SINAN: An Argument Forwarding Multithreaded Architecture*

Soner Önder and Rajiv Gupta
Department of Computer Science
University of Pittsburgh
Pittsburgh PA 15260
{soner,gupta}@cs.pitt.edu

Abstract

The direct execution of data flow graphs by data flow machines exposes the maximum amount of parallelism in a computation. However, it also results in undesirable properties such as: high overhead due to data replication required to implement high fanouts in the data flow graph; poor instruction and data locality since the scheduling of ready instructions is not sensitive to data locality; and less efficient execution of sequentially dependent code sections and vector operations due the inability of the data flow machines to support deep pipelines. SINAN offers a novel approach called argument forwarding *that can eliminate much of the data fanout overhead and provides greatly improved locality. We show that vector operations and sequential segments can be efficiently handled with the dataflow paradigm by a novel approach that* dynamically *forms the activity templates.*

1 Introduction

Since the introduction of the data flow concept in the late 1960s by Adams [1] the concept has evolved into many forms. The resulting architectures can be classified into two broad categories, namely, *pure dataflow architectures* and *dataflow based multithreaded architectures*. Most of the latter architectures are dataflow Von Neumann hybrids while the former are descendants of either static dataflow [7] or dynamic dataflow [3, 4] architectures. Both static and dynamic dataflow machines have similar implementation and efficiency problems which are discussed below.

First, a dataflow node may generate a value which is needed in multiple destinations [8]. The number of destinations is defined to be the *fanout* of an instruction. Some implementations allow an arbitrary number of destinations [7] which result in long running instructions that require cycles proportional to the fanout. On the other hand, some implementations allow only a limited fanout, and introduce explicit forwarding instructions to handle any additional destinations [18]. Second, direct execution of dataflow graphs may lead to poor locality. Poor locality makes the use of caches less effective, and may contribute to the inefficiency of the execution. Finally, well structured program constructs such as vector operations and sequential segments may be executed more efficiently by conventional processors. This is attributable to the well-formed memory access patterns that can be handled efficiently by multiple banks of memory, and more efficient and deeper pipelining of vector operations. On the other hand, incorporation of vector operations into dataflow graphs is non-trivial.

Multithreaded machines aim at solving the above problems by handling dataflow at multiple levels [14, 9], or switching between the Von Neumann and dataflow style of execution [10]. A dataflow hybrid machine designed by Dennis and Gao [6, 11] proposes *argument fetching* as a solution to the fanout and excessive token circulation problems. In this solution, the availability of data values is signalled by using a separate signal graph, while instructions fetch operands from the memory and store results into the memory like a Von Neumann machine. The significance of Gao's approach is that it clearly separates the two fundamental aspects of the computation, namely the data flow and the control flow, from each other. The program counter of Von Neumann machine is replaced by signal graphs, while the data flow aspect of the computation remains Von Neumann. Although this elegant technique eliminates much of the token traffic, the resulting architecture is subject to the same memory latency problems as the Von Neumann machines.

*Supported in part by the National Science Foundation Presidential Young Investigator Award CCR-9157371, Hewlett-Packard Labs, and Intel Corporation to the Univ. of Pittsburgh.

Existing attempts to overcome the implementation and efficiency problems of pure data flow machines have focused on hybridization of the concept by borrowing well-established concepts from Von Neumann machines. In this paper, we present the SINAN architecture that is different from existing approaches in what it borrows from Von Neumann machines. We believe that an effective hybrid model of execution should combine the *data flow* concept from the dataflow machines and the *control dependence* concept from the Von Neumann computer, so that instructions are fired only if both data and control dependences are satisfied. This kind of approach has a number of advantages over existing ones. Partitioning the program into threads by observing the control dependences enable us to exploit coarse grain parallelism [5]. On the other hand, imperatively executing dataflow style instructions greatly improves locality, solves the data fanout problem, and enables us to exploit fine grain parallelism within a thread by using a dynamic instruction scheduling algorithm which is not subject to the limitations of Von Neumann machines that employ dynamic scheduling.

