A Direct Array Handling Technique for Non-strict and Parallel Accesses in a Multithreaded Architecture

Chinhyun Kim and Jean-Luc Gaudiot

CENG Technical Report 93-01

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

A Direct Array Handling Technique for Non-strict and Parallel Accesses in a Multithreaded Architecture*

Chinhyun Kim† *Jean-Luc Gaudiot*†

Abstract

In this paper, we propose an array handling technique called the Direct Access Method (DAM) which enables array elements to be sent *directly* from the producer to the consumer activation while providing non-strict and parallel array accesses. The advantage of this technique over conventional structure storage is that network traffic is reduced and that no global memory space is required. Instead, array elements are produced in the frame memory of the producer activation and forwarded directly to the frame memory of the consumer activation. It is demonstrated here how the technique can be fine-tuned to those cases where an array behaves as a temporary variable and its consumption pattern can be determined at compile-time. Thus, the DAM is proposed as a complement rather than a replacement of the I-Structure representation. Performance measurements obtained by a deterministic simulation of a multithreaded model show that the Direct Access Method indeed performs better than the equivalent I-Structure implementation. The measurements further show that although the I-Structure model has a lower execution time under low network latency conditions, the DAM displays a performance advantage for higher communication network latencies.

1 Introduction

Almost all scientific applications which can benefit from parallel computing entail the manipulation of large data structures. The resulting performance can vary greatly depending on how these large structured data are partitioned and distributed across multiple processing elements. This must be done in such a way as to fully exploit existing parallelism with as little overhead as possible. Since communication accounts for much of this overhead, efficient structure representation and transmission schemes are needed. This paper presents an array handling technique geared mainly for scientific applications in an environment where the computation model is data-driven while the target machine is a multithreaded architecture.

Although every data partition and allocation scheme attempts to render all memory accesses local, remote memory accesses cannot be completely avoided. Frequent remote accesses can have a detrimental effect on the overall performance because of their long latency which can result in low processor utilization. The data-flow computing approach addresses the data partition/allocation problem indirectly by striving to provide a *latency tolerant* computing environment. The adverse effect of long latencies can be indeed minimized by effectively overlapping computation and communication[7]. To this end, three basic requirements must be satisfied:

1. The compiler should be able to expose all the fine/medium grain parallelism contained in the user's program.
2. The architecture of the target machine should be able to exploit such parallelism efficiently.
3. Communication costs should be minimized or at least *tolerable*.

The following is the current data-flow solution to the above requirements:

1. Use a functional language as a programming language which makes it relatively easy for a compiler to extract parallelism.
2. Use a multithreaded architecture to efficiently exploit fine/medium grain parallelism exposed by the compiler.
3. Use I-Structures[4] when arrays can be produced and consumed in parallel.

The concept of I-Structures is an important component of the overall data-flow solution because its objective is to exploit producer-consumer parallelism. This is achieved by providing non-strict and parallel accesses coupled with split-phased memory operations. Specifically, each of the I-Structure memory cell contains presence bits indicating the status of the cell *i.e.*, *full*, *empty*, or *pending*. Initially, each cell is set to empty. If a read request is made to an empty cell, the state of the cell is changed to pending and the request is queued in the deferred read request list. When the cell is written, the state changes to present. If the cell is marked pending prior to being written to, all deferred read requests are serviced.

*This work is supported in part by the National Science Foundation under grant No. CCR-9013965.

†Electrical Engineering-Systems University of Southern California Los Angeles, California 90089-2562

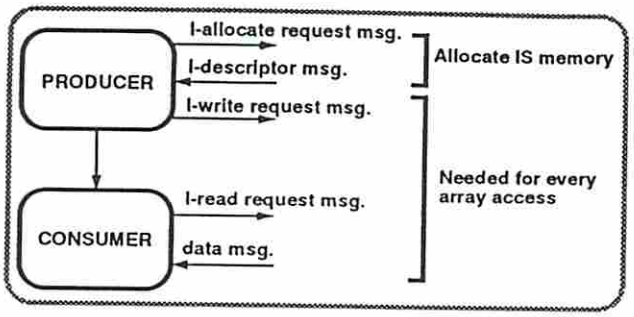

Figure 1: I-Structure and producer-consumer parallelism.

In addition to providing mechanisms to support the status of each memory cell, a memory manager which can efficiently handle dynamically created and deallocated arrays is also required. A complex hardware configuration of the I-Structure memory may be required to provide parallel array accesses as well. For example, the I-Structure memory can be low-order interleaved across the I-Structure nodes[20]. Alternatively, it can be high-order interleaved[14]. However, providing parallel accesses and split-phased read operations can cause excessive network traffic. This means that in the worst case, three remote messages are required for every array element to be written or read.

This overhead associated with the I-Structure representation may render it prohibitively expensive when an array behaves similarly to a temporary variable. This refers to cases where the life-time of an array is short because it is quickly consumed and destroyed after being created. In such situations, the fraction of the total time attributable to the overhead (allocate/deallocate + network traffic) may be relatively large resulting in inefficient operations. This paper thus introduces an alternate array handling mechanism called the *Direct Access Method* which, just like the I-structures, provides non-strict and parallel accesses, minus the overhead. We will show how this method can be used in situations where the arrays behave like temporary variables while the I-Structure representation is used for those situations where the arrays behave more like global variables. We will demonstrate that the Direct Access Method is an efficient array handling scheme in which each array element is sent *directly* from the producer to the consumer without being stored in an intermediate array storage. The mechanism makes the issue of structure memory management moot and reduces the number of remote accesses because write operations are guaranteed to be local memory operations.

In summary, the objective of this paper is to introduce the Direct Access Method for the passing of large chunks of data between producer and consumer processes. It is also intended to demonstrate its performance as well as its most likely context of utilization. Section 2 discusses how the data-flow computing model evolved from pure data-flow to multithreading. Section 3 introduces the Direct Access Method and its predecessor, the Token Relabeling approach. The simulation results which demonstrate the performance and the applicability of the Direct Access Method are presented in section 4. Finally, concluding remarks and directions for future research are offered in section 5.

2 Multithreading

We will now discuss first how the principles of data-flow computing have been evolving from pure data-flow toward multithreading, then we will describe the multithreading execution model. Multithreading is viewed as a practical means to promote data-flow computing rather than as a conceptually different model.

2.1 From Data-flow to Multithreading

One of the first basic tenets of parallel computing must be the ability to expose as much parallelism as possible at compile-time so that parallel (independent) instructions can be distributed across multiple processing elements of a machine and executed concurrently, resulting in higher performance. Traditional approaches to parallel computing using conventional languages such as FORTRAN make it difficult for a compiler to extract large amounts of parallelism automatically. This is due to the nature of the language semantics which are based on the von Neumann computation model: the model is inherently sequential in nature. On the other hand, the data-flow approach to parallel computing consists in starting at the top with languages such as Id[17] and SISAL[16] whose functional semantics make it comparatively easy for a compiler to expose all the parallelism contained in a program.

Being able to exploit large amounts of parallelism is crucial to the data-flow approach because its objective is to tolerate communication latency rather than to reduce it by executing ready instructions whenever long-latency inducing operations are initiated. By effectively overlapping communication with computation, latency caused by remote accesses can be masked[22]. Therefore, it is important to possess excess tasks ready for execution whenever long latency which is unavoidable in parallel computing occurs[17]. First generation data-flow machines were designed to switch context at every instruction, *i.e.*, the length of a thread is one instruction. For this purpose, a complex hardware facility called the *match* unit was developed to synchronize incoming data tokens and dynamically schedule instructions depending on the availability of their input operands. The first generation of data-flow machines is represented by the Manchester Dataflow Machine[1, 23], the Sigma-1[1, 12], and the Tagged Token Dataflow Architecture (TTDA)[3, 1]. However, these machines possessed the following drawbacks which hampered their economic viability and performance:

- The cost of the resulting hardware rendered them not economically viable. Indeed, associative matching of tokens to determine instruction executability requires expensive hardware.
- Dynamic scheduling of every instruction causes unnecessary overhead. Even those instructions which can be determined at compile-time to execute sequentially are needlessly scheduled dynamically.

Subsequent research efforts in the architecture arena have concentrated on addressing these points. The first breakthrough occurred in the design of the matching mechanism. The reason for the associative matching in the first data-flow machines was that the tag field of a token which indicates, among other things, the context of the token, has no correspondence to memory location[2]. Thus tokens were stored

randomly inside the waiting token storage pool of the processor and associative search was performed to find the matching token whenever a new token arrives at the processor. A direct matching scheme called the Explicit Token Storage (ETS) which does not use associative matching was first introduced in the Monsoon[6, 20]. This mechanism allocates at compile-time a memory slot to each arc (of a data-flow graph) belonging to the same *code block*, by assigning an offset from the base of a frame memory. A code block is a group of instructions which becomes activated when the frame memory is allocated to it at run-time. In general, a function body or a loop body corresponds to a code block. Therefore, once a code block is activated, tokens "know" exactly where in the frame memory they need to check for synchronization, making associative matching unnecessary. This mechanism resulted in faster matching speed (since it took locality into consideration) while the hardware could be made simpler.

Further improvement for efficient execution can be achieved by grouping instructions to form threads. One of the drawbacks of the previously mentioned data-flow machines is that every instruction is still scheduled dynamically. This generates an unnecessary burden on the machine when scheduling those instructions that are *known* to be executed in sequence. Such instructions can be better scheduled statically for sequential execution, thereby avoiding the cost of synchronization. In multithreading, instructions of a code block are partitioned to form threads. Although a thread may be scheduled dynamically once activated, instructions belonging to the thread are executed sequentially without further synchronization. Instruction-level parallelism can be further exploited by way of pipelining or superscalar techniques. The first multithreaded architectures that evolved from the data-flow architectures are the Dataflow/von Neumann Hybrid Architecture[13] and the P-RISC[18]. The basic difference between the two is that synchronization is implicit in the Hybrid Architecture provided by the hardware supported presence bits while in P-RISC, synchronization is explicit by the compiler generated codes. More recent proposals such as *T[19, 5] and the Threaded Abstract Machine (TAM)[7] render the concept of thread more explicit by compile-time specification of the beginning and the end of a thread. Clear delineation of threads can make thread scheduling more efficient.

The important point to recognize is that the fundamental data-driven approach to parallel computing has not changed, *i.e.*, extract a lot of parallelism at compile-time so that computations can be effectively overlapped with unavoidable communications (thereby incurring long latency costs) occurring in a large parallel system. Multithreading is an outgrowth of that research aimed at developing an efficient execution model for fine/medium grain parallelism. Initially, every aspect of execution functions were given to the hardware which resulted in costly systems. Gradually, the emphasis is being shifted to software (compiler) resulting in the current version of multithreading execution models.

2.2 Execution Model

The multithreading execution model used for this study is adapted from that of *T and TAM. A code block corresponding to a loop body or a function body in the program text is divided at compile-time into threads. An instance of a code block is said to be *active* when a frame memory is allocated to it and the input parameters are sent. The frame memory

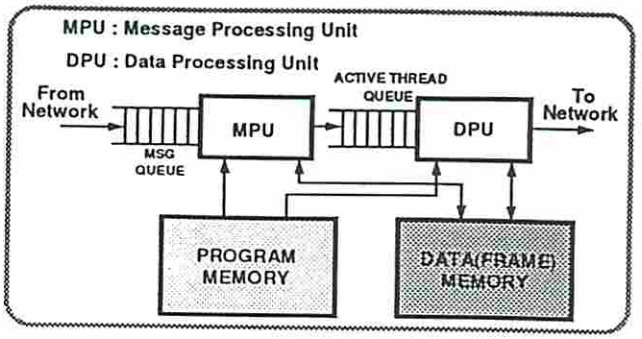

Figure 2: The multithreaded architecture model of a processing element.

request is sent to the *frame manager* residing at each processing node. The amount of memory needed in the frame is determined at compile-time by analyzing the memory requirements of the code block. After a code block is activated, each thread is executed when its activation requirement is satisfied. A thread executes in a non-preemptive mode, *i.e.*, once a thread is activated, its execution continues without suspension until all the instructions of the thread have been executed. The first thread to be executed is the *initialization* thread which receives and stores the input parameters in the appropriate frame locations. It may also initialize synchronization variables used to activate other threads.

In general, an activated code block performs various other functions in addition to the actual computations as specified in the program. The initialization thread as described in the previous paragraph is one. Threads constituting a code block can be classified into the following four classes:

1. Initialization thread: There is always one initialization thread in a code block. It is the first thread to be executed in the code block and its function is to receive input parameters and initialize other local constants and variables.

2. Interface thread: There can be a number of these threads. Their main function is to communicate with the outside world; that is, activations in other processors. For every remote read request, a matching interface thread is required to receive data because of a split-phase operation. An interface thread may update synchronization variables after receiving a token. When the terminal value is reached, the corresponding compute thread is activated.

3. Compute thread: These are the threads that actually perform the computations specified in the program as the programmer sees it. A compute thread may be triggered by the initialization thread, interface threads or other compute threads.

4. Control thread: This thread determines whether the current activation has done all its required work. For instance, if the code block represents a loop body of a parallel loop, the thread determines whether the frame memory should be reused by another loop iteration or be deallocated.

The multithreaded architecture model used for this study (Figure 2) is similar in basic concept to the Decoupled Graph/

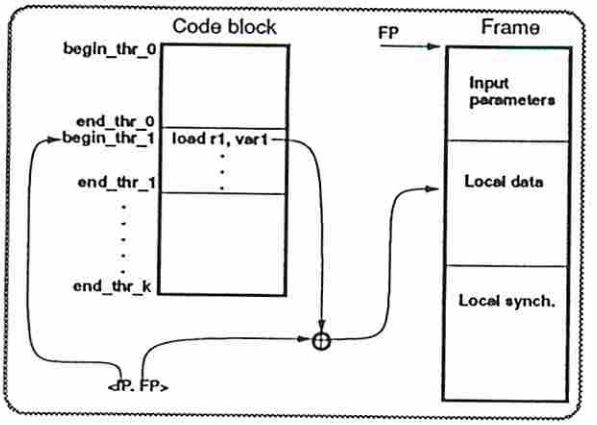

Figure 3: The state of a thread is completely specified by a descriptor consisting of <IP.FP>.

Computation Data-Driven Architecture[8] and *T[19]. The basic concept behind the two models is to decouple the functions of thread activation from that of thread computation. In the Decoupled architecture, the thread executability is determined by the Data-Flow Graph Engine (DFGE) while the activated threads are executed in the Computation Engine (CE). On the other hand, in the *T architecture, the Synchronization Coprocessor (sP) performs thread activation while the Data Processor (dP) executes the activated threads. The difference between the two models is that associative matching is assumed in the Decoupled architecture whereas in *T, the join instruction, first proposed in P-RISC, is used for synchronization.

The multithreaded architecture shown in Figure 2 consists of two processing units. The Message Processing Unit (MPU) is used to execute the interface threads whose main function is to receive tokens and activate compute threads. Once a compute thread is activated, the MPU enqueues the thread descriptor in the Active Thread Queue (ATQ) where it is eventually dequeued by the Data Processing Unit (DPU). The DPU reads a new thread descriptor from the ATQ when the last instruction of the current thread is executed. A thread descriptor consisting of two values, <IP.FP>, completely specifies a thread. IP is an instruction pointer which points at the first instruction of the thread and FP is a frame pointer which points at the base of the frame memory. Once the IP is loaded in the IP register of the DPU, subsequent instructions are executed by incrementing the register value appropriately. Frame locations are accessed using the FP in conjunction with the instruction operands which are offset values. Figure 3 shows how the first instruction of a thread is executed using the thread descriptor.

3 The Direct Access Method

The first subsection describes the Token Relabeling method which inspired the development of our array handling technique called the Direct Access Method. In the second subsection, implementation of this idea in the multithreading context is discussed. Specifically, the direct access mechanism and necessary compile-time analysis is described.

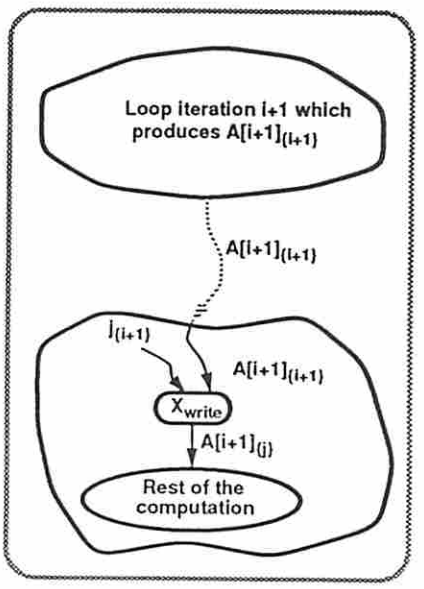

Figure 4: A token carrying an array element needs to be relabeled appropriately in order for it to be forwarded to the correct consumer context.

3.1 Token Relabeling

The Token Relabeling (TR) scheme has been proposed for handling structured data in the tagged token data-flow architecture [9]. Like I-Structures, it is mainly aimed at scientific applications in which arrays are the main structured data types. The central point of this scheme is that, unlike other methods, array elements should not be stored in the structure store facilities. Instead, each array element is treated as a scalar entity and is carried in a token. Individual array elements are uniquely identified by the iteration field of the tag. The basic array accessing mechanism of the TR method is also different from the other array handling methods using a structure store. In the I-Structure scheme, for example, array elements are first sent to the I-Structure controller to be written into the corresponding memory location. Likewise, an array consumer sends a read request to the structure store for consumption. In the Token Relabeling method, each array element is sent directly from the producer to the consumer, bypassing the structure store. Array elements are temporarily stored in the waiting storage pool of the consumer's match unit until they are selected and consumed. Functionally, the Token Relabeling method is identical to the I-Structure approach. Both schemes provide non-strict array accesses and deferred reads. The difference is that the Token Relabeling method does it without special structure storage.

