Data-Driven and Multithreaded Architectures for High-Performance Computing

Jean-Luc Gaudiot and Chinhyun Kim

CENG Technical Report 93-25

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4484

May 1993

Data-Driven and Multithreaded Architectures for High-Performance Computing

Jean-Luc Gaudiot and Chinhyun Kim

Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089

Abstract. Although there seems to be a general agreement among researchers that parallel computing is the most practical means to satisfy the ever increasing desire for higher computing power, the question of *how* to design and program large parallel machines remains. One actively pursued approach is the data-driven computation model. This is a radical departure from the traditional approach based on the von Neumann or control-driven computation model. The justification for this new technique is the inherently parallel nature of the model which makes it a good paradigm for parallel computation. However, it has been learned over the years that implementing a machine based on the data-driven model is not an easy task. In order to succeed, many issues need to be investigated at every level, *i.e.*, algorithm, language, compiler, and machine architecture. This paper discusses past and present research efforts of the data-flow community all aimed at implementing high performance computing systems. We conclude by posing some unanswered, yet important questions for future research.

1 Introduction

Problems of programmability and design of parallel processors can be found to have their root in the model of execution both in its implementation and in its reflection in high-level languages. We will now contrast the features of the conventional model with that of the functional model.

1.1 The von Neumann Model

Despite the fact that a large number of parallel computers are available today, parallel computing is still a reality for only a very small group of people. One of the main reasons for this is the difficulty of efficient parallel programming. Indeed, writing a parallel program, at least in the current programming style, is an arduous job even for an expert programmer. The cause of this difficulty is the von Neumann execution model [6] which lies at the heart of most programming languages.

The simplicity of the von Neumann execution model makes it a good paradigm of a machine architecture. Indeed, most existing processors are based on this computation model with various hardware optimization techniques built-in for

faster execution. Incidentally, the most common techniques are the pipelining and the multiple instruction issue techniques used in many RISC processors. The justification for adopting the von Neumann model by many programming languages is most often efficiency. In other words, a compiler is likely to generate a more efficient code from such languages than it would from a language based on some other computation model. Unfortunately, a language based on the von Neumann model does not migrate well to a multiprocessor environment.

The problem is the mismatch between the computation model reflected in the programming language and the changed computing environment which is no longer sequential. Automatic parallelization of programs written in such languages is very difficult because much of the parallelism existing in the original algorithm is lost during the programming phase. Attempts to automatically parallelize existing programs are still encountering some serious challenges. [8, 32]. A common approach practiced currently by most parallel computer vendors is to provide existing languages, such as C and FORTRAN, with various directives and extensions to specify parallel computations [44].

In many respect, current parallel programming practice is in some cases analogous to the early days of sequential computing when most programs were written in assembly language. To write a program in an assembly language, a programmer must first understand the architecture and the execution mechanism of the processor. Indeed, understanding of the assembly language exactly implies the understanding of these points. Likewise, to program a parallel computer, a programmer must first understand the architecture of the machine. For example, programming a computer which is based on the SIMD model is quite different from programming a machine based on the MIMD model. Even among MIMD computers, programming styles are much different in shared memory machines and in distributed memory machines.

This kind of programming style is not desirable for the same reason that assembly language programming is not desirable. First, software development is difficult because in addition to understanding the algorithm that needs to be implemented, a programmer must be concerned with the low-level machine specifics. Second, once written, porting a program to a machine with a different architecture virtually means rewriting the whole program. In the same manner, parallel computing will not succeed until an application programmer is unburdened from the low level machine specifics.

1.2 The Data-driven Model

Unlike the von Neumann model, there is no concept of memory in the data-driven model. The concept of memory in the von Neumann model introduces states in which every update of a memory location implies a state transition. This *state-driven* nature of the von Neumann model is what makes it a sequential model. In the data-driven model, a data is not represented as a static entity (memory) whose value can be updated time after time. Instead, it is a dynamic entity which is produced once and consumed once by instructions. A value produced by an instruction is valid only until it is consumed by another instruction.

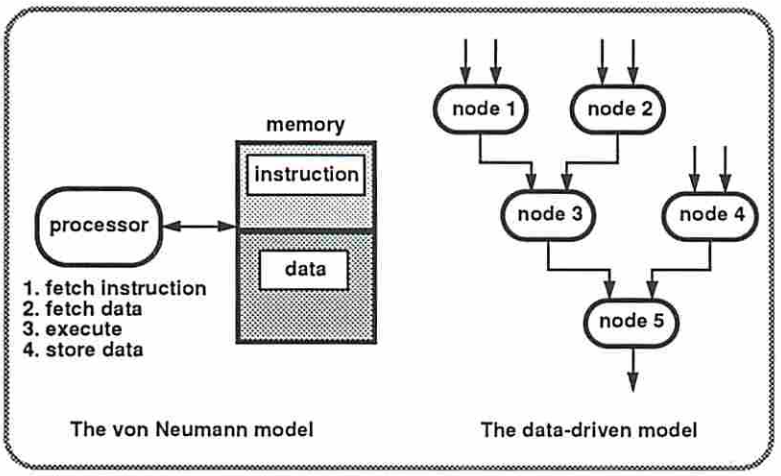

Fig. 1. In the von Neumann model, a processor sequentially fetches instructions and data from memory for execution. In the data-driven execution model, a program is represented as a graph in which nodes can be viewed as virtual processors. They execute as soon as the input operands become available on the input edges.

In the data-driven model, a program is represented as a directed acyclic graph in which nodes represent instructions and the edges represent the data dependency relationship between the connected nodes. A data elementproduced by a node flows on the outgoing edge(s) to the connected node(s) for consumption. Node executability is determined by the *firing rule* which says that a node can be scheduled for execution if and only if its input data become valid for consumption. In this computing abstraction, every node is purely functional in that it causes no side-effect. Also, computation is inherently parallel and asynchronous since any number of nodes can be executed as long as their input data become valid. At this point, we can envision the following general strategy for parallel computing :

1. Adopt a new high-level programming language based on the data-driven model which enables a compiler to extract all existing parallelism from a program and represent them in a data dependency graph. This overcomes the first barrier of parallel computing, *i.e.*, the extraction of parallelism at compile-time.
2. Develop a new architecture that can efficiently exploit at run-time the parallelism extracted by the compiler. Indeed, conventional von Neumann processors are highly optimized for executing a sequential stream of instructions and are not suitable for executing the data dependency graph according to the firing rule.

Using this strategy, computation and communication can be successfully overlapped to hide high latencies which is the bane of multiprocessor machines

[4]. In conventional parallel machines based on von Neumann processors, a processor idles when read operations are initiated. This can reduce processor utilization which in turn reduces the overall performance when the data is not in the register or cache memory. Indeed, latency reduction via caching is the conventional approach. However, this is a very difficult problem since dynamic program behavior in a multiprocessor environment is not well understood. The alternate approach is to provide an execution model which is fundamentally robust against high latencies. The data-driven approach chooses the latter solution by providing an asynchronous execution model that can quickly switch context. Thus, whenever a remote read which may cause high latency is executed, the processor switches context and executes other ready instructions instead of waiting for the read operation to complete.

In this section, the motivations and basic mechanisms of the data-driven approach have been discussed. In section two, we discuss the development of functional languages for data-driven executon model. In sections three and four, we describe the past and present work towards achieving high-performance parallel computing based on the data-driven strategy. We will see that although the basic principle of the computation model has not changed, the machine architecture has gone through major changes. Starting in the beginning from an architecture that is radically different from the conventional von Neumann architecture, it has gradually evolved to accept many characteristics of the von Neumann processor. Also, we will see that the compiler has gradually come to play a key role in the overall solution. In the beginning, its role was shadowed by the emphasis given to the architecture mechanisms. In section five, we discuss two implementations of functional languages on conventional parallel machines. Finally, we discuss issues that require further investigation.

2 High-level Languages for Implicit Parallelism

It would be best if the programs written in existing imperative languages could be directly transformed into fine-grain data dependency graphs. As of now, such efforts have not been successful. There are two notable languages developed mainly for the data-driven execution model. They are Id (Irvine Dataflow language) [38] and SISAL (Streams and Iterations in a Single Assignment Language) [37]. The Id language developement originally started at the University of California at Irvine and was subsequently moved to MIT. The SISAL language development originally started as a collaboration of a number of organizations which included Colorado State University, Digital Equipment Corporation, Lawrence Livermore National Laboratory, and University of Manchester.

Id was designed as a general purpose functional language, suitable for both scientific and symbolic applications. It has characteristics also found in other modern functional languages such as higher-order functions, polymorphism, nonstrict semantics, user-defined abstract data types, etc. [16, 40]. One unique feature of Id, however, is I-structures. The main usage for I-Structures is to handle structured data such as arrays. Functionality of the structure is guaranteed

by the write-once-read-multiple semantics. I-Structures provide non-strictness which can increase parallelism by overlapping the operations of the producer and the consumer [5].

The SISAL language is developed mainly for scientific applications [18, 19, 43, 47]. It does not allow higher order functions or polymorphism. Unlike many other functional languages which only provide recursion, however, SISAL provides loop constructs which are more appropriate when handling arrays in which case the number of iterations is known. Two types of loop constructs are provided. One is a sequential loop construct (non-product form) used when there exist loop carried dependencies. The other is a parallel loop construct (product form) which is used when each iteration is totally independent from other iterations. Unlike Id, SISAL does not take a position on the strictness of the language. Instead, this is left for the implementation level.