In section 2, we present the machine model for SINAN. We show how our idea of forwarding can be used to avoid the problems of data flow machines. In section 3, a brief overview of an implementation of SINAN is presented. In section 4, we conclude with discussion of some other related work.

2 The Machine Model

SINAN envisions the user program as a collection of threads and array objects. Each thread corresponds to a loop free connected region of the flow graph [19]. The *creation* of a thread instance results in allocation of a data segment for the instance. The instance of a thread may be *initiated* as soon as it is created or at a later point in the program. Thus, if a loop body represents a thread, then for each loop iteration a distinct instance of the thread is created. Allowing forward branches in a thread distinguishes our threads from other hybrid architectures such as Dennis's Modern Static Data Flow Machine. SINAN reserves operand slots in the data segment corresponding to each instruction in the code segment. The juxtaposition of a thread's code segment with its data segment forms a structure where each instruction is associated with two source operands from the data segment. Each such tuple represents an activity template shown in Fig. 1. Separate instruction pointer (IP) and data pointers (OPL,OPR) that are moved synchronously keep the association between the instructions and the data.

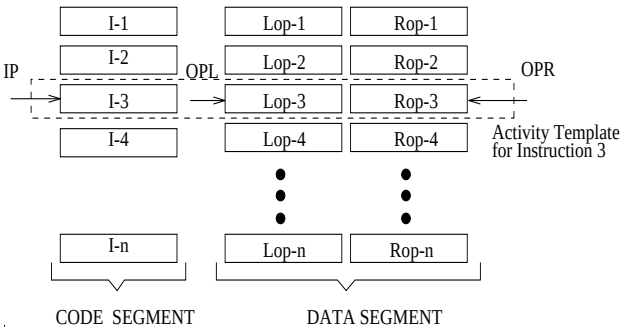

Figure 1: Activity Templates in SINAN.

Within a thread, instructions are arranged by the compiler for fast sequential execution and a program counter is used to execute them. Instead of performing normal load and store operations for scalars, we employ argument forwarding in which scalar values do not have fixed memory locations assigned to them, but flow from one activity template to another as it is the case with pure dataflow machines. Thus, argument forwarding can be described as using a dataflow like instruction format, but employing a program counter to select the next *fired instruction* instead of the usual firing rules. Since argument forwarding is employed for a loop free connected region, the matching overhead is not seen for these segments, while the dataflow aspect of the computation remains dataflow.

Scalar data items always reside in data segments and are forwarded from one data segment to another as required by the execution of the program. However, the array data is stored in separate objects shared by the threads. Thus, the value of an array element must either be explicitly loaded into the thread's data segment, or must be computed and forwarded by another thread. Similarly, the value of an array element computed by a thread is stored into the array object. Therefore, SINAN provides non-blocking load and store instructions. Load instructions accompany the load request with a destination identifier pointing to a specific offset in a thread's data segment. Thus, a thread may perform load requests for other threads, achieving better tolerance to memory latency. Load and store instructions are not required for all array references. A flow sensitive array dependence analysis can be used to achieve the flow of array data using the forwarding mechanism, whenever possible.

In subsequent subsections we show how forwarding addresses some of problems associated with dataflow machines.

2.1 Forwarding and data replication

The fanout problem arises in a program segment when a value computed in one instruction is used by several instructions. For example, in the code segment in Fig. 2 the value of variable *a* computed in I1 is used by both I2 and I3, and must be sent to both of the places. Whether a fixed fanout or an unlimited fanout capability is provided in the machine, the execution of such a sequence will result in replication overhead due to the multiple uses of the value of `a`. Ideally, we would like to pay the overhead of replication only for those cases where the control dependence information guarantees that the destination instruction will be executed, or if two or more computations needing the same value can execute in parallel.

