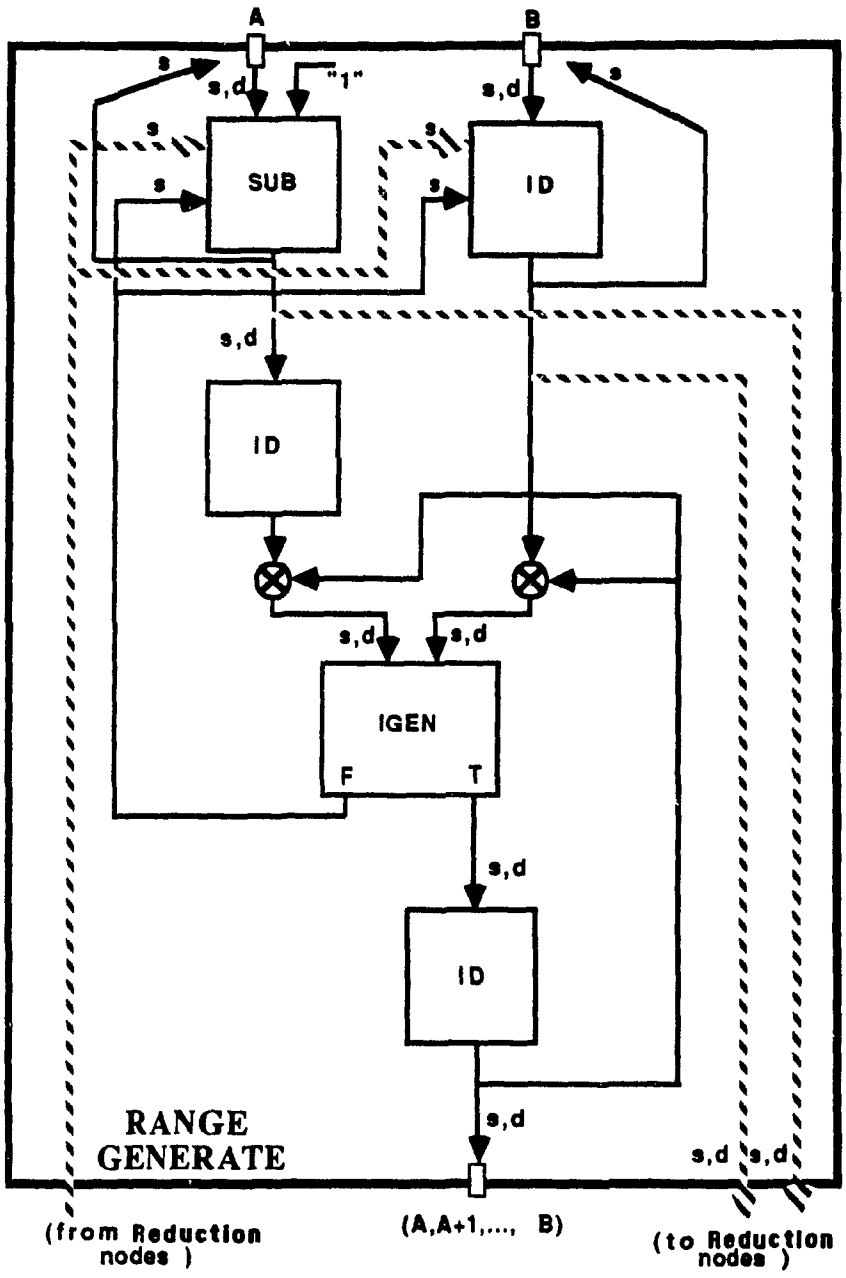

Figure 4.16: Machine graph of the Range Generate operation

of the reduction nodes in the *Returns* subgraph that specify the end of the accumulation, signal back to the driving tuples in the *Generator* subgraph.

For every *Forall* loop, there is one pair of driving tuples. The two driving tuples of a *Forall* can be found in either a *Range Generate* or an *AScatter* mapping scheme, since these two are the only simple nodes that can exist in the *Generator* subgraph. The code generator always picks one node in the *Generator* subgraph and associates the corresponding tuples to be the driving tuples of the loop.

Translation of the Array Scatter operation

SISAL compiler translates the array scatter operation into a node named *AScatter*. Similarly to the *RangeGenerate*, *AScatter* appears only in the *Generate* subgraph of a *Forall* loop. *AScatter* has only one port which holds the base address of the input array. The elements of the input array are placed sequentially in the first output port. Their corresponding index values are placed on the second output port

Figure 4.17 presents the machine graph of the *AScatter* node. This operation repeatedly applies the array select operation to the outermost dimension of the input array. Note that the range of the loop, is specified by the size of the outermost dimension of the input array, as this is represented by the name "High" in figure 4.17. The "IGEN" generate this range of indexes, and depending on the element size and the dimension of the input array, the actual index value is calculated. Like the array selection operation, the first "MULT" instruction will be present only if the elements of the input array are doubles, while the second will be generated only if the input is a multi-dimensional array. The "LOAD" operation is generated only if the input array is one-dimensional array, to fetch the scalar values from the structure memory. The "ADD" operation in right side of figure 4.17, generates the indexes that are placed on the second port, by incrementing by 1 all the values generated from "IGEN".

The tuples in the mapping of figure 4.17 that can serve as the driving tuples of the loop, are the top two "ID"s. The functionality of these tuples in the computation of the loop, has been explained in the *RangeGenerate* mapping presented previously.

Translation of the Reduce operation

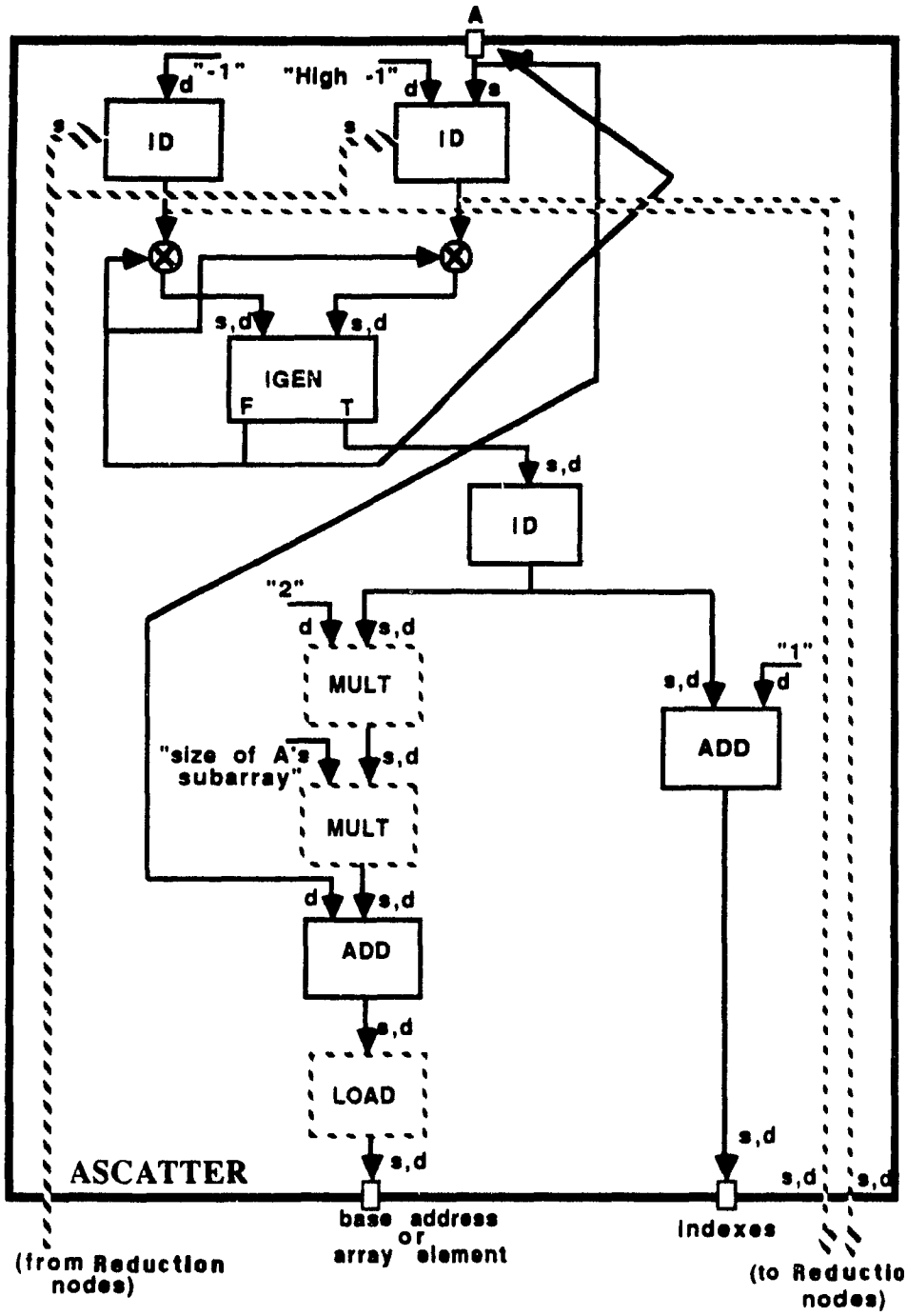

Figure 4.17: Machine graph of the Array Scatter operation

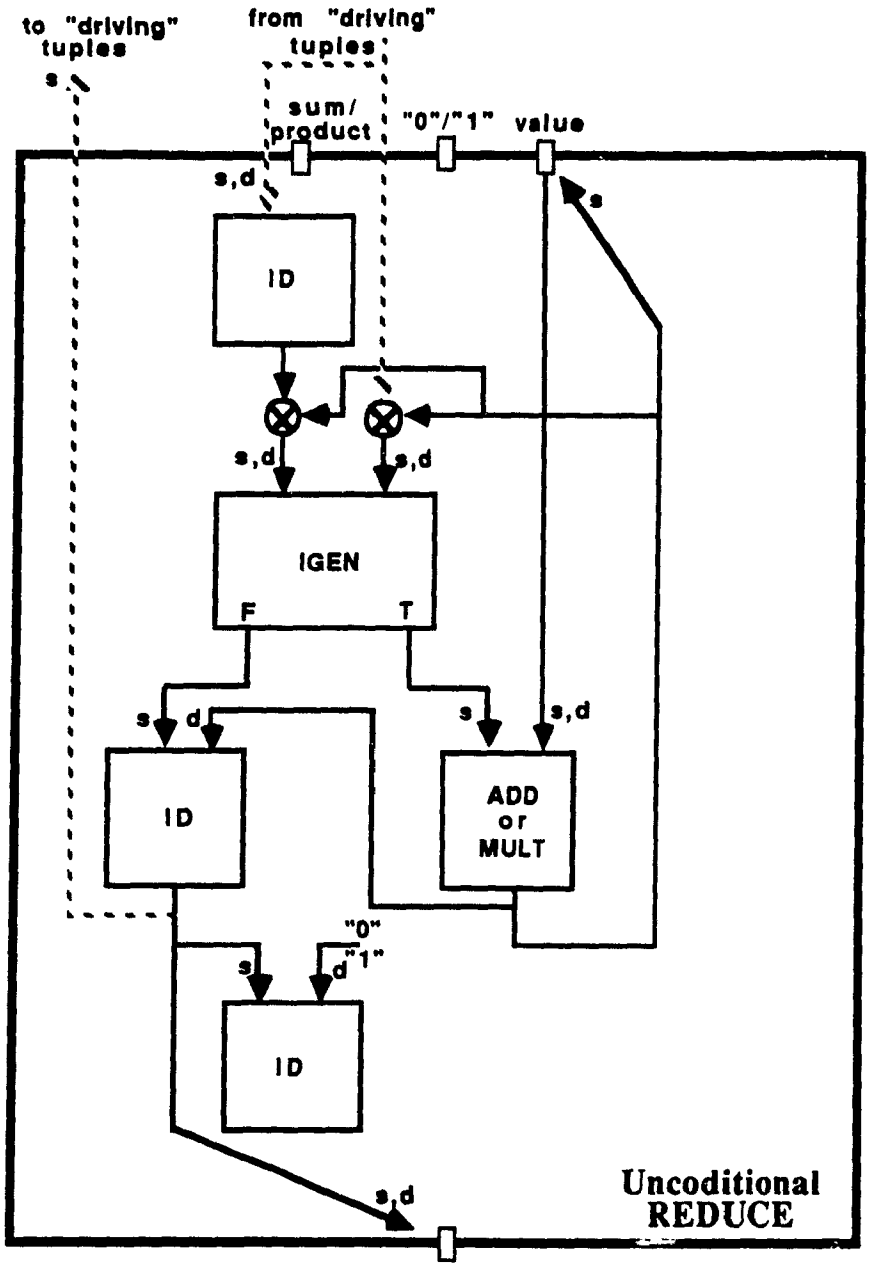

Figure 4.18: Machine graph of the Unconditional Reduce operation

As mentioned earlier, the "*value of*" expression in SISAL is translated to a *Reduce* simple node in the *Returns* subgraph of a *Forall* compound node. The *Reduce* node has four input ports. The first port defines the function that will be applied to the incoming values. These functions correspond to the four *reduction operators* that have been defined in the previous chapter. Recall that these functions are: (1) *sum*, (2) *product*, (3) *least*, and (4) *greatest*. The second input port holds the initial value of the reduction operator. The third port provides the values that are produced from the *Body* subgraph, the reduction function is applied to this sequence of values. The fourth input port is optional; if present, it provides a sequence of boolean values, which defines if the corresponding values in the third port will be used in the reduction.

Figures 4.18 presents the machine graph of the *unconditional Reduce* operation for the "*sum*" and "*product*" reduction operators. The mapping of the other two reduction operators is similar. In this scheme the reduction operation applies to all the values that are coming in the third port. During one activation of a *Forall*, "IGEN" activates so many times the reduction instruction "ADD" or "MULT" as it is defined from the range of this particular activation. The range is provided from the two *driving tuples* in the *Generator* subgraph. The "ADD" instruction is generated when the reduction function in the first port is "*sum*"; the "MULT" is generated when the reduction operation is "*product*".

The "ID" instruction at the top of figure 4.18 is used to initialize the value of the "IGEN" instruction. This value is passed from the driving tuple that produces the low range of the loop. At the end of the accumulation, "IGEN" returns a "false" conditional code, which activates an "ID" instruction that passes the accumulated result value to the other tuples outside the *Forall*. It also signals to the *driving tuples* in the *Generator* subgraph to denote the end of the accumulation. Finally, it signals to a third "ID" instruction which is used to initialize the value of reduction instruction for the next activation of the *Forall*. If the reduction operation is "ADD", then the value is initialized to 0, if it is "MULT", the initialized value is 1. A similar mapping applies for the unconditional *least* and *greatest* reduction operations.

