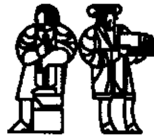
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Evaluation of the MIT Tagged-Token Dataflow Architecture

Computation Structures Group Memo 281
December, 1987

Arvind

Gino Maa

Stephen Brobst

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-0661.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Evaluation of the MIT Tagged-Token Dataflow Architecture

1 Introduction

The dataflow approach offers an elegant framework for exposing inherent program parallelism and providing an execution model for exploiting this parallelism. In the companion papers [2] and [5] we characterized the program parallelism and resource requirements associated with dataflow computation, respectively. These papers explored dataflow execution in the context of an abstract model for executing graphs generated for a tagged-token architecture. Minimal assumptions regarding a specific architectural implementation for executing dataflow graphs were made, albeit that the instruction set used certainly imposes some restrictions. In this paper we will evaluate some of the implementation tradeoffs in the MIT Tagged-Token Dataflow Architecture. We will discuss the explicit costs and benefits of the dataflow execution mechanism as well as the effectiveness of specific resource management policies as implemented on our machine.

We start by introducing a high-level view of the MIT Tagged-Token Dataflow Architecture. In Section 3 we will demonstrate the ability of the Tagged-Token Dataflow Architecture to tolerate latencies and synchronization waits; properties which we believe are imperative to the success of a scalable multiprocessor system. In a multiprocessor system it is also important to consider methods for balancing workload and data across hardware resources in the machine. In Section 4 we investigate a collection of program mapping strategies and give empirical data comparing the performance of the strategies developed. In Section 5 we show that the granularity of task distribution has a much more significant impact upon machine performance than does the policy for distributing tasks. Section 6 examines the issues involved with managing data structure storage and the consequences of structure mapping policies upon machine performance. In Section 7 we will look at the token storage and instruction scheduling facility in the Tagged-Token Dataflow Architecture. A characterization of the token volatility in the storage facility will be given along with some implications for the design of this component of the architecture. In [1] we established the relationship between dataflow MIPs and to von Neumann MIPs. Up to this point we have ignored the *cost* of executing dataflow instructions. Section 8 will discuss the complexity of instruction execution in the dataflow model as compared to the traditional von Neumann framework. The last section gives some remarks regarding the implemention of a realistic and cost-effective dataflow machine.

2 The MIT Tagged-Token Dataflow Architecture

The MIT Tagged-Token Dataflow Architecture, like the Manchester and Sigma-1 machines [6, 8], is a machine to execute dataflow graphs directly. Throughout this paper we will describe the TTDA as if it exists. However, it should be clear that only "soft implementations" exist, *i.e.,* emulators and a simulator. As will be obvious shortly, the TTDA has been specified to the register transfer level but not to the circuit design level.

The machine consists of a number of identical Processing Elements (PEs) and identical I-structure storage Elements interconnected by an n-cube packet network (Figure 1). The I-structure Elements collectively form a global address space. PE's address this global address space uniformly. A single PE and a single I-structure element can constitute a complete dataflow computer. To

simplify the exposition, we will first describe the operation of the machine as if it had only one PE and I-structure module; towards the end of this section we discuss multi-processor operation.

2.1 A TTDA Processing Element

A PE has two main components: the *Control Section* and the *Compute Section* (see Figure 1). The Control Section receives special tokens that can manipulate any part of the PE's state—for example, tokens that initialize Code-Block Registers, store values into Constant Memory and Program Memory, *etc.* These tokens are typically produced by various managers such as the context manager. The Compute Section is the actual execution pipeline of the machine. Input tokens arrive at the top and are routed to one of the two main components based on their contents. Output tokens at the bottom of these sections are routed back to the top of the PE or to the I-structure Element (in a multiprocessor they may go to other PEs or I-structure elements).

The dataflow graph of a code-block is represented as a linear sequence of instructions in Program Memory. The address for each operator in the linear sequence is chosen arbitrarily, except for certain conventions for placing instructions that receive inputs to the code-block. As an engineering decision, we decree that every operator has no more than two inputs. Thus, every operator in the graph is encoded as shown in Figure 2. The *literal/constant* field may be a literal value or an offset into the constant-area. The *destinations* are merely the addresses of the successor operators in the graph. To facilitate relocation, addressing within a code-block is *relative*, *i.e.,* destination addresses are encoded as offsets from the base of the code-block.