Argument forwarding technique augments each instruction with the ability to send its *source operands* as well as its *result* to some destinations. The cost of replication is distributed and amortized over instructions that do not fully utilize the data forwarding capability of the instructions. Under such a scheme, data values flow from an executed instruction to future instructions at the same time that the execution follows the flow of control. Thus data forwarding and the execution of computational instructions are *folded* together, and as long as there is a definite def-use relationship between the forwarding instruction and the receiving instruction, the forwarding overhead is not seen. That is, we can use operand forwarding (or reforwarding) only if the instruction which will perform source operand forwarding is guaranteed to execute. Preliminary experiments we carried out with imperetive languages indicate that even though initial fanout for some instructions may be high, with argument forwarding, most of the overhead can be amortized.

Obviously, the above scheme cannot be used to replace the data replication blindly. In order for the argument forwarding to work, instructions must be arranged in an appropriate order by the compiler. Consider the example in Fig. 2(a) where the second instruction receives the value of `z` from the third instruction. The dataflow approach discovers the correct sequence of execution because it does not rely on the order in which instructions are arranged. The same program segment will deadlock when argument forwarding is used as shown in Fig. 2(b). However, if the instructions are rearranged as shown in Fig. 2(c), the execution proceeds normally. Furthermore, the execution does not involve extra overhead due to replication unlike the first sequence used by a data flow machine. Indeed, this is where our savings come from. With argument forwarding, the compiler is able to

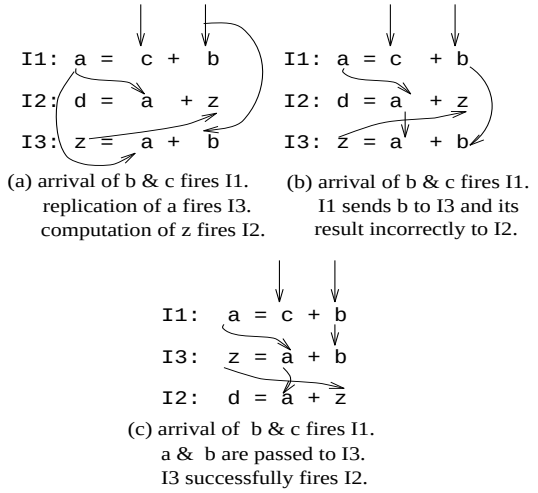

Figure 2: Traditional Data Flow vs Forwarding.

take advantage of information available during compile time to arrange the instructions, while a dataflow machine pays an extra run-time price for determining an instruction order. In the above example, there is no parallelism and therefore there is no benefit to be derived from out-of-order instruction execution. We show that a forwarding-based machine can easily exploit instruction level parallelism within a thread by incorporating a powerful dynamic instruction scheduling scheme. Thus, the flexibility and hence the cost of providing a *total lack of static instruction ordering* in data flow machines does not seem to be justified.

2.2 Forwarding to build locality

In general, instruction caches show better behaviour than data caches in Von Neumann processors. This is because, successive instructions are arranged in successive locations. On the other hand, high cache hit rate for data is much harder to achieve because the data items consumed by successive instructions need not be stored in successive memory locations. When we compare the dataflow execution style with that of Von Neumann, it is not difficult to notice that coupling of instructions with data and a data driven execution forces the instruction references to follow the same reference pattern as the data references.

In argument forwarding technique, instruction references show a similar pattern to Von Neumann machines, because a program counter is used to execute the thread. However, since the data and the instructions are coupled together, a hit for an instruction reference also means a hit for the data reference. Therefore, in a forwarding architecture a code segment accompanied by its data segment results in a simulta-

neous code and data hit within a thread. Moreover, because of argument forwarding, we are able to organize data such that all reads are performed to successive locations (because as the instruction pointer is incremented, so are the operand pointers) for both instruction and data fetches.