Figure 4.19 presents the machine graph of a *conditional Reduce* node for the same reduction operators. This mapping contains all the instructions presented for the unconditional node; in addition, there is an "ID" instruction which is executed whenever the two corresponding input values are presented in the third and the fourth input port. If the boolean value on the fourth port is "true", the accumulation instruction is activated and the value in the third port is added or multiplied to the previously accumulated value;

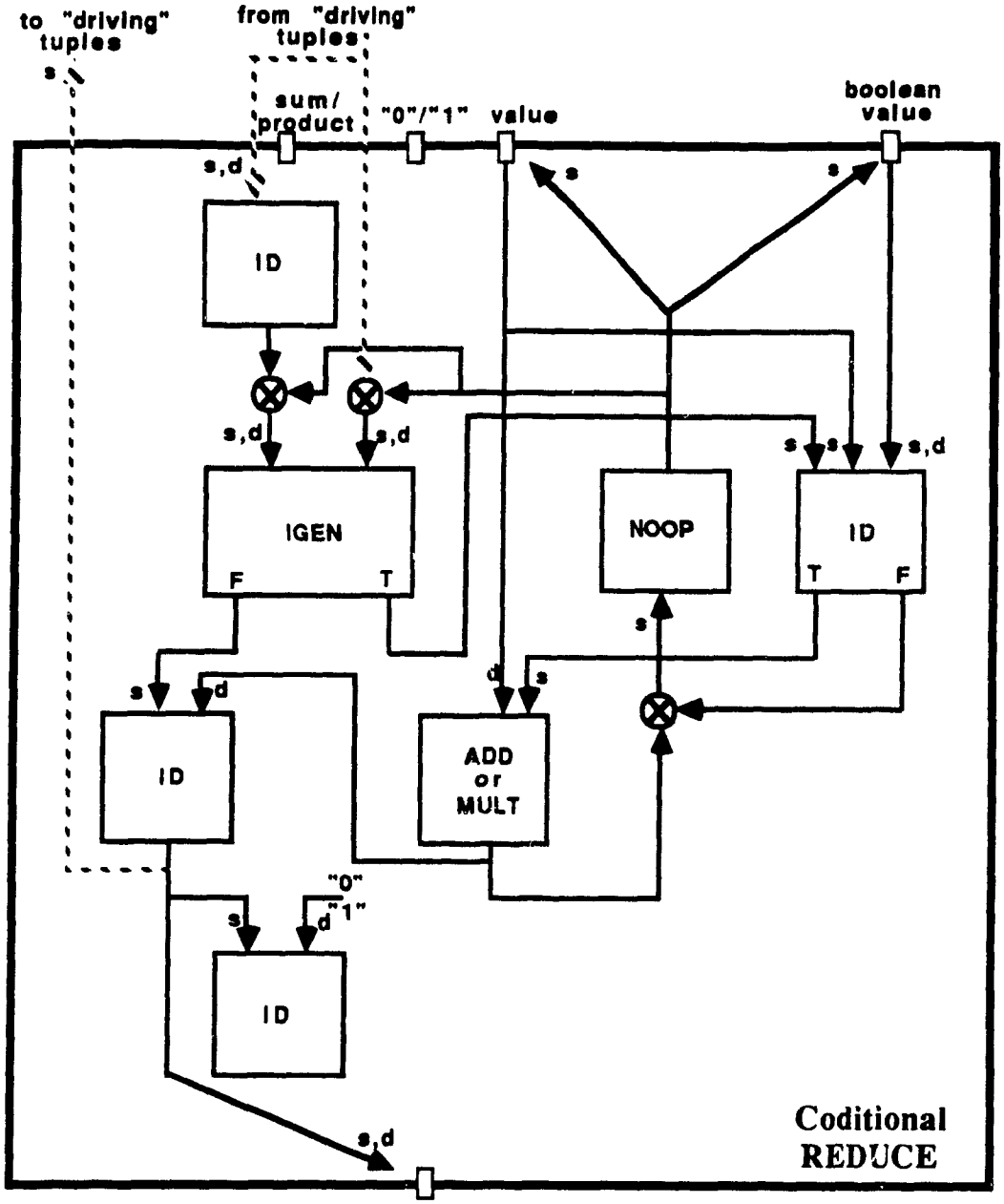

Figure 4.19: Machine graph of the Conditional Reduce operation

otherwise, the value in the third port is discarded. The two mutual exclusive signals are merged in a "NOOP" instruction which is used to send "acknowledgment" signals to the tuples that generates the values in the third and fourth port. A similar mapping applies for the conditional *least* and *greatest* reduction operations.

Translation of the Array Gather operation

Recall that *Forall* is also used for building arrays. The returns expression "*array of*" is translated from the SISAL compiler to an *AGather* simple node, which always exists in the *Returns* subgraph of a *Forall* loop. This node builds an array from the values provided on its second input port. The first input port gives the lower bound of the array.

Figure 4.20 presents the machine graph for the *AGather* operation. Notice that since we are concerned with static arrays, their size is known at compile time. For every element of the array, "IGEN" activates the path of the computation that is used to store the incoming values in the structure memory. The "MULT" instruction multiplies the index that is generated from the "IGEN" by 2, only if the elements are doubles. This value defines the relative offset in the structure memory, where the element will be stored. The actual address is calculated from the "ADD" instruction, which adds the relative index to the base address of the generated array. The "STORE" instruction is used to store the element that is provided from the third port, to the absolute address that has been calculated from the "ADD".

The top two "ID" instructions are used to pass the low and the high index value to the "IGEN". The "ID" in the middle of the figure 4.20, is activated when all the elements of the array have been stored. It is used to signal the top two "ID" instructions that a new computation can start, it also passes the base address of the generated array to the instructions outside the *Forall*.

Note that in the case of the construction of a k-dimensional array in a *Forall* compound node, a number of k *AGather* operations will be combined, each one responsible for building one dimension of the array. In order to avoid the overhead of copying the arrays that every *AGather* builds in the intermediate steps during the construction of the final array, all the elements are gathered in the outermost *AGather*. This implies that all the innermost *AGather* nodes, operate as gates that pass the elements to the next higher dimension. The penalty paid is that this scheme does not allow the exportation of a subarray from an inner *AGather* which exactly follows the philosophy of "monolithic arrays". This should be

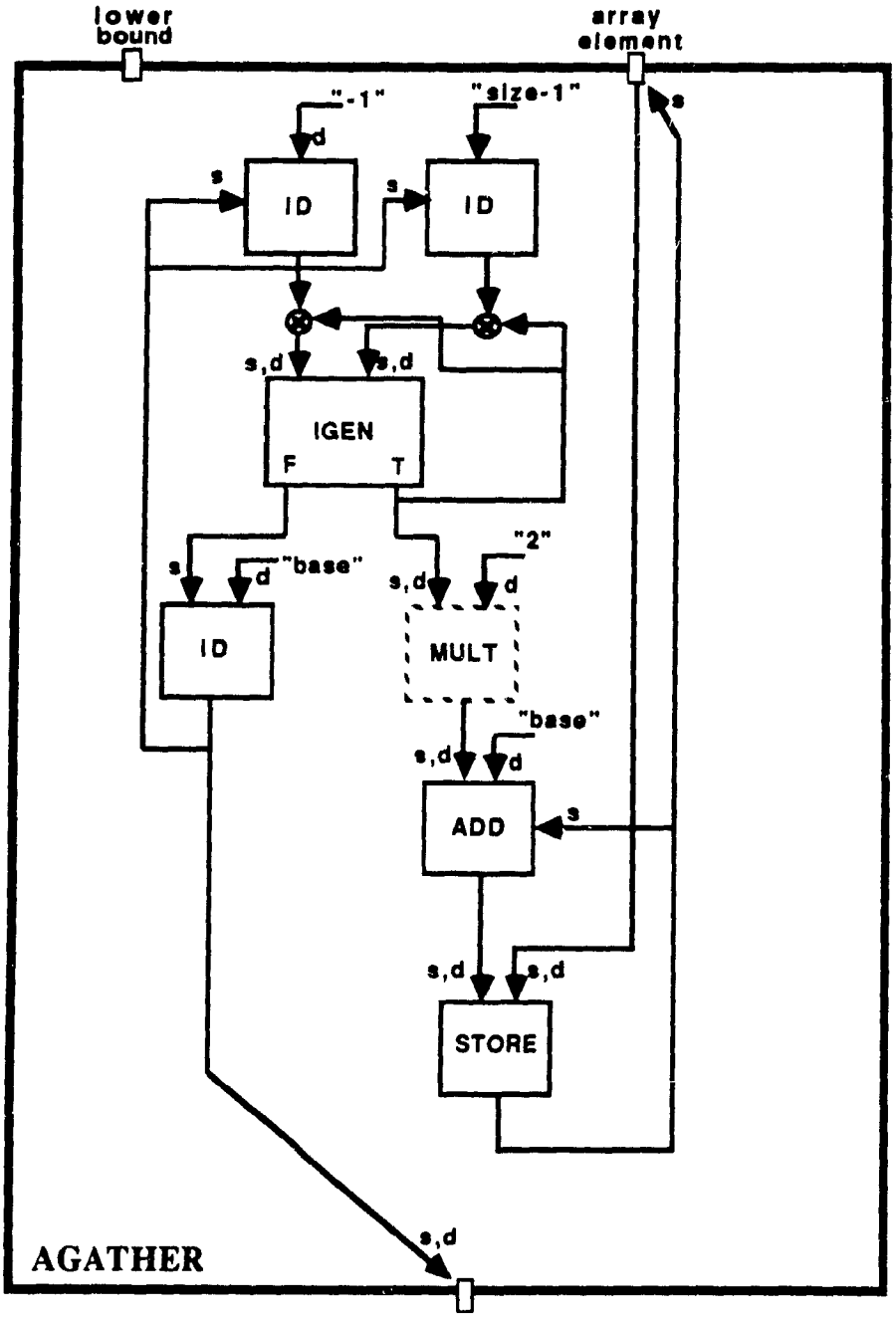

Figure 4.20: Machine graph of the Array Gather operation

consider as a minor restriction compared to the efficiency of generating multi-dimensional arrays without copying.

4.8 Summary

HDDG is an intermediate form of a SISAL program; it stands between the original code and the final machine code. The HDDG form of a program provides an insulation from the features of the programming language and the idiosyncrasies of the target architecture. It can also be used as a framework, where certain kinds of program transformations and optimizations can be applied

The use of data dependence graphs as an intermediate representation of SISAL programs is the basis of generating code for the argument fetching architecture We focus on generating code for the SISAL Kernel, which is the subset of SISAL that has been described in the previous chapter Three are the main principles of generating code for this architecture.

1. preserving the well-behaved property of the graphs;
2. minimizing the number of signals while extracting enough parallelism to keep the resources busy;
3. applying *software pipelining* to loops and to conditional constructs

Well-behaved dataflow programs are these programs that for a unique set of input values, a unique set of output values is determined [14]. It has been proven that in Dennis-Misunas static dataflow architecture [38], dataflow graphs which contain elementary operators, conditional and loop constructs, are well behaved. The machine graphs of the argument fetching architecture presented here, are differentiated from the traditional static dataflow graphs from the fact that signals and data are separated Although we don't have a formal proof of the well-behaved property for the argument fetching architecture, a lot of effort has been invested such that for each separation of signals and data, the mapping schemes will preserve the data dependencies, and assure the correctness and the determinacy of the programs.

Considering the limits in the processing ability of the machine, we try to keep a balance between the number of the generated signals and the potential parallelism of the program. From simulation studies, it has been proven that a key factor which determines the utilization of the argument fetching architecture, is related with the signal traffic. The finite signal processing ability of the DISU, imposes a limitation to the number of signals that can be present in a cycle. A more detailed analysis of how this factor affects the performance of the programs, is discussed in chapter 5.

Finally, the encapsulation of the principle of software pipelining is twofold: (1) to allow multiple activations of compound constructs such as conditionals and loops, (2) to prevent saturation of the machine resources, while making better use of the data memory space. Applying dataflow software pipelining to loops, allows more than one cycles of the loops to be active at a time. The parallel execution using this technique, is realized by the overlapping activations of the same loop in a pipelined fashion. The effects of applying this technique in the overall performance, will be presented in the following two chapters.

Chapter 5

Performance Evaluation

The ideas and techniques presented in the previous chapters have been investigated and validated in a testbed suite for the argument fetching architecture. We choose, as a benchmark, a set of Livermore Loops[19,40], and study the performance of these loops through simulation. Nine of the set of Livermore Loops, all containing *forall* loops, are selected for illustration. The selected set of Livermore Loops is representative of the average performance gain by several other machines for the whole set of the 24 Livermore Loops [37]. By executing compiled code on the testbed, the efficiency of the software pipelining schemes and certain optimization methods have been evaluated. We have seen a substantial speedup, and a high utilization factor of the execution unit. This is achieved by employing only the basic software pipelining for the loops tested, as this has been demonstrated in chapter 2 and chapter 4. In many cases, a balancing technique can be employed as a further optimization on top of the basic software pipelining. The effect of balancing is shown here through an in-depth study of the simulation results of one of the Livermore loops. A similar analysis for an one-dimentional Laplace solver, appears in [32].

A detailed analysis based on the simulation results is presented, addressing two key architectural factors – the signal capacity and scheduling mechanism for enabling instructions – that substantially influence the efficiency of the running programs. These architecture factors represent the ability of the machine to exploit fine-grain parallelism. By experimenting with different architectural configuration the relation between program structure, compiler optimization techniques and the architectural features is addressed.

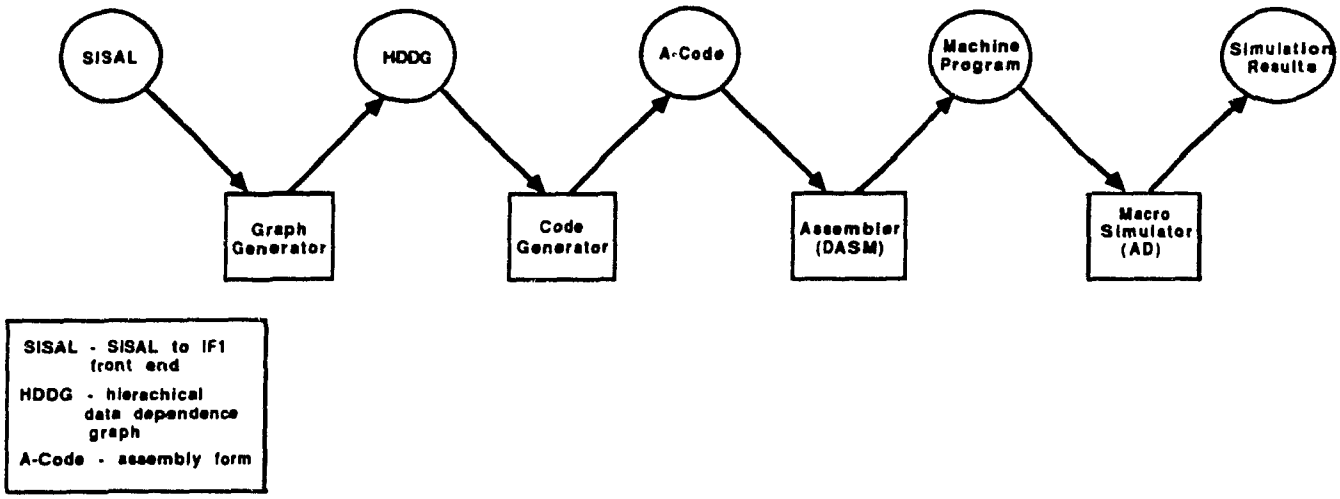

Figure 5.1: The simulation testbed

5.1 The Testbed Suite

An overview of the compiler/architecture testbed that has been used throughout the simulation tests, is presented in Figure 5.1. In chapter 4, we show how starting from the SISAL front end, SISAL Kernel programs are translated into a hierarchical data-dependence graph (HDDG), and subsequently the way that the code generator translates the machine-independent HDDG forms into machine graphs (A-Code). DASM is an assembler for A-code, the assembler form for the argument fetching dataflow architecture. DASM stands for data-driven assembler, and it is used to generate executable code for a macrosimulator of the architecture.