A common scenario for applying the Token Relabeling method is a situation where parallel loops form a producer-consumer relationship such that array(s) produced by the producer loop is input to the consumer loops. The basic token relabeling rule is that an array element $A[i]$ is computed by the producer loop iteration i and is tagged by i, $A[i]_{(i)}$. Likewise, a consumer loop iteration j consumes array element $A[j]_{(j)}$. If $A[i]_{(i)}$ is consumed by the loop iteration $j = i$, no token relabeling is required. However, if $A[i]_{(i)}$ is consumed by the consumer loop iteration $j \neq i$, ar-

ray element $A[i]$ must be appropriately relabeled for correct consumption. For example, assume a consumer iteration j consumes $A[i+1]$, $j \neq i+1$. Then $A[i+1]_{(i+1)}$ produced at iteration $i+1$ must be relabeled to $A[i+1]_{(j)}$ so that the token is correctly consumed by the consumer iteration j. Figure 4 shows how the iteration field of the tag is appropriately relabeled before the array element is consumed.

The advantages of the Token Relabeling method over the I-Structure is that, 1) no special resources are required to provide parallel non-strict array accesses and that 2) it is more efficient. The Token Relabeling method is a more efficient technique because, for every array element transfer from a producer to a consumer, at most only one remote access is required. On the other hand, there are potentially three remote accesses when an I-Structure is used. A study has shown that given the same program, the TR method generates less network traffic than the I-Structure resulting in faster execution time[10]. The Token Relabeling method has been also applied in a scheme which exploits at run-time, parallelism that were hidden at compile-time[15]. However, there are a number of drawbacks to using the Token Relabeling method. One of the drawbacks is that precious match unit storage space is taken by the array elements. Therefore, the TR method is best suited for those situations in which the array in question has the characteristics of a temporary variable. In other words, the TR method may be more appropriate when an array is produced and consumed over a relatively short period of time. For those cases in which an array is accessed over a long period of time, I-Structure representation may be a better choice.

3.2 Direct Access Mechanism in a Multithreaded Execution Model

The implementation of the Token Relabeling scheme in the tagged token data-flow architecture used the following two key features:

- The iteration field is part of the context identifier and its value can be modified at run-time so that a token produced in one context can be made to belong to another context by appropriately changing the iteration tag field.
- Once the context of a token is changed, the token is guaranteed to be forwarded to the processor that executes the new context by a predefined mapping function which uses the tag as its input argument.

Since the multithreaded architecture does not provide the features mentioned above, a run-time system is required if the Token Relabeling scheme is to be implemented. The responsibility of the run-time system is to provide the tagged token abstraction to the programmer. To do this, the run-time system needs to map the context tag field to the context of the multithreaded architecture, the frame pointer FP. Thus, whenever a relabeling occurs, the run-time system must search its mapping table to determine which frame pointer maps to the new context as represented by the modified tag value. This approach, which faithfully reproduces the Token Relabeling mechanism in a multithreaded architecture may, however, result in poor performance due to the overhead of the run-time system.

Our new array handling technique called the Direct Access Method, does not require the type of run-time system

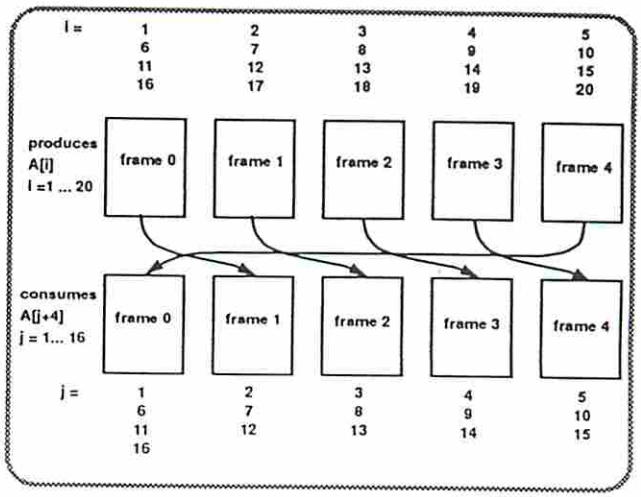

Figure 5: The mapping between the producer loop activation frames and the consumer loop activation frames is static when the array consumption pattern is regular.

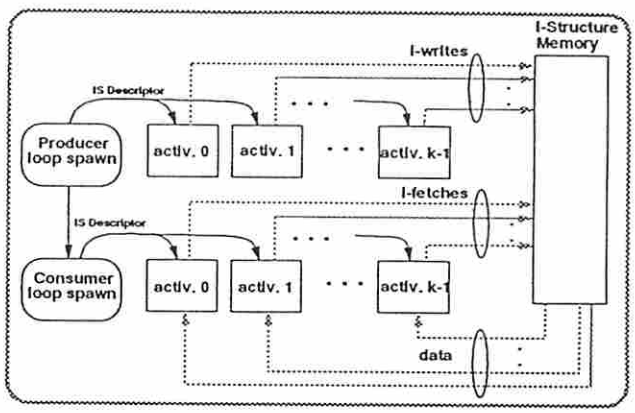

Figure 6: Producer and consumer loop schema using the I-Structure.

mentioned above. This technique, like the Token Relabeling scheme, directly forwards computed array elements from the producer to its consumer. Unlike the Token Relabeling scheme, however, the connection between the producer and the consumer activation is determined at compile-time. This static connection is achieved by determining the consumer's array consumption pattern through array subscript analysis[1]. The consumption pattern in conjunction with the producer-consumer loop bounds and k is used to ascertain which producer and consumer loop activations interact with each other at run-time.

Let us assume that the producer loop ranges from $i = i_{lo} \ldots i_{hi}$ and the consumer loop ranges from $j = j_{lo} \ldots j_{hi}$. In addition, further assume that only k iterations are allowed to be executed in parallel for each loop. Then there are k producer frames, $ProdFrame_l$ and k consumer frames, $ConsFrame_n$, $l, n = 0 \ldots k-1$. Furthermore, assume that the consumer loop consumes the array, A, computed by the producer loop according to the array subscript, $S(j)$. The

[1] It is assumed that Intermediate Form 1 (IF1)[21] data dependency graph produced by the SISAL compiler is used in the analysis.

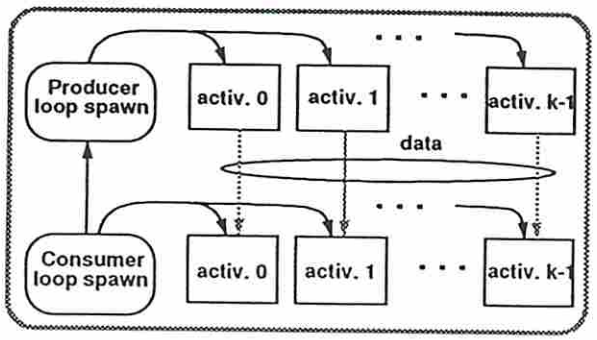

Figure 7: Producer and consumer loop schema using the Direct Access Method.

objective is to connect (at compile-time) the producer iteration $S(j)$ which computes $A[S(j)]$ with the consumer iteration j which consumes it.

Once allocated, a frame is generally reused by multiple iterations without going through the deallocate/allocate process by the frame manager. The reason is to reduce overhead caused by dynamic memory management. Assume, in the context of this discussion that a frame memory used by iteration i is reused by iteration $i + k$. In other words, a frame memory m is reused by iteration i according to the following expression where i_{lo} is the lower loop bound:

$$m = (i - i_{lo}) \ mod \ k$$

An iteration i is said to be connected to iteration j if it has the FP of the frame memory used by iteration j. Therefore, the connection between the producer iteration $i = S(j)$ and the consumer iteration j can be equivalently represented in terms of the corresponding frames $ProdFrame_l$ and $ConsFrame_n$ in which l and n are determined by the following expression:

$$l = (S(j) - i_{lo}) \ mod \ k, \quad n = (j - j_{lo}) \ mod \ k$$

Once k producer frames are connected to k consumer frames according to the array consumption pattern, all iterations are guaranteed to be connected. This is because all iterations map to one of k frames. An example is shown in Figure 5 ($i = 1 \ldots 20$, $j = 1 \ldots 17$, $S(j) = j + 4$, $k = 5$). In the example, $ConsFrame_1$ is mapped to $ProdFrame_0$. This is correct because the consumer iteration 2 consuming $A[6]$ is activated on $ConsFrame_1$ whereas the producer iteration 6 producing $A[6]$ is activated on $ProdFrame_0$.

We also see in the example that the array elements computed by the producer loop are not completely consumed. Assuming that there are no other consumer loops, iterations $i = 1 \ldots 4$ are not consumed. In the Direct Access Method, it is crucial for the compiler to determine the exact number of consumers for each producer loop activation. Otherwise, a deadlock may occur due to the nonterminating loop body.

Figure 6 shows that the I-Structure descriptor is sent as an input parameter to every producer and consumer activations. Each activation uses the descriptor to compute the exact location of the I-Structure memory cell it needs to access. In the Direct Access Method, a frame pointer of the communicating counterpart activation is sent to each activation instead of the I-Structure descriptor. Figure 7 shows that frame pointers of the consumer activations are sent to the producer. Upon receiving the corresponding consumer's frame pointer, a producer activation can transmit computed array elements. After a small initial start-up time, the actual computation performed by the activations is overlapped with the transmission of frame pointers. In the current implementation, a buffer of size k is allocated in the frame of the producer loop spawning code block to store the incoming consumer frame pointers. The function that maps frames to iterations are code generated as part of the loop spawning code block. As each producer activation is created, the corresponding consumer frame pointer is read from the buffer according to the mapping function and sent to the activation along with other input parameters.

4 Performance Measurement

This section reports on the performance of three different implementations of the Livermore Loop 1. Each version is implemented using a different array handling technique and the resulting performance is discussed.

4.1 The Simulated Architecture

The basic configuration of a processing node of the simulated architecture used in the performance measurement is as shown in Figure 2. Each node consists of a Message Processing Unit (MPU) and a Data Processing Unit (DPU) which operate asynchronously to each other. The main function of the MPU is to receive incoming messages and activate threads that are triggered by the incoming data tokens. When a thread is activated, its descriptor is inserted into the Active Thread Queue (ATQ) for eventual execution by the DPU. As thread scheduling is not the main research issue, a simple FIFO policy was adopted. The execution of an instruction is assigned one time unit assuming that one instruction can be issued at each clock cycle. A hypercube topology was selected for the interconnection network.

In the simulator, the I-Structure controller is implemented with the following key implementation parameters:

- Array elements are low-order interleaved across I-Structure (IS) nodes; an array element $A[i]$ is mapped to the IS node, $inode = i \ \ Mod \ \ M$, where M is the total number of the I-Structure nodes in the system. There are as many IS nodes as there are processing nodes.
- The cost of the I-Structure memory allocation and the initialization time is idealized to zero.
- Within the IS node, each memory read and write operation takes one time unit. Every enqueue and dequeue operation to/from the deferred read queue also takes one time unit.
- No extra cost is assigned for accessing an IS node, *i.e.*, no cost is incurred if an activation executing in the processing node k accesses an IS node that is local to the processing node.
- The cost of any remote message including i-write and i-fetch messages is determined by the currently set network latency value and the routed path.
- The processing node in which an array producer activation executes and the IS node in which it writes into are allocated independently. Therefore, i-write or i-fetch operations are not guaranteed to be local.

DAM-MIF

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	2794.0	2687.0	3195.0	4111.0
L = 1.0	2788.0	2711.0	3218.0	4176.0
L = 5.0	3107.0	3217.0	3799.0	4678.0
L = 10.0	5604.0	4790.0	5786.0	6995.0

I-Structure

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	1742.0	1844.0	2121.0	3756.0
L = 1.0	1742.0	1848.0	2168.0	3792.0
L = 5.0	2691.0	3755.0	4751.0	5785.0
L = 10.0	5271.0	7506.0	9330.0	11151.0

DAM-SIF

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	1935.0	2668.0	4603.0	8743.0
L = 1.0	1953.0	2699.0	4624.0	8794.0
L = 5.0	2483.0	3458.0	5043.0	9190.0
L = 10.0	4483.0	6009.0	7091.0	10012.0

Table 1: Execution time of each implementation when the data and the processors are scaled.

4.2 Benchmark and Conditions of Experiments

Three versions of the Livermore Loop 1 are implemented to observe the effects of different array handling techniques on the overall performance of the program. Livermore Loop 1 is a simple parallel loop which produces an array of one dimension. To supply two of its array input parameters, Y and Z, a producer loop is provided. A SISAL version of the program is shown in Figure 8. One of the three versions of the Livermore Loop 1 is implemented using the I-Structure with the assumptions listed in the previous subsection.

The other two implementations are different versions of the Direct Access Method. In the first version, a single iteration is active at a time in a given frame memory and the frame is reused by another activation only after the current activation is terminated. This implementation is called the DAM-Single Iteration Frame (DAM-SIF). In the second version, multiple iterations are active simultaneously in a given frame memory. In this implementation, called the DAM-Multiple Iteration Frame (DAM-MIF), w iterations are simultaneously active using the same frame memory. In this mechanism, memory space to store loop variables for w iterations are provided in a frame. In addition, since different instances of a thread may be active simultaneously in the DAM-MIF, the thread descriptor is provided with an extra field, i, in addition to FP and IP to differentiate between different instances of a thread. Using this new field, different instances of a thread may become activated and executed independently within a given frame memory.

Two different performance measurements were performed on each implementation under various network latency conditions. The first one measures the data scalability inspired by [11]. In the Livermore Loop 1, the amount of parallelism is simply the upper loop bound, n. Ideally, the execution time should stay constant if the problem size and the number of processors were increased simultaneously by the same factor. In reality, the execution time will vary depending

```
define Main

type OneDim = array[real];

function Producer(n:integer returns OneDim,OneDim)
 for k in 1,n
 returns array of real(k) * 0.5
   array of real(k) * 0.8
 end for
end function

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

function Main(n1,n2:integer; Q,R,T:real returns OneDim)
 let
   Answer := Loop1(n2,Q,R,T,Producer(n1))
 in
   Answer
 end let
end function
```

Figure 8: SISAL program of Livermore Loop 1.

on a number of parameters. For example, machine characteristics such as network latency and compiler dependent factors such as the activation spawning mechanism as well as the actual amount of iterations allowed to be active simultaneously all influence the execution time. For the measurements, the data size was varied from 500 to 4000 and the number of processors from 16 to 128. Since the actual amount of parallelism is capped by a loop-bounding mechanism, the maximum number of activations allowed concurrently was made to vary from 96 to 768; on the average, 6 iterations are active concurrently at each processing node. The measurements were repeated for four different communication latencies (L = 0, 1, 5, and 10 time units).

We define the scaling factor SF as the ratio between the execution time of the reference set and the execution time when the processors and the data size are scaled:

$$SF(PE, DATA\ SIZE) = EX_{ref}/EX(PE, DATA\ SIZE)$$

The EX_{ref} is the execution time of the smallest set (the reference set), *i.e.*, $EX(16, 500)$ for each latency condition.

The second is the measurement of speedup. For this measurement, the problem size of the Livermore Loop 1 was fixed at 2000 and the number of processors was varied from 1 to 64. The execution time of each implementation utilizing different numbers of processors was measured for two latency values of 1 and 10. For a fixed number of processors, the execution time can vary greatly depending on how many loop instances are allowed to be active simultaneously. Therefore, for each measurement, the loop bounding value k was varied and the best execution time was selected.

4.3 Observations

Table 1 shows the execution time and Figure 9 shows the data scalability curve of each implementation at different latency conditions. Each point of Figure 9 is computed according to the scaling factor formula. For example, the scaling factor $SF(128, 4000)$ of the DAM-MIF at latency = 10 is computed by $EX(16, 500)/EX(128, 4000) = 0.80$. To avoid cluttering, only the data scalability curve for the DAM-MIF and the I-Structure implementations are shown.

In terms of absolute execution time, the I-Structure implementation performs the best for small latencies (L=0,1).

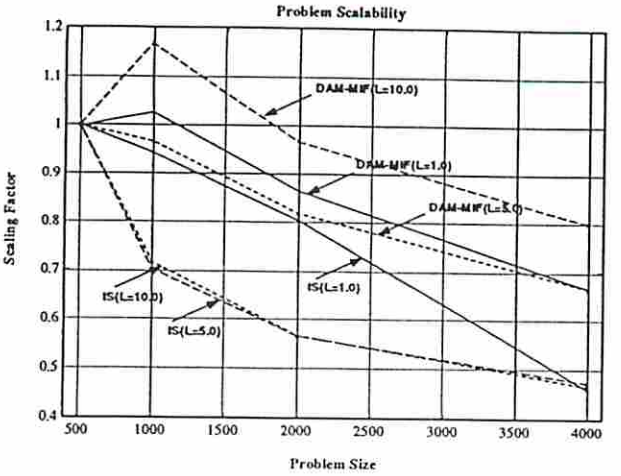

Figure 9: Data scalability graph for DAM-MIF and I-Structures.

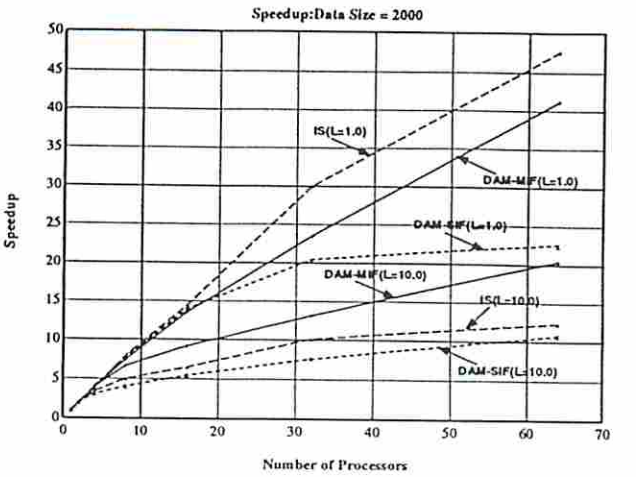

Figure 10: Speedup of three different implementations when latency = 1.0, 10.0.