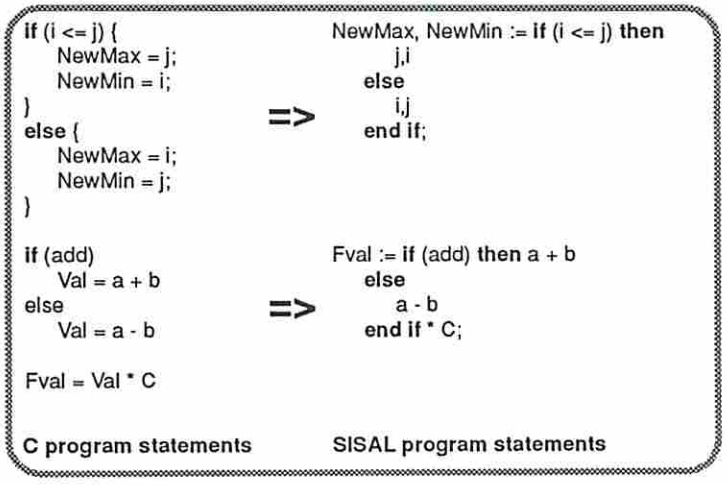

Fig. 2. Examples of the SISAL `if` program structure.

We now introduce some SISAL program structures for comparison. Unlike imperative languages, SISAL program structures are specific expressions with inputs and outputs. Because program structures are treated as expressions, the whole program structure can be bound to a name. An example is given in Figure 2 which shows equivalent C and SISAL program statements. In C, the `if-else` statement is a cotrol-flow statement in which any variable binding is possible within each clause. In SISAL, on the other hand, the data type and the arity of the value(s) produced within each clause must be the same. In the first example, the `if` program structure returns values of arity two.

In SISAL, variables cannot be introduced with assignment statements as in imperative languages. When new values need to be introduced, the `let` program structure is used as shown in Figure 3. The first part of the `let` structure is

used to declare and define one or more value names. The actual computations using the declared values (and other values that are already defined in the outer scope) are specified in the second part of the `let` structure. In the example given in Figure 3, value names `X` and `Y` are declared in the `let` structure. The value returned by the structure is the product of `X` and `Y` and the values `P` and `Q` are assumed to be already defined in the outer scope.

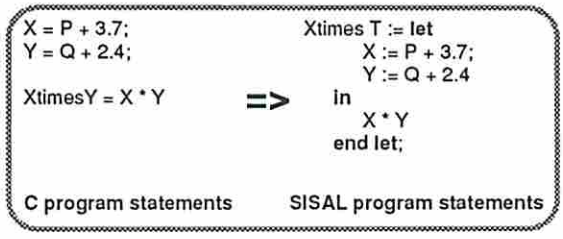

Fig. 3. In SISAL, `let` program structure is used for declaring new values.

As previously mentioned, SISAL provides two types of loop structures. The nonproduct form is used when there exist loop carried dependencies. The example given in Figure 4 is that of histogramming which counts the frequency of some events. Because the content of the array subscript `digit[i]` is not known, the loop must be assumed to contain loop carried dependencies. In the SISAL version, a value name `Histo` is bound to the nonproduct form loop structure. The structure consists of the value initialization part whose function is similar to the value declaration part of the `let` structure. In the main loop body, the array `Temp` is updated at every iteration. The key word `old` is used to specify the value defined in the previous iteration. This keyword is used to enforce the single-assignment semantic of SISAL.

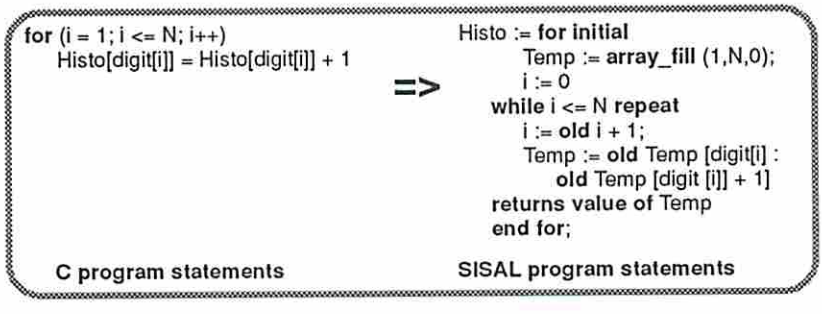

Fig. 4. Nonproduct form of the loop structure is used in the presence of loop carried dependencies.

Figure 5 shows two examples of the product form loop structure. The first (upper) example shows the case in which the returned value of the loop structure is an array. The second (lower) example is a matrix multiplication. SISAL provides a keyword `cross` which specifies the cross product of two vectors. The inner loop of the SISAL version of the matrix multiplication produces the inner product of two vectors `A[i,*]` and `B[*,j]`. The keyword `sum` specifies that the result of the following expression from all iterations are to be added to produce a scalar value. Besides the `sum` keyword, SISAL provides other reduction operations such as `max` and `min` which return the maximum and the minimum element from an array.

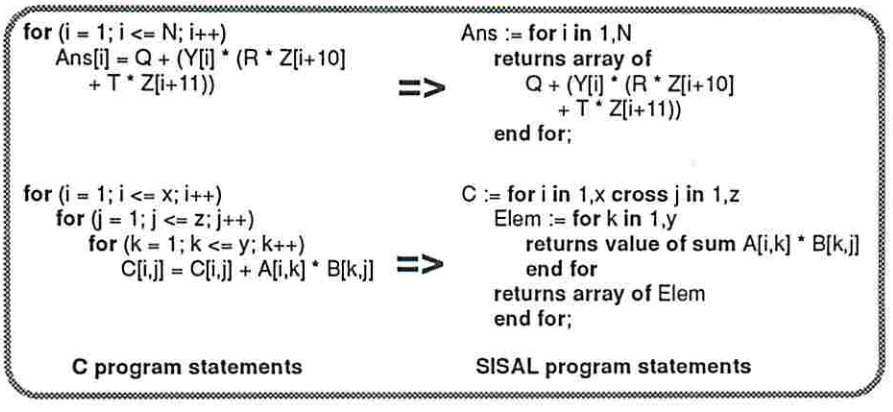

Fig. 5. Product form of the loop structure is used when each loop iteration is independent from each other.

3 Micro Data-Flow : The First Generation

First generation architectures were characterized by an enthusiastic direct implementation of data-flow principle at the architecture level. This means that synchronization and partitioning were all done at the architecture level.

3.1 Some Micro Data-flow Architectures

Some of the first generation data-flow machines include the Manchester Machine [26, 49, 52] and SIGMA-1 [28, 29]. The discussion of the MIT Monsoon dataflow machine [12, 27] is also included in this section although its architecture is quite different from the others in terms of token matching mechanism. In a sense, Monsoon is the last, in chronological terms, of the pure data-flow architectures.

The common objective was to efficiently exploit a large amount of fine grain parallelism extracted by the compiler. Although actual implementation differs from one machine to another, they had the following common characteristics :

- Data elements are physically transported in the form of scalar tokens within a processing element or across different processing elements.
- The *firing rule* as defined in Section 1 is implemented as one of the basic functions provided by the hardware.
- An instruction is scheduled dynamically when its input data tokens arriving at a processing node are properly synchronized (matched).

Manchester Dataflow Computer. The Manchester dataflow computer which is one of the first machines to be actually built was developed at the University of Manchester in the United Kingdom. The processor which consists of four hardware modules to form a pipelined ring is shown in Figure 6. The modules operate asynchronously from each other and the data tokens are transported in packets around the ring structure. Although the packets are transferred at a maximum rate of 4.37M packets / second in the implementation, the ring is rated for up to 10 million packets per second. An I/O Switch is provided for the interface between the host and the machine and also for multi-comuter implementations.

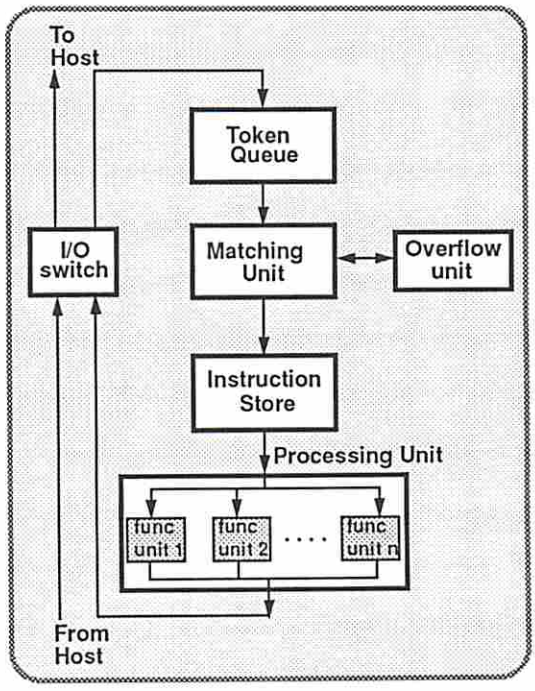

Fig. 6. Processor organization of the Manchester dataflow machine.