The macrosimulator AD can be viewed as a highly instrumented interpreter that represents simulation at the major function level of the architecture [46,47]. The six pipeline stages of the PIPU and the two major units of the DISU are implemented by distinct self-contained code units, which simulate the functionality of each hardware unit. No specific restriction has been made, at the simulation level, to define the timing relation between the PIPU and the DISU machine cycles. The current implementation considers that these two units have the same machine cycle, although in a real machine, timing constraints of the hardware technology may change this proportion.

Two of the simulator's parameters that are used to define a processing element are the

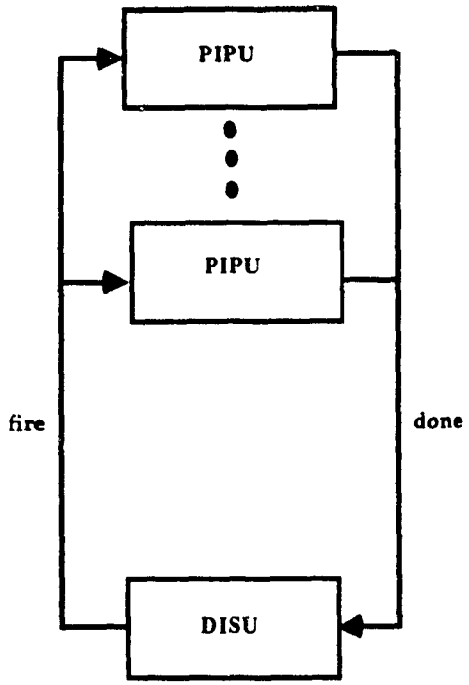

Figure 5.2: Number of PIPUs

following:

- **-np** <*num*>. This parameter sets the number of the PIPUs. The default value is one.
- **-nc** <*num*>. This parameter sets the number of count signals that are allowed per cycle in the DISU unit. The default value is four.

Figures 5.2 and 5.3 illustrate the above two parameters. These two numbers are used to set up the machine configuration each time a test is conducted. By increasing the number of PIPUs, we increase the execution capacity of the processing element proportionally. The second parameter is used to control the propagation of signals in the DISU unit. The more signals that can proceed in a cycle, the more enable instructions can become available in a cycle. The number of signals that can be handled in a cycle by the DISU defines its *signal processing capacity* (or simply, *signal capacity*). The effects of these architectural parameters are investigated in more detail in a latter section.

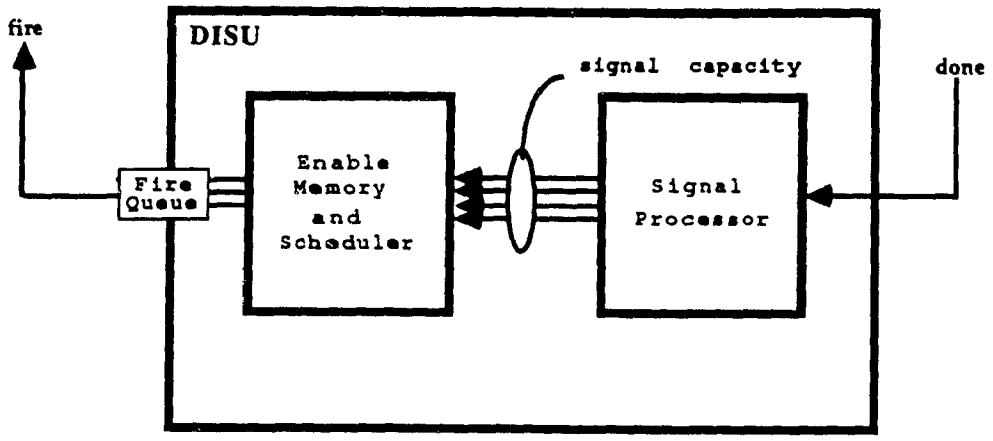

Figure 5.3: The DISU processing capacity

5.2 Performance Metrics

For each program, we present a set of performance metrics, which record the values of various parameters during the execution of the program:

- The *Total Instructions Executed* records the number of the three-address PIPU instructions that have been executed in the PIPU.
- The *Total Run Time* records the number of machine cycles elapsed:
- The *Total Idle Cycles* records the total number of machine cycles where the instruction processing units (PIPUs) were idle, due to the lack of available execution instructions.
- The *Processor Utilization* of N PIPUs is the ratio

$$\frac{(TotalRunTime * N) - TotalIdleCycles}{TotalRunTime * N}$$

- The *Speed-Up* is the ratio

$$\frac{SequentialTime}{TotalRunTime}$$

where the the "SequentialTime" is the product of the "Total Instruction Executed" with the execution time per instruction. In the current implementation, a PIPE unit needs six machine cycles to execute an instruction; so speedup can be achieved by pipelining and multiprocessing.

- *Population* records the average population in the enable memory and in the fire queue (see figure 5.3) of the instructions that were ready to fire but which were delayed for more than one machine cycle, waiting in the enable memory or in the fire queue

- The *Total Count Signals* records the number of count signals in the DISE produced by the executed program Conceptually this metric represents the signal traffic in the DISE

5.3 Pipelining of the Livermore Loops

In order to test the proposed software pipelining schemes a number of benchmark programs have been selected for execution on the simulated model We choose scientific applications because of their simplicity and maturity. Scientific computing provides a rich set of important applications which are both computationally intensive and well-suited for parallel execution. Moreover the demand for supercomputers able to accelerate computations in the scientific and engineering fields is substantial and ever-increasing

The Livermore loops [19,40] are a collection of typical loops extracted from important scientific applications developed at Lawrence Livermore National Laboratory that consume as many cycles as can be provided by the world's fastest supercomputers The kernels capture the inner loop calculations which constitute the most computationally intensive portions of the applications from which they are extracted The nine kernels executed on the simulated version of the argument fetching architecture embody the following algorithms.

1. Loop 1: Excerpt from a Hydrodynamics Code
2. Loop 3: Inner Product
3. Loop 7: Equation of State Fragment
4. Loop 10: Difference Predictor

5. Loop 12: First Difference

6. Loop 16: Monte Carlo Search Loop

7. Loop 21: Matrix by Matrix Product

8. Loop 22: Planckian Distribution

9. Loop 24. Find Location of the First Minimum in Array

These are the benchmark programs that are used for the performance evaluation study presented in this chapter. The SISAL source code for the above set of Livermore kernels, can be found in Appendix B. Although the kernels capture only small fragments of much larger applications, they are nevertheless representative of the dynamic instruction mix for these programs. The implication of running the kernels efficiently is that corresponding programs will have the potential to exhibit similar performance figures since most of the computation takes place within these loops. The common feature of the set of programs chosen is that all contain at least one *forall* code block [19] which may be nested with other forall and conditional code blocks This is a suitable set of programs to test the effectiveness of the software pipelining in the code produced by the code generator The static dataflow model effectively supports the software pipelining without requiring much complexity in the compiler. In this study we rely on dataflow software pipelining to expose program parallelism for overlapped execution The effect of further compiler optimization is presented in the next section.

The performance of Livermore loops on a processing element of one PIPU and a DISU signal capacity of four is presented in Table 5.1. The average utilization is approximately 70% and the average speedup is 4. Notice that since one PIPU is a six-stage pipeline, the optimum speedup that can be achieved is 6. Moreover the space that the softwaie pipelined program requires is small, since only one copy of the program is needed for the execution. Only one set of data memory locations are needed and they are reused by the separate iterations - a contrast from the loop unraveling suggested in the dynamic dataflow architecture [7,9]. Note the difference in performance between some loops While loop1 and loop7 achieve almost optimum performance, loop16 and loop24 have relative poor performance. This difference in speedup and utilization is related to the structure of each kernel. In a static data flow architecture, where only one instantiation of an instruction is allowed, the following two factors that affect the performance in forall loops where softwaie pipelining is applied:

PIPU: 1, DISU signal capacity: 4					
Kernels	**Total Instructions Executed**	**Total Run-time**	**Population**	**Utilization**	**Speedup**
Loop1	6,258	6,329	1.7	98.0%	5.9
Loop3	352	486	0.9	72.4%	4.3
Loop7	1,057	1,141	7.5	92.6%	5.5
Loop10	78,090	114,022	0.7	68.4%	4.1
Loop12	3,013	5,476	0.5	55.0%	3.3
Loop16	845	2,206	0.4	38.3%	2.2
Loop21	10,319	23,102	1.0	78.8%	4.7
Loop22	5,240	7,613	0.8	68.8%	4.1
Loop24	4,218	14,001	0.3	30.1%	1.8

Table 5.1: Performance of Livermore Loops

1. The *size of the loop*. Loops with small sizes may have limited parallelism.
2. The degree of *balancing*. If the program graph is unbalanced, fully pipelined execution cannot be achieved [21].

One way to exploit the parallelism in forall loops with small sizes is to generate statically more than one copy of the loop body. Since there are no data dependencies among the loop iterations, the separate pieces of code can run in parallel. The space/time tradeoff should be considered if such an optimization is to be applied to the generated code. A detailed analysis of how the performance is related with the program structure of each kernel, will be presented in the next chapter.

5.4 Optimization by Balancing Techniques

Unequal path lengths in machine graph between any two instructions, is a major limitation in fine grain software pipelining scheduling. It can be effectively solved by applying *balancing techniques* that have been proposed for static dataflow computers. The goal of

the compiler to keep the minimum amount of space can be preserved by applying an optimal balancing which introduces the minimum buffering in data flow graphs such that their execution can be fully pipelined [21,26].

The purpose of this section is to demonstrate the effects of balancing dataflow programs. The goal of balancing is to introduce the minimum buffering into data flow graphs such that their execution can be fully pipelined. A consequence is that a balanced graph can run in a maximally pipelined fashion [26,42]. Therefore, to achieve maximum pipelining, a basic technique is to transform an unbalanced signal-flow graph into a balanced graph by introducing FIFO buffers or a chain of identity tuples on certain arcs. The principle of balancing for software pipelining is discussed in [21].

We choose to apply balancing to Livermore loop 7. The reason for selecting this loop was its higher degree of unbalancing compared to the rest of the loops Loop 7 and loop 16 are the most unbalanced loops from the selected set of kernels Due to its simplicity, loop7 is more suitable for demonstrating the effects of balancing in a relatively unbalanced code. Two techniques have been used to balance the machine graph of loop 7

1. Breaking artificial data dependencies that had been introduced from the code generator.

2. Inserting dummy nodes in the unbalanced paths of the graph.

Table 5.1 shows that the utilization and the speedup of loop 7 is close to optimum in an processing element consisted of one PIPU and one DISU with a signal capacity of 4. In order to test the effects of balancing obviously we have to use more PIPUs and a larger capacity in the DISU. Experiments with three different architecture configurations are presented in tables 5.2, 5.3, and 5.4.

From these tables we observe that in all three machine configuration a considerable performance improvement has been observed by using the balanced program. Especially in the model where two PIPUs and a signal capacity of five are used, the balanced scheme achieves almost optimum performance. In this machine configuration the balanced program is faster by 30% than the unbalanced while it keeps the two PIPUs busy during the whole computation When three PIPUs and signal capacity of five are used (Table 5.4), the speedup of the balanced program is by a factor of 1.6 better compare to the unbalanced.

PIPU: 2, DISU signal capacity: 4		
Performance Metrics	**Pipelined**	**Balanced Pipelined**
Total Instructions Executed	12,821	14,811
Total Run-time	9,236	8,377
Total Count Signals	34,648	32,827
Utilization	69.4%	88.4%
Speedup	8.3	10.6
Population	1.9	2.6

Table 5.2: Performance of the balanced loop7

PIPU: 2, DISU signal capacity: 5		
Performance Metrics	**Pipelined**	**Balanced Pipelined**
Total Instructions Executed	12,821	14,811
Total Run-time	8,523	7,588
Total Count Signals	34,648	32,827
Utilization	75.2%	97.5%
Speedup	9.0	11.7
Population	2.7	6.8

Table 5.3: Performance of the balanced loop7(cont.)

PIPU: 3, DISU signal capacity: 5		
Performance Metrics	**Pipelined**	**Balanced Pipelined**
Total Instructions Executed	12,821	14,811
Total Run-time	8,320	5,957
Total Count Signals	34,648	32,827
Utilization	51.3%	82.8%
Speedup	9.2	15
Population	2.5	4.1

Table 5.4. Performance of the balanced loop7(cont.)

The way that balancing has been implemented in this experiment by introducing dummy nodes causes an increase in the program size. In the future work, we plan to investigate an efficient implementation of FIFOs in order to avoid executing instructions that are not serving any computational purpose. It is noticeable that, even in the current implementation where the dummy nodes cause the balanced program to execute almost one thousand instructions more than the unbalanced, the total execution time of the balanced code is still shorter by more than one thousand machine cycles

Another interesting observation is that in the configuration where a model of two PIPUs and a signal capacity of four is used, the balanced program does not achieve the optimum performance as one might expect In order to achieve the optimum performance the signal capacity of the DISU should be increased by one, in this case the benefits of the balancing are realized in full extent This small change in signal capacity causes an increase from 88 4% to 97 5% in the utilization while the speedup increases from 10 6 to 11 7 From this test it is clear that the the underlying architecture has a direct impact on the benefits of compiler optimizations such as balancing. This observation motivated us to study in more detail the behavior of this loop under different machine configurations. The results of this study are presented in the next section.

5.5 The Impacts of Two Key Architectural Factors

As described above, the argument fetching architecture consists of a pipelined processing element where a large pool of enabled instructions should be available for execution by the PIPUs. In reality, a machine may only have limited parallelism In a balanced design, the DISU will be able to supply fire signals to the PIPU just fast enough to keep the instruction execution pipelines operating continuously in full capacity

As the previous test showed, there may be cases where the demand of signal processing in the DISU cannot be satisfied fast enough - it may becomes a bottleneck of the throughput. In such cases, the optimization by code balancing can not achieve the desired results Moreover, even if a large signal capacity is provided, poor design of the enable memory scheduler can also degrade the parallelism offered by the running programs

In this test we study the effects on performance of these two crucial architectural factors by running the balanced version of loop7 on different machine configurations in which the number of PIPUs and the signal capacity of the DISU are varied

5.5.1 Architectural Factor I : The DISU Signal Capacity

Figure 5.4 illustrates the speedup curves under different machine configurations for the balanced loop 7 When the signal capacity is less than 8 the program cannot achieve the maximum speedup no matter how many PIPUs are used In this case the computation is *DISU bounded* This means that the DISU is unable to satisfy the demand for manipulating the incoming signals, causing a degradation in the performance of the running program. The observed relatively low speedup is caused by the limited number of fired instructions generated by the DISU per cycle.