This is indeed a very desirable property, because multiple banks of memory can be effectively used to reduce the memory latency when memory accesses are stride one accesses. As execution proceeds and additional threads are activated, data values flow from one thread to another thread's data segment, effectively creating the locality for both running and waiting threads. Obviously, this scheme will only work if the data segment for the destination thread is allocated statically, or if the compiler ensures that the data segment is allocated before any potential producer threads are scheduled for execution.

2.3 Forwarding for vectorization

Vector machines are typically equipped with special vector units that implement the vector operations at high speed. These units take advantage of deep pipelines to cope with the floating point latency and exploit uniform memory access patterns to cope with the memory latency problem. On the other hand in SINAN, due to the unique forwarding mechanism, vector operations do not require specialized vector units. Furthermore, there is no need for separate vector instructions. Consider the vector summation given in Fig. 3(a). If we unroll the loop n times, and assume that the values of the vector `a` has been computed and forwarded to the data segment containing the vector loop, we obtain the set of activity templates shown in Fig. 3(b).

(a) for (i=1; i < n; i++) a[i]=a[i]+c
(b) add → a[1] c
add → a[2] c
...
add → a[n] c.

Figure 3: Vector addition.

In the unrolled example, one of the source operands remains the same, and the same instruction is repeatedly used. Although the instructions have different destination addresses, each instruction's result is separated from the preceding one by a constant stride. By including minimal additional hardware support we can *dynamically* realize the activity templates shown in the unrolled loop. By including three flags in the instruction format indicating that the instruction must be re-issued, left operand must be reused, and right operand must be reused, we achieve a scheme where the activity templates can be dynamically formed. Thus, there is no need to unroll a vectorizable loop. If we selectively *hold* the IP, OPL and OPR pointers based on the above flags, all vector operations may be realized. Thus a vector-vector operation will specify only hold program counter flag, while a vector-scalar operation will specify an additional flag regarding the scalar operand. The destination address portions of any result tokens produced must be incremented if a vector result is being formed. Special registers are provided to hold the constant stride and the vector size.

2.4 Program partitioning

Compiler techniques already developed for lenient languages [20] can be successfully employed on SINAN architecture. In order to provide an understanding of how SINAN achieves the balance between the right amount of hardware and compiler support for multithreading, we give a brief sketch of the process of partitioning an imperative program into its threads.

Although thread instances are created and activated dynamically, partitioning of a program is carried out by the compiler. In general, a loop free code segment consisting of one or more basic blocks is a candidate for being compiled as a thread. Thread creation and initiation are treated as two distinct operations. A thread is initiated by issuing a continuation token consisting of a code descriptor (C) and a data descriptor (D). An executing thread (ET) can initiate another thread (T) as soon as it is certain that the thread T will be executed in the current execution of the program. If the control flow dictates that the execution of thread ET guarantees that thread T must be executed, then ET can initiate the execution of T early; thus allowing parallel execution of ET and T. On the other hand, if the execution of T is dependent upon the outcome of a conditional branch in ET, then thread T cannot be initiated till the outcome of the branch is determined.