The *Token Queue Unit* consists of a 32K word circular FIFO store with three buffer registers. The main function of the Token Queue is to store initial data tokens as well as to smooth out any uneveness in the rate of token generation

and consumption between different modules. A word is 96 bits long which is the length of a token. The format of a token is shown below. The length of each field, in number of bits, is indicated in parenthesis. The `marker` field indicates whether the token belongs to a user or a system operation. Other fields are self-explanatory :

`Token` ≡ `<data (37).tag (36).destination (22).marker (1)>.`

The tokens that are temporarily stored in the Token Queue Unit eventually arrive at the *Matching Unit* which is responsible for scheduling instructions. An instruction is deemed executable when the Matching unit finds all of its input operands available. The input operands of an instruction are found by associatively matching the arriving token with the waiting tokens. The matching is achieved by using the subfields of the token which are 54 bits long. They are the `tag` field (36 bits) and the `instruction address` field (18 bits) of the `destination` field. A 16 bit hashing function is applied and the resulting value is used to address the eight hash tables in parallel. If a match occurs, one would produce the matching token. The matching token and the incoming token are then formed into a single token and are sent to the *Instruction Store Unit*. If a matching operand is not found, the arriving token is stored in the Matching Unit or in the *Overflow Unit* if all table entries are full. Incidentally, all instructions are either monadic or dyadic operations. The resulting token is 133 bits long and consists of :

`Token` ≡ `<data (37).data (37).tag (36).destination (22).marker (1)>.`

The Instruction Unit uses the `instruction address` field of the incoming token to access the instruction. The `instruction address` field is divided into the `segment` field (6 bits) and the `offset` field (12 bits). As in conventional processors, the `segment` field is used to access a field in the segment table to retrieve a 20 bit segment base address. The offset value is then added to the base address to fetch the instruction that would operate on the input operands. The instruction consists of two fields. One is the `opcode` field (10 bits) and the other is the destination field. There can be up to two destination fields and one can contain a literal data. Again, the instruction along with the rest of the token fields are formed into a packet and sent to the *Processing Unit* for execution :

`Token` ≡ `<data (37).data (37).opcode (10).tag (36).destination (22).`
`destination (optional).marker (1)>.`

The Instruction Unit can contain an array of up to 20 functional units in which each unit is implemented by a microcoded bit-slice processor. The maximum instruction rate for each functional unit is 0.27 MIPS which makes the processor capable of reaching the maximum throughput of about 5 MIPS. The

resulting data is formed into a 96 bit token which is then either sent to the host or circulated back to the Token Queue Unit for further processing. This completes the pipeline cycle.

Several conclusions were drawn from the Manchester Project :

1. A wide variety of programs indeed contain sufficient parallelism to fully utilize the functional units.
2. Compiler optimization is required to generate more efficient code.
3. Storing of all data structures in the Matching Unit can create a high overhead.
4. Even without data structures, the Token Queue and the Matching Unit utilization is still high.

ETL SIGMA-1. The ETL SIGMA-1 is a data-flow computer developed by the Electro Technical Laboratory (ETL) in Japan. Currently, it is the largest operational data-flow computer based on the tagged token data-flow architecture. The system consists of 128 processing elements and 128 structure store elements connected in a two-level network. A two stage global network connects 32 group nodes in which each group node consists of four PEs and four SEs connected in a local network. In addition, there are 16 maintenance and a host processor to handle operations such as I/O, system monitoring, performance measurements, etc.

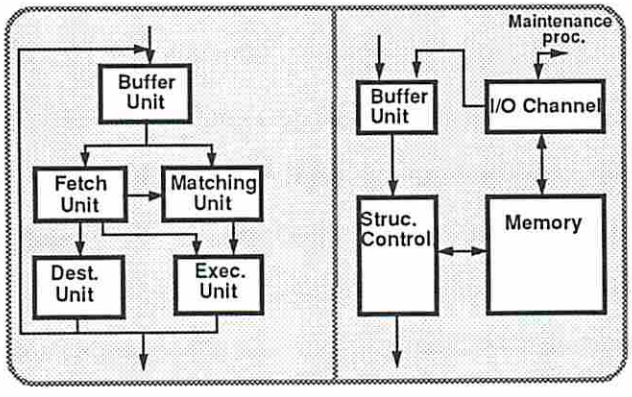

Fig. 7. Sigma-1 has dedicated hardware structure handlers in addition to the processing elements. The organization of the processing element is shown at left while that of the structure controller is shown at right.

A processing element consisting of five units which constitute two pipeline stages is shown in Figure 7. The first pipeline stage (firing stage) consists of the Buffer Unit, the Matching Unit and the instruction Fetch Unit. The Buffer Unit is a FIFO queue that can store up to 8K tokens in which a token is 89 bits in

length. The function of this unit is similar to that of the Token Queue Unit of the Manchester machine. When an input operand token of a dyadic operation arrives at the Matching Unit, a search is initiated to find the matching token from the waiting pool of tokens. The associative search is done using the tag field of the token. The token format is as follows :

```
Token ≡ <pe (8).itag (8).tag (32).c (1).type (8).data (32)>.
```

The `pe` field indicates the processing element which the token consuming instruction is to be executed. The `itag` indicates the type of a token, *e.g.*, a token may be a user data token or a token for a maintenance function. The `c` field is used to indicate the validity of the token *i.e.*, a token is invalid when the field is set. The `type` field indicates the data type. The current implementation of the SIGMA-1 only allows for a single precision floating point representation of 32 bits. The `tag` field is further divided into the following fields :

```
Tag ≡ <i (10).base (8).offset (10).flg (4)>.
```

The `i` field indicates the loop iteration to which the token belongs. The `base` and the `offset` fields are used to fetch an instruction. The `flg` field indicates the type of matching functions. For example, the SIGMA-1 machine provides a "sticky" token matching function for loop invariants. While regular tokens are deleted from the Matching Unit once a match occurs, a sticky token is not deleted. This function enhances performance since a token does not need to be recirculated.

An instruction format is shown below. The length of an instruction can vary from a word to three words in length in which a word is 40 bits long. The fan-out of an instruction is three at the maximum, *i.e.*, there can be at most three destinations. Also, a literal value can be included in an instruction. The `ndest` field indicates the number of destinations :

```
Instruction ≡ <literal (40).opcode (8).ndest (2).dest0 (20).
              dest1 (20).dest2 (20)>.
```

The Execution Unit and the Distribution Unit belong to the second stage (execution stage) of the processor pipeline. The Execution Unit consists of an integer ALU, a floating point ALU, a multiplier, and a structure address generator. The Distribution Unit is responsible for forming a resulting data into a token and distributing it to its destinations as specified in the instruction. The two stage processor is clocked at 10 MHz. and is estimated to provide 1.7 MFLOPS. The programming language chosen for the SIGMA-1 project is a single assignment language called DFC (Data-Flow C) which is a derivative of C.

MIT Monsoon. The common drawback of the aforementioned data-flow machines is the overly complex and expensive hardware of the matching unit. The key architectural breakthrough of the Monsoon data-flow machine which evolved from the Tagged Token Dataflow Architecture (TTDA) [5] is the Explicit Token Store (ETS) mechanism. Using this mechanism, a matching function becomes a simple read-write operations on a conventional memory device instead of an associative search. The net result is simpler processor hardware and faster token matching.

The ETS mechanism relies on a compile-time analysis which assigns to each token a synchronization point. This information is encoded into the instruction to be used at run-time to directly access the synchronization location of a token. The assigned synchronization point is really an offset from the base of a memory block which is dynamically allocated to a group of instructions called a *code block*. A code block can be a function body or a loop body in the source program. The allocated memory block is called a *frame* memory and it provides the synchronization space to the tokens that belong to the same code block.

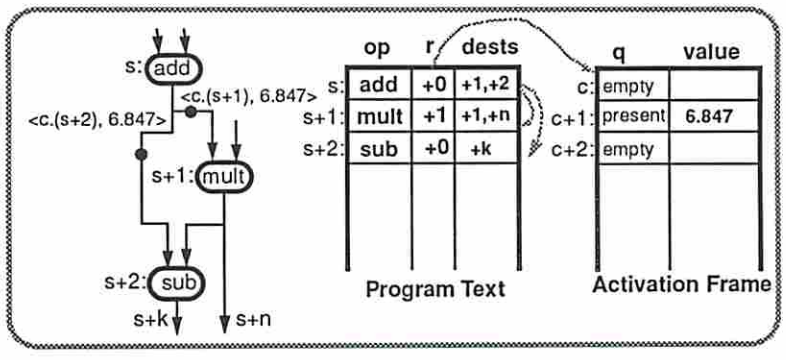

Fig. 8. In the ETS mechanism, tokens are matched directly by using the offset generated at compile-time.

In the Monsoon, two values FP and IP constitute a tag. The FP is the base address of the frame memory and the IP is the instruction pointer. The FP uniquely identifies the instance of a code block because every code block activation gets allocated its own frame memory block. When a token arrives, a processor can compute the exact synchronization location within the frame memory using these two values :

1. The IP is used to fetch the instruction.
2. The offset stored as part of an instruction is fetched and added to the FP.
3. The computed frame memory location is accessed and the presence bit is checked.
4. If the presence bit is set, a matching token exists. The matching token is read from the frame memory and the two data values are sent to the execution

unit. If the presence bit is not set, the incoming token is stored.

The Monsoon processor architecture is shown in Figure 9. Unlike the previous data-flow machines, the first unit in the pipeline is the Instruction Fetch Unit. This of course is due to the ETS mechanism which uses the offset value to compute the synchronization location. Because of the simpler token matching hardware, the processing time of the match unit is comparable to the other units in the pipeline which results in a more efficiently executing pipe. For most other data-flow machines which use associative matching, the match unit is the bottleneck which hampers the pipeline throughput.