	DAM-MIF		I-Structure		DAM-SIF	
Number of PEs	L = 1.0	L = 10.0	L = 1.0	L = 10.0	L = 1.0	L = 10.0
1	108083.0	108083.0	104155.0	104155.0	76455.0	76455.0
2	53982.0	53978.0	52109.0	52139.0	38392.0	38483.0
4	27137.0	27265.0	27045.0	30700.0	19384.0	24976.0
8	14599.0	16329.0	13076.0	20908.0	9997.0	19021.0
16	7970.0	11755.0	7188.0	16594.0	5511.0	14000.0
32	4597.0	8230.0	3469.0	10274.0	3758.0	10090.0
64	2623.0	5283.0	2187.0	8434.0	3396.0	7091.0

Table 2: Execution time of each implementation when the data and the processors are scaled.

On the other hand, DAM-MIF produces the best result for longer latencies (L=5,10) and larger processors ($\geq$ 32). In terms of data scalability, DAM-MIF consistently performs the best for all latencies. As can be seen from the graph of Figure 9, the scalability of the DAM-MIF degrades to 0.8 (L=10) and to 0.67 (L=1,5); the scalability of the I-Structure implementation degrades to around 0.47 for all three latencies. For a higher latency value, DAM-MIF performs better in both the execution time and data scalability. However, the DAM-SIF performs poorly against other implementations. Its only best execution time is for two cases when 16 processors (L=5,10) are utilized.

Table 2 lists the execution time of each implementation when the problem size is fixed at 2000 and the processors are increased from 1 to 64. As observed in the data scalability measurement, the DAM-MIF performs the best when a larger number of processors are used under longer latency condition. In the measurement, the DAM-MIF produced the best execution time when the processors $\geq$ 8 and the latency is 10. For a latency of 1, the I-Structure had the fastest execution time when the number of processors were 32 and 64. For a smaller number of processors, the DAM-SIF implementation resulted in the best execution time. Figure 14 shows the speedup curve of each implementation at different latency conditions. At latency = 1, the I-Structure results in the best speedup ($\approx$ 47.5). Because of the inefficient frame usage, the speedup curve of the DAM-SIF implementation quickly saturates after 32 processors. When the latency is increased by 10-fold to 10, the DAM-MIF produces best speedup ($\approx$ 20.5). At this latency condition, the speedup of the I-Structure version is only 25 % of the speedup at latency = 1 while the speedup of the DAM-MIF and the DAM-SIF is only reduced by half.

4.4 Interpretation

The key to achieving good performance is to effectively overlap computation and communication. At the same time, available resources must be utilized efficiently. In both the data scalability and the speedup measurement, the performance of the I-Structure version degraded rapidly for longer latency conditions. The cause of such performance characteristic is that the I-Structure implementation over-utilizes the network. Although split-phased remote memory operations provide environment for efficient processor utilization, network resources may easily be overloaded with remote message traffic becoming a bottleneck. In the simulated program, the producer loop creates two array elements and the consumer loop (Livermore Loop 1) consumes three array elements at every iteration. Assuming the array size is N, the worst case expression for the total number of remote array operations is:

$$\begin{aligned} \textit{Total remote msgs} &= \textit{i-write msgs} + \textit{i-fetch msgs} + \textit{data msgs} \\ &= 2N + 3N + 3N = 8N \end{aligned}$$

In the DAM-SIF implementation, the network resource is better utilized than in the I-Structure implementation because the message traffic is reduced. This is achieved by the direct array accessing mechanism in which the array elements are sent directly from the producer to the consumer activations; write operation is always a fast local memory write operation to a frame memory. The expression for the total number of remote messages is,

$$\begin{aligned} \textit{Total remote msgs} &= \textit{data msgs} + \textit{sync msgs} \\ &= 3N + 3N = 6N \end{aligned}$$

Synchronization message in the above expression is required by the very mechanism of the Direct Access Method

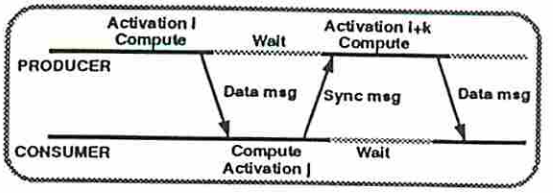

Figure 11: In the DAM-SIF implementation, a frame cannot be reused immediately after an activation until the sync msg is received.

when a frame memory is reused[2]. Its function is to synchronize the producer and the consumer activations before an array element is sent directly from the producer to the consumer activation. Without synchronization, an array element may be sent before the next consumer iteration becomes active. Unfortunately, this mechanism makes effective overlapping of computation and communication difficult, resulting in low processor utilization. More specifically, a frame used by a previous producer activation cannot be reused immediately by a new producer activation until a sync msg is received, indicating that the new consumer instance is active (Figure 11). In the meantime, the frame memory is not utilized by any activation. A relatively poor performance of the DAM-SIF in both performance metrics is the result of low processor utilization caused by this synchronization mechanism.

The DAM-MIF mechanism solves the low processor utilization problem by allowing multiple iterations to be active simultaneously using the same frame memory. At the same time, remote message traffic is further reduced. In this implementation, a thread descriptor is represented by `<IP.FP.i>`. The `i` field is used to distinguish different instances of a same thread. Figure 12 shows that for w iteration instances, the producer activation sends the data to the consumer activation without exchanging synchronization messages. The total number of remote messages for the DAM-MIF is thus:

$$\begin{aligned} Total\ remote\ msgs &= data\ msgs + sync\ msgs \\ &= 3N + 3\lceil N/w \rceil \end{aligned}$$

The DAM-SIF is a special case of the DAM-MIF in which the value of w is one.

Figures 13 and 14 show the processor and the network utilization of the three implementations at different latency values when the same number of instances are allowed to be active simultaneously. Because of the reasons discussed, the I-Structure implementation saturates the network for a latency of 5 and above, although it performs well under low latency conditions. It is difficult to increase processor utilization under longer latency condition because the network is already saturated. In the DAM-SIF implementation, processors are consistently underutilized. In the DAM-MIF implementation, the network utilization is still around 50 % even at latency=10 leaving more room for extra activations. Among the three implementations, the DAM-MIF implementation utilizes the processor and the network resources most efficiently given the same conditions.

[2]Synchronization messages are not required in an ideal case where a parallel loop can always be unraveled. However, in most real situations where memory resources are finite, a memory frame must always be reused.

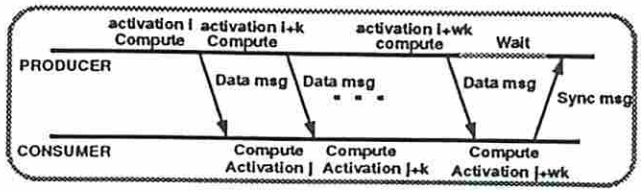

Figure 12: A processor is better utilized in the DAM-MIF implementation because a frame memory is reused immediately after an activation is terminated. At the same time, network load is reduced.

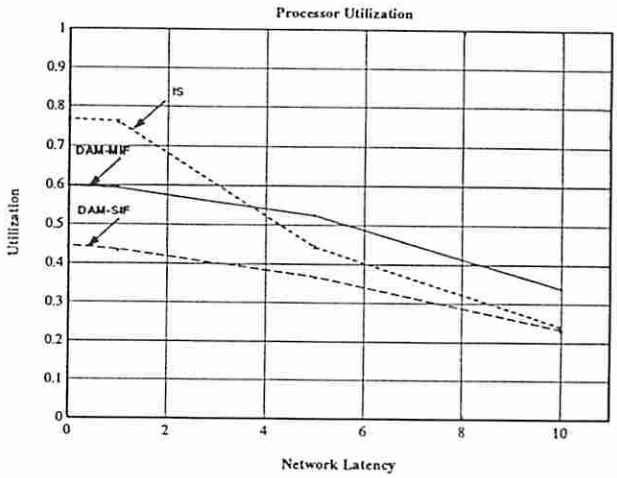

Figure 13: Processor utilization of the three implementations for different latency conditions.

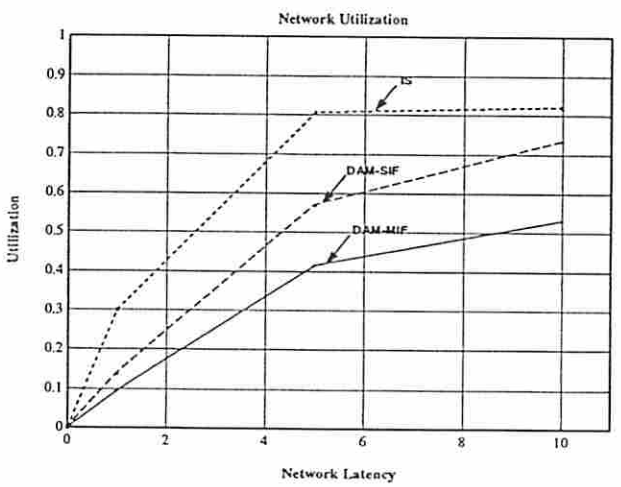

Figure 14: Network utilization of the three implementations for different latency conditions.

5 Conclusions

We have described in this paper a new array handling technique called the Direct Access Method, measured its performance, and compared it with that of the I-Structure representation on a simple parallel loop. It was found that the chief application of this technique would be in those cases where arrays display the characteristics of temporary variables (*i.e.,* the array is consumed "shortly" after being produced). It has been shown that this technique produced better performance under more realistic conditions by efficiently utilizing the system resources. At the same time, a large amount of the I-Structure memory is freed to be used by other portions of the program.

However, the Direct Access Method cannot be used in large programs at the exclusion of other approaches. For those arrays which are accessed over a long period of time after being produced, the I-Structure approach is still the most appropriate array handling method. Therefore, the Direct Access Method can be used selectively in conjunction with the I-Structures in order to enhance performance. A preliminary examination of large programs such as SIMPLE shows that the Direct Access Method can be applied to arrays produced and consumed within a function. However, further research is needed to determine the criteria for using the Direct Access Method. Using this criteria, the data dependence graph of a program can be analyzed at compile-time to determine the most appropriate array handling method.

References

[1] Arvind, L. Bic, and T. Ungerer. Evolution of dataflow computers. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 1, pages 3–33. Prentice Hall, 1991.

[2] Arvind and K. Gostelow. The U-interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[3] Arvind and R. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Parallel Architectures and Languages Europe*. Springer-Verlag, August 1987.

[4] Arvind and R. Thomas. I-structures: An efficient data type for functional languages. Technical Report LCS/TM-178, MIT, Laboratory for Computer Science, June 1980.

[5] 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.

[6] D. Culler and G. Papadopoulos. The explicit token store. In L. Bic and J-L. Gaudiot, editors, *Journal of Parallel and Distributed Comuting, Special Issue:Data-Flow Processing*, pages 289–308. Academic Press, Inc, December 1990.

[7] D. Culler, A. Sah, and et. al. 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.

[8] 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.

[9] J-L. Gaudiot. Structure handling in data-flow systems. *IEEE Transactions on Computer*, C-35(6):489–502, June 1986.

[10] J-L. Gaudiot and Y. Wei. Token relabeling in a tagged-token data-flow architecture. *IEEE Transactions on Computer*, 38(9), 1989.

[11] J. Gustafson. Reevaluating amdahl's law. *Communication of ACM*, 31(5):532–533, May 1988.

[12] 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.

[13] R. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *The 15th Annual International Symposium on Computer Architecture*, 1988.

[14] K. Kawakami and J. Gurd. A scalable dataflow structure store. In *The 13th Annual International Symposium on Computer Architecture*, pages 243–250. ACM and IEEE, ACM Press, June 1986.

[15] 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, 1991.

[16] J. McGraw, S. Skedzielewski, and et.al. *SISAL Language Reference Manual Version 1.2*, March 1985.

[17] R. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT, July 1991.

[18] R. Nikhil and Arvind. Can dataflow subsume von neumann computing? In *The 16th Annual International Symposium on Computer Architecture*, 1989.

[19] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. Technical Report 325-1, MIT, November 1991.

[20] G. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[21] 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.

[22] T. von Eicken, D. Culler, and et. al. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM and IEEE, ACM Press, May 1992.

[23] I. Watson and J. Gurd. A practical data-flow computer. *IEEE Computer*, 15(2):51–57, February 1982.

A Direct Array Handling Technique for Non-strict and Parallel Accesses in a Multithreaded Architecture

Chinhyun Kim and Jean-Luc Gaudiot

CENG Technical Report 93-01

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

A Direct Array Handling Technique for Non-strict and Parallel Accesses in a Multithreaded Architecture*

Chinhyun Kim† *Jean-Luc Gaudiot*†

Abstract

In this paper, we propose an array handling technique called the Direct Access Method (DAM) which enables array elements to be sent *directly* from the producer to the consumer activation while providing non-strict and parallel array accesses. The advantage of this technique over conventional structure storage is that network traffic is reduced and that no global memory space is required. Instead, array elements are produced in the frame memory of the producer activation and forwarded directly to the frame memory of the consumer activation. It is demonstrated here how the technique can be fine-tuned to those cases where an array behaves as a temporary variable and its consumption pattern can be determined at compile-time. Thus, the DAM is proposed as a complement rather than a replacement of the I-Structure representation. Performance measurements obtained by a deterministic simulation of a multithreaded model show that the Direct Access Method indeed performs better than the equivalent I-Structure implementation. The measurements further show that although the I-Structure model has a lower execution time under low network latency conditions, the DAM displays a performance advantage for higher communication network latencies.

1 Introduction

Almost all scientific applications which can benefit from parallel computing entail the manipulation of large data structures. The resulting performance can vary greatly depending on how these large structured data are partitioned and distributed across multiple processing elements. This must be done in such a way as to fully exploit existing parallelism with as little overhead as possible. Since communication accounts for much of this overhead, efficient structure representation and transmission schemes are needed. This paper presents an array handling technique geared mainly for scientific applications in an environment where the computation model is data-driven while the target machine is a multithreaded architecture.

Although every data partition and allocation scheme attempts to render all memory accesses local, remote memory accesses cannot be completely avoided. Frequent remote accesses can have a detrimental effect on the overall performance because of their long latency which can result in low processor utilization. The data-flow computing approach addresses the data partition/allocation problem indirectly by striving to provide a *latency tolerant* computing environment. The adverse effect of long latencies can be indeed minimized by effectively overlapping computation and communication[7]. To this end, three basic requirements must be satisfied:

1. The compiler should be able to expose all the fine/medium grain parallelism contained in the user's program.
2. The architecture of the target machine should be able to exploit such parallelism efficiently.
3. Communication costs should be minimized or at least *tolerable*.

The following is the current data-flow solution to the above requirements:

1. Use a functional language as a programming language which makes it relatively easy for a compiler to extract parallelism.
2. Use a multithreaded architecture to efficiently exploit fine/medium grain parallelism exposed by the compiler.
3. Use I-Structures[4] when arrays can be produced and consumed in parallel.

The concept of I-Structures is an important component of the overall data-flow solution because its objective is to exploit producer-consumer parallelism. This is achieved by providing non-strict and parallel accesses coupled with split-phased memory operations. Specifically, each of the I-Structure memory cell contains presence bits indicating the status of the cell *i.e., full, empty,* or *pending*. Initially, each cell is set to empty. If a read request is made to an empty cell, the state of the cell is changed to pending and the request is queued in the deferred read request list. When the cell is written, the state changes to present. If the cell is marked pending prior to being written to, all deferred read requests are serviced.

*This work is supported in part by the National Science Foundation under grant No. CCR-9013965.

†Electrical Engineering-Systems University of Southern California Los Angeles, California 90089-2562

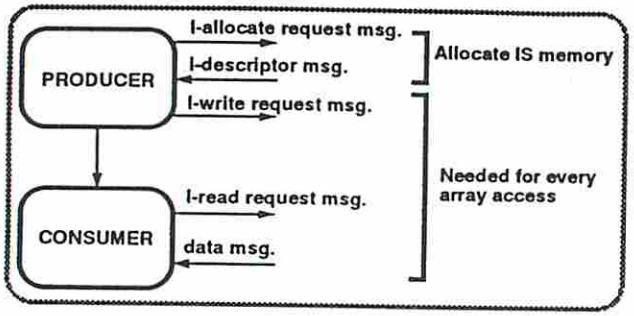

Figure 1: I-Structure and producer-consumer parallelism.

In addition to providing mechanisms to support the status of each memory cell, a memory manager which can efficiently handle dynamically created and deallocated arrays is also required. A complex hardware configuration of the I-Structure memory may be required to provide parallel array accesses as well. For example, the I-Structure memory can be low-order interleaved across the I-Structure nodes[20]. Alternatively, it can be high-order interleaved[14]. However, providing parallel accesses and split-phased read operations can cause excessive network traffic. This means that in the worst case, three remote messages are required for every array element to be written or read.

This overhead associated with the I-Structure representation may render it prohibitively expensive when an array behaves similarly to a temporary variable. This refers to cases where the life-time of an array is short because it is quickly consumed and destroyed after being created. In such situations, the fraction of the total time attributable to the overhead (allocate/deallocate + network traffic) may be relatively large resulting in inefficient operations. This paper thus introduces an alternate array handling mechanism called the *Direct Access Method* which, just like the I-structures, provides non-strict and parallel accesses, minus the overhead. We will show how this method can be used in situations where the arrays behave like temporary variables while the I-Structure representation is used for those situations where the arrays behave more like global variables. We will demonstrate that the Direct Access Method is an efficient array handling scheme in which each array element is sent *directly* from the producer to the consumer without being stored in an intermediate array storage. The mechanism makes the issue of structure memory management moot and reduces the number of remote accesses because write operations are guaranteed to be local memory operations.

In summary, the objective of this paper is to introduce the Direct Access Method for the passing of large chunks of data between producer and consumer processes. It is also intended to demonstrate its performance as well as its most likely context of utilization. Section 2 discusses how the data-flow computing model evolved from pure data-flow to multithreading. Section 3 introduces the Direct Access Method and its predecessor, the Token Relabeling approach. The simulation results which demonstrate the performance and the applicability of the Direct Access Method are presented in section 4. Finally, concluding remarks and directions for future research are offered in section 5.