Studying the speedup with respect to the number of PIPUs, we can observe two effects For small DISU throughput (1-2 signals) the speedup remains constant with respect to the number of PIPUs that are used This is due to the fact that although the program offers enough parallelism and the machine has enough PIPUs to execute the enable instructions, the small signal capacity could not provide enough enabled instructions to exploit the parallelism of the program and of the available hardware On the other hand, when the signal capacity is large enough to satisfy the demand of the incoming signals, the speedup increases up to a threshold value (15 2 for this program). Increasing the execution power

from 4 to 8 and the DISU signal capacity from 8 to 12 doesn't affect the speedup which remains constant to 15.2

A related effect can be seen in figure 5.5 where the utilization is plotted as a function of the number of PIPUs and the signal capacity of DISU. When the capacity is small the utilization drops with the number of PIPUs that are used. When the capacity is large enough to exploit the parallelism of the program, the utilization of PIPUs is kept high with respect to the available parallelism of the program. In this test, loop7 could keep up to 2 pipelines busy working almost at full capacity. The number of PIPUs that can be kept busy depends on the parallelism of the running programs.

From the simulation studies conducted running several programs, there is a strong experimental evidence that for each program there is an optimum machine configuration which can best exploit its parallelism in a cost effective fashion. In order to achieve such optimality, the following are important:

1. The PIPU capacity (measured by the number of PIPU execution pipelines and denoted by P) must match the parallelism of the program (*computation parallelism*);
2. The DISU signal capacity, denoted by C, must match the demand of manipulating the signals required for exploiting the computational parallelism (*synchronization requirement*).

This study shows that, even for a program with computation parallelism high enough to keep the PIPU filled under an idealized DISU (with infinite signal capacity), the actual performance observed in a real machine may be far below the ideal value. The outcome depends on the average number of signals S that are needed to fire an instruction. This number, which we call *average signal density*, is given by the following formula.

$$S = \frac{TotalCountSignals}{TotalInstructionsExecuted}$$

For a given machine configuration, the condition to keep the PIPU pipeline usefully busy is:

$$\frac{C}{S} \geq P$$

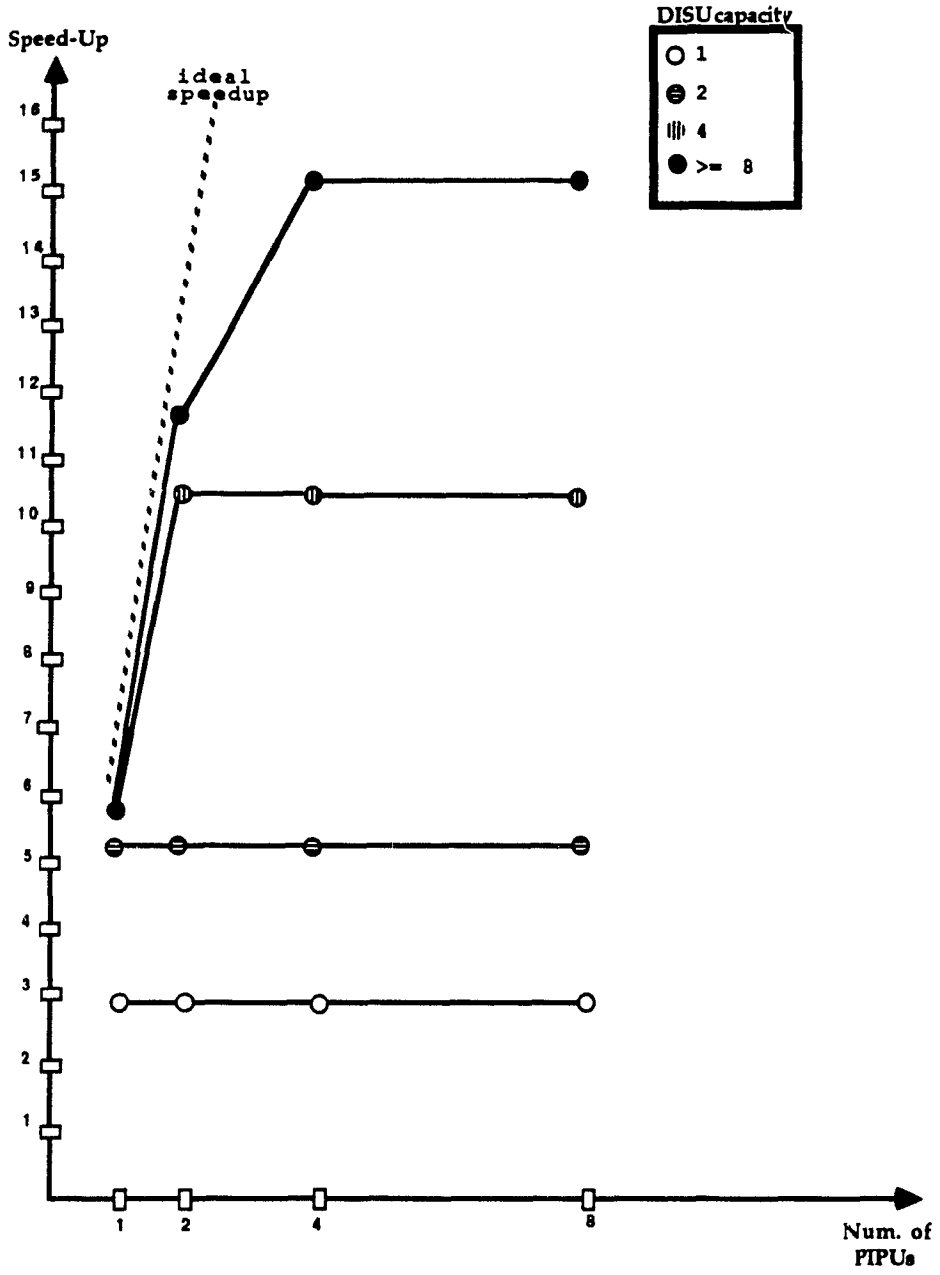

Figure 5.4: Speed up on different machine configurations

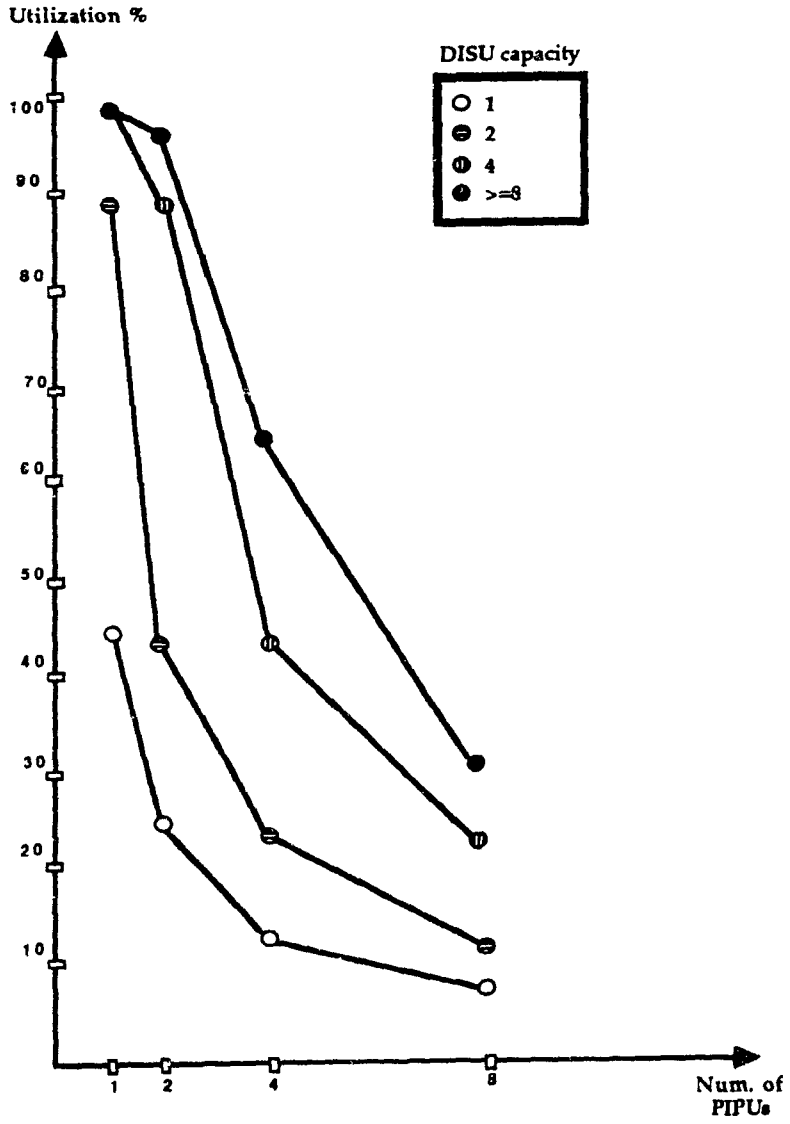

Figure 5.5: Utilization on different machine configurations

Conceptually this means that to fully exploit the computation parallelism of a given program, the DISU capacity should be at least equal to the product of the average signal traffic density of the program with the number of PIPUs of the given configuration. This is not surprising nothing is free Fine-grain parallelism has a price and the DISU must pay it!

The results of our simulation have verified this condition In the balanced pipelined version of loop7, we can derive from the tables 5 2, and 5 3 that the average signal traffic S is 2 3 In the case where two PIPUs (P = 2) and a size four signal pipe is used (C = 4) the ration $\frac{C}{S}$ is 1 7, which is less than the number of PIPUs This causes the signal pipe to become a bottleneck degrading the utilization and the speedup of the program 5 2 By increasing the DISU signal capacity by one, the ratio becomes 2 02 which meets the above condition We can observe that this small increase in the signal capacity causes the computation to run almost in optimum utilization and speedup 5 3

This is an encouraging step toward understanding the dynamics of program execution behavior and their relationship to architectural parameters However, there are still difficult problems to be dealt with In particular, we would like to characterize the relation between the DISU signal capacity and the synchronization requirement of a program, which fluctuates greatly from one program to another, and even between parts of the same program

Both the DISU capacity and PIPU capacity should be taken as important parameters in compiler optimization for the argument fetching architecture. In a machine where the signal traffic plays such an important rule in the computation, the compiler should try to minimize the number of signals without sacrificing the correctness of the implementation. This was one of the design principles that has been followed during the design and the implementation of the code generator

5.5.2 Architectural Factor II: The Enable Memory Scheduler

The other architectural factor that affects the performance of the running programs is the *enable memory scheduler* [24] During all the simulation tests a "Row Watcher" scheduler has been used This enable memory scheduler organizes instructions by row and column according to their address , and selects the enabled instructions from one row at a time , scheduling them for execution. If there are not enough PIPUs available, then the

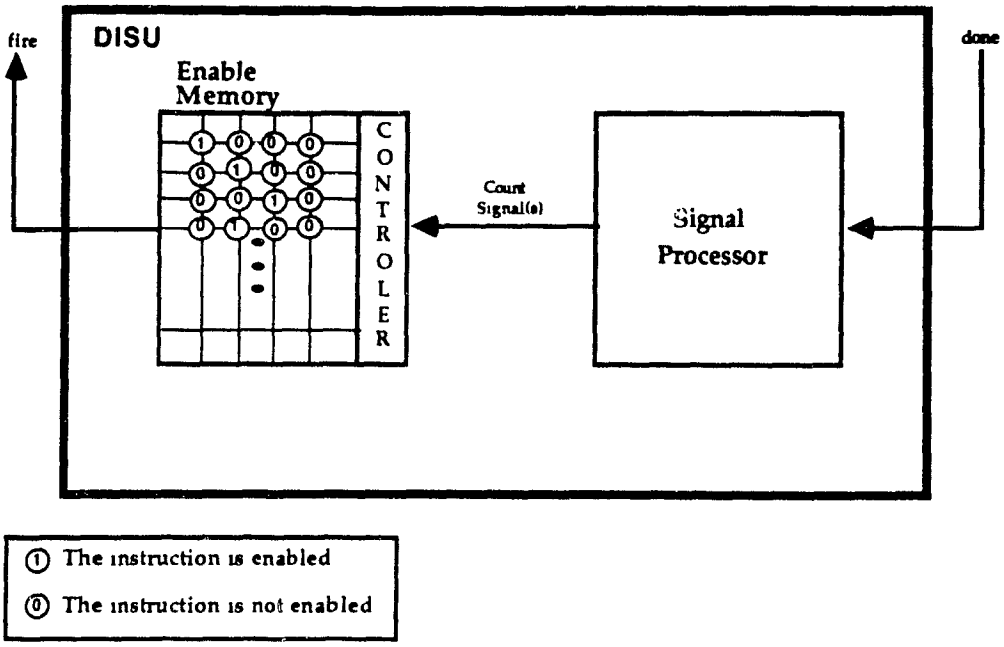

Figure 5 6 The enable scheduler bottleneck

selected enabled instructions stay in the *FIFO fire queue* until there are available PIPUs to execute them

Figure 5 6 illustrates a situation in the enable memory where such a scheduler becomes a bottleneck of the machine There are four instructions that are enabled in four different rows in the enable memory The Row Watcher cannot dispatch them all in the same cycle to the PIPUs for execution In this snapshot of computation, although the running program has the potential to keep four PIPUs busy at the next machine cycle, the Row Watcher can only exploit 25% of the parallelism that the program offers at that stage.

The metric that allows us to study the behavior of the enable memory scheduler is the "average population" as this is reported from the simulator at the end of the execution of each program This metric records the average number of enabled instructions that wait more than one cycle either in the enable memory or in the fire queue Assuming that the size of the fire queue and the size of the enable memory are large enough to accommodate the degree of parallelism of the running programs an increase in the population can have two causes·

1. *Increased Population in the Fire Queue.* If the number of the PIPUs are not enough to execute all the fired instructions that are scheduled from the enable memory

scheduler, then the increase of the waiting instructions in the fire queue causes an increase in the average population metric.

2. *Increased Population in the Enable Memory* The number of enabled instructions that are waiting for more than one cycle in the enable memory causes an increase in this metric.

In order to study the impact of the scheduler in the performance of the running programs, we not only have to look at the value of the population metric but also to the utilization of the PIPUs. High population accompanied with high utilization implies that the computation is PIPU bound and that the increase in the population is caused from the enable instructions that are waiting in the fire queue. On the other hand, high population accompanied with low utilization implies that the scheduling mechanism works poorly and does not exploit the parallelism of the computation.

The average population on different configuration models for loop 7 is shown in figure 5.7. The four peaks in this bar graph are caused by the two different reasons explained above. The 14.7 and 6.8 peaks of the average population appears when a signal capacity of eight is used with 1 and 2 execution pipelines respectively. The utilization of PIPUs for both configurations is 99.1% (figure 5.5), therefore the computation in these two configurations is PIPU bound. The other two peaks in the average population appears when 4 and 8 PIPUs are operating with a DISU of signal capacity 8. Under both configurations the value of this metric is 4 while the utilization is 63% and 31.6% respectively. In this case although the execution pipes are not fully utilized the population remains relatively high. In the case of 4 PIPUs, the unexploited parallelism corresponds to a 27% loss of utilization, and is expressed by the average of 4 enable instructions waiting in the enable memory to be scheduled.