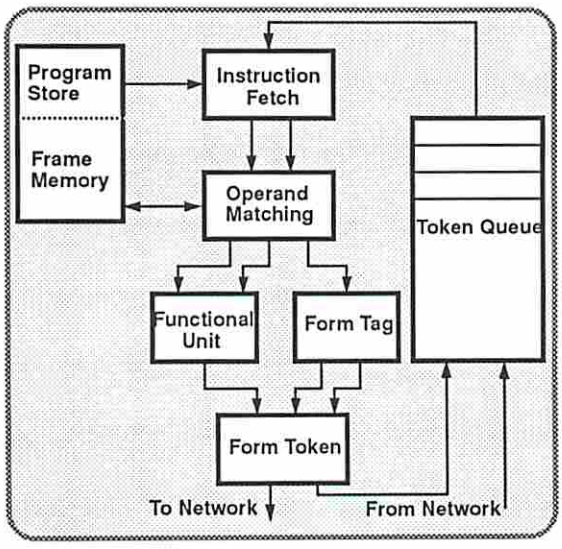

Fig. 9. Organization of the Monsoon processor.

Other Architectures. Other projects are being pursued elsewhere. They include, but are not limited to the following list:

- The RMIT/CSIRO Data-flow architecture.
- The Datarol.
- The Epsilon project.
- The Rapid.

The RMIT/CSIRO Data-flow machine was studied at the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organization in Australia. The architecture of the machine is evolved from the Manchester Data-flow machine. The design objective is to provide an efficient execution environment for both static and dynamic execution styles. Thus, the

architecture provides two modes of token matching. In addition to token matching by tag, tokens can be queued and executed in the arriving order. The motivation for such a hybrid architecture is that applications such as DSP are more suited for static execution while other applications may benefit from a dynamic execution style [1].

A data-flow architecture called Datarol is under study at Kyushu University in Japan. The datarol architecture has some features characteristic of recent hybrid architectures. For example, instead of dispatching data token to a destination instruction, the datarol provides a *by-reference mechanism* which allows the destination instruction to fetch its input operands. This mechanism makes token copying unnecessary when there are multiple destination instructions. In addition, compile-time optimization can be performed to have an internal register as the operand store for improved performance [3].

The Epsilon-2 architecture is developed at Sandia National Laboratories. It is an outgrowth of the Epsilon-1 project. However, unlike the Epsilon-1, Epsilon-2 implements a dynamic data-flow execution model. The overall architecture is similar to that of Monsoon in which direct token matching is used. Its salient feature is the *repeat token* mechanism which is also available on the Epsilon-1 architecture. The purpose of this feature is to reduce the overhead associated with data fanout [24, 25].

A data-flow processor chip set is designed and implemented by a collaboration of Osaka University, Mitsubishi Electric Corporation, and Sharp Corporation [35, 42]. The processor called Q-v1 is based on the dynamic data-driven execution model and consists of five chips. The Rapid processor is implemented on a single board using the chip set which can provide up to 20 MFLOPS of computing power. A unique feature of the Qv-1 processor is its asynchronous pipelining scheme. In the Qv-1 chips, data transfer between two pipeline stages is achieved by the exchange of send and acknowledge signals between the self-timed data-transfer control circuits. In the future version, five chip set will be integrated into a single chip.

3.2 The Id Compiler

This section discusses the Id compiler for Monsoon dataflow machine. The objective of the Id compiler is to compile programs in such a way that fine-grain parallelism is efficiently exploited with minimum overhead. Its main function is the controlling of asynchronous parallel execution and the managing of resources (such as frame memory and heap memory). Scheduling is done by the hardware which dynamically schedules instructions by way of token matching. The current Id compiler for the Monsoon relies on user annotations which tells the compiler how to control parallelism and manage resources [27]. Two user annotations are currently in use. The first is the *loop bound* value which is used to control the maximum number of loop iterations that are allowed to be active concurrently. The loop bound value k has a direct relationship to the amount of frame memory needed because k parallel iterations require k frame memory blocks. The second type of annotation is used to reclaim heap storage when it is no longer used.

Program	Source Lines	Application
MMT	130	Matrix Multiplication
GAMTEB	750	Neutron Transport
SIMPLE	1000	Hydrodynamics
PARAFFINS	300	Isomer Enumeration

Table 1. The list of benchmark programs used in the performance measurements of Monsoon.

Program	Feb. '91	Aug. '91	Mar. '92	Jun. '92
	(hr:min:sec)			
MMT(500 x 500)	4:04	3:58	3:55	2:57
4 x 4 Blocked MMT	–	–	–	1:46
GAMTEB(40,000 particle)	17:13	10:42	5:36	–
(1,000,000 particles)	7:13:20	4:17:14	2:36:00	2:22:00
SIMPLE(100 x 100, 1 iter.)	0:19	0:15	0:10	0:06
(100 x 100, 100 iter.)	4:48:00	–	–	1:19:49
PARAFFINS (n=19)	0:50	0:31	–	0:02.4
PARAFFINS (n=22)	–	–	–	0:32.2

Table 2. Performance improvements of the benchmark programs due to the compiler improvements.

Results reported in [27] (Table 2) show that recent compiler improvements have resulted in improving the benchmark program performances. One compiler improvement is the reduction of heap allocation. In previous version of the compiler, a called function used the heap memory to return the values to the calling function. Changing the compiler strategy which avoids the expensive heap allocation/deallocation operations by directly sending the return values to the calling function had significant contributions to benchmarks such as Simple and Gamteb. Another compiler improvement is the lifting of loop initialization code. This optimization is used to move the initialization code (such as frame memory allocation and storing of loop constants) of inner loops as far up as possible to the outer loop. This reduces the initialization cost of the inner loop by overlapping the operations with that of the outer loop operations.

Performance measurements using benchmark programs have shown that the compiler can expose enough parallelism to keep the Monsoon's processor pipeline busy. The experiment has also verified that latency can be effectively hidden by overlapping computations and communications. On the other hand, execution of sequential code was not competitive with that of C on a conventional processor (MIPS R3000). However, this may be due to the Monsoon architecture rather than the compiler [27].

3.3 Summary

So far, we have discussed the first research efforts aimed at attaining high-performance computing based on the data-driven execution model. The strategy was to extract as much parallelism as possible, of every granularity, at compile-time and efficiently exploiting them at run-time. As part of this strategy, functional languages such as Id and SISAL were developed to allow a compiler to easily extract all the static parallelism as expressed in the program.

The conventional von Neumann processor architectures are highly optimized for executing a sequential stream of instructions and are not capable of efficiently exploiting a large number of parallel tasks. Thus a new architecture was developed that can efficiently exploit a large amount of fine-grain parallelism extracted at compile-time. Actually, it can be said that emphasis was given to the development of the data-flow architecture during this research era.

As far as the performance (in terms of execution time) is concerned, the first research efforts did not really produce overwhelming results in favor of the data-flow approach. On the other hand, they yielded some valuable insights. First, the compiler is still an important (perhaps major) issue. In addition to extracting parallelism, conventional optimization techniques such as invariant code removal, common subexpression elimination, constant folding, etc. are still required for improving performance. Second, pure data-flow architectures may not be the ultimate answer. Although data-flow architectures satisfactorily exploit parallelism, their performance in executing sequential code was below standards. This was to be expected since data-flow architectures have no mechanism to exploit locality which is the key to efficient sequential code execution. Thus, the objective for the second stage of the research is set.

4 Hybrids : The Second Generation

Several points became clear as a result of experimenting with the data-flow architectures discussed in the previous section.

First, too many functionalities have been given to the architecture resulting in overly complex hardware and making them economically infeasible. The match unit became a bottleneck in the circular pipeline of the data-flow architectures. The ETS mechanism used in Monsoon is one solution in the right direction. By relying more on a compile-time analysis, a simpler synchronization mechanism can be developed to reduce the hardware costs and matching time.

Second, dynamic scheduling of every instruction is expensive and in many instances unnecessary. By analyzing the data dependency graphs at compile-time, many instructions can be statically scheduled without losing parallelism. By doing so, many von Neumann processor optimization techniques may be applied. Figure 10 shows a portion of a scientific application represented by an IF1 graph [48]. Assuming that the availability of the data tokens which represent `i` and `h` cannot be determined at compile-time, the only instructions that require dynamic scheduling are instructions 1 and 4. The instructions 2,3, and 5 can be scheduled statically according to the data dependency relationship.

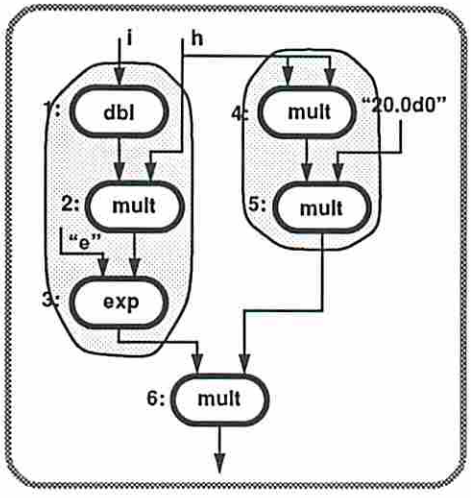

Fig. 10. A portion of a scientific application is shown in the IF1 graph.