The example in Fig. 4 illustrates the partitioning of a program graph into threads, selection of points at which the threads are to be created and initiated, and the flow of values among threads and between memory and threads. During the execution of any given thread the instruction level parallelism is exploited using a dynamic scheduling mechanism discussed in the next section. By simultaneous execution of threads parallelism across threads, and thus across basic blocks and loops, is also exploited. In our example first the execution of threads 1 and 2 is overlapped. Next, in-

```
Read N, B[1..N]
A[0] = SUM[0] = 0
l = 1; h = N;
repeat
    l = l + 1
    h = h − 1
    for i = l to h do
        A[i] = A[i − 1] + B[i]
    end
    for j = l to h do
        SUM[j] = SUM[j − 1] + A[j]
    end
    total = sum[h]
    print l, h, total
until l = h
```

```
Thread 1:
    Create and Initiate Thread 2
    Read N, B[1..N]
    A[0] = SUM[0] = 0
    l = 1; Forward l to Thread 2
    h = N; Forward h to Thread 2
Thread 2:
    Create and Initiate Threads 3,4, and 5
    i = j = l = l + 1
    Forward i to Thread 3, j to Thread 4,
        and l to Thread 5
    h = h − 1
    Forward h to Threads 3, 4 and 5
Thread 3:
    A[i] = A[i − 1] + B[i]
    i = i + 1
    if i ≤ h
        Create next instance of Thread 3
        Forward h, i, A[i − 1] to next instance
            of Thread 3
        Forward A[i − 1] to Thread 4
        Initiate next instance of Thread 3
Thread 4:
    SUM[j] = SUM[j − 1] + A[j]
    j = j + 1
    if j ≤ h
        Create next instance of Thread 4
        Forward h, j, SUM[j − 1] to next instance
            of Thread 4
        Initiate next instance of Thread 4
    else Forward SUM[j − 1] to Thread 5
Thread 5:
    if l ≠ h
        Create and Initiate Thread 2
        Forward l, h to Thread 2
    total = SUM[h]
    print l, h, total
```

Figure 4: A Partitioning of a Program Graph into Threads.

stances of threads 3 and 4 are created thus allowing simultaneous execution of an iteration each from the two for loops. Finally the execution of thread 5 and next instance of thread 2 is overlapped. In Fig. 4 the forwarding of values to a given thread is carried out as soon as the values become available and the destination thread has been created. This process allows threads to execute in parallel.

3 Implementation Overview

Many machine designs incorporate both a cache and a register file to capture the locality at various levels of program execution. We believe this type of memory hierarchy arrangement is a direct consequence of Von Neumann style program execution where memory references usually follow a non-uniform pattern, except for the code and vector data references. On the other hand, because of argument forwarding, an executing thread performs non-uniform memory references only when load/store instructions are used. As we pointed out earlier, a code segment and a data segment are well formed entities that can be efficiently streamed into the execution units. This type of arrangement calls for an *active memory subsystem*, because many of the memory references can be completed even before the execution of a thread starts.

On another front, assuming one-cycle instruction execution, the execution of an instruction may require at least three memory reads and two or more writes depending on the forwarding capability of the machine every cycle. This is a demanding requirement. Considering the fact that a quite significant portion of these writes stay within a thread, the architecture can be better realized by using only one fast store to hold instructions and data, and not utilizing caches. An implementation of a register file that allows storing of instructions and data directly to the registers is given by Anido *et al.* in [2]. In fact, these decisions are consistent with the practice of many vector supercomputers which also do not employ caches but rely on fast register files and interleaved accesses to multiple banks of memory.

As shown in Fig. 5, the SINAN architecture is modeled as an array of identical processors connected through an interconnection network similar to Iannucci's machine [13]. However, each node of SINAN contains multiple processing elements called register units which contain one such fast store and communicate over a local bus with the thread manager and local memory. SINAN uses a uniform, descriptor based addressing scheme. Descriptors are used

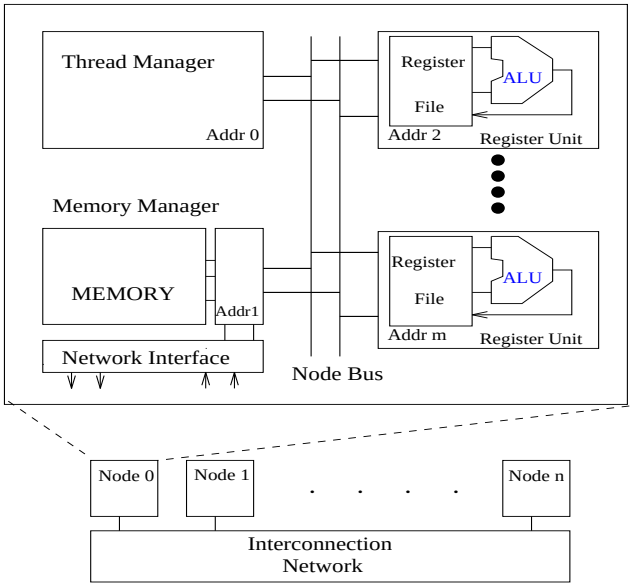

Figure 5: SINAN - Main Components.