2 Multithreading

We will now discuss first how the principles of data-flow computing have been evolving from pure data-flow toward multithreading, then we will describe the multithreading execution model. Multithreading is viewed as a practical means to promote data-flow computing rather than as a conceptually different model.

2.1 From Data-flow to Multithreading

One of the first basic tenets of parallel computing must be the ability to expose as much parallelism as possible at compile-time so that parallel (independent) instructions can be distributed across multiple processing elements of a machine and executed concurrently, resulting in higher performance. Traditional approaches to parallel computing using conventional languages such as FORTRAN make it difficult for a compiler to extract large amounts of parallelism automatically. This is due to the nature of the language semantics which are based on the von Neumann computation model: the model is inherently sequential in nature. On the other hand, the data-flow approach to parallel computing consists in starting at the top with languages such as Id[17] and SISAL[16] whose functional semantics make it comparatively easy for a compiler to expose all the parallelism contained in a program.

Being able to exploit large amounts of parallelism is crucial to the data-flow approach because its objective is to tolerate communication latency rather than to reduce it by executing ready instructions whenever long-latency inducing operations are initiated. By effectively overlapping communication with computation, latency caused by remote accesses can be masked[22]. Therefore, it is important to possess excess tasks ready for execution whenever long latency which is unavoidable in parallel computing occurs[17]. First generation data-flow machines were designed to switch context at every instruction, *i.e.*, the length of a thread is one instruction. For this purpose, a complex hardware facility called the *match* unit was developed to synchronize incoming data tokens and dynamically schedule instructions depending on the availability of their input operands. The first generation of data-flow machines is represented by the Manchester Dataflow Machine[1, 23], the Sigma-1[1, 12], and the Tagged Token Dataflow Architecture (TTDA)[3, 1]. However, these machines possessed the following drawbacks which hampered their economic viability and performance:

- The cost of the resulting hardware rendered them not economically viable. Indeed, associative matching of tokens to determine instruction executability requires expensive hardware.
- Dynamic scheduling of every instruction causes unnecessary overhead. Even those instructions which can be determined at compile-time to execute sequentially are needlessly scheduled dynamically.

Subsequent research efforts in the architecture arena have concentrated on addressing these points. The first breakthrough occurred in the design of the matching mechanism. The reason for the associative matching in the first data-flow machines was that the tag field of a token which indicates, among other things, the context of the token, has no correspondence to memory location[2]. Thus tokens were stored

randomly inside the waiting token storage pool of the processor and associative search was performed to find the matching token whenever a new token arrives at the processor. A direct matching scheme called the Explicit Token Storage (ETS) which does not use associative matching was first introduced in the Monsoon[6, 20]. This mechanism allocates at compile-time a memory slot to each arc (of a data-flow graph) belonging to the same *code block*, by assigning an offset from the base of a frame memory. A code block is a group of instructions which becomes activated when the frame memory is allocated to it at run-time. In general, a function body or a loop body corresponds to a code block. Therefore, once a code block is activated, tokens "know" exactly where in the frame memory they need to check for synchronization, making associative matching unnecessary. This mechanism resulted in faster matching speed (since it took locality into consideration) while the hardware could be made simpler.

Further improvement for efficient execution can be achieved by grouping instructions to form threads. One of the drawbacks of the previously mentioned data-flow machines is that every instruction is still scheduled dynamically. This generates an unnecessary burden on the machine when scheduling those instructions that are *known* to be executed in sequence. Such instructions can be better scheduled statically for sequential execution, thereby avoiding the cost of synchronization. In multithreading, instructions of a code block are partitioned to form threads. Although a thread may be scheduled dynamically once activated, instructions belonging to the thread are executed sequentially without further synchronization. Instruction-level parallelism can be further exploited by way of pipelining or superscalar techniques. The first multithreaded architectures that evolved from the data-flow architectures are the Dataflow/von Neumann Hybrid Architecture[13] and the P-RISC[18]. The basic difference between the two is that synchronization is implicit in the Hybrid Architecture provided by the hardware supported presence bits while in P-RISC, synchronization is explicit by the compiler generated codes. More recent proposals such as *T[19, 5] and the Threaded Abstract Machine (TAM)[7] render the concept of thread more explicit by compile-time specification of the beginning and the end of a thread. Clear delineation of threads can make thread scheduling more efficient.

The important point to recognize is that the fundamental data-driven approach to parallel computing has not changed, *i.e.*, extract a lot of parallelism at compile-time so that computations can be effectively overlapped with unavoidable communications (thereby incurring long latency costs) occurring in a large parallel system. Multithreading is an outgrowth of that research aimed at developing an efficient execution model for fine/medium grain parallelism. Initially, every aspect of execution functions were given to the hardware which resulted in costly systems. Gradually, the emphasis is being shifted to software (compiler) resulting in the current version of multithreading execution models.

2.2 Execution Model

The multithreading execution model used for this study is adapted from that of *T and TAM. A code block corresponding to a loop body or a function body in the program text is divided at compile-time into threads. An instance of a code block is said to be *active* when a frame memory is allocated to it and the input parameters are sent. The frame memory

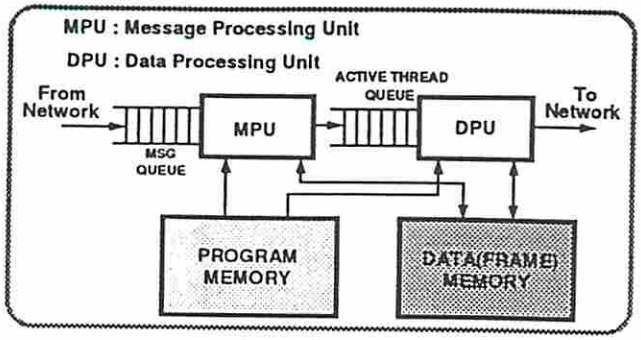

Figure 2: The multithreaded architecture model of a processing element.

request is sent to the *frame manager* residing at each processing node. The amount of memory needed in the frame is determined at compile-time by analyzing the memory requirements of the code block. After a code block is activated, each thread is executed when its activation requirement is satisfied. A thread executes in a non-preemptive mode, *i.e.*, once a thread is activated, its execution continues without suspension until all the instructions of the thread have been executed. The first thread to be executed is the *initialization* thread which receives and stores the input parameters in the appropriate frame locations. It may also initialize synchronization variables used to activate other threads.

In general, an activated code block performs various other functions in addition to the actual computations as specified in the program. The initialization thread as described in the previous paragraph is one. Threads constituting a code block can be classified into the following four classes:

1. Initialization thread: There is always one initialization thread in a code block. It is the first thread to be executed in the code block and its function is to receive input parameters and initialize other local constants and variables.

2. Interface thread: There can be a number of these threads. Their main function is to communicate with the outside world; that is, activations in other processors. For every remote read request, a matching interface thread is required to receive data because of a split-phase operation. An interface thread may update synchronization variables after receiving a token. When the terminal value is reached, the corresponding compute thread is activated.

3. Compute thread: These are the threads that actually perform the computations specified in the program as the programmer sees it. A compute thread may be triggered by the initialization thread, interface threads or other compute threads.

4. Control thread: This thread determines whether the current activation has done all its required work. For instance, if the code block represents a loop body of a parallel loop, the thread determines whether the frame memory should be reused by another loop iteration or be deallocated.

The multithreaded architecture model used for this study (Figure 2) is similar in basic concept to the Decoupled Graph/

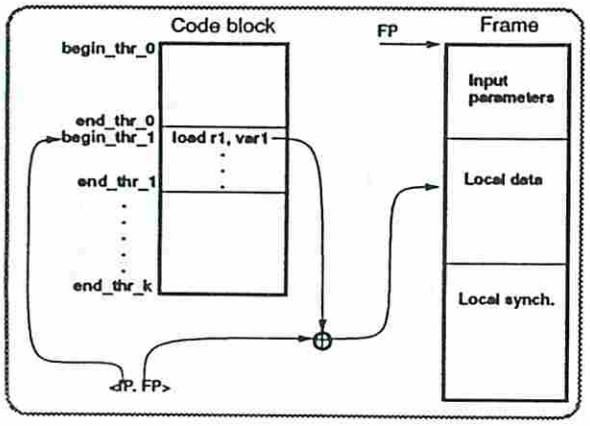

Figure 3: The state of a thread is completely specified by a descriptor consisting of <IP.FP>.

Computation Data-Driven Architecture[8] and *T[19]. The basic concept behind the two models is to decouple the functions of thread activation from that of thread computation. In the Decoupled architecture, the thread executability is determined by the Data-Flow Graph Engine (DFGE) while the activated threads are executed in the Computation Engine (CE). On the other hand, in the *T architecture, the Synchronization Coprocessor (sP) performs thread activation while the Data Processor (dP) executes the activated threads. The difference between the two models is that associative matching is assumed in the Decoupled architecture whereas in *T, the `join` instruction, first proposed in P-RISC, is used for synchronization.

The multithreaded architecture shown in Figure 2 consists of two processing units. The Message Processing Unit (MPU) is used to execute the interface threads whose main function is to receive tokens and activate compute threads. Once a compute thread is activated, the MPU enqueues the thread descriptor in the Active Thread Queue (ATQ) where it is eventually dequeued by the Data Processing Unit (DPU). The DPU reads a new thread descriptor from the ATQ when the last instruction of the current thread is executed. A thread descriptor consisting of two values, <IP.FP>, completely specifies a thread. IP is an instruction pointer which points at the first instruction of the thread and FP is a frame pointer which points at the base of the frame memory. Once the IP is loaded in the IP register of the DPU, subsequent instructions are executed by incrementing the register value appropriately. Frame locations are accessed using the FP in conjunction with the instruction operands which are offset values. Figure 3 shows how the first instruction of a thread is executed using the thread descriptor.

3 The Direct Access Method

The first subsection describes the Token Relabeling method which inspired the development of our array handling technique called the Direct Access Method. In the second subsection, implementation of this idea in the multithreading context is discussed. Specifically, the direct access mechanism and necessary compile-time analysis is described.

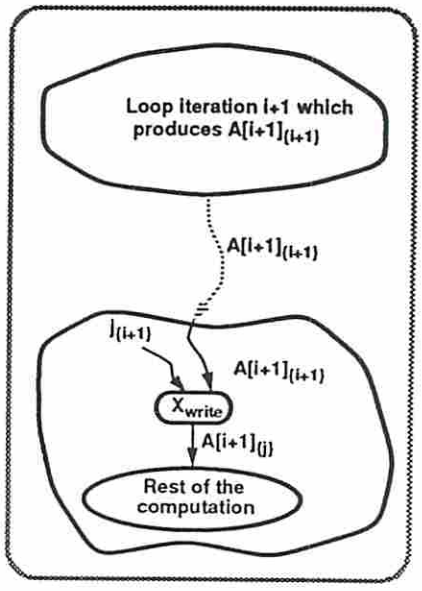

Figure 4: A token carrying an array element needs to be relabeled appropriately in order for it to be forwarded to the correct consumer context.

3.1 Token Relabeling

The Token Relabeling (TR) scheme has been proposed for handling structured data in the tagged token data-flow architecture [9]. Like I-Structures, it is mainly aimed at scientific applications in which arrays are the main structured data types. The central point of this scheme is that, unlike other methods, array elements should not be stored in the structure store facilities. Instead, each array element is treated as a scalar entity and is carried in a token. Individual array elements are uniquely identified by the iteration field of the tag. The basic array accessing mechanism of the TR method is also different from the other array handling methods using a structure store. In the I-Structure scheme, for example, array elements are first sent to the I-Structure controller to be written into the corresponding memory location. Likewise, an array consumer sends a read request to the structure store for consumption. In the Token Relabeling method, each array element is sent directly from the producer to the consumer, bypassing the structure store. Array elements are temporarily stored in the waiting storage pool of the consumer's match unit until they are selected and consumed. Functionally, the Token Relabeling method is identical to the I-Structure approach. Both schemes provide non-strict array accesses and deferred reads. The difference is that the Token Relabeling method does it without special structure storage.

A common scenario for applying the Token Relabeling method is a situation where parallel loops form a producer-consumer relationship such that array(s) produced by the producer loop is input to the consumer loops. The basic token relabeling rule is that an array element $A[i]$ is computed by the producer loop iteration i and is tagged by i, $A[i]_{(i)}$. Likewise, a consumer loop iteration j consumes array element $A[j]_{(j)}$. If $A[i]_{(i)}$ is consumed by the loop iteration $j = i$, no token relabeling is required. However, if $A[i]_{(i)}$ is consumed by the consumer loop iteration $j \neq i$, ar-

ray element $A[i]$ must be appropriately relabeled for correct consumption. For example, assume a consumer iteration j consumes $A[i+1]$, $j \neq i+1$. Then $A[i+1]_{(i+1)}$ produced at iteration $i+1$ must be relabeled to $A[i+1]_{(j)}$ so that the token is correctly consumed by the consumer iteration j. Figure 4 shows how the iteration field of the tag is appropriately relabeled before the array element is consumed.

The advantages of the Token Relabeling method over the I-Structure is that, 1) no special resources are required to provide parallel non-strict array accesses and that 2) it is more efficient. The Token Relabeling method is a more efficient technique because, for every array element transfer from a producer to a consumer, at most only one remote access is required. On the other hand, there are potentially three remote accesses when an I-Structure is used. A study has shown that given the same program, the TR method generates less network traffic than the I-Structure resulting in faster execution time[10]. The Token Relabeling method has been also applied in a scheme which exploits at run-time, parallelism that were hidden at compile-time[15]. However, there are a number of drawbacks to using the Token Relabeling method. One of the drawbacks is that precious match unit storage space is taken by the array elements. Therefore, the TR method is best suited for those situations in which the array in question has the characteristics of a temporary variable. In other words, the TR method may be more appropriate when an array is produced and consumed over a relatively short period of time. For those cases in which an array is accessed over a long period of time, I-Structure representation may be a better choice.

3.2 Direct Access Mechanism in a Multithreaded Execution Model

The implementation of the Token Relabeling scheme in the tagged token data-flow architecture used the following two key features:

- The iteration field is part of the context identifier and its value can be modified at run-time so that a token produced in one context can be made to belong to another context by appropriately changing the iteration tag field.
- Once the context of a token is changed, the token is guaranteed to be forwarded to the processor that executes the new context by a predefined mapping function which uses the tag as its input argument.

Since the multithreaded architecture does not provide the features mentioned above, a run-time system is required if the Token Relabeling scheme is to be implemented. The responsibility of the run-time system is to provide the tagged token abstraction to the programmer. To do this, the run-time system needs to map the context tag field to the context of the multithreaded architecture, the frame pointer FP. Thus, whenever a relabeling occurs, the run-time system must search its mapping table to determine which frame pointer maps to the new context as represented by the modified tag value. This approach, which faithfully reproduces the Token Relabeling mechanism in a multithreaded architecture may, however, result in poor performance due to the overhead of the run-time system.

Our new array handling technique called the Direct Access Method, does not require the type of run-time system

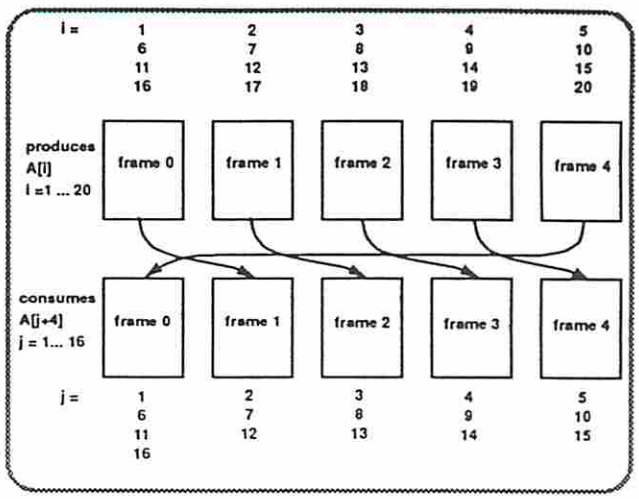

Figure 5: The mapping between the producer loop activation frames and the consumer loop activation frames is static when the array consumption pattern is regular.

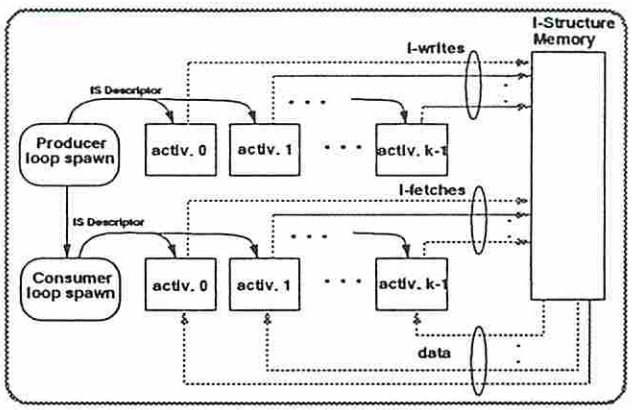

Figure 6: Producer and consumer loop schema using the I-Structure.

mentioned above. This technique, like the Token Relabeling scheme, directly forwards computed array elements from the producer to its consumer. Unlike the Token Relabeling scheme, however, the connection between the producer and the consumer activation is determined at compile-time. This static connection is achieved by determining the consumer's array consumption pattern through array subscript analysis[1]. The consumption pattern in conjunction with the producer-consumer loop bounds and k is used to ascertain which producer and consumer loop activations interact with each other at run-time.

Let us assume that the producer loop ranges from $i = i_{lo} \ldots i_{hi}$ and the consumer loop ranges from $j = j_{lo} \ldots j_{hi}$. In addition, further assume that only k iterations are allowed to be executed in parallel for each loop. Then there are k producer frames, $ProdFrame_l$ and k consumer frames, $ConsFrame_n$, $l, n = 0 \ldots k-1$. Furthermore, assume that the consumer loop consumes the array, A, computed by the producer loop according to the array subscript, $S(j)$. The

[1] It is assumed that Intermediate Form 1 (IF1)[21] data dependency graph produced by the SISAL compiler is used in the analysis.