This section discusses the recent research activities aimed at these points. The common approach taken by many researchers is a more balanced distribution of functionalities between the compiler and the hardware. It is driven by the fact that, at least in today's technology, pure data-flow architectures are not economically feasible. The resulting solution is also known as *multithreading*.

4.1 Architectures

After extensive experimentations with pure data-flow architectures, the second generation data-flow architectures gradually converged to hybrids in an attempt to combine the best features of the conventional von Neumann architecture and of pure data-flow architectures. With such architectures, a sequential stream of instructions would be efficiently executed according to the von Neumann model while still exploiting fine-grain parallelism with the help from the data-driven model.

P-RISC. The objective of the P-RISC (Parallel-RISC) was to come up with an architecture suitable for multiprocessing by starting out with a von Neumann architecture and extending it by borrowing necessary features from the data-flow architecture to exploit fine-grain parallelism [39]. The resulting architecture has four new instructions in addition to its conventional RISC instruction sets. They are :

1. `fork IPt`
2. `join x`
3. `start v c d`
4. `loadc a x IPr`

The first two instructions `fork` and `join` are the result of decomposing the implicit data-flow operations into primitive instructions which can be explicitly controlled by the compiler. In a data-flow architecture, token matching is an implicit operation which is transparent to the compiler. Also, forwarding the output tokens to their respective destinations is implicit to the compiler. The `fork IPt` instruction creates two active threads by queueing two *continuations* `<FP.IP+1>` and `<FP.IPt>` in the token queue. The `join x` instruction toggles the value at frame location `FP+x`. If the result is one, nothing happens. It the result is zero, a continuation `<FP.IP+1>` is inserted in the token queue. Figure 11 shows an example in which these two instructions are used to effect data-driven execution style.

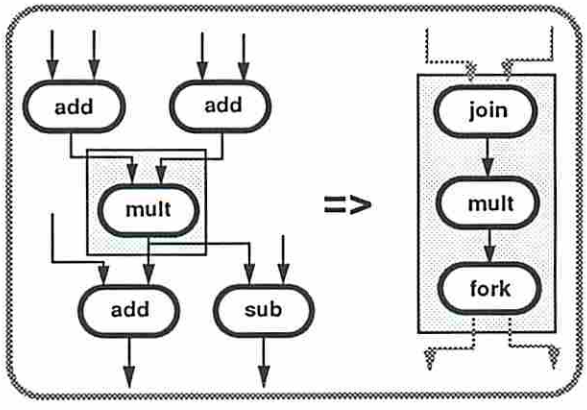

Fig. 11. In P-RISC, token matching and forwarding is explicit under the control of a compiler.

The `start v c d` instruction is used to activate a thread that belongs to a different activation which may be executed on the same processor or on a different processor. This is accomplished by sending a message of the form `<START,[FP+v],[FP+c],[FP+d]>`. The second message operand is the descriptor of the thread to be activated. Upon arriving at the destination, the value at the first message operand is stored at the frame memory location whose offset is indicated by the third message operand and the thread descriptor is inserted into the token queue. This instruction can be used for a procedure call linkage mechanism, *i.e.*, a returned value is sent as the first operand and the second operand indicates the thread that becomes upon return.

The `start v c d` instruction is also used to implement *split-phase* remote read operations. This operation is used in data-driven execution to efficiently utilize a processor when a long latency inducing operation is activated. To implement a split-phase read operation, a P-RISC processor executes an active thread from the token queue instead of idling after dispatching a read request message of the form `<READ,a,FP.IP+1,x>`. In the message, `a` is the address of

the requested memory location, `x` is the offset of the frame memory location which the returning value is to be stored. The `FP.IP+1` is the thread descriptor which is to be activated upon arrival of the requested data. When a `READ` message is received, the memory handler saves the thread descriptor and the offset. After reading the value from address `a`, it executes a `start` instruction which sends the value along with the thread descriptor and the offset value.

The `loadc a x IPr` is just a variation of a `load` instruction. In a `load` instruction, no continuation is generated by the instruction after the `READ` message is dispatched. The processor simply fetches a continuation from the queue. The `loadc a x IPr` instruction on the other hand, generates a `READ` message with `IPr` as the return address and continues execution from `IP+1`. In many cases, this instruction simplifies the code [39].

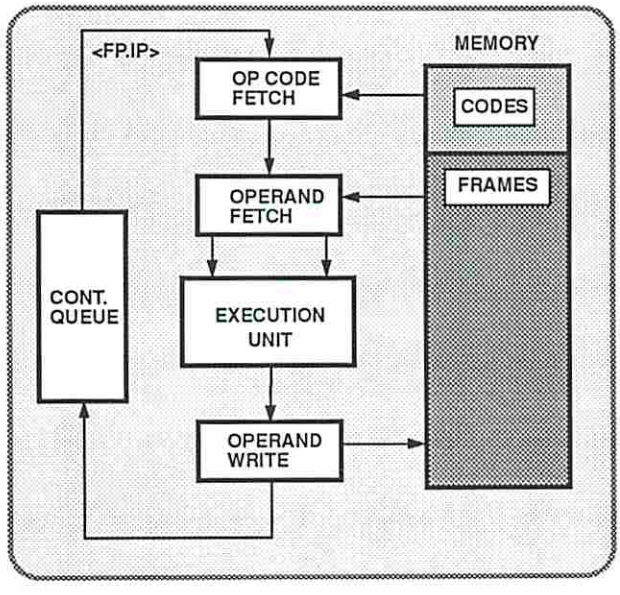

Fig. 12. Organization of the P-RISC processor.

Work is in progress to implement a processor based on the P-RISC concept. The processor is designated *T (pronounced "start") and is based on the Motorola 88110 superscalar processor [7, 15, 41]. The *T processor has a functional unit called the Message and Synchronization Unit (MSU) in addition to the existing functional units. The main functions of the MSU is to handle messages and scheduling of threads. The MSU provides a close coupled register-set model of network messaging by providing a set of transmit and receive registers. User level instructions are available so that messages can be sent and received without the operating system support. In *T, the maximum message length is fixed at 24 words.

In addition to network support, the *T processor provides mechanisms for multithreading. In the MSU, 64 entry Microthread Stack is provided to store thread descriptors. Scheduled threads are stored in the 8 Scheduled Microthread Descriptor Registers. The threads are scheduled via fixed priority scheme of the Scheduled Microthread Descriptor Registers, the network receiver, and the Microthread Stack.

USC Decoupled Architecture Model. Two factors have influenced the development of the USC Decoupled Architecture Model. First is the need to develop an efficient architecture model for data dependency graphs with variable resolution actors [17]. It has been shown that appropriate fusing of simple nodes into a macro node can improve overall performance without significant loss of parallelism [22]. In such execution model, however, there can be a great difference in the execution time according to the granularity of the node. In turn, this requires a large buffer between the graph stages (match and token formating/routing) and the computation stages (fetch and execute) in conventional data-flow architectures [17]. Second, it would be beneficial to borrow advanced pipeline techniques from von Neumann processors in executing instructions within the macro nodes. That is, the instructions inside a macro node are executed in sequence and the von Neumann processors are specifically optimized for efficient execution of sequential codes.

Figure 14 shows the Decoupled Architecture model which consists of two processing units. They are the *Computation Engine* (CE) and the *Data-Flow Graph Engine* (DFGE). The CE is responsible for executing the instructions which belong to a node once the node is scheduled for execution. A high performance conventional processor is appropriate for the Computation Engine since the instructions within the node are executed in sequence. The role of the DFGE is that of the scheduler. It performs token matching and once all input tokens of a node is available, the node is scheduled for execution by the CE. The two processing units are connected by two queues. The *Ready Queue* (RQ) holds the descriptor of the ready nodes inserted by the DFGE. The *Acknowledge Queue* (AQ) holds the descriptor of the node that has been executed by the CE. Upon receiving a descriptor of an executed node, the DFGE can schedule other nodes that depend on the node.

EM-4. The EM-4 is a multiprocessor machine developed at ETL of Japan. Currently, a machine containing 80 processing elements is in operation at ETL. Similar to other recent data-flow architectures, the EM-4 architecture based on a multithreading model which is called the *strongly connected arc model.* A thread is called a *strongly connected block* (SCB). Once a SCB is initiated, execution continues without preemption until the SCB terminates. Instructions within the SCB are executed sequentially in a control-driven fashion [45].

As in the Monsoon, a direct token matching scheme is used in the EM-4 [46]. However, the actual mechanism is different from that of the ETS used in the Monsoon. There are two main differences from the ETS scheme. First, in

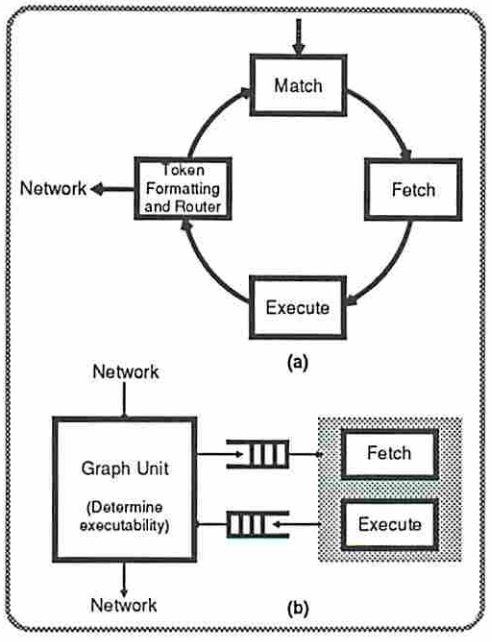

Fig. 13. Basic data-flow architecture organized as a (a) cyclic pipeline and (b) Decoupled Graph/Computation configuration. Courtesy of Prentice Hall, *Advanced Topics in Data-flow Computing*, J-L. Gaudiot and L. Bic, 1991.