From these simulation studies it becomes apparent that the effect in the performance of the enable memory scheduler makes this component one of the most crucial modules of the architecture. The limitations and complexities that are imposed by the underlying hardware cause a tradeoff between efficiency and complexity in the design and implementation of the enable memory scheduler.

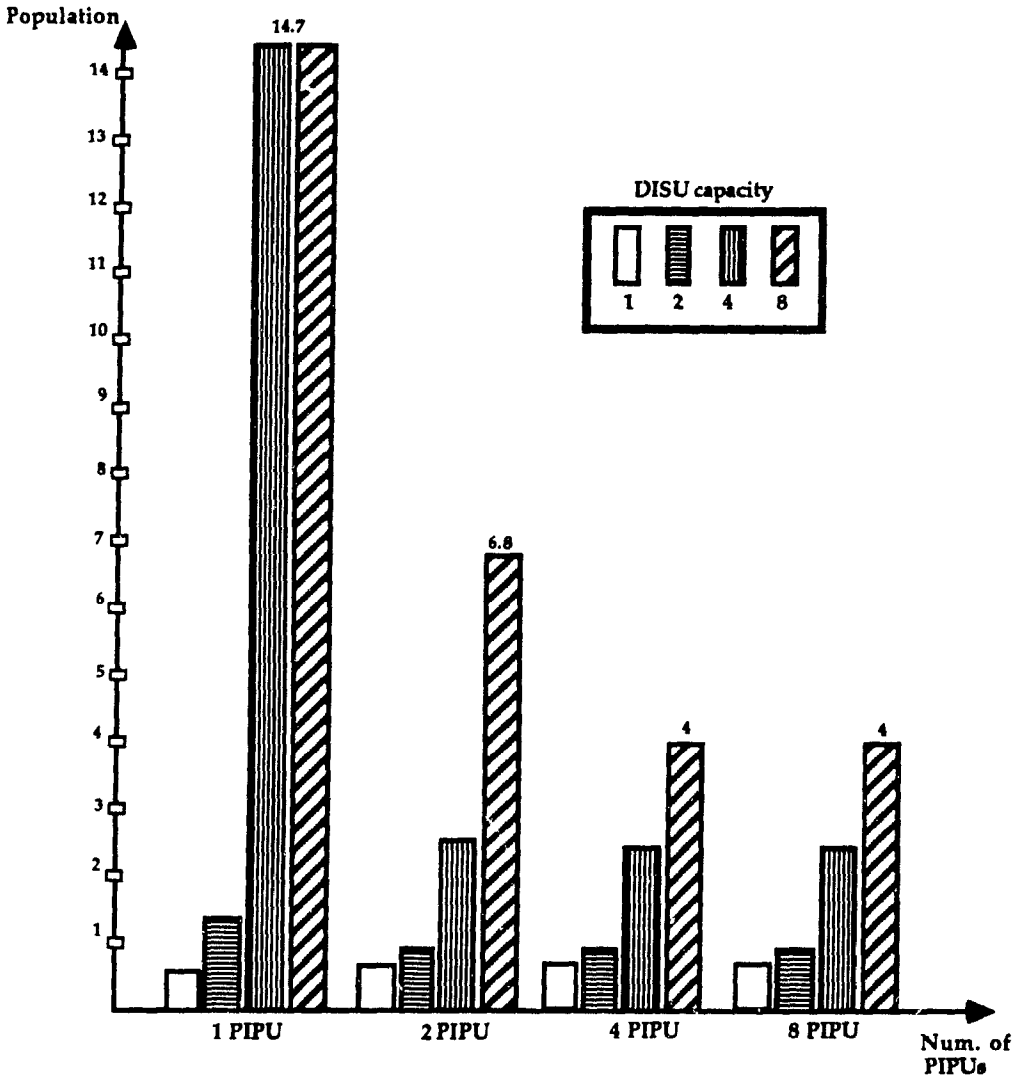

Figure 5.7: Average population on different machine configurations

5.6 Summary

In this chapter we argue that fine-grain software pipelining is an effective approach in exploiting the parallel/pipelined processing power of high-performance dataflow architectures. Without any further optimization, the basic software pipelining alone achieves a 70% utilization and speedup of 4 for nine Livermore loops with *forall* and conditional code blocks in their computational body. We believe the basic dataflow software pipelining can be applied similarly to loops with dependencies between iterations, although this is left as a future research direction.

The effect of balancing as a global optimization method has been exposed through the detailed study of the simulation results of one of the Livermore loops. An optimum performance has been achieved under a certain machine configuration. The balancing technique has the potential to yield maximally pipelined code and significant speedup of the computation. Based on these observation, we believe that the compiler of the argument-fetching dataflow architecture should consider balancing as a major compile-time global code optimization. By combining software pipelining with the balancing technique, the parallelism of many loops can be effectively explored with minimum space requirements.

The simulation studies conducted have also revealed the relation between DISU signal capacity and program synchronization requirements. We also gained a deeper insight of the impact of the enable memory scheduler. In addition to PIPU capacity, the power to exploit fine-grain parallelism in the architecture is determined by these two key factors. It is critical to engineer the design of these components such that a compiler can generate code to fully utilize the parallelism in both the program and the machine.

Chapter 6

A comparison study of software pipelining in von-Neumann architectures

Optimal mapping of loops has long been a challenge for von Neumann style architectures. Recently, there has been considerable interest in scheduling techniques that exploit the repetitive nature of innermost loops to generate highly efficient code for conventional pipelined processor architectures [44,48,41]. A technique called "software pipelining", has been proposed [2,13,41] where an iteration of a loop is activated before its preceding iteration is completed – thus multiple instructions are in concurrent execution. Apart from the differences in target architecture models, a major distinction here is that dataflow software pipelining is done at "*fine-grain*" level – an instruction is a unit of scheduling, while the software pipelining cited above is done in a "*coarse-grain*" level – an iteration is a unit for scheduling.

Software pipelining can be studied as a scheduling technique that exploits the repetitive nature of loops to generate highly efficient code for processors with parallel, pipelined functional units. In that perspective a meaningful comparison can be applied to different architectures which use the same technique to exploit the parallelism in the loops. One of the von-Neumann architectures that uses the coarse-grain scheduling is the Warp architecture [41]. The Warp machine is a high-performance programmable systolic linear array of VLIW processors. In software pipelining for this architecture, iterations of a loop in the

source program are continuously initiated at constant intervals, before the preceding iterations complete. The objective of software pipelining is to minimize the interval at which iterations are initiated; the initiation interval determines the throughput for the loop.

There is strong evidence that "fine-grain" scheduling has an advantage over "coarse scheduling" in pipelining of loops, a fact that is recognized to be true even in conventional pipelined architectures as described in [43]. It is argued that fine-grain software pipelining and relatively "clean" hardware pipelines can together exploit parallelism in loops in an optimal or suboptimal fashion (in term of time complexity) not possible by the coarse-grain methods. Dataflow software pipelining (based on balancing techniques) has another important (unfortunately often ignored) advantage that it uses a predictable and small amount of storage bounded by the size of the loop body. Furthermore the locations are effectively reused, thus the hard problem of register allocation is avoided. This is due to the fact that the dataflow architecture model ensures that the memory locations for the instructions (arguments/results) are also used in a pipelined fashion.

In this chapter we study the effectiveness of applying the software pipelining into these two different architectures, the Warp architecture and the argument fetching architecture.

6.1 Software Pipelining in the Warp Architecture

The Warp machine [3,41] is a high performance systolic array computer designed for computation intensive applications. In a typical configuration, Warp consists of a linear systolic array of ten identical cells, each of which is a 10 MFLOPS programmable processor.

Each Warp cell has its own sequencer and program memory. Its data path consists of a floating-point multiplier, a floating-point adder, an integer ALU, three register files (one for each arithmetic unit), a 512-word queue for each of the two inter-cell data communication channels, and a 32 Kword data memory. All these components are connected through a crossbar, and can be programmed to operate concurrently via wide instructions of over 200 bits. The multiplier and adder are both 5-stage pipelined; together with the two cycle delay through the register file, multiplications and additions take 7 cycles to complete.

The compiler for the Warp machine has been extensively used in many applications such as robot navigation, image and signal processing and scientific computing [3,4]. It consists of two major phases: (1) a machine independent front end translates the source

programs written in a Pascal-like language into machine independent flow graphs (2) and the back end translates the flow graph into code for the Warp cells. The code generator as a part of the back end of the Warp compiler, follows two steps: (1) the transformation of the machine independent flow graph produced by the front end into machine dependent flow graph, where generic operator in the former are mapped onto micro-operations and (2) the scheduling of the operations

Software pipelining is the the scheme that is used from the Warp code generator in order to schedule the operations. In software pipelining for the Warp machine, the iterations of a loop in the source program are continuously initiated at constant intervals before the preceding iterations complete. The objective of the software pipelining here is to minimize the interval at which iterations are initiated; the initiation interval determines the throughput for the loop. The basic units of scheduling are indivisible sequences of micro-instructions This reveals the coarse-grain nature of this software pipelining.

6.2 Effectiveness of Software Pipelining

In our perspective there are four parameters that should be taken under consideration when we study the effectiveness of software pipelining for a specific architecture:

1. **The scheduling efficiency** as this is defined from the rate that instructions/iterations are scheduled.
2. The **scheduling limitations** which specify in what extent the software pipelining can be applied.
3. The **space** that is needed to apply the software pipelining in terms of the number of register that are used.
4. **Compiler complexity** as a measure of the work that the code generator must perform in order to apply this scheduling technique to a specific architecture.

Based on the above four factors, we make a comparison of the effectiveness of the software pipelining in the Warp machine and in the argument fetching architecture.

6.2.1 Efficiency of Software Pipelining

In the Warp compiler the efficiency of software pipelining is defined by the minimum rate that different iterations can be scheduled in an overlapped fashion. The scheduling problem is to find a schedule of the operations within an iteration, such that the same schedule can be pipelined with the shortest, constant initiation interval. As mentioned earlier, the objective of the software pipelining in this architecture is to minimize the interval at which iterations are initiated. A *lower bound of initiation interval* can be calculated either statically or dynamically based on some scheduling constraints. These constraints refer to the resource requirements that are needed from the overlapped iterations and to the precedence constraints that defines the data dependency relations between the iterations [41].

Starting from the lower bound and based on the scheduling constraints, the code generator for the Warp machine tries to find the best schedule. The scheduling process is repeated with a greater interval value when an attempt to find a schedule for a given initiation interval is aborted due to the resource conflicts. The efficiency of the software pipelining depends on the interval value that is derived by the compiler, when the schedule meets the lower bound then the best software pipelining schedule has been achieved.

The performance gain by applying the software pipelining in a static dataflow architecture depends on the following two factors:

1. the *balancing factor*, and
2. the *size of the loop*.

The balancing factor determines the activation rate of successive runs, i.e., the rate at which input tokens can be consumed. Therefore, the efficiency of the dataflow software pipelining can be measured as the degree of unbalancing of the generated code. In a static dataflow architecture, the maximally pipelined throughput for any machine graph is 1/2, i.e. every instruction can be ready for execution every second machine cycle [21]. In the previous chapter, we saw that an optimum performance has been achieved when the balancing technique has been applied to one of the Livermore loops. It is proved that applying this technique to acyclic graphs, a maximally pipelined throughput of a loop can be achieved [21]. Therefore the efficiency of the software pipelining is directly related to how well the body of a loop is balanced. Although we conjectured that this optimization

can also be applied to certain cyclic graphs with the same effectiveness, making balancing a general technique for maximally pipelined the loops, this still remains an open research topic.

One effect of balancing is to increase the computational parallelism of the program by allowing more instructions to be enabled in each step of the computation As mentioned before, the computational parallelism, as this is explored by applying the software pipelining technique, depends upon the size of the loop. The size of the loop puts an upper bound to the achieved efficiency.

As explained earlier, software pipelining uses only one copy of the loop body making the best use of space and demanding only a small number of registers to pipeline the loop. The tradeoff here is that the maximum number of overlapped computations that can be executed is equal to the number of the instructions in the loop body. The length of the pipeline is equal to the depth of the machine graph for the loop body, and its width will vary according to the spatial parallelism of the graph. Therefore, the amount of parallelism that can be exploited by software pipelining is bounded by the size of the loop

The effects of the balancing factor and the size of the loops in the performance on the selected set of Livermore Loops, will be explored latter in this chapter.

6.2.2 Scheduling Limitations

There are several factors that determinate the applicability and the effectiveness of software pipelining in the Warp machine. The Warp code generator does not make any attempt to pipeline the loops when their length exceeds a threshold value. Also if the statically calculated lower bound of the initiation interval is closed to the length of the unpipelined loop [1], then software pipelining is not applied. Furthermore, due to the coarse-grain scheduling, the Warp compiler in many cases disallows overlap of loop and conditional constructs with other operations outside these two constructs

Software pipelining as implemented in the argument fetching architecture does not apply any limitation due to the length size or to the degree of nesting of the compound control constructs. Also, the loops can be overlapped with other operations no matter what their execution stage is. The pipelined scheme of the conditional expressions allows more

[1] Unpipelined loop is a loop where only one iteration is allowed per time.

than one activation of a conditional statement to be executed at the same time. There is no limitation applied due to the nesting of the conditional expressions. In addition, the pipelined scheme does not prevent overlapping the conditional statement with other operations outside the conditional.

6.2.3 Space Requirements

In order to apply software pipelining in loops, the code generator for the Warp machine needs sometimes to unroll the loops. An optimization method is used to reduce the number of locations allocated to a variable by reusing the same location in non-overlapping iterations. One implication of applying software pipelining in the Warp machine is the increase of program size. If the number of iterations is known at compile time, the code size of the pipelined loop is within three to five times the code size of one iteration of the loop. If the number of iterations is not known at compile time then additional code must be generated. In addition to that, any code scheduled in parallel with any conditional statement should be duplicated in both branches

In the fine grain software pipelining the code size that is needed to execute a loop is equal to the code size that is needed for one iteration of the loop. The code generator does not need to unroll the loops or to apply any other optimization method in order to achieve the minimum number of registers that are needed for the software pipelining. The argument fetching dataflow model ensures that the memory locations for the instructions are also used in a pipelined fashion.

6.2.4 Compiler Complexity

The code generator for the Warp architecture must do a detailed analysis to find the minimum initiation interval. As explained earlier, this interval defines the best schedule that can be applied to all the loop iterations. Then it repeatedly tries to find a schedule for a given initiation interval, starting from the predefined best schedule. If an attempt to find a schedule for a given initiation interval due to resource conflicts is aborted then the scheduling process is repeated with a greater interval value. The compiler uses a linear search to find the schedule: first establish a lower and an upper bound of the initiation

interval, and then use a linear search to find the smallest initiation interval, starting from the lower bound. Although empirical results show that a schedule meeting the lower bound can often be found, the overhead of the compiler to apply this iterative approach could be substantial. The initiation interval can be defined either statically if the number of iterations is known at compile time or dynamically by generating code that handles the scheduling at run time.

The benefit of fine grain software pipelining for the argument fetching architecture is that a compiler does not need to make analysis to determine the best schedule. The software pipelining scheduling is done totally at run-time. In order to achieve the optimum scheduling, the compiler simply performs code balancing, whose effects have been demonstrating in the previous chapter.