to describe physical units as well as software objects such as code segments and data segments. The address space is global and is physically and logically distributed among multiple nodes. The higher order bits of the address field of descriptors are used to select the node while the lower order bits locate the object within that node. Each unit is assigned a unit address. The thread manager is assigned the unit address zero, the memory manager is assigned the unit address one, and the register units have addresses ranging from 2 to m. The units contained in a node operate asynchronously and communicate with each other by message passing. These messages perform activities such as acquiring a register unit, loading a thread, performing loads and stores, initiating thread execution and freeing of register units.

As it is pointed out in [16], a direct consequence of message passing style communication is that transactions must be split phase transactions, i.e., a transaction is split into request and response messages. SINAN's split phase transactions involve simultaneous transmission of an opcode and two descriptor words. The opcode field determines the type of the transaction and the descriptors specify the source and the destination units involved. Since a thread is composed of a data segment and code segment, two descriptors are used to define a thread. A result computed by an instruction in a thread is either forwarded to another instruction in the same thread or it may be required by another thread.

A scalar value forwarded to an instruction within the same thread stays within the register unit. However, a scalar value required by another thread is sent to memory where it is stored in the data segment of the appropriate thread. If the receiving thread is also loaded in a register unit and currently executing, then it would be beneficial to directly send the scalar data value to the register unit and avoid the need for a load in the receiving thread. This is achieved by having each register unit *snoop* for the data belonging to the data segment currently loaded in the register unit. Thus, the data values are always sent to memory and picked up by another register unit if its destination segment matches to some data segment loaded in a register unit. This mechanism guarantees that the values of a data segment loaded in the register unit and corresponding values in the copy of the data segment in the memory are always consistent. It should be noted that a register unit must monitor the node bus for descriptors containing one of the two following ids: it's physical id; and the id of the data segment currently loaded in the register unit. Since the latter id changes as different data segments are loaded in the register unit, we refer to this process as unit *renaming*. Special attention should be paid to the situation in which a thread sends values to the data segment corresponding to another thread which is in the process of being loaded. In this situation, the value received from the memory is stale and the value received from another thread is the true value. Tag bits are provided for registers to ensure that the value received from a thread executing in another register unit is not overwritten by the stale value received from the memory.

The thread scheduling and resource allocation process is implemented by a five stage pipeline. In the *token fetch* stage, a continuation token is dequeued from the token queue and in the remaining stages an attempt is made to schedule the execution of this thread. In the *resource fetch* stage the RU-Status table is examined to determine if a free register unit can be found. If no free units are available, the pipeline stalls until one becomes available. If a register unit is found, first the unit is renamed to the data segment descriptor during the *unit rename* stage. Next the loading of the data segment and the code segment for the current thread is initiated by the *initiate data fetch* and *initiate code fetch* stages respectively. If the chosen register unit already contains the appropriate code/data segment, then the code/data fetch stages are by-passed. The by-passing is analogous to having a cache hit for data/instructions in the register unit. Thus, if a thread representing a loop body is repeatedly initiated the code for the thread is likely to be

loaded into the register unit once and reused by the subsequent instances of the thread.

The instruction format used by SINAN is shown in Fig. 6. The opcode is merely a coding of the control lines for the ALU. All instruction address fields, namely, D, NS1, and NS2 are offsets from the start of a data segment of a thread and normally reference to the current thread and therefore they are actually indices to the register file. The D field points at either the left or the right operand of the instruction that will use the result of the computation. The NS1, and NS2 fields point at the instructions that will use the source operands next. S1I and S2I point at the locations in the register file where the source operands for the current instruction are located. Finally, the control field contains bits such as HoldPC, HoldS1, and HoldS2.