A specific invocation of a code-block is determined by a *context*, which identifies two registers: the Code-Block Register (CBR) which points to the base address in Program Memory for the code-block's instructions, and the Data Base Register (DBR) which points to the base address in Constant Memory for the constant area. When a code-block is invoked, the caller first obtains resources for the callee from a resource manager, which allocates a CBR/DBR pair and space in Constant Memory for the callee, initializes the CBR/DBR to point to the instruction- and constant-base addresses, and returns the CBR/DBR number as the context. To execute the code-block, t okens carrying inputs are sent to the Wait-Match unit of the PE.

Wait-Match Unit: The *Wait-Match Unit* (WM) is a memory containing a pool of waiting tokens. If the entering token is destined for a monadic operator, it goes straight through WM to the Instruction-Fetch unit. Otherwise, the token is destined for a dyadic operator and the tokens in WM are examined to see if the other operand has already arrived, *i.e.,* WM contains another token with the same tag. If so, a *match* occurs: the matching token is extracted from WM, and the two tokens are passed on to the Instruction-Fetch unit. If WM does not contain the partner, then this token is left in WM. The Wait-Match unit is thus the *rendezvous* point for pairs of arguments for dyadic operators. It has the semantics of an associative memory.

Instruction-Fetch Unit: The tag on the operand tokens entering the Instruction-Fetch unit identifies an instruction to be fetched from Program Memory (*via* a CBR). The fetched instruction may include a literal or offset into the constant area, and a list of destination instruction offsets. The tag also specifies a DBR; using the constant base address found there and the constant offset in the instruction, any required constants from the constant area are now fetched from Constant Memory. The data part of the token, any constants or literals, the opcode, destination offsets, and the context itself are all passed on to the next stage (ALU and Compute-Tag unit).