6.3 Performance Evaluation

In the analysis of this chapter we assume an ideal hardware such that the signal processing capacity and the scheduler do not impose any restriction of exploiting the parallelism. This section presents the effectiveness of software pipelining in the two under study architectures. The comparison study attempting here is mainly focused on the achievable performance and the scheduling limitations. The performance for the nine Livermore loops are given for the compiler generated code in both architectures.

6.3.1 Performance Statistics

Table 6.1 shows the performance of the Livermore loops in a single Warp cell. The speedup factors given in the second column are the ratios of the execution time between an unpipelined and the pipelined kernel. The utilization given in the third column of Table 6.1 are for single precision floating point arithmetic. They measure the ratio of the achieved MFLOPS over 10, which is the maximum number of the floating point operations capable in a cell. The last column contains lower bound figures of the efficiency of the software pipelining technique. They were obtained by dividing the lower bound of the initiation interval by the achieved interval value, and represent a lower bound of the achieved efficiency.

Kernels	Speedup	Utilization	Efficiency (lower bound)
Loop1	8.25	62%	1.00
Loop3	2.71	14%	1.00
Loop7	6.00	79%	1.00
Loop10	5.31	34%	0.85
Loop12	4.00	17%	1.00
Loop16	1.00	3%	1.00
Loop21	6.00	30%	1.00
Loop22	1.00	11%	0.56
Loop24	1.33	4%	1.00
Average	**3.9**	**28%**	0.93

Table 6.1: Performance of Livermore loops in a single Warp cell

	Speedup		Utilization		Efficiency
Kernels	Non Pipelined	Pipelined	Non Pipelined	Pipelined	(unbalancing factor)
Loop1	2.21	7.26	18.4%	60.5%	0.4
Loop3	1.83	4.84	15.2%	40.3%	1.0
Loop7	5.03	9.02	41.9%	75.2%	0.3
Loop10	2.10	4.43	17.5%	36.9%	1.0*
Loop12	1.60	3.55	13.3%	29.6%	0.7
Loop16	2.11	2.42	17.6%	20.3%	0.07
Loop21	1.92	5.13	16.0%	42.8%	0.7
Loop22	1.47	3.82	12.2%	31.9%	0.3*
Loop24	1.19	1.84	9.9%	15.5%	1.0
Average	2.16	**4.70**	18.0%	**39.2%**	**0.6**

⋆: loops that contain cyclic graphs.

Table 6.2: Performance of Livermore loops in the argument fetching architecture

Table 6.2 presents the performance of the same set of Livermore loops in a processing element of the argument fetching architecture For a meaningful comparison between the two architectures, the argument fetching processor in this test contains two PIPUs such that its execution power is very close to the execution power of a single Warp cell The signal processing capacity in the DISU is set to eight which is enough to satisfy the synchronization requirement of all the selected kernels By setting the signal processing capacity to eight, we eliminate the restriction imposed by the second architectural factor as this has been explored in the previous chapter

The utilization and the speedup factors as have been defined in the previous chapter are presented in table 6 2. For both of these metrics we give the performance of two implementations of each kernel In the first implementation the software pipelining is not applied to the iterative constructs of the kernels ("non-pipelined" column) while in the second the software pipelining is applied to the loops and to the conditional constructs of each kernel ("pipelined" column) Since the dataflow model of computation exploits the parallelism in the fine-grain level, it is important in this test to distinguish the spatial parallelism from the temporal parallelism as this is exploited from the software pipelining. In the "non-pipelined" programs, only one iteration of the loops is allowed to proceed at a time These programs exploit only the spatial parallelism that exists in the kernels. The benefits of the software pipelining are exploited in the "pipelined" programs, where multiple instantiations of a loop is allowed by pipelining the data through the dataflow actors.

The efficiency of software pipelining is measured by the degree of unbalancing in the machine graph of each kernel. The unbalancing factor for each kernel in table 6 2 has been derived by taking the maximum unbalancing factor between two tuples The unbalancing factor between two tuples is obtained by dividing the minimum length over the maximum length of all the distinct paths between the two tuples [21]. Some kernels in table 6 2 that contain cyclic graphs are marked with an asterisk As mentioned earlier it is still an open problem to maximize the throughput of programs that contain cyclic graphs. The effect of balancing in these programs can not be predicted. More research is needed to investigate how to balance programs with cyclic graphs

In order to have a better understanding of the achieved performance, the size of each loop should be taken under consideration. The size of the loop determines the maximum parallelism that the software pipelining can exploit from the loop. The bigger the size is, the more parallelism can be exploited. The effect of the balancing factor and of the loop size, will be investigated in more detail in the following section.

Analyzing the statistics of the fine-grain software pipelining

The benefit of applying the fine-grain software pipelining is revealed by the difference in the performance rates between the non-pipelined and the pipelined mapping of each loop. As table 6.2 illustrates, this difference in the performance between the two implementations is related to the unbalancing factor and with the size of the loop

For loop 7, although its size is large enough to keep the 2 PIPUs busy, the unbalancing factor of 3/10 prevents this loop to achieve the maximum speedup and utilization. As has been shown in the previous chapter, the maximum performance is achieved when the loop becomes balanced using a configuration of 2 PIPUs and a DISU with at least 4 signal processing capacity The effect of the small size of a loop in the achieved performance is revealed by looking at the performance numbers of loop 3 and loop 24. These two loops although fully balanced (the unbalancing factor is equal to 1) achieve a speedup of 4 8 and 1.8 respectively. This is caused by the small size of the loops as this can be measured by the number of machine instructions that these two loops contain In this case the size of the loops sets a upper limit to the maximum speedup that can be achieved by applying the dataflow software pipelining A similar observation can be derived for loop 12 Although its degree of unbalancing is small (2/3), its small size dominates the achieved speedup (3.85). On the other hand, loop 1 with a higher degree of unbalancing achieves double the speedup and utilization due to its larger size

Two loops belong to a special category: loop 10 and loop 22 are kernels that contain cyclic graphs. As mentioned earlier there is no way to assure a maximum throughput for programs that contain cycles. Assuming that the cycles in the machine graphs are executed only once, the efficiency of the software pipelining in these loops represents the lower bound of the unbalancing factor. Despite the existence of cycle in their machine graph, the pipelined version of loop 10 and loop 22 achieves double speedup compare to the non-pipelined, as table 6.2 illustrates.

6.3.2 A Comparison Analysis Based on the Performance Statistics

Table 6.3 correlates the achieved performance in the argument fetching machine and in the Warp machine for each individual loop. By looking at the average numbers in the

	Speedup		Utilization		Efficiency	
	Warp arch.	Arg fetch.	Warp arch.	Arg. fetch.	Warp arch.	Arg. fetch.
Loop1	8.25	7.26	62%	60 5%	1.00	0.4
Loop3	2.71	4.84	14%	40 3%	1 00	1.0
Loop7	6.00	9 02	79%	75.2%	1.00	0.3
Loop10	5 31	4.43	34%	36.9%	0.85	1.0*
Loop12	4.00	3 55	17%	29.6%	1.00	0.7
Loop16	1.00	2.42	3%	20 3%	1.00	0.07
Loop21	6.00	5 13	30%	42.8%	1.00	0.7
Loop22	1.00	3.82	11%	31 9%	0.56	0.3*
Loop24	1.33	1.84	4%	15.5%	1.00	1.0
Average	**3.9**	**4.70**	**28.0%**	**39.2%**	**0.93**	**0.6**

Table 6 3. Performance of the argument fetching vs the Warp architecture

last lines of table 6 3, we observe a 3 9 speedup in the Warp machine as opposed to 4.7 achieved in the argument fetching simulator This difference in the performance becomes bigger if we consider that the length of each floating point pipe in the Warp cell is equal to seven while the length of each execution pipeline in the argument fetching architecture is six Since the speedup is limited by the length of the pipe, the maximum speedup that a program can get in a Warp cell is fourteen while in the argument fetching processing element of two PIPE's is twelve

A similar observation for the utilization is derived for the two architectures. The average utilization in the Warp cell is 28% as opposed to 39 2% in the two execution pipelines of the argument fetching simulator. This difference in the performance is achieved while for most of the kernels the Warp compiler obtains the theoretical optimum schedule In seven out of nine loops, the achieved efficiency is equal to one This means that for these kernels the optimum schedule has been achieved under Warp's compiler On the other hand the fine-grain software pipelining does not meet the same degree of efficiency due to the unbalancing factor as this presented in table 6 3 Therefore we should expect the performance advantage to become even bigger when balancing is encapsulated in the code generator of the argument fetching architecture

Comparing the achieved performance of loops 22 and 16 (table 6 3), we realize the generality and applicability of the fine-grain as opposed to the coarse-grain software pipelining.

Looking at the speedup of loop 22 in table 6.1, we see that the Warp compiler cannot find a satisfactory schedule and therefore the speedup for this loop is equal to 1. The reason is that the code is too large and too deeply nested. On the other hand the software pipelined program in the argument fetching architecture achieves a 3.82 speedup. It is also worth noting that even the "non-pipelined" program achieves a 1.47 speedup due to the spatial parallelism that this kernel offers.

Examining the performance of loop 16 in the two architectures, we see that this kernel cannot be pipelined from the Warp compiler because the statically calculated lower bound of the initiation interval was within 99% of the length of the unpipelined loop. The code generator of the argument fetching architecture can achieve a 2.42 speedup. Looking at the performance of the non-pipelined and the pipelined program of this kernel (table 6.2) we conclude that the speedup is derived mostly from the exploitation of the spatial parallelism. Considering the high degree of unbalancing of this kernel, we should expect a considerable increase of the speedup and utilization if balancing is applied to this kernel.

6.4 Summary

The superiority of fine-grain as opposed to coarse grain software pipelining has been investigated through the study of the compiler generated code of two architectures: the Warp systolic array and the argument fetching architecture. The power of the dataflow model for exploiting computationally intensive programs is even more important, if one consider that the Warp systolic architecture is a feasible and extensively tested parallel machine organization. Moreover the software pipelining in this architecture has been developed into a complete algorithm that is based on software heuristics. The Warp compiler is capable to apply the software pipelining in the innermost loops and in conditional statements something that most horizontally microcoded or VLIW machine cannot apply.

The effectiveness of software pipelining in the Warp architecture relies on several assumptions on the architecture and program characteristics. Although these assumptions may be realistic for a class of programs, the feasibility of software pipelining is restricted. On the other hand the flexibility and the generality of fine-grain software pipelining make this technique a powerful code mapping strategy for exploiting temporal parallelism without sacrificing the simplicity of the static dataflow model.

The results presented in this chapter show the benefit of fine-grain scheduling, which outperformed the coarse grain scheduling as both were applied to a set of nine Livermore

loops. Moreover, we observe that the performance of fine-grain software pipelining depends on the characteristics of the program. The exploitation of parallelism is limited by the unbalancing factor and by the size of the program. On the other hand the minimum space requirements due to the single copy of the loop body constitute a tradeoff between the maximum parallelism and the space efficiency achieved from exploiting the parallelism in loops.

Chapter 7

Conclusions

The dataflow model offers a very powerful framework for exploiting program parallelism to yield high speed computation. The static dataflow architecture, which is the target of this study, is attractive for its hardware simplicity and cost effectiveness However, many criticisms have arisen mainly from its inability to express temporal parallelism

In this thesis, we examined a recently proposed static dataflow architecture, the *argument fetching dataflow architecture*. In this architecture, the data and the signaling roles of the information packets are separated, and an instruction fetches its own arguments from the data memory just like in conventional processor architectures. This eliminates the token traffic.

Dataflow software pipelining proved to be a powerful mapping scheme for exploiting the parallelism in a static dataflow architecture. This can be achieved by arranging the machine code such that the successive computations can follow each other through one copy of the code.

A code generator has been implemented which automatically generates code for a subset of SISAL. The main principles in the design and implementation have been focused on producing well-behaved graphs and minimizing the signal traffic of the generated code.

A collection of Livermore loops has been chosen as the basis for the empirical research presented here. The performance results gained by executing this set of benchmark programs are very promising. We demonstrated that software pipelining is very effective, improving drastically the performance of the benchmark programs. The superiority of the

fine-grain software pipelining as opposed to the coarse grain software pipelining, has also been investigated through a comparison study with the compiler for the Warp systolic array architecture

Moreover, we use the compiler generated code as a vehicle for testing several optimizations methods and identifying the bottlenecks of the underline architecture The balancing technique has been applied and an optimum performance has been achieved under a certain machine configuration By modifying the configuration parameters of the architecture, a better understanding of processor design constraints has been established

7.0.1 Directions for Future Research

This thesis has laid the groundwork for the design and implementation of a code generator based on the principle of dataflow software pipelining However, code generation is one of the latest phases in the compilation process. To produce more efficient object code for a parallel dataflow architecture, many decisions and optimizations should be considered at earlier stages of compilation. The developing and the undergoing research of the compiler is very crucial for exploiting the inherent parallelism that the underline argument fetching architecture offers

Figure 7 1 shows an overview of the compiler project. Research will be focused mainly in four levels. In the language level, an applicative programming language *EVAL* is under development at the Advanced Computer Architecture and Program Structures Group at McGill University. The goal is to design a general purpose language, which facilitates productive programming in scientific numerical computations The core of this language will be based on SISAL and VAL These two languages are well-known for their emphasis in providing array operations and having substantial body of real programming done in large-scale scientific applications Some features that are being considered for this new language is the stream data type, high-order functions, type polymorphism, non-strict functions and error handling

At the program graph level, HDDG form is convenient for certain kinds of program analyzers and transformers. Currently, HDDG parses the SISAL front-end intermediate form, and generates a dataflow graph where nodes and edges represents operations and data dependencies respectively. HDDG can be further analyzed and all information which are considered important could be extracted and attached to respective nodes in HDDG as their attributes.