OPCODE	D	NS1	NS2	S1I	S2I	CONTROL

Figure 6: SINAN Instruction Format.

The above instruction format is based upon the data-flow model of computing. Every instruction has its own set of data sources, instructions may be executed out of order and may run to completion out of order. Therefore, we are able to employ a very simple algorithm to exploit fine grain parallelism within a thread. Data hazards which occur in traditional pipelined architectures do not occur here. Because the variables are continuously renamed, write-after-read and write-after-write hazards cannot occur. Read-after-write dependencies are enforced by the tags associated with the registers. These tags show if the corresponding location is empty or not. An instruction is not issued when the empty/full bit for an operand is set to empty. Two additional bits in the control portion of the instruction specifies the value of tag bits for the left and right operand after the instruction is issued.

The execution of a thread can be in one of two states. During the first state we scan the instructions by using the program counter and issue those instructions that are ready for execution. This will typically result in the issuing and completion of integer instructions and the issuing of floating point operations that are ready. The instructions that are not issued are inserted into a FIFO during this state. When a branch instruction that cannot be issued is encountered, the execution switches to the second state in which instructions are taken from the FIFO one by one and their issuing is attempted. If the instruction cannot be issued, it is returned to the FIFO. When the FIFO becomes empty, the branch that caused the execution to switch to the second state is executed and the execution returns to the first state. It should be noted that if the code contains no conditional branches, the scanning continues unhindered. If a conditional branch is encountered, the scanning continues if the conditional branch can be executed. The execution is only stalled if the operands for the conditional branch are not yet available. Typically the FIFO will only contain instructions that are waiting for operands from long latency operations such as memory loads and floating point operations. To improve the likelihood that the scanning is not stalled at conditional branches and an instruction is likely to be ready when it is examined for issuing, the compiler employs code reordering techniques. Although SINAN executes threads in a Von Neumann fashion, our approach to dynamic scheduling is far more powerful than the dynamic scheduling techniques used in superscalar processors. In superscalar processors out of order issuing of instructions can be carried out within a small window of instructions. In SINAN this window extends for the entire length of the thread provided the FIFO is long enough to hold all unissued instructions. Furthermore, the hazards due to register usage encountered in superscalar processors are not encountered in SINAN.

The instruction format and the use of a register file makes forwarding quite efficient within a thread. However, in order to handle inter-thread communication efficiently, references to other threads require a different approach. In order to send a value another thread, a descriptor which points at that thread must be loaded into the *memory address register out* (MAROUT). Thus a store instruction can be emulated by specifying the destination descriptor as the source operand 1 and the value to be stored as the source operand 2 for an instruction and executing a no-op. The NS1 field of the instruction must address the MAROUT and the NS2 field of the instruction must address MDROUT. When the instruction is executed, the source operands are forwarded to MAROUT and MDROUT respectively, which in effect pushes the new values of these registers to a FIFO. From the FIFO, these values are broadcast on the bus. This technique works well for storing single scalar values to other threads and array objects. However, this technique does not allow efficient streaming between two threads. When vectors flow between different threads in a streamed fashion (as during *vector chaining*), or two threads are closely coupled such that they frequently exchange values, the same descriptor

can be reused, and only its offset field is modified.

4 Concluding Remarks

The *argument forwarding* concept is equally applicable to both dynamic and the static data flow models. As shown by Dennis [8], static and dynamic dataflow models are not that different. The difference between the models lies mainly in the way frame allocation is handled. Although Sinan is based on the static dataflow concept in the way the activation templates are formed, activation templates can also be formed dynamically. In addition, frame allocation is done dynamically as it is the case with dynamic dataflow machines. In this respect, it is also a hybrid of static and dynamic dataflow machines.