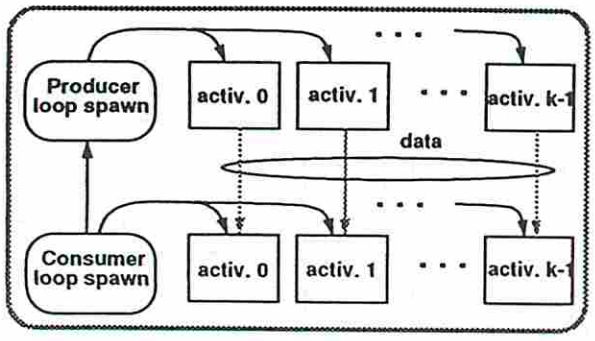

Figure 7: Producer and consumer loop schema using the Direct Access Method.

objective is to connect (at compile-time) the producer iteration $S(j)$ which computes $A[S(j)]$ with the consumer iteration j which consumes it.

Once allocated, a frame is generally reused by multiple iterations without going through the deallocate/allocate process by the frame manager. The reason is to reduce overhead caused by dynamic memory management. Assume, in the context of this discussion that a frame memory used by iteration i is reused by iteration $i + k$. In other words, a frame memory m is reused by iteration i according to the following expression where i_{lo} is the lower loop bound:

$$m = (i - i_{lo}) \ mod \ k$$

An iteration i is said to be connected to iteration j if it has the FP of the frame memory used by iteration j. Therefore, the connection between the producer iteration $i = S(j)$ and the consumer iteration j can be equivalently represented in terms of the corresponding frames $ProdFrame_l$ and $ConsFrame_n$ in which l and n are determined by the following expression:

$$l = (S(j) - i_{lo}) \ mod \ k, \quad n = (j - j_{lo}) \ mod \ k$$

Once k producer frames are connected to k consumer frames according to the array consumption pattern, all iterations are guaranteed to be connected. This is because all iterations map to one of k frames. An example is shown in Figure 5 ($i = 1 \ldots 20$, $j = 1 \ldots 17$, $S(j) = j + 4$, $k = 5$). In the example, $ConsFrame_1$ is mapped to $ProdFrame_0$. This is correct because the consumer iteration 2 consuming $A[6]$ is activated on $ConsFrame_1$ whereas the producer iteration 6 producing $A[6]$ is activated on $ProdFrame_0$.

We also see in the example that the array elements computed by the producer loop are not completely consumed. Assuming that there are no other consumer loops, iterations $i = 1 \ldots 4$ are not consumed. In the Direct Access Method, it is crucial for the compiler to determine the exact number of consumers for each producer loop activation. Otherwise, a deadlock may occur due to the nonterminating loop body.

Figure 6 shows that the I-Structure descriptor is sent as an input parameter to every producer and consumer activations. Each activation uses the descriptor to compute the exact location of the I-Structure memory cell it needs to access. In the Direct Access Method, a frame pointer of the communicating counterpart activation is sent to each activation instead of the I-Structure descriptor. Figure 7 shows that frame pointers of the consumer activations are sent to the producer. Upon receiving the corresponding consumer's frame pointer, a producer activation can transmit computed array elements. After a small initial start-up time, the actual computation performed by the activations is overlapped with the transmission of frame pointers. In the current implementation, a buffer of size k is allocated in the frame of the producer loop spawning code block to store the incoming consumer frame pointers. The function that maps frames to iterations are code generated as part of the loop spawning code block. As each producer activation is created, the corresponding consumer frame pointer is read from the buffer according to the mapping function and sent to the activation along with other input parameters.

4 Performance Measurement

This section reports on the performance of three different implementations of the Livermore Loop 1. Each version is implemented using a different array handling technique and the resulting performance is discussed.

4.1 The Simulated Architecture

The basic configuration of a processing node of the simulated architecture used in the performance measurement is as shown in Figure 2. Each node consists of a Message Processing Unit (MPU) and a Data Processing Unit (DPU) which operate asynchronously to each other. The main function of the MPU is to receive incoming messages and activate threads that are triggered by the incoming data tokens. When a thread is activated, its descriptor is inserted into the Active Thread Queue (ATQ) for eventual execution by the DPU. As thread scheduling is not the main research issue, a simple FIFO policy was adopted. The execution of an instruction is assigned one time unit assuming that one instruction can be issued at each clock cycle. A hypercube topology was selected for the interconnection network.

In the simulator, the I-Structure controller is implemented with the following key implementation parameters:

- Array elements are low-order interleaved across I-Structure (IS) nodes; an array element $A[i]$ is mapped to the IS node, $inode = i \ Mod \ M$, where M is the total number of the I-Structure nodes in the system. There are as many IS nodes as there are processing nodes.
- The cost of the I-Structure memory allocation and the initialization time is idealized to zero.
- Within the IS node, each memory read and write operation takes one time unit. Every enqueue and dequeue operation to/from the deferred read queue also takes one time unit.
- No extra cost is assigned for accessing an IS node, *i.e.*, no cost is incurred if an activation executing in the processing node k accesses an IS node that is local to the processing node.
- The cost of any remote message including i-write and i-fetch messages is determined by the currently set network latency value and the routed path.
- The processing node in which an array producer activation executes and the IS node in which it writes into are allocated independently. Therefore, i-write or i-fetch operations are not guaranteed to be local.

DAM-MIF

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	2794.0	2687.0	3195.0	4111.0
L = 1.0	2788.0	2711.0	3218.0	4176.0
L = 5.0	3107.0	3217.0	3799.0	4678.0
L = 10.0	5604.0	4790.0	5786.0	6995.0

I-Structure

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	1742.0	1844.0	2121.0	3756.0
L = 1.0	1742.0	1848.0	2168.0	3792.0
L = 5.0	2691.0	3755.0	4751.0	5785.0
L = 10.0	5271.0	7506.0	9330.0	11151.0

DAM-SIF

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	1935.0	2668.0	4603.0	8743.0
L = 1.0	1953.0	2699.0	4624.0	8794.0
L = 5.0	2483.0	3458.0	5043.0	9190.0
L = 10.0	4483.0	6009.0	7091.0	10012.0

Table 1: Execution time of each implementation when the data and the processors are scaled.

4.2 Benchmark and Conditions of Experiments

Three versions of the Livermore Loop 1 are implemented to observe the effects of different array handling techniques on the overall performance of the program. Livermore Loop 1 is a simple parallel loop which produces an array of one dimension. To supply two of its array input parameters, Y and Z, a producer loop is provided. A SISAL version of the program is shown in Figure 8. One of the three versions of the Livermore Loop 1 is implemented using the I-Structure with the assumptions listed in the previous subsection.

The other two implementations are different versions of the Direct Access Method. In the first version, a single iteration is active at a time in a given frame memory and the frame is reused by another activation only after the current activation is terminated. This implementation is called the DAM-Single Iteration Frame (DAM-SIF). In the second version, multiple iterations are active simultaneously in a given frame memory. In this implementation, called the DAM-Multiple Iteration Frame (DAM-MIF), w iterations are simultaneously active using the same frame memory. In this mechanism, memory space to store loop variables for w iterations are provided in a frame. In addition, since different instances of a thread may be active simultaneously in the DAM-MIF, the thread descriptor is provided with an extra field, `i`, in addition to FP and IP to differentiate between different instances of a thread. Using this new field, different instances of a thread may become activated and executed independently within a given frame memory.

Two different performance measurements were performed on each implementation under various network latency conditions. The first one measures the data scalability inspired by [11]. In the Livermore Loop 1, the amount of parallelism is simply the upper loop bound, n. Ideally, the execution time should stay constant if the problem size and the number of processors were increased simultaneously by the same factor. In reality, the execution time will vary depending

```
define Main

type OneDim = array[real];

function Producer(n:integer returns OneDim,OneDim)
 for k in 1,n
 returns array of real(k) * 0.5
   array of real(k) * 0.8
 end for
end function

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

function Main(n1,n2:integer; Q,R,T:real returns OneDim)
 let
   Answer := Loop1(n2,Q,R,T,Producer(n1))
 in
   Answer
 end let
end function
```

Figure 8: SISAL program of Livermore Loop 1.

on a number of parameters. For example, machine characteristics such as network latency and compiler dependent factors such as the activation spawning mechanism as well as the actual amount of iterations allowed to be active simultaneously all influence the execution time. For the measurements, the data size was varied from 500 to 4000 and the number of processors from 16 to 128. Since the actual amount of parallelism is capped by a loop-bounding mechanism, the maximum number of activations allowed concurrently was made to vary from 96 to 768; on the average, 6 iterations are active concurrently at each processing node. The measurements were repeated for four different communication latencies (L = 0, 1, 5, and 10 time units).

We define the scaling factor SF as the ratio between the execution time of the reference set and the execution time when the processors and the data size are scaled:

$$SF(PE, DATA\ SIZE) = EX_{ref}/EX(PE, DATA\ SIZE)$$

The EX_{ref} is the execution time of the smallest set (the reference set), *i.e.*, $EX(16, 500)$ for each latency condition.

The second is the measurement of speedup. For this measurement, the problem size of the Livermore Loop 1 was fixed at 2000 and the number of processors was varied from 1 to 64. The execution time of each implementation utilizing different numbers of processors was measured for two latency values of 1 and 10. For a fixed number of processors, the execution time can vary greatly depending on how many loop instances are allowed to be active simultaneously. Therefore, for each measurement, the loop bounding value k was varied and the best execution time was selected.

4.3 Observations

Table 1 shows the execution time and Figure 9 shows the data scalability curve of each implementation at different latency conditions. Each point of Figure 9 is computed according to the scaling factor formula. For example, the scaling factor $SF(128, 4000)$ of the DAM-MIF at latency = 10 is computed by $EX(16, 500)/EX(128, 4000) = 0.80$. To avoid cluttering, only the data scalability curve for the DAM-MIF and the I-Structure implementations are shown.

In terms of absolute execution time, the I-Structure implementation performs the best for small latencies (L=0,1).

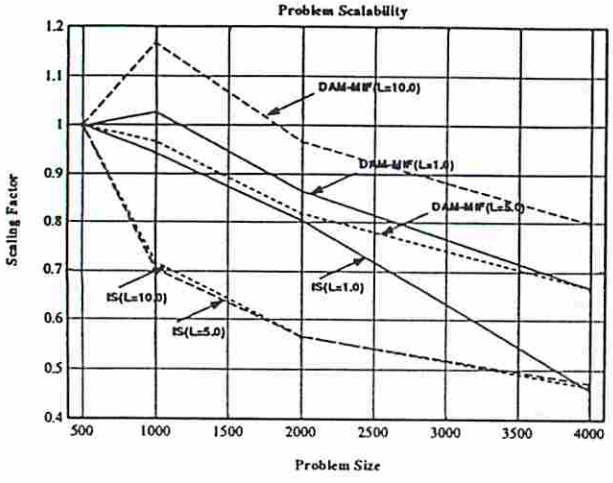

Figure 9: Data scalability graph for DAM-MIF and I-Structures.

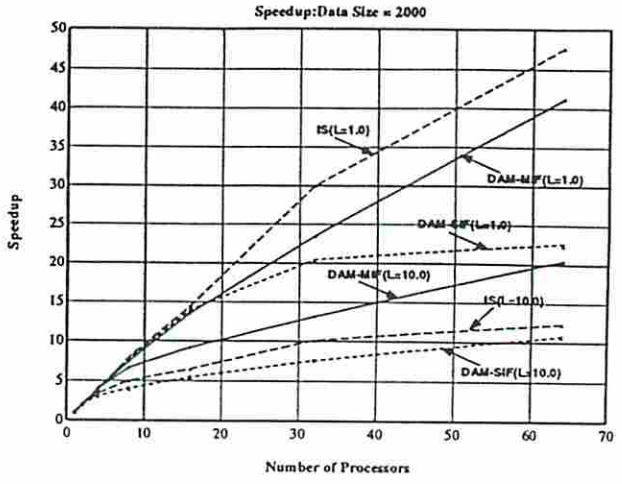

Figure 10: Speedup of three different implementations when latency = 1.0, 10.0.

	DAM-MIF		I-Structure		DAM-SIF	
Number of PEs	L = 1.0	L = 10.0	L = 1.0	L = 10.0	L = 1.0	L = 10.0
1	108083.0	108083.0	104155.0	104155.0	76455.0	76455.0
2	53982.0	53978.0	52109.0	52139.0	38392.0	38483.0
4	27137.0	27265.0	27045.0	30700.0	19384.0	24976.0
8	14599.0	16329.0	13076.0	20908.0	9997.0	19021.0
16	7970.0	11755.0	7188.0	16594.0	5511.0	14000.0
32	4597.0	8230.0	3469.0	10274.0	3758.0	10090.0
64	2623.0	5283.0	2187.0	8434.0	3396.0	7091.0

Table 2: Execution time of each implementation when the data and the processors are scaled.

On the other hand, DAM-MIF produces the best result for longer latencies (L=5,10) and larger processors ($\geq$ 32). In terms of data scalability, DAM-MIF consistently performs the best for all latencies. As can be seen from the graph of Figure 9, the scalability of the DAM-MIF degrades to 0.8 (L=10) and to 0.67 (L=1,5); the scalability of the I-Structure implementation degrades to around 0.47 for all three latencies. For a higher latency value, DAM-MIF performs better in both the execution time and data scalability. However, the DAM-SIF performs poorly against other implementations. Its only best execution time is for two cases when 16 processors (L=5,10) are utilized.

Table 2 lists the execution time of each implementation when the problem size is fixed at 2000 and the processors are increased from 1 to 64. As observed in the data scalability measurement, the DAM-MIF performs the best when a larger number of processors are used under longer latency condition. In the measurement, the DAM-MIF produced the best execution time when the processors $\geq$ 8 and the latency is 10. For a latency of 1, the I-Structure had the fastest execution time when the number of processors were 32 and 64. For a smaller number of processors, the DAM-SIF implementation resulted in the best execution time. Figure 14 shows the speedup curve of each implementation at different latency conditions. At latency = 1, the I-Structure results in the best speedup ($\approx$ 47.5). Because of the inefficient frame usage, the speedup curve of the DAM-SIF implementation quickly saturates after 32 processors. When the latency is increased by 10-fold to 10, the DAM-MIF produces best speedup ($\approx$ 20.5). At this latency condition, the speedup of the I-Structure version is only 25 % of the speedup at latency = 1 while the speedup of the DAM-MIF and the DAM-SIF is only reduced by half.

4.4 Interpretation

The key to achieving good performance is to effectively overlap computation and communication. At the same time, available resources must be utilized efficiently. In both the data scalability and the speedup measurement, the performance of the I-Structure version degraded rapidly for longer latency conditions. The cause of such performance characteristic is that the I-Structure implementation over-utilizes the network. Although split-phased remote memory operations provide environment for efficient processor utilization, network resources may easily be overloaded with remote message traffic becoming a bottleneck. In the simulated program, the producer loop creates two array elements and the consumer loop (Livermore Loop 1) consumes three array elements at every iteration. Assuming the array size is N, the worst case expression for the total number of remote array operations is:

$$\begin{aligned} \textit{Total remote msgs} &= \textit{i-write msgs} + \textit{i-fetch msgs} + \textit{data msgs} \\ &= 2N + 3N + 3N = 8N \end{aligned}$$

In the DAM-SIF implementation, the network resource is better utilized than in the I-Structure implementation because the message traffic is reduced. This is achieved by the direct array accessing mechanism in which the array elements are sent directly from the producer to the consumer activations; write operation is always a fast local memory write operation to a frame memory. The expression for the total number of remote messages is,

$$\begin{aligned} \textit{Total remote msgs} &= \textit{data msgs} + \textit{sync msgs} \\ &= 3N + 3N = 6N \end{aligned}$$

Synchronization message in the above expression is required by the very mechanism of the Direct Access Method

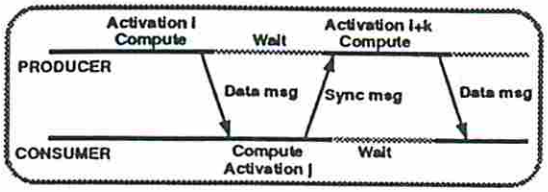

Figure 11: In the DAM-SIF implementation, a frame cannot be reused immediately after an activation until the sync msg is received.

when a frame memory is reused[2]. Its function is to synchronize the producer and the consumer activations before an array element is sent directly from the producer to the consumer activation. Without synchronization, an array element may be sent before the next consumer iteration becomes active. Unfortunately, this mechanism makes effective overlapping of computation and communication difficult, resulting in low processor utilization. More specifically, a frame used by a previous producer activation cannot be reused immediately by a new producer activation until a sync msg is received, indicating that the new consumer instance is active (Figure 11). In the meantime, the frame memory is not utilized by any activation. A relatively poor performance of the DAM-SIF in both performance metrics is the result of low processor utilization caused by this synchronization mechanism.

The DAM-MIF mechanism solves the low processor utilization problem by allowing multiple iterations to be active simultaneously using the same frame memory. At the same time, remote message traffic is further reduced. In this implementation, a thread descriptor is represented by <IP.FP.i>. The i field is used to distinguish different instances of a same thread. Figure 12 shows that for w iteration instances, the producer activation sends the data to the consumer activation without exchanging synchronization messages. The total number of remote messages for the DAM-MIF is thus:

$$\begin{aligned} Total\ remote\ msgs &= data\ msgs + sync\ msgs \\ &= 3N + 3\lceil N/w \rceil \end{aligned}$$

The DAM-SIF is a special case of the DAM-MIF in which the value of w is one.

Figures 13 and 14 show the processor and the network utilization of the three implementations at different latency values when the same number of instances are allowed to be active simultaneously. Because of the reasons discussed, the I-Structure implementation saturates the network for a latency of 5 and above, although it performs well under low latency conditions. It is difficult to increase processor utilization under longer latency condition because the network is already saturated. In the DAM-SIF implementation, processors are consistently underutilized. In the DAM-MIF implementation, the network utilization is still around 50 % even at latency=10 leaving more room for extra activations. Among the three implementations, the DAM-MIF implementation utilizes the processor and the network resources most efficiently given the same conditions.

[2]Synchronization messages are not required in an ideal case where a parallel loop can always be unraveled. However, in most real situations where memory resources are finite, a memory frame must always be reused.