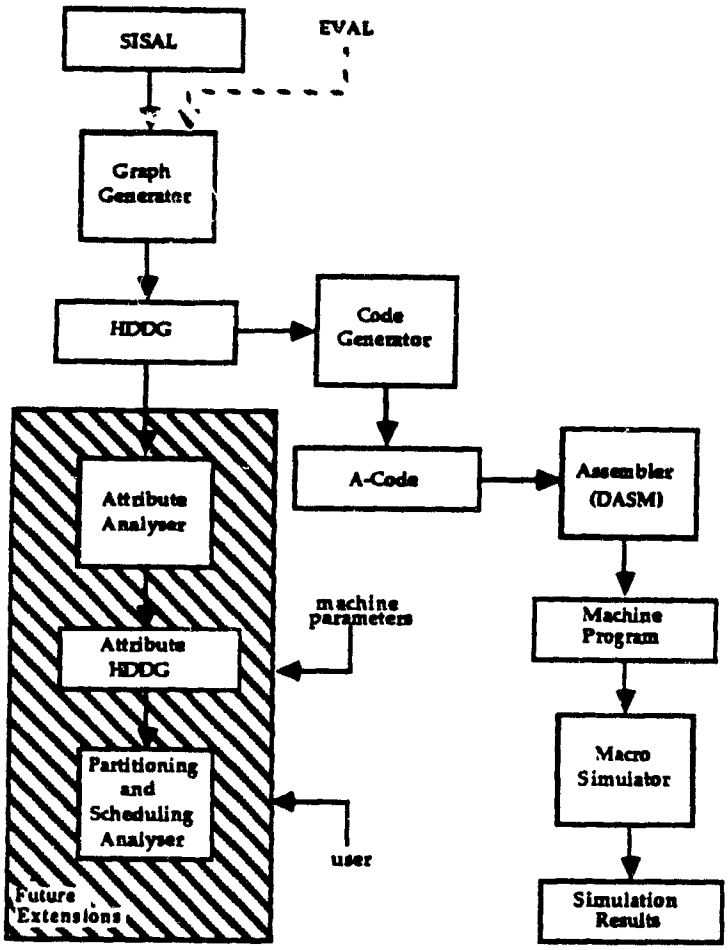

Figure 7.1: A compiler overview

Following the extraction of attributes, the program can be partitioned into units called *code blocks*. The main interest at this stage will be focused in optimizing array operations. Array operations in large numerical computations usually take place in a regular and repetitive pattern. For example, in many situations, where two code blocks generate and use an array in a producer-consumer fashion, the elements of the array can be transmitted between two blocks in a pipelined fashion without using memory as an intermediate storage. This not only substantially saves memory space, but also removes all array operations. Other kind of array optimizations should also be considered.

Another crucial issue that should be considered at this stage, is the scheduling of tasks and instructions on processors. Although the problem of optimal scheduling is NP-complete, compile-time analysis based on the data-dependence and the structure of the program graph, can be used to minimize the parallel execution time. Careful attention is required to balance between the complexity of the code scheduling problems and the overhead and feasibility to generate efficient code. During this phase the machine parameters are considered as part of the input.

In this work, we assume that data constructors are strict, i.e. the component expressions are evaluated before building the structure. The same applies to loop-driven array constructors, where the evaluation of their bodies starts when all the inputs have been computed. A *non-strict* evaluation would allow data constructors to build the structure before evaluating the component expressions, and loop-driven constructors to evaluate their bodies, receiving the arguments unevaluated. We are currently investigating efficient implementation schemes for structure operations to support effectively this type of computation. We expect the impact on the performance will be considerable.

Another primary issue in the code generation, is the error handling. Error values can be produced at any stage of computation; therefore code should be generated to detect these exceptional situations and handle them effectively. Other related directions address the issues of debugging, separate compilation and an arithmetic function library support. Finally, the whole compiler will be integrated in an programming environment, providing the facilities to the user to interchange information interactively with the compiler, which will be used for generating more efficient code.

At the architecture level, the simulation studies conducted here have shown that the DISU signal capacity and the enable memory scheduler are the most critical components of the architecture. Simulation of much larger computational problems is needed to establish a better understanding of the limitation that are imposed from the underline architecture.

A tradeoff between efficiency and complexity in the design and implementation of these two crucial parts of the architecture should also be considered. Active study is being pursued in our group in the area of enable memory architecture and scheduling mechanisms [35].

Finally, further study will be done to develop a rigorous characterization of program structure in terms of both its computation parallelism and synchronization requirement. New performance metrics and measuring methods may also be required. With such research underway, more advanced code optimization such as balancing techniques can be effectively employed in compilers for high-performance dataflow machines.

Appendix A

Machine Instruction Set

PIPU operations correspond to the *p-instructions* in the program tuple. A-Code supports a reduced set of instructions which can be divided into arithmetic, logic, comparison and data transfer operations. Both signed and unsigned operations are supported. Arithmetic operations, unless otherwise stated, all return a condition code of T ("true") if the result is non-zero, else they return F ("false"). Boolean logic operations return the result of the conditional test. Data transfer operations, by default, return only Z ("unconditional"). In the following descriptions, the "Code Returned" field is included for those operations whose return condition code is not immediately clear.

ABS
ABSF

Usage: ABS *negative absolute*

ABS returns the absolute value of an integer. ABS takes a single argument, a signed word or doubleword integer, and places its absolute value in the result register. ABSF performs the same operation for a floating point number.

ADD
ADDF
ADDU

Usage: ADD *arg1 arg2 sum*

ADD, ADDU and ADDF perform the signed, unsigned and floating point addition functions respectively. Both take as arguments a pair of single or double word number (integers for ADD and UADD and a floating point real for FADD) and produce a sum which is placed into the result register. Users should note that carries and borrows out of the high order bit and other exception conditions are not currently detected by this operation.

AND

Usage: AND *bit-arg1 bit-arg2 result*

Code Returned: T if the bitwise logical AND is non-zero
F if the result is zero

AND performs a bitwise logical and operation on the first and second arguments and places the result in the register specified by the result operand.

ASL
ASR

Usage: ASL *bit-arg count result*

Code Returned: Z

ASL and ASR perform the arithmetic shift operations. ASL shifts *bit-arg count* bit positions to the left and replicates the least significant bit into vacated bit positions, while ASR shifts *bit-arg count* positions right and replicates the sign bit. As with all shift and rotate operations, the condition code returned is always "Z".

CHS
CHSF

Usage: CHS *arg1 result*

CHS expects a signed integer operand and performs a two's complement operation on that argument. The net effect is a change of sign of the operand (i.e. from negative to positive). CHSF is similar to CHS except that the change of sign is performed on a floating point operand. Therefore CHSF does not complement all of the word or doubleword operand, but only the mantissa.

COMP

Usage: COMP *bit-arg result*

Code Returned: T if the result is non-zero
F if the result is zero

COMP performs a bitwise one's complement of the first argument and places the result in the second (result) argument. COMP tests the result operation and returns false (true) if the result is zero (non-zero).

DIV
DIVF
DIVU

Usage: DIV *dividend divisor quotient*

DIV performs the division operation on signed, unsigned and floating point operations. Both single and double word division is supported in all cases. If *divisor* equals zero, an *is_error* condition is returned in *quotient*.

DOUBLEF

Usage: DEC *real double*

DOUBLEF takes a real number and returns the corresponding double real.

EQ
EQF
EQU

Usage: EQ *comparand comparator test-result*

Code Returned: T if the two arguments are equal
F otherwise

EQ is similar to the AND operation in that the two input arguments are logically ANDed together However, the results of the AND are not placed in the result register. Instead, a "0" is placed in *test-result* and a "False" is returned as condition code if the logical AND produces a non-zero result, and a "1" is placed in the result register and the condition code returns "True" otherwise. Hence this a boolean operation, with "0" representing "false" and "1" representing "true". EQF and EQU performs the floating point and the unsigned operations respectively.

GEQ
GEQF
GEQU

Usage: GEQ *comparand comparator test-result*

Code Returned: T if comparator is greater than or equal to comparand
F otherwise

GEQ compares *comparand* against *comparator* and places a "1" in the result register and returns a condition code of "True" if *comparand* is greater than or equal to *comparator*. Otherwise, a "0" is placed in the result register and the operation returns "False". GEQF and GEQU are the floating point and unsigned versions of GEQ, and expects two floating point and unsigned integer operands respectively.

GT
GTF
GTU

Usage: GT *comparand comparator test-result*

Code Returned: T if comparator is greater than comparand
F otherwise

GT is similar to GEQ, except the operation returns "True" if *comparator* is strictly greater than *comparand*. Otherwise, a "0" is placed in the result register and the operation returns "False". GTF and GTU are the floating and unsigned integer comparison instructions respectively.

ID
IDF
IDU
GATE

Usage: ID *noop-argument noop-result*

Code Returned: T if noop-argument is non-zero
F if noop-argument is zero

ID and GATE both perform the same operation. They are, in fact, simply aliases for each other. These operations simply transfer the contents of the first argument to the result register specified by the second argument. Thus, they may be used as the "T" and "F" gates found in static dataflow architectures The condition code is returned by ID is "false" if the input argument is zero; otherwise, it returns "true" IDF and IDU are used when the input argument is a floating point number or an unsigned number respectively.

LEQ
LEQF
LEQU

Usage: LEQ *comparand comparator test-result*

Code Returned: T if comparand $\leq$ comparator
F otherwise

LEQ performs the integer "Less Than or Equal to" operation, returning "True" if the first argument is less than or equal to the second A "1" is placed in the result register if LEQ returns true, and a "0" is placed otherwise. LEQF and LEQU perform the same operation on floating point and unsigned arguments respectively.

LINDEX

Usage: LINDEX *base index absolute-address*

Code Returned: Z

LINDEX is used in conjunction with the LOAD operation on structured memory elements. Its effect is somewhat like that of I-structure array accesses in tagged-token dynamic dataflow architectures When an LINDEX instruction is fired, an absolute address is calculated from the base and index given by the first and second operand respectively, and then the "valid/invalid" bit of the actual location itself is checked If this bit is reset, it is invalid, and LINDEX will place itself in a list of pending instructions waiting for that particular datum. No "done" signal is released in that case. If the bit is set, LINDEX will continue and release a "done" signal

See also: LOAD, SINDEX, STORE.

LT
LTF
LTU

Usage: LT *comparand comparator test-result*

Code Returned: T if comparand < comparator
F otherwise

LT is similar to LEQ, returning "True" if the first argument is strictly less than the second. LTF and LTU perform the same operation on floating point and unsigned arguments respectively.

LOAD

Usage: LOAD *absolute-address memory-element*

Code Returned: Z

The first argument passed to LOAD is an address into structure memory. From this address, a single or double word memory element is retrieved and placed into the result register specified by the second element. LOAD returns no condition code (i.e. returns "Z").

MOD
MODU

Usage: MOD *dividend divisor modulo*

MOD returns the remainder of an integer division. *dividend* is divided by *divisor* and the remainder is placed into *modulo*. MOD expects two signed integers as arguments and returns a signed modulo result. MODU performs the same operation on two unsigned operands.

MULT
MULTF
MULTU

Usage: MULT *multipicand multiplier product*

MULT returns the product of the first two arguments, with the result placed into the third register. MULTF and MULTU are the floating point and unsigned integer equivalents of MULT respectively.

NOOP

Usage: NOOP

Code Returned: Z

NOOP is a true no-operation instruction. If receives and produces no arguments. NOOP instructions are usually used in nodes used for *AND*-type signal merges. input operand is non-zero, and "false" if it is zero.

NOT

Usage: NOT *argument result*

Code Returned: T if the input argument is zero
F if the input argument is non-zero

NOT is similar to the COMP operation, however, while COMP performs a bitwise negation of its input operand and places the result in the result register, NOT is a boolean negation function, hence the input operand is compared to "0" (false), and if equal, a "1" (true) is placed into the result register. Otherwise, a "0" (false) is placed in the result register.

OR

Usage: OR *bit-arg1 bit-arg2 result*

Code Returned: T if the result is non-zero
F if the result is zero

OR performs a bitwise logical OR of the two input arguments, placing the result in the third argument.

ROL
ROR

Usage: ROL *bit-arg count result*

Code Returned: Z

ROL (ROR) rotates the bit pattern specified by the first argument *count* number of bits to the left (right). Each rotation shifts the bit pattern one bit left (right) and places the most (least) significant bit into the vacated bit position at the right (left).

ROUNDF

Usage: ROUNDF *real integer*

ROUNDF returns as result the integer value that is closer to input real number. For reals with integral part less than .5, it returns the largest integer not greater than the input real number; otherwise, it returns the smallest integer that is greater than the input real number.

SHL
SHR

Usage: SHL *bit-arg count result*

Code Returned: Z

SHL (SHR) is similar to the rotate instructions (ROL and ROR), except that the most (or least) significant bits are not rotated into vacated bit positions. Instead, SHL and SHR shifts the bit pattern left and right respectively, and places zero into vacated bit positions. The total number of shifts performed is specified by the second operand.

SINDEX

Usage: SINDEX *base index absolute-address*

Code Returned: Z

SINDEX is used in conjunction with STORE to store elements of an array into structure memory. When an SINDEX instruction is executed, the absolute address is calculated from the base and index given by the first two arguments, and the "valid/invalid" bit of that memory location in structure memory is set. This will cause the release of all pending instructions (i.e. a "done" signal will be sent for all the blocked instructions). SINDEX usually uses the "short-cut" fire mechanism to fire its partner STORE instruction to prevent hazards.

See also: LINDEX, LOAD, STORE

SINGLE
SINGLEF

Usage: SINGLE *integer real* Usage: SINGLEF *double real*

SINGLE converts the input integer to the corresponding real number, while SINGLEF converts the input double to the corresponding real. The conversion from double to real is rounded.

STORE

Usage: STORE *datum absolute-address*

Code Returned: Z

STORE places the datum specified by the first argument into the structure memory location specified by the second argument. STORE always returns Z.

SUB
SUBF
SUBU

Usage: SUB *arg1 arg2 result*

SUB subtracts the second argument from the first and places the result in the result register specified by the third argument.

TRUNCF

Usage: TRUNCF *real integer*

TRUNCF converts the input real number to the corresponding integer by deleting any non-integral portion of the real number.

XOR

Usage: XOR *bit-arg1 bit-arg2 result*

Code Returned: T if the result is non-zero
F if the result is zero

XOR performs the bitwise exclusive OR function on the first two arguments, placing the result into the result register. A "True" is returned if the result is non-zero, and a "False" is returned otherwise.

Appendix B

The Livermore Kernel Benchmarks

The Livermore Loops are 24 loops from actual production codes run at Lawrence Livermore National Laboratory. The Loops represent the type of computation kernels typically found in large-scale scientific computing.