The design of SINAN has been inspired by many other processors [13, 12, 17, 16, 15], and is primarily based upon concepts of data-flow computing. However, we believe SINAN is one of the first architecture to show the close relationship between the way flow of data is handled, locality and vector operations. We share the ideas with the TAM approach [5] and the Monsoon [17] in that, effective solutions can only be found if a good model of execution is superimposed on the hardware and software design.

References

[1] D. A. Adams, "A model for parallel computations," In L.C. Hobbs et al., editor, *Parallel Processor Systems, Technologies, and Applications*, pages 311–333. Spartan, 1970.

[2] M. L. Anido, D. J. Allerton, and E. J. Zaluska, "A three-port/three-access register file for concurrent processing and I/O communication in a RISC-like graphics engine," *Proc. 16th Int. Symp. on Computer Architecture*, pages 354–361, 1989.

[3] Arvind and K. P. Gostelow, "Some relationships between asynchronous interpreters of a dataflow language," *Proc. IFIP WG2.2 Conf. on Formal Description of Programming Concepts*, 1978.

[4] Arvind, V. Kathail, and K. Pingali, "A dataflow architecture with tagged tokens," Technical Report 174, MIT, Lab for Computer Science, 1980.

[5] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[6] J. B. Dennis and G. R. Gao, "An efficient pipelined dataflow processor architecture," *Proc. IEEE/ACMConf. on Supercomputing*, 1988.

[7] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data flow computer," *Proc. 2nd Annual Symp. on Computer Architecture*, 1975.

[8] J. B. Dennis, "The evolution of "static" data-flow architecture," In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 35–91. New Jersey: Prentice Hall, 1991.

[9] P. Evripidou and J-L. Gaudiot, "The USC decoupled multilevel data-flow execution model," In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 347–379. New Jersey: Prentice Hall, 1991.

[10] G. R. Gao, "A flexible architecture model for hybrid data-flow and control-flow evaluation," In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 327–346. New Jersey: Prentice Hall, 1991.

[11] G. R. Gao, R. Tio, and H. H. J. Hum, "Design of an efficient dataflow architecture without data flow," Technical Report TR-SOCS-88.14, School of Computer Science, McGill University, 1988.

[12] V.G. Grafe, G.S. Davidson, J.E. Hoch, and V.P. Holmes, "The ϵpsilon dataflow processor," *Proc. 16th Int. Symp. on Computer Architecture*, pages 36–45, 1989.

[13] R.A. Iannucci, "Toward a dataflow/von Neumann hybrid architecture," *Proc. 15th Int. Symp. on Computer Architecture*, pages 131–140, 1988.

[14] W. Najjar and J-L. Gaudiot, "Macro data-flow architecture," In J.A. Sharp, editor, *Data Flow Computing: Theory and Practice*, pages 272–291. Ablex Publishing Corporation, 1992.

[15] R. Nikhil and Arvind, "Can dataflow subsume von neumann computing?," *Proc. 16th Int. Symp. on Computer Architecture*, pages 262–272, 1989.

[16] R. S. Nikhil, G.M. Papadopoulos, and Arvind, "*T:a multithreaded massively parallel architecture," *Proc. 19th Int. Symp. on Computer Architecture*, pages 156–167, 1992.

[17] G.M. Papadopoulos and D.E. Culler, "Monsoon: An explicit token-store architecture," *Proc. 17th Int. Symp. on Computer Architecture*, pages 82–91, 1990.

[18] G. M. Papadopoulos, "Implementation of a general purpose dataflow multiprocessor," Technical Report TR–432, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[19] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, "Thread-based programming for the EM–4 hybrid dataflow machine," *Proc. 19th Int. Symp. on Computer Architecture*, pages 146–155, 1992.

[20] K. E. Schauser, D. E. Culler, and T. von Eicken, "Compiler-controlled multithreading for lenient parallel architectures," Technical Report TR–91–640, EECS Dept., Univ. of California, Berkeley, 1991.