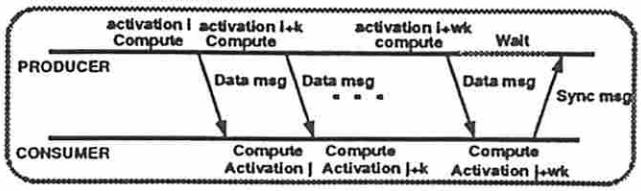

Figure 12: A processor is better utilized in the DAM-MIF implementation because a frame memory is reused immediately after an activation is terminated. At the same time, network load is reduced.

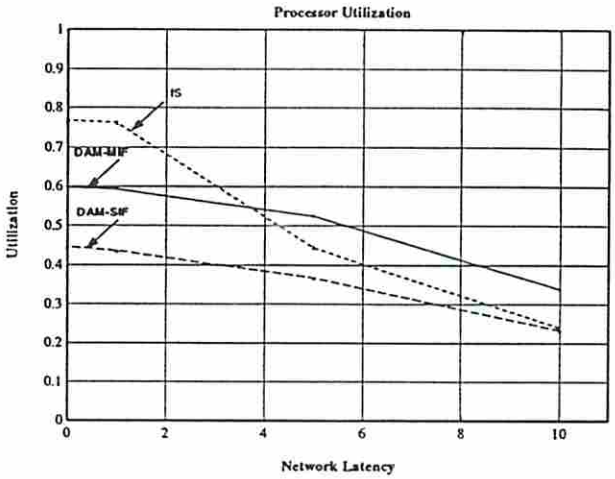

Figure 13: Processor utilization of the three implementations for different latency conditions.

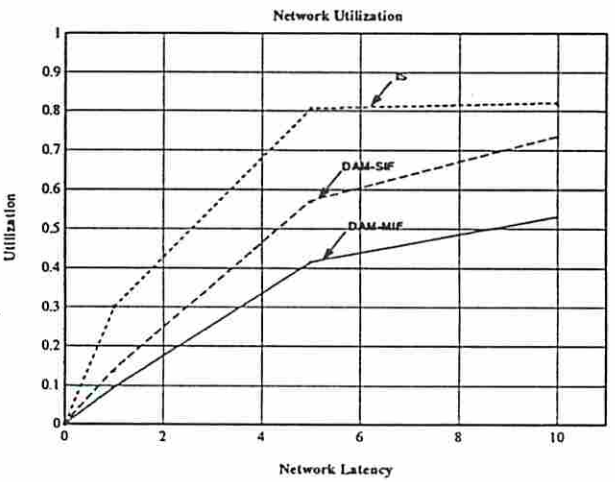

Figure 14: Network utilization of the three implementations for different latency conditions.

5 Conclusions

We have described in this paper a new array handling technique called the Direct Access Method, measured its performance, and compared it with that of the I-Structure representation on a simple parallel loop. It was found that the chief application of this technique would be in those cases where arrays display the characteristics of temporary variables (*i.e.*, the array is consumed "shortly" after being produced). It has been shown that this technique produced better performance under more realistic conditions by efficiently utilizing the system resources. At the same time, a large amount of the I-Structure memory is freed to be used by other portions of the program.

However, the Direct Access Method cannot be used in large programs at the exclusion of other approaches. For those arrays which are accessed over a long period of time after being produced, the I-Structure approach is still the most appropriate array handling method. Therefore, the Direct Access Method can be used selectively in conjunction with the I-Structures in order to enhance performance. A preliminary examination of large programs such as SIMPLE shows that the Direct Access Method can be applied to arrays produced and consumed within a function. However, further research is needed to determine the criteria for using the Direct Access Method. Using this criteria, the data dependence graph of a program can be analyzed at compile-time to determine the most appropriate array handling method.

References

[1] Arvind, L. Bic, and T. Ungerer. Evolution of dataflow computers. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 1, pages 3–33. Prentice Hall, 1991.

[2] Arvind and K. Gostelow. The U-interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[3] Arvind and R. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Parallel Architectures and Languages Europe*. Springer-Verlag, August 1987.

[4] Arvind and R. Thomas. I-structures: An efficient data type for functional languages. Technical Report LCS/TM-178, MIT, Laboratory for Computer Science, June 1980.

[5] 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.

[6] D. Culler and G. Papadopoulos. The explicit token store. In L. Bic and J-L. Gaudiot, editors, *Journal of Parallel and Distributed Comuting, Special Issue:Data-Flow Processing*, pages 289–308. Academic Press, Inc, December 1990.

[7] D. Culler, A. Sah, and et. al. 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.

[8] 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.

[9] J-L. Gaudiot. Structure handling in data-flow systems. *IEEE Transactions on Computer*, C-35(6):489–502, June 1986.

[10] J-L. Gaudiot and Y. Wei. Token relabeling in a tagged-token data-flow architecture. *IEEE Transactions on Computer*, 38(9), 1989.

[11] J. Gustafson. Reevaluating amdahl's law. *Communication of ACM*, 31(5):532–533, May 1988.

[12] 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.

[13] R. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *The 15th Annual International Symposium on Computer Architecture*, 1988.

[14] K. Kawakami and J. Gurd. A scalable dataflow structure store. In *The 13th Annual International Symposium on Computer Architecture*, pages 243–250. ACM and IEEE, ACM Press, June 1986.

[15] 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, 1991.

[16] J. McGraw, S. Skedzielewski, and et.al. *SISAL Language Reference Manual Version 1.2*, March 1985.

[17] R. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT, July 1991.

[18] R. Nikhil and Arvind. Can dataflow subsume von neumann computing? In *The 16th Annual International Symposium on Computer Architecture*, 1989.

[19] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. Technical Report 325-1, MIT, November 1991.

[20] G. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[21] 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.

[22] T. von Eicken, D. Culler, and et. al. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM and IEEE, ACM Press, May 1992.

[23] I. Watson and J. Gurd. A practical data-flow computer. *IEEE Computer*, 15(2):51–57, February 1982.

A Direct Array Handling Technique for Non-strict and Parallel Accesses in a Multithreaded Architecture

Chinhyun Kim and Jean-Luc Gaudiot

CENG Technical Report 93-01

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

A Direct Array Handling Technique for Non-strict and Parallel Accesses in a Multithreaded Architecture*

Chinhyun Kim† *Jean-Luc Gaudiot*†

Abstract

In this paper, we propose an array handling technique called the Direct Access Method (DAM) which enables array elements to be sent *directly* from the producer to the consumer activation while providing non-strict and parallel array accesses. The advantage of this technique over conventional structure storage is that network traffic is reduced and that no global memory space is required. Instead, array elements are produced in the frame memory of the producer activation and forwarded directly to the frame memory of the consumer activation. It is demonstrated here how the technique can be fine-tuned to those cases where an array behaves as a temporary variable and its consumption pattern can be determined at compile-time. Thus, the DAM is proposed as a complement rather than a replacement of the I-Structure representation. Performance measurements obtained by a deterministic simulation of a multithreaded model show that the Direct Access Method indeed performs better than the equivalent I-Structure implementation. The measurements further show that although the I-Structure model has a lower execution time under low network latency conditions, the DAM displays a performance advantage for higher communication network latencies.

1 Introduction

Almost all scientific applications which can benefit from parallel computing entail the manipulation of large data structures. The resulting performance can vary greatly depending on how these large structured data are partitioned and distributed across multiple processing elements. This must be done in such a way as to fully exploit existing parallelism with as little overhead as possible. Since communication accounts for much of this overhead, efficient structure representation and transmission schemes are needed. This paper presents an array handling technique geared mainly for scientific applications in an environment where the computation model is data-driven while the target machine is a multithreaded architecture.

Although every data partition and allocation scheme attempts to render all memory accesses local, remote memory accesses cannot be completely avoided. Frequent remote accesses can have a detrimental effect on the overall performance because of their long latency which can result in low processor utilization. The data-flow computing approach addresses the data partition/allocation problem indirectly by striving to provide a *latency tolerant* computing environment. The adverse effect of long latencies can be indeed minimized by effectively overlapping computation and communication[7]. To this end, three basic requirements must be satisfied:

1. The compiler should be able to expose all the fine/medium grain parallelism contained in the user's program.
2. The architecture of the target machine should be able to exploit such parallelism efficiently.
3. Communication costs should be minimized or at least *tolerable*.

The following is the current data-flow solution to the above requirements:

1. Use a functional language as a programming language which makes it relatively easy for a compiler to extract parallelism.
2. Use a multithreaded architecture to efficiently exploit fine/medium grain parallelism exposed by the compiler.
3. Use I-Structures[4] when arrays can be produced and consumed in parallel.

The concept of I-Structures is an important component of the overall data-flow solution because its objective is to exploit producer-consumer parallelism. This is achieved by providing non-strict and parallel accesses coupled with split-phased memory operations. Specifically, each of the I-Structure memory cell contains presence bits indicating the status of the cell *i.e.*, *full*, *empty*, or *pending*. Initially, each cell is set to empty. If a read request is made to an empty cell, the state of the cell is changed to pending and the request is queued in the deferred read request list. When the cell is written, the state changes to present. If the cell is marked pending prior to being written to, all deferred read requests are serviced.

*This work is supported in part by the National Science Foundation under grant No. CCR-9013965.

†Electrical Engineering-Systems University of Southern California Los Angeles, California 90089-2562

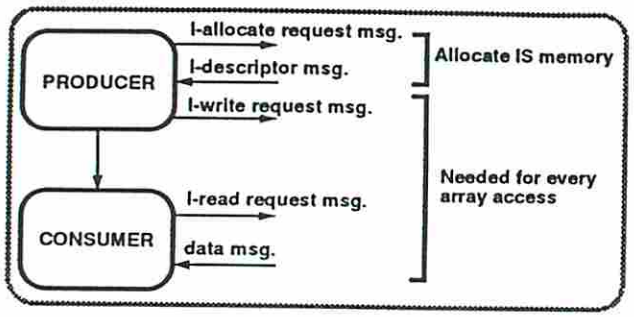

Figure 1: I-Structure and producer-consumer parallelism.

In addition to providing mechanisms to support the status of each memory cell, a memory manager which can efficiently handle dynamically created and deallocated arrays is also required. A complex hardware configuration of the I-Structure memory may be required to provide parallel array accesses as well. For example, the I-Structure memory can be low-order interleaved across the I-Structure nodes[20]. Alternatively, it can be high-order interleaved[14]. However, providing parallel accesses and split-phased read operations can cause excessive network traffic. This means that in the worst case, three remote messages are required for every array element to be written or read.

This overhead associated with the I-Structure representation may render it prohibitively expensive when an array behaves similarly to a temporary variable. This refers to cases where the life-time of an array is short because it is quickly consumed and destroyed after being created. In such situations, the fraction of the total time attributable to the overhead (allocate/deallocate + network traffic) may be relatively large resulting in inefficient operations. This paper thus introduces an alternate array handling mechanism called the *Direct Access Method* which, just like the I-structures, provides non-strict and parallel accesses, minus the overhead. We will show how this method can be used in situations where the arrays behave like temporary variables while the I-Structure representation is used for those situations where the arrays behave more like global variables. We will demonstrate that the Direct Access Method is an efficient array handling scheme in which each array element is sent *directly* from the producer to the consumer without being stored in an intermediate array storage. The mechanism makes the issue of structure memory management moot and reduces the number of remote accesses because write operations are guaranteed to be local memory operations.

In summary, the objective of this paper is to introduce the Direct Access Method for the passing of large chunks of data between producer and consumer processes. It is also intended to demonstrate its performance as well as its most likely context of utilization. Section 2 discusses how the data-flow computing model evolved from pure data-flow to multithreading. Section 3 introduces the Direct Access Method and its predecessor, the Token Relabeling approach. The simulation results which demonstrate the performance and the applicability of the Direct Access Method are presented in section 4. Finally, concluding remarks and directions for future research are offered in section 5.

2 Multithreading

We will now discuss first how the principles of data-flow computing have been evolving from pure data-flow toward multithreading, then we will describe the multithreading execution model. Multithreading is viewed as a practical means to promote data-flow computing rather than as a conceptually different model.

2.1 From Data-flow to Multithreading

One of the first basic tenets of parallel computing must be the ability to expose as much parallelism as possible at compile-time so that parallel (independent) instructions can be distributed across multiple processing elements of a machine and executed concurrently, resulting in higher performance. Traditional approaches to parallel computing using conventional languages such as FORTRAN make it difficult for a compiler to extract large amounts of parallelism automatically. This is due to the nature of the language semantics which are based on the von Neumann computation model: the model is inherently sequential in nature. On the other hand, the data-flow approach to parallel computing consists in starting at the top with languages such as Id[17] and SISAL[16] whose functional semantics make it comparatively easy for a compiler to expose all the parallelism contained in a program.

Being able to exploit large amounts of parallelism is crucial to the data-flow approach because its objective is to tolerate communication latency rather than to reduce it by executing ready instructions whenever long-latency inducing operations are initiated. By effectively overlapping communication with computation, latency caused by remote accesses can be masked[22]. Therefore, it is important to possess excess tasks ready for execution whenever long latency which is unavoidable in parallel computing occurs[17]. First generation data-flow machines were designed to switch context at every instruction, *i.e.*, the length of a thread is one instruction. For this purpose, a complex hardware facility called the *match* unit was developed to synchronize incoming data tokens and dynamically schedule instructions depending on the availability of their input operands. The first generation of data-flow machines is represented by the Manchester Dataflow Machine[1, 23], the Sigma-1[1, .12], and the Tagged Token Dataflow Architecture (TTDA)[3, 1]. However, these machines possessed the following drawbacks which hampered their economic viability and performance:

- The cost of the resulting hardware rendered them not economically viable. Indeed, associative matching of tokens to determine instruction executability requires expensive hardware.
- Dynamic scheduling of every instruction causes unnecessary overhead. Even those instructions which can be determined at compile-time to execute sequentially are needlessly scheduled dynamically.

Subsequent research efforts in the architecture arena have concentrated on addressing these points. The first breakthrough occurred in the design of the matching mechanism. The reason for the associative matching in the first data-flow machines was that the tag field of a token which indicates, among other things, the context of the token, has no correspondence to memory location[2]. Thus tokens were stored

randomly inside the waiting token storage pool of the processor and associative search was performed to find the matching token whenever a new token arrives at the processor. A direct matching scheme called the Explicit Token Storage (ETS) which does not use associative matching was first introduced in the Monsoon[6, 20]. This mechanism allocates at compile-time a memory slot to each arc (of a data-flow graph) belonging to the same *code block*, by assigning an offset from the base of a frame memory. A code block is a group of instructions which becomes activated when the frame memory is allocated to it at run-time. In general, a function body or a loop body corresponds to a code block. Therefore, once a code block is activated, tokens "know" exactly where in the frame memory they need to check for synchronization, making associative matching unnecessary. This mechanism resulted in faster matching speed (since it took locality into consideration) while the hardware could be made simpler.

Further improvement for efficient execution can be achieved by grouping instructions to form threads. One of the drawbacks of the previously mentioned data-flow machines is that every instruction is still scheduled dynamically. This generates an unnecessary burden on the machine when scheduling those instructions that are *known* to be executed in sequence. Such instructions can be better scheduled statically for sequential execution, thereby avoiding the cost of synchronization. In multithreading, instructions of a code block are partitioned to form threads. Although a thread may be scheduled dynamically once activated, instructions belonging to the thread are executed sequentially without further synchronization. Instruction-level parallelism can be further exploited by way of pipelining or superscalar techniques. The first multithreaded architectures that evolved from the data-flow architectures are the Dataflow/von Neumann Hybrid Architecture[13] and the P-RISC[18]. The basic difference between the two is that synchronization is implicit in the Hybrid Architecture provided by the hardware supported presence bits while in P-RISC, synchronization is explicit by the compiler generated codes. More recent proposals such as *T[19, 5] and the Threaded Abstract Machine (TAM)[7] render the concept of thread more explicit by compile-time specification of the beginning and the end of a thread. Clear delineation of threads can make thread scheduling more efficient.

The important point to recognize is that the fundamental data-driven approach to parallel computing has not changed, i.e., extract a lot of parallelism at compile-time so that computations can be effectively overlapped with unavoidable communications (thereby incurring long latency costs) occurring in a large parallel system. Multithreading is an outgrowth of that research aimed at developing an efficient execution model for fine/medium grain parallelism. Initially, every aspect of execution functions were given to the hardware which resulted in costly systems. Gradually, the emphasis is being shifted to software (compiler) resulting in the current version of multithreading execution models.

2.2 Execution Model

The multithreading execution model used for this study is adapted from that of *T and TAM. A code block corresponding to a loop body or a function body in the program text is divided at compile-time into threads. An instance of a code block is said to be *active* when a frame memory is allocated to it and the input parameters are sent. The frame memory

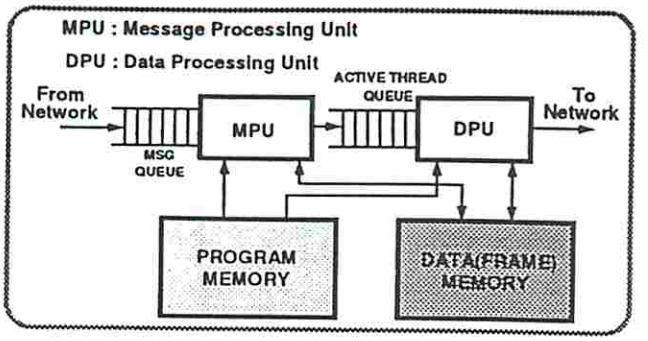

Figure 2: The multithreaded architecture model of a processing element.

request is sent to the *frame manager* residing at each processing node. The amount of memory needed in the frame is determined at compile-time by analyzing the memory requirements of the code block. After a code block is activated, each thread is executed when its activation requirement is satisfied. A thread executes in a non-preemptive mode, i.e., once a thread is activated, its execution continues without suspension until all the instructions of the thread have been executed. The first thread to be executed is the *initialization* thread which receives and stores the input parameters in the appropriate frame locations. It may also initialize synchronization variables used to activate other threads.