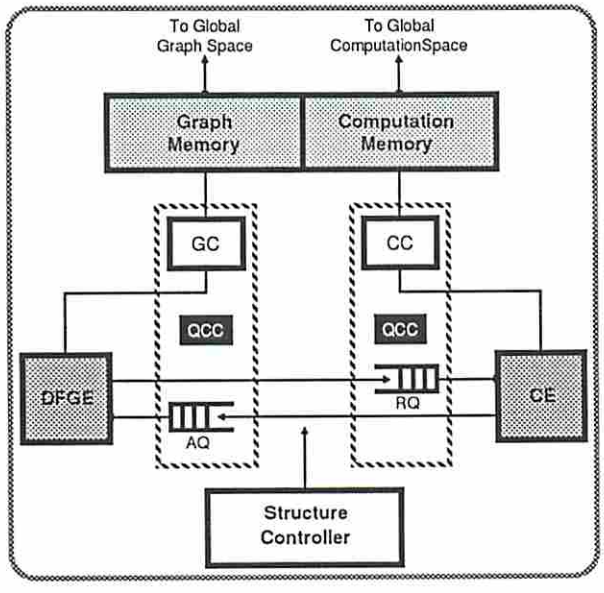

Fig. 14. Data-flow processor with decoupled graph and computation units. Courtesy of Prentice Hall, *Advanced Topics in Data-flow Computing*, J-L. Gaudiot and L. Bic, 1991.

the EM-4, the offset value which specifies the storage/synchronization location of an instruction's operand is carried in a data token. In the ETS, the offset value is stored as part of an instruction. Second, the offset not only indicates the synchronization point of tokens, but also acts as the offset for the instruction which operates on the tokens. In the ETS, the offset value is only valid for accessing operands within the frame memory. Figure 8 describes the direct matching scheme of the EM-4.

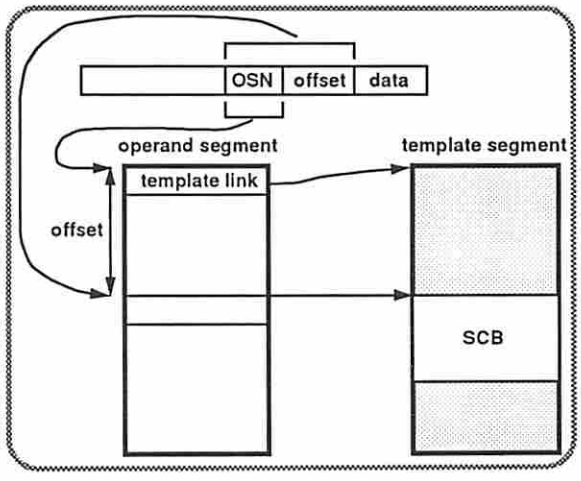

Fig. 15. The direct matching scheme of EM-4. The operand segment refers to the frame memory and the template segment refers to the code block. The operand segment number OPN and the offset are carried in the data token packet.

The heart of the EM-4 machine is the EMC-R single chip processor. The EMC-R processor architecture provides both the data-flow and the von Neumann execution mechanism within the same processor. The processor contains four pipeline stages in which the first two stages are used for token matching functions and the latter two stages are used for the fetching and the execution of the instructions. For the instructions that belong to a thread (or SCB) use only the latter two pipeline stages once the thread is activated. The Operand Match Unit, the Instruction Fetch Unit, Destination Unit and the Execution Unit shown in Figure 16 correspond to the four stages.

The EMC-R processor chip also contains the Switching Unit (SU) which performs a three-by-three packet switching function. The tokens that are destined for the local processor are first stored in the Input Buffer Unit (IBU) which is a 32 word FIFO buffer. An 8K word secondary buffer is located in an off-chip memory in case of an overflow. Although the EMC-R does not have a floating-point execution unit, a commercially available floating-point chip can be inserted.

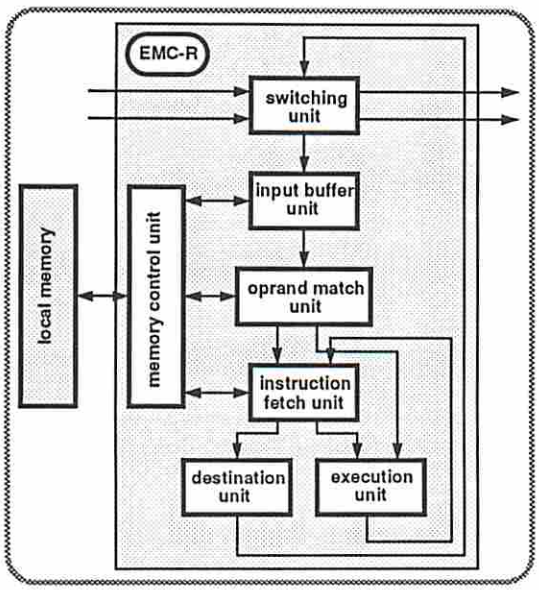

Fig. 16. The organization of the EMC-R single chip processor.

Other Architectures. The McGill Dataflow Architecture Model (MDFA) is another hybrid architecture model developed at McGill University in Canada. MDFA has a decoupled architecture similar to the USC Decoupled Architecture. In MDFA, a processor is divided into the Instruction Processing Unit (IPU) and the Instruction Scheduling Unit (ISU). The two units interact with each other by exchanging *fire* and *done* signals. A fire signal is sent from the ISU to the IPU and it consists of the instruction pointer and the base address of the frame memory, *i.e.*, the continuation. A done signal is sent from the IPU to the ISU and it also consists of the continuation of the executed instruction [20].

So far, we have discussed hybrid architectures which originally evolved from the data-flow architecture. There are other hybrid architectures which started from the von Neumann architecture. They will not be discussed in this paper, but they include the Hybrid Architecture [30, 31], the Message-Driven Processor (MDP) [14], and the Tera machine [2].

4.2 Summary

The driving force behind the second generation architectures were efficient sequential execution and simpler and faster synchronization mechanism. The result is a hybrid architecture which provides von Neumann execution model in addition to the data-driven execution model. With the exception of the EM-4, these architectures decouple computing tasks from the synchronization tasks. In *T, this is handled by the Message and Synchronization Unit while the Data-Flow

Graph Engine is responsible for the job in the USC Decoupled Architecture Model.

Compilers play an important role since the hybrid architectures depend on compilation strategy more than the first generation data-flow architectures. Performance can vary greatly depending on the criteria used for thread partitioning. Due to relatively recent introduction of the hybrid architectures, however, not much work is reported for compilation strategy.

5 Compiling for Conventional Parallel Machines

As it turns out, the principles of functional programs do not necessarily require hardware support. Implicit parallelism through functional programs can indeed be exploited on conventional architectures. Two projects are discussed.

5.1 SISAL Compiler

This section discusses the compilation strategy and the performance of the Optimizing SISAL Compiler (OSC) [9]. Currently, OSC is available on many shared memory parallel machines from companies such as Sequent, Silicon Graphics, CRAY, etc. Incidentally, OSC can also be used in uniprocessor workstations running Unix operating system. With this compiler, a SISAL program originally written to run on a small workstation can be recompiled to run on a parallel supercomputer such as CRAY C-90 without any program modification.

Program	Source Lines	Time Steps	Problem Size	Application
KIN16	225	42,000	2 x 14285	Gel Electrophoresis
RICARD	297	40,000	5 x 1315	Gel Chromatography
CFFT	517	1	524288	Fourier Transform
BMK11A	1003	200	2560	Particle Transport
LOOPS	1116	4000	100 - 1000	Scientific Kernel
SIMPLE	1527	62	100 x 100	Hydrodynamics
UNSPLIT	1937	40	160 x 80	Hydrodynamics
WEATHER	2712	20	420 km	Weather Model

Table 3. The list of benchmark programs used to compare the performance of OSC to FORTRAN on a CRAY Y-MP supercomputer.

Traditionally, the main drawback of functional languages (such as SISAL) has been the performance. Although these languages give programmers a high level of abstraction, the lack of performance made them unattractive [36]. However, OSC has shown that high performance parallel computing is possible with functional programming. Experimenting with a number of numerical application benchmark programs showed that OSC produced code performs competitive to

CRAY Y-MP/864	Concurrent-Vector					Seconds
	One CPU		Four CPUs		Eight CPUs	
Program	Sisal	Fortran	Sisal	Fortran	Sisal	Fortran
KIN16	74.8	79.9	19.7	20.7	11.6	11.2
RICARD	39.8	38.9	13.8	12.5	8.7	7.4
CFFT	0.33	0.42	0.10	0.15	0.07	0.13
BMK11A	4.8	4.2	3.4	3.0	3.3	2.9
LOOPS	16.1	12.9	12.3	11.4	11.3	11.0
SIMPLE	9.4	9.0	3.1	8.8	2.2	8.8
UNSPLIT	10.1	8.0	2.8	5.0	1.7	4.9
WEATHER	7.3	7.0	2.0	6.7	1.2	6.8
Total	162.6	160.3	57.2	68.3	40.1	53.1

Table 4. Performance comparison of SISAL and FORTRAN benchmark programs on a CRAY Y-MP supercomputer.

that of a FORTRAN code on a single head CRAY Y-MP supercomputer [10]. However, the real advantage is that programs can be run on multi-head CRAY computers without modification with correspondingly improved performance. Table 3 describes the type of benchmark programs used in the experiment. Performance results of the benchmarks on a CRAY Y-MP/864 is reported in [10] (Table 4). It demonstrates that the programs written in FORTRAN perform slightly better when one processor is used. However, the programs written in SISAL perform better when multiple processors are used.