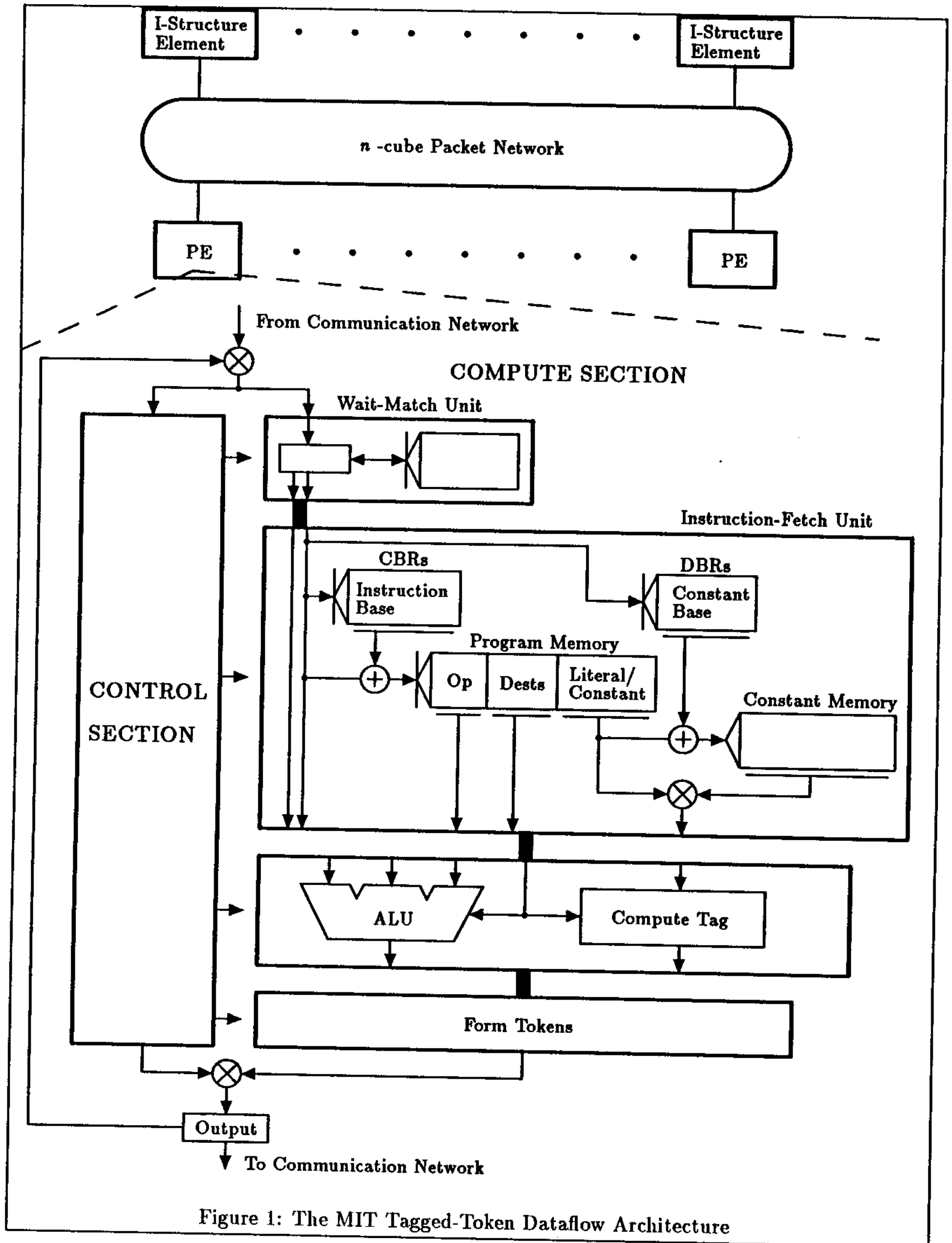


Figure 1: The MIT Tagged-Token Dataflow Architecture

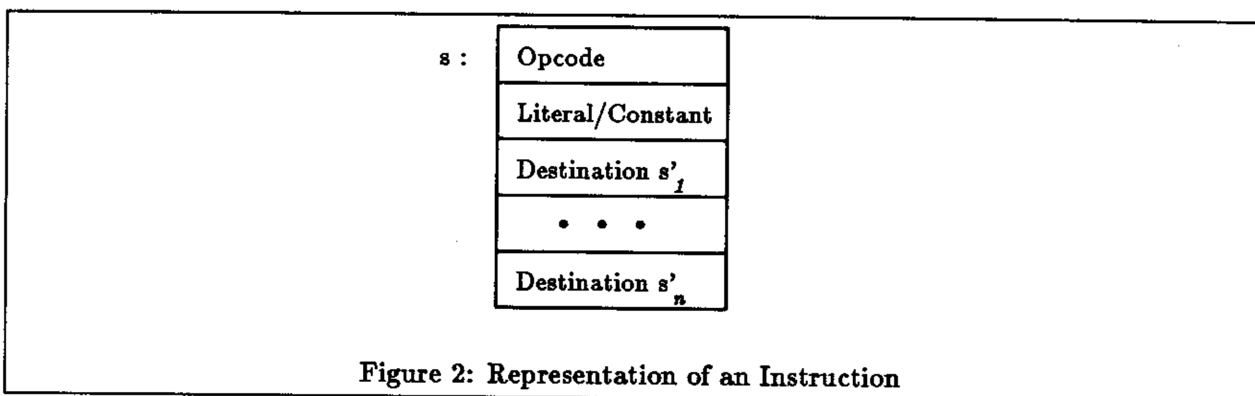


Figure 2: Representation of an Instruction

ALU and Compute-Tag Unit: The ALU and Compute-Tag units are actually two ALUs operating in parallel. The ALU unit itself is a conventional ALU that takes the operand/literal/constant data values and the opcode and produces a result. The Compute-Tag unit takes the CBR and DBR numbers from the context and the instruction offsets for the destinations, and computes the tags for the output of the operator. Recall that the tag for two instructions in the same code-block invocation will differ only in the instruction offset. The ALU result and the destination tags are passed to the Form-Tokens unit .