In general, an activated code block performs various other functions in addition to the actual computations as specified in the program. The initialization thread as described in the previous paragraph is one. Threads constituting a code block can be classified into the following four classes:

1. Initialization thread: There is always one initialization thread in a code block. It is the first thread to be executed in the code block and its function is to receive input parameters and initialize other local constants and variables.

2. Interface thread: There can be a number of these threads. Their main function is to communicate with the outside world; that is, activations in other processors. For every remote read request, a matching interface thread is required to receive data because of a split-phase operation. An interface thread may update synchronization variables after receiving a token. When the terminal value is reached, the corresponding compute thread is activated.

3. Compute thread: These are the threads that actually perform the computations specified in the program as the programmer sees it. A compute thread may be triggered by the initialization thread, interface threads or other compute threads.

4. Control thread: This thread determines whether the current activation has done all its required work. For instance, if the code block represents a loop body of a parallel loop, the thread determines whether the frame memory should be reused by another loop iteration or be deallocated.

The multithreaded architecture model used for this study (Figure 2) is similar in basic concept to the Decoupled Graph/

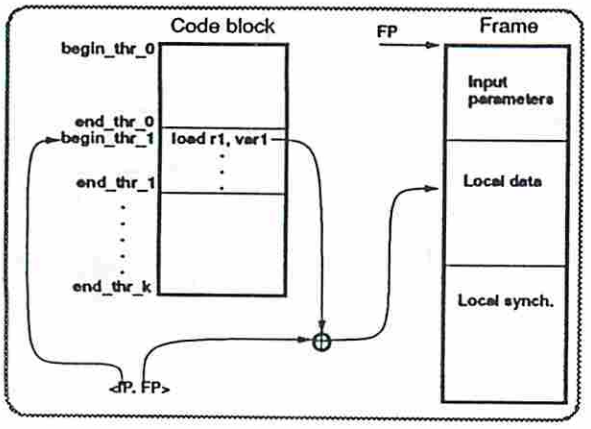

Figure 3: The state of a thread is completely specified by a descriptor consisting of <IP.FP>.

Computation Data-Driven Architecture[8] and *T[19]. The basic concept behind the two models is to decouple the functions of thread activation from that of thread computation. In the Decoupled architecture, the thread executability is determined by the Data-Flow Graph Engine (DFGE) while the activated threads are executed in the Computation Engine (CE). On the other hand, in the *T architecture, the Synchronization Coprocessor (sP) performs thread activation while the Data Processor (dP) executes the activated threads. The difference between the two models is that associative matching is assumed in the Decoupled architecture whereas in *T, the `join` instruction, first proposed in P-RISC, is used for synchronization.

The multithreaded architecture shown in Figure 2 consists of two processing units. The Message Processing Unit (MPU) is used to execute the interface threads whose main function is to receive tokens and activate compute threads. Once a compute thread is activated, the MPU enqueues the thread descriptor in the Active Thread Queue (ATQ) where it is eventually dequeued by the Data Processing Unit (DPU). The DPU reads a new thread descriptor from the ATQ when the last instruction of the current thread is executed. A thread descriptor consisting of two values, <IP.FP>, completely specifies a thread. IP is an instruction pointer which points at the first instruction of the thread and FP is a frame pointer which points at the base of the frame memory. Once the IP is loaded in the IP register of the DPU, subsequent instructions are executed by incrementing the register value appropriately. Frame locations are accessed using the FP in conjunction with the instruction operands which are offset values. Figure 3 shows how the first instruction of a thread is executed using the thread descriptor.

3 The Direct Access Method

The first subsection describes the Token Relabeling method which inspired the development of our array handling technique called the Direct Access Method. In the second subsection, implementation of this idea in the multithreading context is discussed. Specifically, the direct access mechanism and necessary compile-time analysis is described.

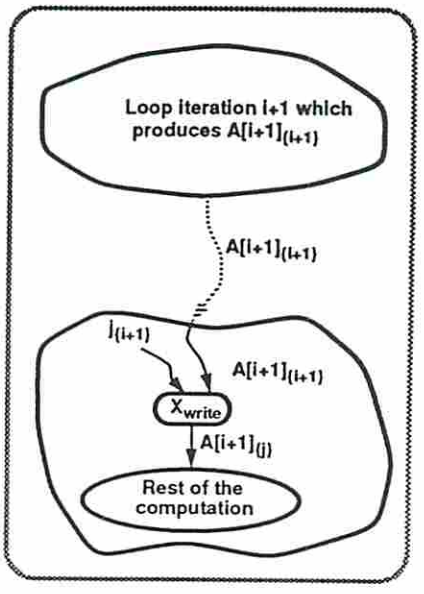

Figure 4: A token carrying an array element needs to be relabeled appropriately in order for it to be forwarded to the correct consumer context.

3.1 Token Relabeling

The Token Relabeling (TR) scheme has been proposed for handling structured data in the tagged token data-flow architecture [9]. Like I-Structures, it is mainly aimed at scientific applications in which arrays are the main structured data types. The central point of this scheme is that, unlike other methods, array elements should not be stored in the structure store facilities. Instead, each array element is treated as a scalar entity and is carried in a token. Individual array elements are uniquely identified by the iteration field of the tag. The basic array accessing mechanism of the TR method is also different from the other array handling methods using a structure store. In the I-Structure scheme, for example, array elements are first sent to the I-Structure controller to be written into the corresponding memory location. Likewise, an array consumer sends a read request to the structure store for consumption. In the Token Relabeling method, each array element is sent directly from the producer to the consumer, bypassing the structure store. Array elements are temporarily stored in the waiting storage pool of the consumer's match unit until they are selected and consumed. Functionally, the Token Relabeling method is identical to the I-Structure approach. Both schemes provide non-strict array accesses and deferred reads. The difference is that the Token Relabeling method does it without special structure storage.

A common scenario for applying the Token Relabeling method is a situation where parallel loops form a producer-consumer relationship such that array(s) produced by the producer loop is input to the consumer loops. The basic token relabeling rule is that an array element $A[i]$ is computed by the producer loop iteration i and is tagged by i, $A[i]_{\{i\}}$. Likewise, a consumer loop iteration j consumes array element $A[j]_{\{j\}}$. If $A[i]_{\{i\}}$ is consumed by the loop iteration $j = i$, no token relabeling is required. However, if $A[i]_{\{i\}}$ is consumed by the consumer loop iteration $j \neq i$, ar-

ray element $A[i]$ must be appropriately relabeled for correct consumption. For example, assume a consumer iteration j consumes $A[i+1]$, $j \neq i+1$. Then $A[i+1]_{(i+1)}$ produced at iteration $i+1$ must be relabeled to $A[i+1]_{(j)}$ so that the token is correctly consumed by the consumer iteration j. Figure 4 shows how the iteration field of the tag is appropriately relabeled before the array element is consumed.

The advantages of the Token Relabeling method over the I-Structure is that, 1) no special resources are required to provide parallel non-strict array accesses and that 2) it is more efficient. The Token Relabeling method is a more efficient technique because, for every array element transfer from a producer to a consumer, at most only one remote access is required. On the other hand, there are potentially three remote accesses when an I-Structure is used. A study has shown that given the same program, the TR method generates less network traffic than the I-Structure resulting in faster execution time[10]. The Token Relabeling method has been also applied in a scheme which exploits at run-time, parallelism that were hidden at compile-time[15]. However, there are a number of drawbacks to using the Token Relabeling method. One of the drawbacks is that precious match unit storage space is taken by the array elements. Therefore, the TR method is best suited for those situations in which the array in question has the characteristics of a temporary variable. In other words, the TR method may be more appropriate when an array is produced and consumed over a relatively short period of time. For those cases in which an array is accessed over a long period of time, I-Structure representation may be a better choice.

3.2 Direct Access Mechanism in a Multithreaded Execution Model

The implementation of the Token Relabeling scheme in the tagged token data-flow architecture used the following two key features:

- The iteration field is part of the context identifier and its value can be modified at run-time so that a token produced in one context can be made to belong to another context by appropriately changing the iteration tag field.
- Once the context of a token is changed, the token is guaranteed to be forwarded to the processor that executes the new context by a predefined mapping function which uses the tag as its input argument.

Since the multithreaded architecture does not provide the features mentioned above, a run-time system is required if the Token Relabeling scheme is to be implemented. The responsibility of the run-time system is to provide the tagged token abstraction to the programmer. To do this, the run-time system needs to map the context tag field to the context of the multithreaded architecture, the frame pointer FP. Thus, whenever a relabeling occurs, the run-time system must search its mapping table to determine which frame pointer maps to the new context as represented by the modified tag value. This approach, which faithfully reproduces the Token Relabeling mechanism in a multithreaded architecture may, however, result in poor performance due to the overhead of the run-time system.

Our new array handling technique called the Direct Access Method, does not require the type of run-time system

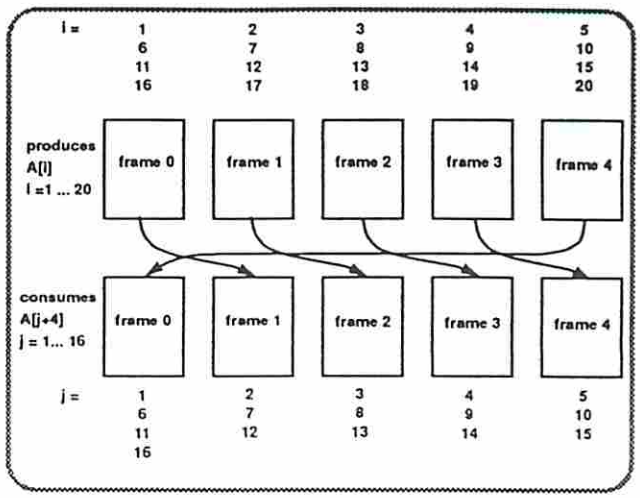

Figure 5: The mapping between the producer loop activation frames and the consumer loop activation frames is static when the array consumption pattern is regular.

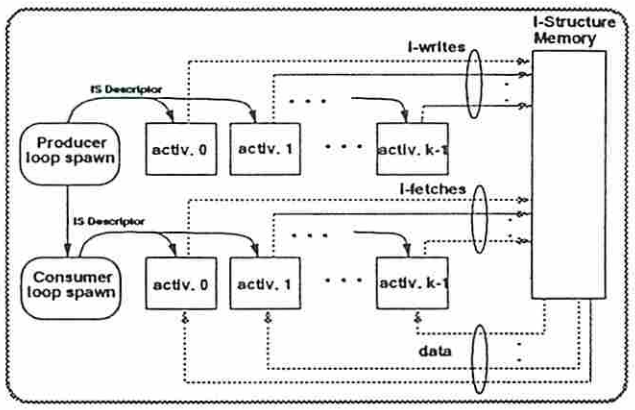

Figure 6: Producer and consumer loop schema using the I-Structure.

mentioned above. This technique, like the Token Relabeling scheme, directly forwards computed array elements from the producer to its consumer. Unlike the Token Relabeling scheme, however, the connection between the producer and the consumer activation is determined at compile-time. This static connection is achieved by determining the consumer's array consumption pattern through array subscript analysis[1]. The consumption pattern in conjunction with the producer-consumer loop bounds and k is used to ascertain which producer and consumer loop activations interact with each other at run-time.

Let us assume that the producer loop ranges from $i = i_{lo} \ldots i_{hi}$ and the consumer loop ranges from $j = j_{lo} \ldots j_{hi}$. In addition, further assume that only k iterations are allowed to be executed in parallel for each loop. Then there are k producer frames, $ProdFrame_l$ and k consumer frames, $ConsFrame_n$, $l, n = 0 \ldots k-1$. Furthermore, assume that the consumer loop consumes the array, A, computed by the producer loop according to the array subscript, $S(j)$. The

[1] It is assumed that Intermediate Form 1 (IF1)[21] data dependency graph produced by the SISAL compiler is used in the analysis.

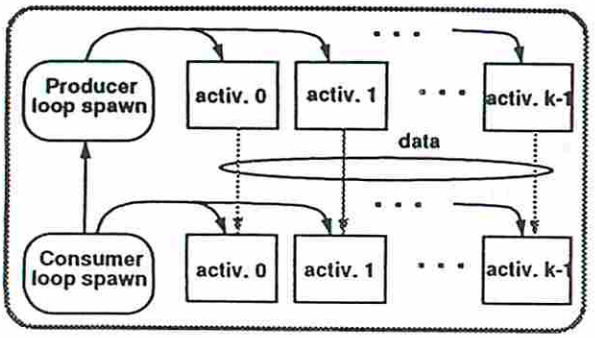

Figure 7: Producer and consumer loop schema using the Direct Access Method.

objective is to connect (at compile-time) the producer iteration $S(j)$ which computes $A[S(j)]$ with the consumer iteration j which consumes it.

Once allocated, a frame is generally reused by multiple iterations without going through the deallocate/allocate process by the frame manager. The reason is to reduce overhead caused by dynamic memory management. Assume, in the context of this discussion that a frame memory used by iteration i is reused by iteration $i + k$. In other words, a frame memory m is reused by iteration i according to the following expression where i_{lo} is the lower loop bound:

$$m = (i - i_{lo}) \ mod \ k$$

An iteration i is said to be connected to iteration j if it has the FP of the frame memory used by iteration j. Therefore, the connection between the producer iteration $i = S(j)$ and the consumer iteration j can be equivalently represented in terms of the corresponding frames $ProdFrame_l$ and $ConsFrame_n$ in which l and n are determined by the following expression:

$$l = (S(j) - i_{lo}) \ mod \ k, \quad n = (j - j_{lo}) \ mod \ k$$

Once k producer frames are connected to k consumer frames according to the array consumption pattern, all iterations are guaranteed to be connected. This is because all iterations map to one of k frames. An example is shown in Figure 5 ($i = 1 \ldots 20$, $j = 1 \ldots 17$, $S(j) = j + 4$, $k = 5$). In the example, $ConsFrame_1$ is mapped to $ProdFrame_0$. This is correct because the consumer iteration 2 consuming $A[6]$ is activated on $ConsFrame_1$ whereas the producer iteration 6 producing $A[6]$ is activated on $ProdFrame_0$.

We also see in the example that the array elements computed by the producer loop are not completely consumed. Assuming that there are no other consumer loops, iterations $i = 1 \ldots 4$ are not consumed. In the Direct Access Method, it is crucial for the compiler to determine the exact number of consumers for each producer loop activation. Otherwise, a deadlock may occur due to the nonterminating loop body.

Figure 6 shows that the I-Structure descriptor is sent as an input parameter to every producer and consumer activations. Each activation uses the descriptor to compute the exact location of the I-Structure memory cell it needs to access. In the Direct Access Method, a frame pointer of the communicating counterpart activation is sent to each activation instead of the I-Structure descriptor. Figure 7 shows that frame pointers of the consumer activations are sent to the producer. Upon receiving the corresponding consumer's frame pointer, a producer activation can transmit computed array elements. After a small initial start-up time, the actual computation performed by the activations is overlapped with the transmission of frame pointers. In the current implementation, a buffer of size k is allocated in the frame of the producer loop spawning code block to store the incoming consumer frame pointers. The function that maps frames to iterations are code generated as part of the loop spawning code block. As each producer activation is created, the corresponding consumer frame pointer is read from the buffer according to the mapping function and sent to the activation along with other input parameters.

4 Performance Measurement

This section reports on the performance of three different implementations of the Livermore Loop 1. Each version is implemented using a different array handling technique and the resulting performance is discussed.

4.1 The Simulated Architecture

The basic configuration of a processing node of the simulated architecture used in the performance measurement is as shown in Figure 2. Each node consists of a Message Processing Unit (MPU) and a Data Processing Unit (DPU) which operate asynchronously to each other. The main function of the MPU is to receive incoming messages and activate threads that are triggered by the incoming data tokens. When a thread is activated, its descriptor is inserted into the Active Thread Queue (ATQ) for eventual execution by the DPU. As thread scheduling is not the main research issue, a simple FIFO policy was adopted. The execution of an instruction is assigned one time unit assuming that one instruction can be issued at each clock cycle. A hypercube topology was selected for the interconnection network.

In the simulator, the I-Structure controller is implemented with the following key implementation parameters:

- Array elements are low-order interleaved across I-Structure (IS) nodes; an array element $A[i]$ is mapped to the IS node, $inode = i \ Mod \ M$, where M is the total number of the I-Structure nodes in the system. There are as many IS nodes as there are processing nodes.
- The cost of the I-Structure memory allocation and the initialization time is idealized to zero.
- Within the IS node, each memory read and write operation takes one time unit. Every enqueue and dequeue operation to/from the deferred read queue also takes one time unit.
- No extra cost is assigned for accessing an IS node, *i.e.*, no cost is incurred if an activation executing in the processing node k accesses an IS node that is local to the processing node.
- The cost of any remote message including i-write and i-fetch messages is determined by the currently set network latency value and the routed path.
- The processing node in which an array producer activation executes and the IS node in which it writes into are allocated independently. Therefore, i-write or i-fetch operations are not guaranteed to be local.

DAM-MIF

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	2794.0	2687.0	3195.0	4111.0
L = 1.0	2788.0	2711.0	3218.0	4176.0
L = 5.0	3107.0	3217.0	3799.0	4678.0
L = 10.0	5604.0	4790.0	5786.0	6995.0

I-Structure

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	1742.0	1844.0	2121.0	3756.0
L = 1.0	1742.0	1848.0	2168.0	3792.0
L = 5.0	2691.0	3755.0	4751.0	5785.0
L = 10.0	5271.0	7506.0	9330.0	11151.0

DAM-SIF

PE:DATA SIZE	16:500	32:1000	64:2000	128:4000
L = 0.0	1935.0	2668.0	4603.0	8743.0
L = 1.0	1953.0	2699.0	4624.0	8794.0
L = 5.0	2483.0	3458.0	5043.0	9190.0
L = 10.0	4483.0	6009.0	7091.0	10012.0

Table 1: Execution time of each implementation when the data and the processors are scaled.