The main villain in SISAL has been the aggregate data types such as arrays. Because of the language semantic which requires *referential transparency*, arrays often had to be copied. For example, a simple replacement operation on an array element causes array copying which makes the execution time a function of the array size instead of being a constant. The problem becomes much worse if the array replacement operation happens to occur inside a loop. It was observed on a Sequent Balance that a SISAL program which initializes an array of 50,000 elements took approximately one hour while a FORTRAN version took less than one second to complete [8].

The main strategy of the OSC is to minimize array copying via automatic static analysis of the program graphs. Specifically, the objective is to transform array operations that cause copying to either *build-in-place* operations or to *update-in-place* operations. Array operations that cause dynamically growing arrays can be optimized into build-in-place operations and the array replacement operations can be optimized into update-in-place operations. The optimizers IF2MEM and IF2UP as shown in Figure 17 are responsible for these transformations. The final output of the array optimizers are program graphs in IF2 [53] format with explicit memory management operations.

Although the actual graph analysis that optimize various array operations are nontrivial, the basic concept is easy to understand. Two simple examples are used to demonstrate the basic intuition. In the array initialization example

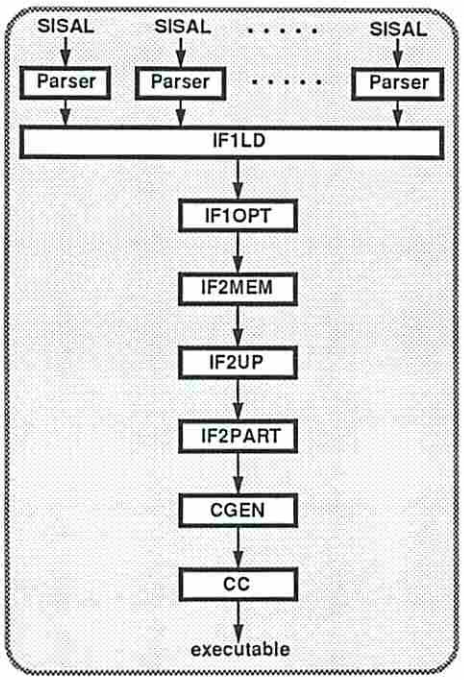

Fig. 17. The current Optimizing SISAL Compiler goes through a number of optimization phases. IF1OPT performs conventional optimization transformations on the IF1 graphs. The resulting graphs are input to the IF2MEM and IF2UP for efficient array usage. After partitioning by IF2PART, C code is generated for final compilation.

```
type OneDim = array [integer];

function Main (N : integer returns OneDim)
    for initial
        I := 1;
        A := array [1:0]
    while (I < N) repeat
        I := old I + 1;
        A := array_addh (old A, 0)
    returns value of A
    end for
end function
```

Fig. 18. A SISAL program which initializes an array to value zero.

```
for initial
      I := 1
while (I <= 1000) repeat
      I := old I + A[I];
returns array of I
end for
```

Fig. 19. IF2MEM optimizer is not able to transform this program segment to preallocate memory.

of Figure 18, we see that the array size grows dynamically. Initially the array contains a single element and grows until there are N array elements. A naive implementation would allocate memory for one element. Then another memory allocation is needed to store two array elements. After the memory is created, the old values need to be copied over. This would repeat until all array elements are initialized. Clearly, this is a very time consuming process. However, this can be avoided if we preallocate memory for N array elements. Figure 19 shows another case in which this is not possible. Because the content of array A cannot be determined, it is not possible to preallocate memory.

```
type TwoDim = array [array [real]];

function Main (I,J:integer; A:TwoDim returns TwoDim, real)
      A[I,J : 0.0]I, A[J,I]
end function
```

Fig. 20. Without optimization, the two array operations in function Main causes array copying.

In SISAL semantics, execution of the two array operations of Figure 20 is implicitly parallel. One is an array replacement operation and the other is an array read operation. Since the relationship between the array subscripts of the two operations is not known, array copying results. In this example, if it can be determined that the array read operation shown is the only array read, array replacement operation can be transformed into update-in-place operation by forcing the array read operation to occur first. The IF2UP optimizer looks for such cases during its graph analysis. In general, this analysis is quite complicated.

In addition to the array optimization analysis performed by IF2MEM and IF2UP, IF1OPT performs well known conventional optimization techniques such as :

- Function inline expansion.

- Loop invariant removal.
- Record fission.
- Common subexpression elimination.
- Loop fusion.
- Constant folding.
- Dead code elimination.

5.2 Threaded Abstract Machine

In the first generation pure data-flow architectures, fine-grain parallel execution is provided by special mechanisms that are built into the hardware. For example, matching hardware is provided to dynamically schedule instructions through token matching. Also, a router hardware is provided to automatically route a computed data token to its destination. The Threaded Abstract Machine (TAM) [11, 13] takes the opposing approach in that it tries to provide fine-grain parallel computations on conventional parallel machines which provide none of these mechanisms in hardware.

To minimize the overhead of frequent transfer of control, a fast context switching capability is a prerequisite to providing fine-grain parallel execution. Otherwise, the benefit of parallel execution can be wasted by the time spent in context switching. Due to large states, however, fast context switching is not one of the strong points of modern processors. TAM overcomes this problem by addressing two key issues, namely, the scheduling strategy and the communication mechanisms.

The key point of the TAM scheduling strategy is that the memory hierarchy of the machine is taken into account. To explain this idea, let us first assume that there are a number of active code blocks in a processor. A code block is active when a frame has been allocated to it and is ready to execute, *i.e.*, at least one of its threads is ready for execution. An active code block is called an *activation*. Although there can be many activations in a processor, only one thread belonging to one activation is executed by a processor at a time.

Now, assume that a thread has just been terminated and a new thread needs to be scheduled. There are a number of possible strategies. For example, a round robin scheduling of activations can be assumed. In that case, a ready thread from a new activation is scheduled. In TAM, ready threads that belong to the current activation have higher priority than the threads belonging to other activations. A new activation is scheduled only when no other thread from the current activation can be scheduled . This strategy is better because it takes advantage of locality. In other words, by scheduling threads from current activation, values stored in processor registers and cache can be used. On the other hand, if a thread from a different activation is scheduled, the current state of the processor must be saved and loaded with the state of the new activation incurring a large overhead.

Communication is another important element in the TAM model. Without efficient communicating mechanisms, fine-grain parallel execution is not possible due to processor idling caused by long latency. In TAM, communication is

tightly coupled to computation. Thus, communication chores are handled by the compiler generated *message handler* instead of relying on the operating system services. A message handler is called an *inlet* and its objective is to quickly inject the received data into the currently executing activation threads. By doing so, activation that is already resident on the processor can continue executing instead of being switched out and brought back again later [51].

Program	Source Lines	Problem Size	Application
QS	–	–	Quicksort
GAMTEB	750	8192	Neutron Transport
PARAFFINS	300	–	Isomer Enumeration
SIMPLE	1000	128 x 128	Hydrodynamics
SPEECH	–	–	Speech Processing
MMT	130	60 x 60	Matrix Multiplication

Table 5. The list of benchmark programs used in the benchmarking of TAM on a 64-node CM-5 machine.

To verify the TAM concept, an intermediate language representation called TL0 has been developed. The backend of the Id compiler has been modified to produce multithreaded code in TL0 format instead of data-flow instructions. The resulting TL0 is then translated to C code and compiled by the native C compiler for execution. Seven benchmark programs as shown in Table 5 are used in the performance measurements. Dynamic scheduling characteristics of the benchmark programs are reported in [11] (Table 6). The 64-node CM-5 was used as a target machine. Preliminary results are encouraging in that the performance on the 64-node CM-5 is comparable to a 16-node Monsoon. Recall that the Monsoon has many special mechanisms built into the hardware [11].

The average number of TL0 instructions per thread on the compiled benchmark programs is approximately six which is comparable to the run-length of a basic block in conventional programs [11]. With such short threads, it is easy to incur a large context switching overhead. Through the scheduling strategy and the communication mechanism described above, the TAM model keeps the context switching cost to minimum by restricting the thread switching to those within the current activation as much as possible.

6 Conclusions

Research in the data-driven model has gone through two distinct evolutionary phases. The first generation data-flow architecture is motivated by the efforts to execute data-flow graphs directly on a hardware with minimum compile-time analysis. The hardware of the first generation architecture was quite complex

	QS	GAMTEB	PARAFFINS	SIMPLE	SPEECH	MMT
Avg. Thread length	2.6	3.2	3.1	5.3	6.3	17.6
Threads per Quanta	11.5	13.5	215.5	7.5	16.7	530.0
Quanta per Invocation	4.1	3.4	2.7	4.8	21.7	3.4

Table 6. Dynamic scheduling characteristics of the benchmark programs on a 64 node CM-5 machine.

because the mechanisms to provide dynamic execution model is mostly implemented into the hardware. The second generation architecture is characterized by its hybrid nature. Driven by the need to provide efficient execution mechanism for both sequential and parallel code segments, hybrid architectures possess features of both the von Neumann and the data-flow architectures. While the hardware has become simpler, more compile-time analysis is required to determine the execution mode of different progam segments.