Form-Tokens Unit: The Form-Tokens unit takes the data value from the ALU and the tags from the Compute-Tag unit and combines them into result tokens.

The Compute Section can be viewed as two simple pipelines— the Wait-Match unit, and everything below it. Once tokens enter the lower pipeline, there is nothing further to block it. Pipeline stages do not share any state, so there are no complications such as reservation bits.

2.2 I-Structure Storage

Memory-reads in the TTDA are *split-phase* reads. In the first phase the *I-fetch* instruction sends a read-request (a token) to the memory (I-structure storage). The token contains not only a, the address to be read, but also s, the address to which the result should be sent. The memory unit sends the required value v directly to the destination instruction at s in the second phase of the read.

An I-structure Element is a memory module with a controller that handles I-structure read and write requests, as well as requests to initialize the storage. The structure of the memory is shown in Figure 3. In the Data Storage area each location has some extra *Presence Bits* that specify its state: *Present*, *Absent* or *Waiting*. When an I-structure is allocated in this area, all its locations are initialized to the Absent state.

When a read-token arrives it contains the address of the location to be read and the tag for the instruction that is waiting for the value. If the designated location's state is Present, the datum is read and sent to the waiting instruction. If the state is Absent or Waiting, the read is *deferred*, *i.e.,* the tag is *queued* at that location. The queue is simply a linked list of tags in the *Deferred Read Requests* area.

When a write-token arrives it contains the address of the location to be written and the datum to be written there. If the location's state is Absent, the value is written there and the state