4.2 Benchmark and Conditions of Experiments

Three versions of the Livermore Loop 1 are implemented to observe the effects of different array handling techniques on the overall performance of the program. Livermore Loop 1 is a simple parallel loop which produces an array of one dimension. To supply two of its array input parameters, Y and Z, a producer loop is provided. A SISAL version of the program is shown in Figure 8. One of the three versions of the Livermore Loop 1 is implemented using the I-Structure with the assumptions listed in the previous subsection.

The other two implementations are different versions of the Direct Access Method. In the first version, a single iteration is active at a time in a given frame memory and the frame is reused by another activation only after the current activation is terminated. This implementation is called the DAM-Single Iteration Frame (DAM-SIF). In the second version, multiple iterations are active simultaneously in a given frame memory. In this implementation, called the DAM-Multiple Iteration Frame (DAM-MIF), w iterations are simultaneously active using the same frame memory. In this mechanism, memory space to store loop variables for w iterations are provided in a frame. In addition, since different instances of a thread may be active simultaneously in the DAM-MIF, the thread descriptor is provided with an extra field, `i`, in addition to FP and IP to differentiate between different instances of a thread. Using this new field, different instances of a thread may become activated and executed independently within a given frame memory.

Two different performance measurements were performed on each implementation under various network latency conditions. The first one measures the data scalability inspired by [11]. In the Livermore Loop 1, the amount of parallelism is simply the upper loop bound, n. Ideally, the execution time should stay constant if the problem size and the number of processors were increased simultaneously by the same factor. In reality, the execution time will vary depending

```
define Main

type OneDim = array[real];

function Producer(n:integer returns OneDim,OneDim)
 for k in 1,n
 returns array of real(k) * 0.5
   array of real(k) * 0.8
 end for
end function

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

function Main(n1,n2:integer; Q,R,T:real returns OneDim)
 let
   Answer := Loop1(n2,Q,R,T,Producer(n1))
 in
   Answer
 end let
end function
```

Figure 8: SISAL program of Livermore Loop 1.

on a number of parameters. For example, machine characteristics such as network latency and compiler dependent factors such as the activation spawning mechanism as well as the actual amount of iterations allowed to be active simultaneously all influence the execution time. For the measurements, the data size was varied from 500 to 4000 and the number of processors from 16 to 128. Since the actual amount of parallelism is capped by a loop-bounding mechanism, the maximum number of activations allowed concurrently was made to vary from 96 to 768; on the average, 6 iterations are active concurrently at each processing node. The measurements were repeated for four different communication latencies (L = 0, 1, 5, and 10 time units).

We define the scaling factor SF as the ratio between the execution time of the reference set and the execution time when the processors and the data size are scaled:

$$SF(PE, DATA\ SIZE) = EX_{ref}/EX(PE, DATA\ SIZE)$$

The EX_{ref} is the execution time of the smallest set (the reference set), *i.e.*, $EX(16, 500)$ for each latency condition.

The second is the measurement of speedup. For this measurement, the problem size of the Livermore Loop 1 was fixed at 2000 and the number of processors was varied from 1 to 64. The execution time of each implementation utilizing different numbers of processors was measured for two latency values of 1 and 10. For a fixed number of processors, the execution time can vary greatly depending on how many loop instances are allowed to be active simultaneously. Therefore, for each measurement, the loop bounding value k was varied and the best execution time was selected.

4.3 Observations

Table 1 shows the execution time and Figure 9 shows the data scalability curve of each implementation at different latency conditions. Each point of Figure 9 is computed according to the scaling factor formula. For example, the scaling factor $SF(128, 4000)$ of the DAM-MIF at latency = 10 is computed by $EX(16, 500)/EX(128, 4000) = 0.80$. To avoid cluttering, only the data scalability curve for the DAM-MIF and the I-Structure implementations are shown.

In terms of absolute execution time, the I-Structure implementation performs the best for small latencies (L=0,1).

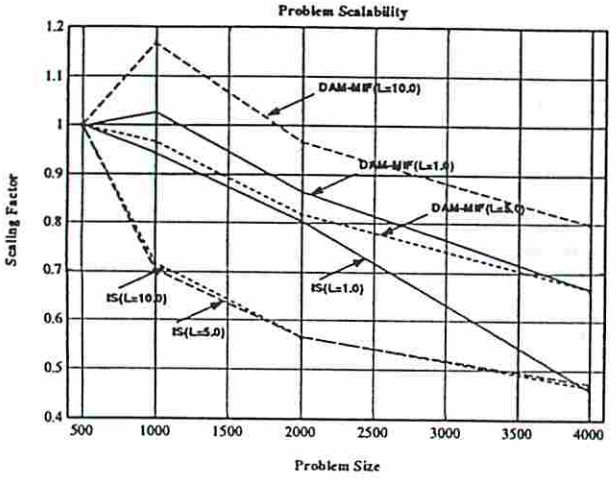

Figure 9: Data scalability graph for DAM-MIF and I-Structures.

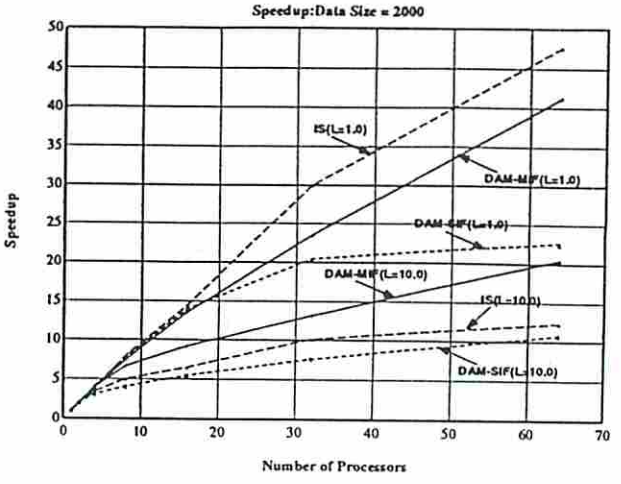

Figure 10: Speedup of three different implementations when latency = 1.0, 10.0.

	DAM-MIF		I-Structure		DAM-SIF	
Number of PEs	L = 1.0	L = 10.0	L = 1.0	L = 10.0	L = 1.0	L = 10.0
1	108083.0	108083.0	104155.0	104155.0	76455.0	76455.0
2	53982.0	53978.0	52109.0	52139.0	38392.0	38483.0
4	27137.0	27265.0	27045.0	30700.0	19384.0	24976.0
8	14599.0	16329.0	13076.0	20908.0	9997.0	19021.0
16	7970.0	11755.0	7188.0	16594.0	5511.0	14000.0
32	4597.0	8230.0	3469.0	10274.0	3758.0	10090.0
64	2623.0	5283.0	2187.0	8434.0	3396.0	7091.0

Table 2: Execution time of each implementation when the data and the processors are scaled.

On the other hand, DAM-MIF produces the best result for longer latencies (L=5,10) and larger processors ($\geq$ 32). In terms of data scalability, DAM-MIF consistently performs the best for all latencies. As can be seen from the graph of Figure 9, the scalability of the DAM-MIF degrades to 0.8 (L=10) and to 0.67 (L=1,5); the scalability of the I-Structure implementation degrades to around 0.47 for all three latencies. For a higher latency value, DAM-MIF performs better in both the execution time and data scalability. However, the DAM-SIF performs poorly against other implementations. Its only best execution time is for two cases when 16 processors (L=5,10) are utilized.

Table 2 lists the execution time of each implementation when the problem size is fixed at 2000 and the processors are increased from 1 to 64. As observed in the data scalability measurement, the DAM-MIF performs the best when a larger number of processors are used under longer latency condition. In the measurement, the DAM-MIF produced the best execution time when the processors $\geq$ 8 and the latency is 10. For a latency of 1, the I-Structure had the fastest execution time when the number of processors were 32 and 64. For a smaller number of processors, the DAM-SIF implementation resulted in the best execution time. Figure 14 shows the speedup curve of each implementation at different latency conditions. At latency = 1, the I-Structure results in the best speedup ($\approx$ 47.5). Because of the inefficient frame usage, the speedup curve of the DAM-SIF implementation quickly saturates after 32 processors. When the latency is increased by 10-fold to 10, the DAM-MIF produces best speedup ($\approx$ 20.5). At this latency condition, the speedup of the I-Structure version is only 25 % of the speedup at latency = 1 while the speedup of the DAM-MIF and the DAM-SIF is only reduced by half.

4.4 Interpretation

The key to achieving good performance is to effectively overlap computation and communication. At the same time, available resources must be utilized efficiently. In both the data scalability and the speedup measurement, the performance of the I-Structure version degraded rapidly for longer latency conditions. The cause of such performance characteristic is that the I-Structure implementation over-utilizes the network. Although split-phased remote memory operations provide environment for efficient processor utilization, network resources may easily be overloaded with remote message traffic becoming a bottleneck. In the simulated program, the producer loop creates two array elements and the consumer loop (Livermore Loop 1) consumes three array elements at every iteration. Assuming the array size is N, the worst case expression for the total number of remote array operations is:

$$\begin{aligned} \textit{Total remote msgs} &= \textit{i-write msgs} + \textit{i-fetch msgs} + \textit{data msgs} \\ &= 2N + 3N + 3N = 8N \end{aligned}$$

In the DAM-SIF implementation, the network resource is better utilized than in the I-Structure implementation because the message traffic is reduced. This is achieved by the direct array accessing mechanism in which the array elements are sent directly from the producer to the consumer activations; write operation is always a fast local memory write operation to a frame memory. The expression for the total number of remote messages is,

$$\begin{aligned} \textit{Total remote msgs} &= \textit{data msgs} + \textit{sync msgs} \\ &= 3N + 3N = 6N \end{aligned}$$

Synchronization message in the above expression is required by the very mechanism of the Direct Access Method

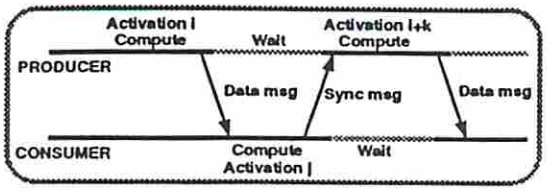

Figure 11: In the DAM-SIF implementation, a frame cannot be reused immediately after an activation until the sync msg is received.

when a frame memory is reused[2]. Its function is to synchronize the producer and the consumer activations before an array element is sent directly from the producer to the consumer activation. Without synchronization, an array element may be sent before the next consumer iteration becomes active. Unfortunately, this mechanism makes effective overlapping of computation and communication difficult, resulting in low processor utilization. More specifically, a frame used by a previous producer activation cannot be reused immediately by a new producer activation until a sync msg is received, indicating that the new consumer instance is active (Figure 11). In the meantime, the frame memory is not utilized by any activation. A relatively poor performance of the DAM-SIF in both performance metrics is the result of low processor utilization caused by this synchronization mechanism.

The DAM-MIF mechanism solves the low processor utilization problem by allowing multiple iterations to be active simultaneously using the same frame memory. At the same time, remote message traffic is further reduced. In this implementation, a thread descriptor is represented by <IP.FP.i>. The i field is used to distinguish different instances of a same thread. Figure 12 shows that for w iteration instances, the producer activation sends the data to the consumer activation without exchanging synchronization messages. The total number of remote messages for the DAM-MIF is thus:

$$\begin{aligned} Total\ remote\ msgs &= data\ msgs + sync\ msgs \\ &= 3N + 3\lceil N/w \rceil \end{aligned}$$

The DAM-SIF is a special case of the DAM-MIF in which the value of w is one.

Figures 13 and 14 show the processor and the network utilization of the three implementations at different latency values when the same number of instances are allowed to be active simultaneously. Because of the reasons discussed, the I-Structure implementation saturates the network for a latency of 5 and above, although it performs well under low latency conditions. It is difficult to increase processor utilization under longer latency condition because the network is already saturated. In the DAM-SIF implementation, processors are consistently underutilized. In the DAM-MIF implementation, the network utilization is still around 50 % even at latency=10 leaving more room for extra activations. Among the three implementations, the DAM-MIF implementation utilizes the processor and the network resources most efficiently given the same conditions.

[2]Synchronization messages are not required in an ideal case where a parallel loop can always be unraveled. However, in most real situations where memory resources are finite, a memory frame must always be reused.

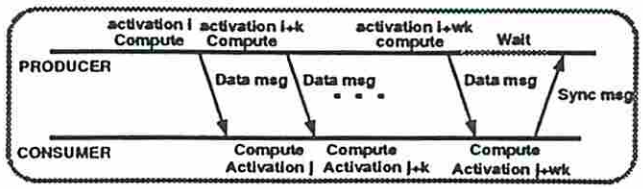

Figure 12: A processor is better utilized in the DAM-MIF implementation because a frame memory is reused immediately after an activation is terminated. At the same time, network load is reduced.

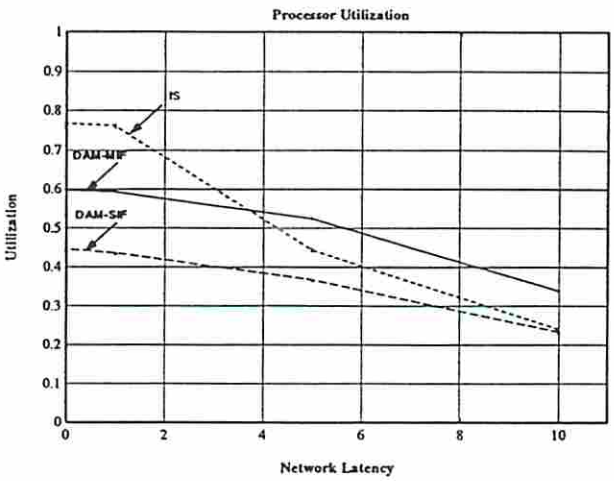

Figure 13: Processor utilization of the three implementations for different latency conditions.

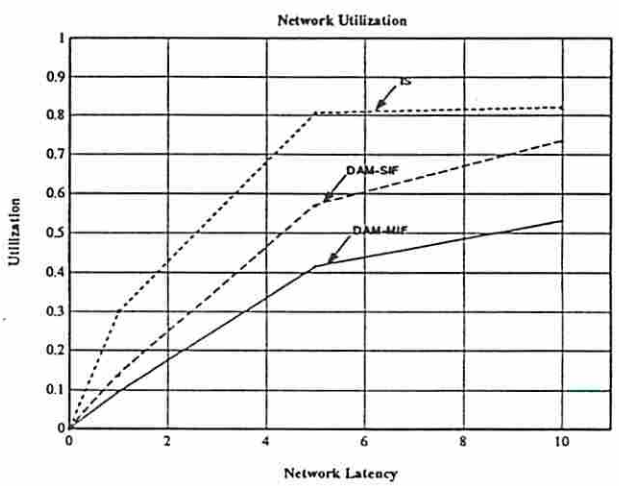

Figure 14: Network utilization of the three implementations for different latency conditions.

5 Conclusions

We have described in this paper a new array handling technique called the Direct Access Method, measured its performance, and compared it with that of the I-Structure representation on a simple parallel loop. It was found that the chief application of this technique would be in those cases where arrays display the characteristics of temporary variables (*i.e.*, the array is consumed "shortly" after being produced). It has been shown that this technique produced better performance under more realistic conditions by efficiently utilizing the system resources. At the same time, a large amount of the I-Structure memory is freed to be used by other portions of the program.

However, the Direct Access Method cannot be used in large programs at the exclusion of other approaches. For those arrays which are accessed over a long period of time after being produced, the I-Structure approach is still the most appropriate array handling method. Therefore, the Direct Access Method can be used selectively in conjunction with the I-Structures in order to enhance performance. A preliminary examination of large programs such as SIMPLE shows that the Direct Access Method can be applied to arrays produced and consumed within a function. However, further research is needed to determine the criteria for using the Direct Access Method. Using this criteria, the data dependence graph of a program can be analyzed at compile-time to determine the most appropriate array handling method.

References

[1] Arvind, L. Bic, and T. Ungerer. Evolution of dataflow computers. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 1, pages 3–33. Prentice Hall, 1991.

[2] Arvind and K. Gostelow. The U-interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[3] Arvind and R. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Parallel Architectures and Languages Europe*. Springer-Verlag, August 1987.

[4] Arvind and R. Thomas. I-structures: An efficient data type for functional languages. Technical Report LCS/TM-178, MIT, Laboratory for Computer Science, June 1980.

[5] 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.

[6] D. Culler and G. Papadopoulos. The explicit token store. In L. Bic and J-L. Gaudiot, editors, *Journal of Parallel and Distributed Comuting, Special Issue:Data-Flow Processing*, pages 289–308. Academic Press, Inc, December 1990.

[7] D. Culler, A. Sah, and et. al. 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.

[8] 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.

[9] J-L. Gaudiot. Structure handling in data-flow systems. *IEEE Transactions on Computer*, C-35(6):489–502, June 1986.

[10] J-L. Gaudiot and Y. Wei. Token relabeling in a tagged-token data-flow architecture. *IEEE Transactions on Computer*, 38(9), 1989.

[11] J. Gustafson. Reevaluating amdahl's law. *Communication of ACM*, 31(5):532–533, May 1988.

[12] 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.

[13] R. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *The 15th Annual International Symposium on Computer Architecture*, 1988.

[14] K. Kawakami and J. Gurd. A scalable dataflow structure store. In *The 13th Annual International Symposium on Computer Architecture*, pages 243–250. ACM and IEEE, ACM Press, June 1986.

[15] 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, 1991.

[16] J. McGraw, S. Skedzielewski, and et.al. *SISAL Language Reference Manual Version 1.2*, March 1985.

[17] R. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT, July 1991.

[18] R. Nikhil and Arvind. Can dataflow subsume von neumann computing? In *The 16th Annual International Symposium on Computer Architecture*, 1989.

[19] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. Technical Report 325-1, MIT, November 1991.

[20] G. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[21] 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.

[22] T. von Eicken, D. Culler, and et. al. Active messages: a mechanism for integrated communication and computation. In *The 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM and IEEE, ACM Press, May 1992.

[23] I. Watson and J. Gurd. A practical data-flow computer. *IEEE Computer*, 15(2):51–57, February 1982.