Although many improvements have been made, more investigation is required to determine the necessary architectural features for fine, medium-grain asynchronous data-driven execution model. The work of [10] and [11] discussed in the previous section can be used as basis for future research to address this question. The Optimizing SISAL Compiler work concentrated on efficient handling of structured data (mainly arrays) on shared memory systems consisting of a relatively small number of processors. A static execution model is employed while exploiting coarse-grain parallelism. This work can be further expanded to provide multithreading execution model for a massively parallel system by integrating the TAM execution model.

In such an environment, strict array handling scheme used in the current OSC may not be sufficient. To effectively overlap computation and communication, non-strict array operation may be necessary. I-Structures currently supported in the TAM implementation provide non-strictness. This technique, however, has some drawbacks. First, I-Structures must first be allocated and initialized before using and must be deallocated after use. An efficient heap manager is a necessity, but in any case frequent memory alloc/deallocation operations will become overheads. Second, due to its split-phase read operation, using I-Structure can result in three remote messages per read/write of an array element.

An alternative array handling scheme is available that can complement I-Structures. A technique called the Direct Access Method (DAM) is developed for use in multithreading framework [34]. This technique is based on the Token Relabeling Scheme which is used in a tagged token data-flow architecture [21, 23, 33]. Unlike, I-Structures, the DAM does not use heap memory. Instead, activation frame memory is used for array element storage. Thus, overhead of heap memory alloc/deallocation is avoided. In addition, since an array element is sent *directly* from a source to a destination, number of remote messages is reduced. By analyzing program graphs at compile-time, an optimizer can decide the most appropriate array handling techniques to use for improved performance.

There exist many scientific applications which conventional parallel machines

do not perform well. This is due to the inherently dynamic run-time behavior of the applications. For example, adaptive methods for partial differential equations dynamically migrate around the simulation space depending on intermediate results [50]. By providing efficient mechanisms for sequential execution while still retaining its ability to dynamically exploit fine-grain parallelism, the data-driven execution model may be the natural solution for such applications.

References

1. D. Abramson and G. Egan. Design of a high performance data-flow multiprocessor. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 4, pages 121–141. Prentice Hall, 1991.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, pages 1–6. ACM, ACM Press, June 1990.
3. M. Amamiya. An ultra-multiprocessing architecture for functional languages. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 3, pages 95–119. Prentice Hall, 1991.
4. Arvind and R.A. Iannucci. Two fundamental issues in multiprocessing. In *Conference on Parallel Processing in Science and Engineering*, 1987.
5. Arvind and R. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Parallel Architectures and Languages Europe*. Springer-Verlag, August 1987.
6. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
7. M. Beckerle. An overview of the START(*T) computer system. Technical Report MCRC-TR-28, Motorola Inc., Cambridge Research Center, One Kendall Square, Building 200 Cambridge MA 02139, July 1992.
8. D. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, 1989.
9. D. Cann. *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550, 1992.
10. D. Cann. Retire Fortran: A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.
11. D. Culler, S. Goldstein, K. Schauser, and T. von Eicken. TAM-a compiler controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, July 1993.
12. D. Culler and G. Papadopoulos. The Explicit Token Store. In J-L. Gaudiot and L. Bic, editors, *Journal of Parallel and Distributed Computing, Special Issue:Data-Flow Processing*, pages 289–308. Academic Press, Inc, December 1990.
13. D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *ASPLOS-IV Proceedings*, pages 164–175. ACM and IEEE, ACM Press, April 1991.
14. W. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–38, April 1992.

15. K. Diefendorff and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.
16. K. Ekanadham. A perspective on id. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 6, pages 197–253. ACM Press, 1991.
17. P. Evripidou and J-L. Gaudiot. A decoupled graph/ computation data-driven architecture with variable-resolution actors. In B. Wah, editor, *Proceedings of 1990 International Conference on Parallel Processing : Vol 1 Architecture*, pages 405–414. The Pennsylvania State University, The Pennsylvania State University Press, August 1990.
18. J. Feo. Sisal. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing*, chapter 9, pages 337–386. North-Holland, 1992.
19. J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, December 1990.
20. G. Gao, H. Hum, and J-M. Monti. Towards an efficient hybrid dataflow architecture model. In E. Arts, J. van Leeuwen, and M. Rem, editors, *PARLE'91 Parallel Architectures and Languages Europe*, pages 355–371. Springer-Verlag, June 1991.
21. J-L. Gaudiot. Structure handling in data-flow systems. *IEEE Transactions on Computer*, C-35(6):489–502, June 1986.
22. J-L. Gaudiot and M. Ercegovac. Performance evaluation of a simulated data-flow computer with low-resolution actors. *Journal of Parallel and Distributed Computing*, 2:321–351, 1985.
23. J-L. Gaudiot and Y. Wei. Token relabeling in a tagged-token data-flow architecture. *IEEE Transactions on Computer*, 38(9), 1989.
24. V. Grafe and J. Hoch. The Epsilon-2 multiprocessor system. *Journal of Parallel and Distributed Computing*, 10:309–318, December 1990.
25. V. Grafe, J. Hoch, G. Davidson, V. Holmes, D. Davenport, and K. Steele. The Epsilon project. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 6, pages 175–205. Prentice Hall, 1991.
26. J. Gurd, C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.
27. J. Hicks, D. Chiou, B. Ang, and Arvind. Performance studies of the Monsoon dataflow processor. *Journal of Parallel and Distributed Computing*, July 1993.
28. K. Hiraki, S. Sekiguchi, and T. Shimada. Status report of SIGMA-1: A data-flow supercomputer. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 7, pages 207–223. Prentice Hall, 1991.
29. K. Hiraki, T. Shimada, and K. Nishida. A hardware design of the SIGMA 1- a data flow computer for scientific computations. In *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984.
30. R. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *The 15th Annual International Symposium on Computer Architecture*, May 1988.
31. R. Iannucci. *Parallel Machines: Parallel Machine Languages, The Emergence of Hybrid Dataflow Computer Architecture.* Engineering and Computer Science. Kluwer Academic Publishers, 1990.
32. M. Kallstrom and S. Thakkar. Programming three parallel computers. *IEEE Software*, pages 11–22, January 1988.
33. C. Kim and J-L. Gaudiot. A scheme to extract run-time parallelism from sequential loops. In *Proceedings of the 5th ACM International Conference on Supercomputing.* ACM, ACM Press, June 1991.

34. C. Kim and J-L. Gaudiot. A direct array handling technique for non-strict and parallel accesses in a multithreaded architecture. Technical Report CENG-93-01, University of Southern California, January 1993.
35. S. Komori, K. Shima, S. Miyata, T. Okamoto, and H. Terada. The data-driven microprocessor. *IEEE Micro*, 9(3):45–59, June 1989.
36. J. McGraw, D. Kuck, and M. Wolfe. A debate: Retire FORTRAN? *Physics Today*, 37(5):66–75, May 1984.
37. J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL Language Reference Manual Version 1.2*, March 1985.
38. R. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT, July 1991.
39. R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *The 16th Annual International Symposium on Computer Architecture*, 1989.
40. R. Nikhil and Arvind. Id: a language with implicit parallelism. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing*, chapter 5, pages 169–215. North-Holland, 1992.
41. R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *The 19th Annual International Symposium on Comuter Architecture*, pages 156–167. ACM, ACM Press, May 1992.
42. H. Nishikawa, H. Terada, S. Komori, K. Shima, T. Okamoto, and S. Miyata. Architecture of a VLSI-oriented data-driven processor: the Q-v1. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 9, pages 247–264. Prentice Hall, 1991.
43. R. Oldehoeft and D. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, 1988.
44. A. Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., second edition, 1987.
45. S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. Design of the dataflow single-chip processor EMC-R. *Journal of Information Processing*, 13(2):165–173, 1990.
46. M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *The 19th Annual International Symposium on Computer Architecture*, pages 146–155. ACM, ACM Press, May 1992.
47. S. Skedzielewski. Sisal. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 4, pages 105–157. ACM Press, 1991.
48. S. Skedzielewski and J. Glauert. *IF1 An Intermediate Form for Applicative Languages.* Computing Research Group, Lawrence Livermore National Laboratory, P.O. Box 808, L-306, Livermore, CA 94550, 1985.
49. V. Srini. An architectural comparison of dataflow systems. *Computer*, pages 68–88, March 1986.
50. T. Sterling and M. MacDonald. The realities of high performance computing and dataflow's role in it: Lessons from the NASA HPCC program. In M. Cosnard, K. Ebcioglu, and J-L. Gaudiot, editors, *IFIP Transactions: Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 165–176. IFIP, North-Holland, January 1993.
51. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual Inter-*

national Symposium on Computer Architecture, pages 256–266. ACM and IEEE, ACM Press, May 1992.

52. I. Watson and J. Gurd. A practical data-flow computer. *IEEE Computer*, 15(2):51–57, February 1982.
53. M. Welcome, S. Skedzielewski, R. Yates, and J. Ranelletti. *IF2: An Applicative Language Intermediate Form with Explicit Memory Management.* University of California Lawrence Livermore National Laboratory, December 1986.

This article was processed using the LaTeX macro package with LLNCS style