Here, we present the SISAL source code for the set of loops that has been used for the performance evaluation, studied in this thesis. The SISAL functions faithfully implement the computations of the loops, which originally have been written in Fortran. All the loops presented here, contain at least one **for-in** expression, which indicates that these loops can potentially executed in parallel since there are no dependencies across iterations

Benchmark 1: Excerpt from a Hydrodynamics Code (loop 1)

This code fragment is the first Livermore Kernel and is excerpted from a hydrodynamics code. The values Q, R, T are scalar coefficients while Y and Z are one dimensional arrays. This loop returns an one-dimensional array of size n. Note that for the construction of a static array, the value of n should be known at compile time.

```
type OneDim = array[double],
 function Loop1 (n: integer; Q,R,T: integer;
                 Y,Z: OneDim; returns OneDim)
          for k in 1,n
          returns
                 Q + (Y[k] * (R * Z[k+10] + T * Z[k+11]))
          end for
end function
```

Benchmark 2: Inner product of two arrays (loop 3)

This code fragment is the third Livermore Kernel and calculates the inner product of two arrays. The inputs X and Z represent one-dimensional arrays whose inner product is calculated.

```
type OneDim = array[double];
 function Loop3(n: integer; X,Z: OneDim; returns real)
          for i in 1,n
          returns value of sum X[i] * Z[i]
          end for
end function
```

Benchmark 3: Equation of State Fragment (loop 7)

This code fragment is the seventh Livermore Kernel and returns a one-dimensional array of size n. R and T are coefficients, while U, Y, and Z are input arrays that are used for the construction of the returned array. The value of n should be known at compile time.

```
type OneDim = array[double];
 function Loop7(n: integer; R,T:real; U,Y,Z: OneDim; returns OneDim)
          for k in 1,n
          returns array of U[k] + R * (Z[k] + R * Y[k])
                           + T * (U[k+3] + R * (U[k+2] + R * U[k+1])
                           + T * (U[k+6] + R * (U[k+5] + R * U[k+4])))
          end for
end function
```

Benchmark 4: Difference Predictors (loop 10)

This code fragment is the tenth Livermore Kernel that returns a two-dimensional array of size *[5..14, 1..n]*. A doubly-nested **for all** expression computes the instances of the equation and gathers the results into an array. A single expression in the returns clause states the equation. The sum over "PX" is returned by a third **forall** expression which fetches the values in parallel and reduces them via the **value of sum** reduction operator. The result is then subtracted directly from CX(5,j). The value of n should be known at compile time.

```
type TwoDim = array[OneDim];
 function Loop10 (n: integer; CX,PX: TwoDim; returns TwoDim)
          for i in 5,14 cross j in 1,n
          returns array of
                   CX[5,j] - for k in 6,i
                             returns value of sum PX[k,j]
                             end for
          end for
end function
```

Benchmark 5: First Difference Calculation (loop 12)

This code fragment is the twelfth Livermore Kernel and performs a first difference calculation. It returns a one-dimensional array of size n. For the construction of a static array, the value of n should be known at compile time. The i^{th} element of the array is set to Y[i+1]-Y[i].

```
type OneDim = array[double];
 function Loop12(n: integer; Y:OneDim; returns OneDim)
          for i in 1,n
          returns array of Y[i+1] - Y[i]
          end for
end function
```

Benchmark 6: Monte Carlo Search Loop (loop 16)

This code fragment is the sixteenth Livermore Kernel, which searches for a particle in a two-dimensional gird divided into *ZONE[1]* zones. Each zone is subdivided into n groups. If the particle is found, the bf function returns the zone and group number of the location; else, it returns ZONE 1, GROUP 0.

```
type IntOneDim = array[integer];
 function Loop16(n: integer; R,S,T: integer; D.PLAN:IntOneDim;
                  ZONE:IntOneDim; returns integer,integer)
          let Y := for i in 1,ZONE[1] cross j in 1,n
                   j4 := 2 * (n * (i-1) + j - 1) + 3;
                   j5 := ZONE[2 * (n * (i-1) + j - 1) + 3];
                   test := D[j5] - (D[j5-1] * exp(T - D[j5-2], 2) +
                           exp(S - D[j5-3], 2) + exp(R - D[j5-4], 2));
                   C1 := if j5 < n/3 then
                                 if PLAN[j5] < T then ZONE[j4-1]
                                 elseif PLAN[j5] = T then 0
                                 else -ZONE[j4-1]
                                 end if
```

```
                elseif j5 < 2*n/3 then
                        if PLAN[j5] < S then ZONE[j4-1]
                        elseif PLAN[j5] = S then 0
                        else -ZONE[j4-1]
                        end if
                elseif j5 < n then
                        if PLAN[j5] < R then ZONE[j4-1]
                        elseif PLAN[j5] = R then 0
                        else -ZONE[j4-1]
                        end if
                elseif j5 = n then 0
                elseif test < 0 then ZONE[j4-1]
                else -ZONE[j4-1]
                end if
            returns value of least if C1 = 0 then j4
                                  else 2 * n * ZONE[1] + 2
                                  end if
            end for
        in if Y = 2 * n * ZONE[1] + 2 then 1, 0
           else (Y - 3) / (2 * n) + 1, Y
           end if
        end let
end function
```

Benchmark 7: Matrix Multiply (loop 21)

This code fragment is the twenty-first Livermore Kernel and performs a standard matrix multiplication. The calculation is expressed entirely in the **returns** clause. It computes the inner products of the rows of *VY* and the columns of *CX* in parallel and then added the results directly the appropriate *PX* element. The value of n, which specifies the size of the innermost dimension of the array, should be known at compile time.

```
type OneDim = array[double];
type TwoDim = array[OneDim];
 function Loop21(n: integer; CX,PX,VY:TwoDim; returns TwoDim)
        for i in 1,15 cross j in 1,n
        returns
                array of PX[i,j] + for k in 1,15
                                   returns value of sum VY[i,k] * CX[k,j]
                                   end for
        end for
end function
```

Benchmark 8: Planckian Distribution (loop 22)

This code fragment is the twenty-second Livermore Kernel and returns two one-dimensional arrays of size *n*: *Y* and *W*. Note, that the value of *n* should be known at compile time, such that the construction of static arrays will be possible.

```
type OneDim = array[double];
 function Loop22(n: integer; U,V,X:OneDim; returns OneDim, OneDim)
        for i in 1,n
            Y := if V[i] = 0.0 then 20.0
                 else min(U[i]/V[i], 20.0)
                 end if;
            W := X[i] / (exp(2.71828182845905, Y) - 1.0)
        returns array of W
                array of Y
```

```
        end for
end function
```

Benchmark 9: Find Location of First Minimum in Array (loop 24)

This code fragment is the twenty-fourth Livermore Kernel and returns the first location of the minimum value of the input array A. The first loop finds the minimum value of X, and the second loop returns the index of the first of those values. Both loops are parallel and use the *value of least* reduction operator to return the smallest value.

```
type OneDim = array[double];
function Loop24(n: integer; A OneDim, returns integer)
        let
            x := for y in A
                 returns value of least y
                 end for
        in for y in A at i
           returns value of least i when y = x
           end for
        end let
end function
```

Bibliography

[1] Ackerman, W. B., *Dataflow Languages*, AFIPS Procc., Vol 48, National Computer Conference, 1979.

[2] Aiken, A. and Nicolau, A , *Perfect Pipelining: A new loop parallelization technique.*, In Procc. 1988 European Symposium on programming, Springer Verlag Lecture Notes in Cor puter Science no. 300, March.

[3] Annaratone, M., Arnould, E , Gross, T., Kung, H. T., Lam, M., Menzilcioglu, O., Webb, J. A., *The Warp Computer: Architecture, Implementation, and Performance*, CMU-RI-TR-87-18, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, July 1987.

[4] Annaratone, M , Bitz, F , Clune, E , Kung, H. T., Maulik, P., Ribas, H., Tseng, P., and Webb, J. A., *Applications and Algorithm Partitioning on Warp*, Proc Compcon Spring 87, San Francisco, February, 1987, pp. 272-275.

[5] Arvind and Ianucci, R.A., Culler, D. E., *The Tagged Token Dataflow Architecture*, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July, 1983.

[6] Arvind and Ianucci, R.A., *A Critique on Multiprocessing von Newmann Style*, Proceedings 10th Annual Symposium on Computer Architecture, pp 426-436, ACM 1983.

[7] Arvind and D.E. Culler. Managing resources in a parallel machine In J.V Woods, editor, *Fifth Generation Computer Architecture*, pages 103–121, Elsevier Science Publishers, 1986.

[8] Arvind and Iannucci, R., *Two Fundamental Issues in Multiprocessing*, DFVLR Proceedings, Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W.Germany, June 25-29, 1987.

[9] Arvind and R S. Nikhil *Executing a Program on the MIT Tagged-Token Dataflow Architecture.* Computation Structures Group Memo 271, Laboratory for Computer Science, MIT, 1987

[10] Arvind, Dertouzos, M , Nikhil, R S , Papadopoulos, G. M , *Project Dataflow A Parallel Computing System based on the Monsoon Architecture and the Id Programming Language* , Computation Structures Group Memo 285, MIT, Cambridge, MA, March 25, 1988

[11] Backus, J , *Can Programming Be Liberated from the Von Neumann Style? A Function Style and Its Algebra of Programs*, CACM, Vol. 21, No 8, Aug 1978

[12] Cocke, J , *The search for performance in scientific processors* , ACM A M turing award recipient, Commun. of the ACM, Vol 31, Number 3, March 1988

[13] Ebcioglu, K , *A compilation technique for software pipelining of loops with conditional jumps*, In Procc. of the 20th Annual Workshop on Microprogramming, pages 69-79, Dec., 1987

[14] Dennis, J.B, *First Version of a Data Flow Procedure Language*, Proceedings of the Colloque sur la Programmation, Vol 19, Lecture Notes in Computer Science, Springer-Verlag 1974, pp 362-376

[15] Dennis, J B, and Misunas, D *A Preliminary Architecture for a Basic Dataflow Processor*, Computation Structures Group Memo 102, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, Aug., 1974.

[16] Dennis, J.B, *Dataflow supercomputers*, IEEE Computer 13, pp 48-56, 11 November 1980.

[17] Dennis, J.B., Gao, G R. and Todd, K.W , *Modeling the Weather with a Data Flow Supercomputer*, IEEE Trans, on Computers, C-33, 7, July 1984

[18] J.B. Dennis and G R. Gao *An Efficient Pipelined Dataflow Processor Architecture*, in Joint Conf. on Supercomputing, pp 368-373, IEEE Computer Society and ACM SIGARCH, Florida, Nov. 1988

[19] Feo, John T , *The Livermore Loops in Sisal* UCID-21159, Lawrence Livermore National Laboratory Livermore California, December 1986.

[20] Feo, John. T.: An Analysis of the Computational and Parallel Complexity of the Livermore Loops, in Parallel Computing 1987.

[21] Gao, G. R., *Algorithmic Aspects of Balancing Techniques for Pipelined Data Flow Code Generation*, Journal of Parallel and Distributed Computing, pp. 39-61, 1989

[22] Gao, G. R., *A Pipelined Code Mapping Strategy for Data Flow Supercomputers*, to appear on the Proceedings of the Third International Conference on Supercomputing, pp. 209-215, Boston, Mass, May, 1988.

[23] Gao, G. R and Rene, T., *Instruction Set Design an Efficient Dataflow Architecture*, to appear on the Proceedings of the 22nd International Conference, of System Science, pp. 383-393, Hawaii, Jan., 1989.

[24] Gao, G. R , Rene, T. and Hum, H., *Design an Efficient Dataflow Architecture without Dataflow*, Proc. of the International Conf. on Fifth-Generation Computers, pp. 861-868, Tokyo, Japan, Dec 1988.

[25] Gao, G.R., *A Maximally Pipelined Tridiagonal Linear Equation Solver*, Journal of Parallel and Distributed Computing, Aug. 1986.

[26] Gao, G.R., *A Pipelined Code Mapping Scheme for Static Data Flow Computers*, Ph.D dissertation, Laboratory for Computer Science, Aug 1986.

[27] Gao, G R., *Maximum Pipelining Linear Recurrences on Static Data Flow Computers*, International Journal on Parallel Programming 15(2), 1987

[28] Gao, G. R., *Pipelining of Array Computation - A Data Flow Approach for Exploiting Fine-Grain Parallelism*, on the Proceedings of ICPS Edmonton 87 Conference, Nov. 1987.

[29] Gao, G. R., *Monolithic Array and Its Efficient Eager Evaluation*, ACAPS Technical Memo 06, School of Computer Science, McGill University, Montreal, Que., Nov. 2, 1988.

[30] Gao, G.R., Tio, R.; *Instruction Set Definition for the Argument Fetching Dataflow Architecture*, Technical Memo 01, Advanced Computer Architecture and Program Structures Group, School of Computer Science, McGill University, Montreal, Que.; Feb 1988.

[31] Gao, G. R., Paraskevas, Z . *Efficient Software Pipelining in the Argument-Fetching Dataflow Architecture*, to appear on the Proceedings of the Canadian Conference On Electrical and Computer Engineering, Sept. 17-20, 1989 Also ACAPS Technical Memo 02, School of Computer Science, McGill University, Montreal, Que , March, 1988

[32] Gao, G R., Paraskevas, Z . *Dataflow Software Pipelining A Case Study* ACAPS Technical Memo 6, School of Computer Science, McGill University, Montreal, Que , May 3, 1989 Also presented to the International Conference on Supercomputing, June 5-9, 1989

[33] Gao, G R . Paraskevas, Z : *Exploitation of Fine-Grain Parallelism by Dataflow Software Pipelining*, ACAPS Technical Memo 12, School of Computer Science, McGill University, Montreal, Que , May 3, 1989.

[34] Gao, G R., Tremblay G . *A Formal Operational Model for Data-Driven Program Tuples* , ACAPS Technical Memo 08, School of Computer Science, McGill University, Montreal, Que., April 18, 1989

[35] Gao, G. R., Warshawski, A.. *A VLSI Enable Memory Architecture*, ACAPS Technical Note 12 (in preparation), School of Computer Science, McGill Univ , School of Computer Science, McGill University, Montreal, Que., May, 1989.

[36] Hockey, R. W.: *Performance of Parallel Computers*, in High Speed Computation, ed J.S. Kowalik, Berlin, 159-175, 1984

[37] Kuck, David *Keynote Address*, 15th Annual International Symposium on Computer Architecture, May 30th - June 3rd, Hawaii, 1988.

[38] Lynn, B., M., *Safety and Optimization Transformations for Data Flow Programs*, MIT/LCS/TR-240, January, 1980.

[39] J.R. McGraw, S. Skedzielewski, et al *SISAL Streams and Iteration in a Single Assignment Language — Language Reference Manual Version 1 2* Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[40] McMahon, F H . *Livermore Fortran Kernels A Computer Test of Numerical Performance Range* UCRL-53745, University of California-Lawrence Livermore National Laboratory, Livermore California, December 1986

[41] Monica, L. *Software Pipelining: An Efficient Scheduling Technique for VLIW Machines* , in Proceeding of SIGPLAN'88 Conference on Programming Language and Implementation, Atlanta, Georgia, June 22-24, 1988.

[42] Montz, L. B *Safety and Optimization Transformations for Dataflow Programs*, Tech. Rep. 240, Laboratory for Computer Science, MIT, Cambridge, MA, Jan. 1980.

[43] Pingali, K. *Fine-Grain Compilation for Pipelined Machines* TR 88-934, Depart. of Computer Science, Cornell Univesity, August 1988.

[44] Rau, B.R. and Glaeser, C.D. *Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performnace Scientific Computing* , in Proc. 14th Annual Workshop on Microprogramming, pages 183-198, Oct. 1981

[45] S.K. Skedzielewski and J Glauert. *IF1. An Intermediate Form for Applicative Languages.* Technical Report M-170, Version 1 0, Lawrence Livermore National Laboratory, 1985.

[46] R. Tio. *The A-Code Assembly Language Reference Manual.* ACAPS Design Note 02, School Of Computer Science, McGill University, Montreal, Que., July 1988.

[47] R. Tio. *DASM: The A-Code Data-Driven Assembler Program Reference Manual.* ACAPS Design Note 03, School Of Computer Science, McGill University, Montreal, Que., July 1988

[48] Touzeau, R F., *A Fortran Compiler for the FPS-164 Scientific Computer* , in Proc. ACM SIGPLAN'84 Symp on Compiler Construction, pages 48-57, June 1984.

[49] Won, K. H : *IF1 Parser For HDDG* Technical Note 01, Advanced Computer Architecture and Program Structures Group, School of Computer Science, McGill University, Montreal, Que.; June 1988.

[50] Hong, W,, K.,; *Information Structures for an Experimental Multiprocessor System Compiler*, Technical Memo 05, Advanced Computer Architecture and Program Structures Group, School of Computer Science, McGill University, Montreal, Que.; Que., June 1988.