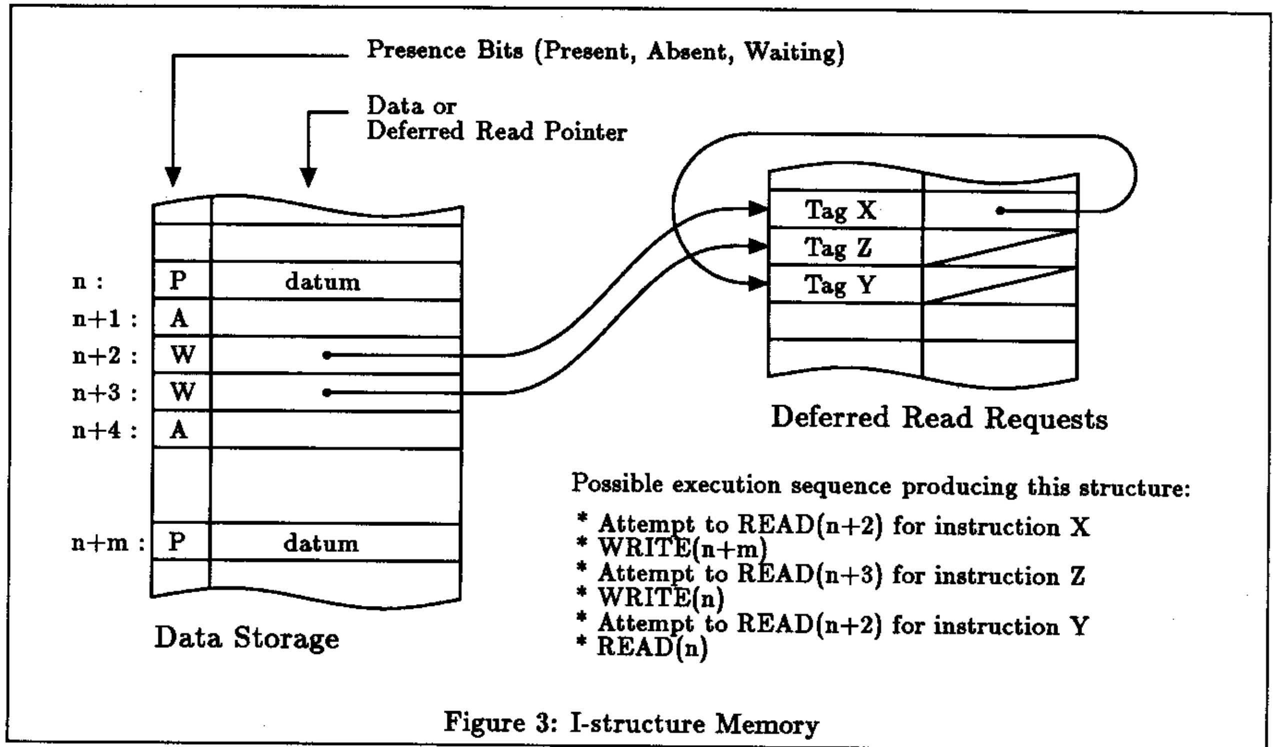


Figure 3: I-structure Memory

changed to Present. If the location's state is Waiting, the value is written there, the state changed to Present, and the value is also sent to all the destinations queued at the location. If the location's state is already Present, it is an error.

2.3 Multiprocessor Operation

Dataflow machines, unlike von Neumann multiprocessors, treat inter- and intra- processor communications identically, that is, identical token formats are used for both. A natural consequence of this fact is that no special support is required for multiprocessor operations.

The simplest form of distribution of code on the TTDA is to allocate an entire code-block to a PE. However, the TTDA has a sophisticated *mapping* mechanism that also allows a code-block to be allocated *across* a group of PEs, thereby exploiting the parallelism internal to a code-block. For example, it is possible to load a copy of a loop procedure on several PE's, and distribute different iterations of the loop on different PE's according to some pre-specified hash function on the tags. The group of PE's that cooperate in a procedure activation are called a *domain*. The hash function to be used is stored in a MAP register which, like the CBR and DBR, is loaded at the time of procedure activation. The architecture requires that the same triple of MAR, CBR and DBR, be assigned to an activation on each PE of the domain. In fact it is also possible to execute a code-block on several PE's by loading only parts of it on different PE's. When a token is produced at the output of either a PE or an I-structure Element, its tag, which is computed using the MAP information, specifies exactly which PE or I-structure Element it must go to. The token is sent there *directly* (there is no broadcasting). Since the I-structure storage address space is global, memory modules can be interleaved on any combination of address bits.

It is important to note that the program does not have to be recompiled for different mapping

schemes. The number of instructions executed in a TTDA program is *independent* of the number of PEs it is executed on[1].

2.4 Simulator for the MIT Tagged-Token Dataflow Architecture

Our most detailed "soft implementation" of the MIT Tagged-Token Dataflow Architecture is in the form of a register transfer level simulator which has been described in [3]. The simulator models TTDA as a system of PEs and I-structure storage elements, interconnected by a multi-stage (butterfly) interconnection network of size $(n \log_2 n)/2$, where n is the number of processors plus the number of storage elements. It models each PE as an interconnection of stations (such as Wait-Match unit, Instruction Fetch unit, *etc.*) separated by buffers. The simulation facility allows one to assign detailed timing parameters to each of the components in the architecure to model the latency of each execution station. However, for the purposes of clarity, we will simplify the description of the model in our presentation[2]. Timing parameters are set such that each station requires 1τ of processing time and all buffers take no time.

Given this model if, for example, we assume that $1\tau = 100nsec$ then each station in the model *at best* would be able to process at 10 million operations per second. A dyadic instruction would have a latency of 6τ because each of the stations: Input, WM, Instruction Fetch, ALU and Compute Tag (in parallel), Form Tokens, and Output would charge 1τ through the pipeline. Monadic instructions would have a latency of only 5τ because they bypass the Wait-Match station.

In the SIMPLE code, where 51% of the instructions executed are dyadic, under the timing assumptions given above we can get at most 7.5 million dataflow instructions per second since enabling each dyadic instruction requires two tokens to be processed through the WM. The extent to which we can reach the 75% ALU utilization possible in a dataflow processing element is one measure of its efficiency.

The latency in memory accesses and communication between processors is simulated by inserting k unit-delay elements at the input of each processor and storage element. The effect of this arrangement is that each access (i.e., a read operation or a write plus an acknowledgment) is actually deferred by $2k\tau$ since it involves two passes through the network. The artificially induced latency affects only network-bound tokens, which are related to I-structure access or manager request, but not the majority of the tokens which circulate, via a short-circuit, back to the input of the local processor pipeline. Processor to processor communication incurs a $k\tau$ latency for the one way trip bewteen processors.

In experimenting with program execution on a multiprocessor system one must select parameters for task granularity in allocating work to processing elements, the allocation policy for distributing tasks, and the policies for distributing and mapping data structures. The experiments throughout this paper assume that all domain sizes are uniformly set to one; this means that the granularity of task distribution is such that a code-block instance is executed entirely on one processor and iterations of a loop all unfold on the same processor. We have made rather strightforward assumptions regarding the allocation policies for tasks and data structures which will be justified later in the paper.

[1] In saying that the number of instructions executed is independent of the number of PEs we are ignoring the set-up instructions for loading program memory, CBRs, DBRs, etc.

[2] Although the true model implemented is more detailed and realistic than the one presented, the results do not change with the simplifications given.

The level of simulation detail forces the simulator to demand a remarkable amount of resources: due to the size of the available address space (12 Mbytes) of the IBM 4381/VM host and the simulation speed ($\approx$80 ops/sec), the practical upper limit for the simulated system size is around 16 processors, and the biggest SIMPLE problem we can run is about one iteration on a 10 by 10 grid, which generates just over 200,000 dataflow instructions. The intent, given these limitations, is to observe the behavior of a relatively small workload[3] on a fairly small system and to extrapolate the probable behavior of a life-size problem running on a full-scale TTDA multi-processor. The conclusions of such extrapolations will be fair as long as both the problem size, and thus the amount of parallelism, and the system size are scaled simultaneously.

3 Tolerence of Latencies and Synchronization Waits

Our goal in designing the MIT Tagged-Token Dataflow Architecture is to minimize the performance penalties (*i.e.*, induced processor idle time) associated with *latency* and *synchronization* during multiprocessor operation. Latency is defined as the time between issuing a request and receiving the associated response. Of particular interest is memory latency since in a multiprocessor system this determines the time required for execution of an instruction involving a remote operand reference. Synchronization is required when a sequencing of events between two tasks is needed to coordinate resources or to communicate data. The cost of synchronization includes the time for synchronization itself as well as the idle cycles incurred while waiting for satisfaction of the synchronization event.

The primary source of memory latency in the TTDA is I-structure operand fetches from remote structure elements. The latency incurred includes the time for a round-trip flight through the network (request and response) as well as the time taken to satisfy the request. Note that this latency can be arbitrarily long because the time to satisfy the request will, in part, depend on when the data becomes available for the requested I-structure slot. In other words, both network latency *and* synchronization between the write and read requests to an I-structure slot will affect the time it takes to satisfy a memory request for an I-structure operand. The other instance where synchronization occurs is in the Wait-Match unit where the arrival of operands is synchronized in order to enable each instruction execution. In order to tolerate these memory latencies and synchronization events we will invest a certain amount of parallelism in hiding the potential induced processor idle time by keeping the processor busy with other threads of computation.

The measure by which we can assess our success in hiding memory latency and synchronization waits is the extent to which the processor is kept busy doing useful work rather than incurring idle cycles. Synchronization, we will argue in section 3.3, has a fixed cost per instruction in our architecture and therefore does not affect the scalability of the machine as long as we have sufficient parallelism to keep the processors busy during synchronization events. Latency, on the other hand, increases as we scale the machine due to longer transit times through the network, possible memory contention, and so on. As we increase latency with the scaling of the machine we would like to still retain approximately the same level of utilization in each processing element in the system for an overall increase of throughput in the machine.

[3]The computation itself is by no means more trivial; the *small workload* only refers to the reduced number of iterations of the algorithm in order to limit the simulation time and resources needed.

3.1 Verification of Tolerence to Memory Latency

In attempt to verify the effects of memory access latency on the performance of the TTDA, we studied the behavior of SIMPLE on the TTDA simulator[10]. These studies used a fixed number of processors and allowed us to examine the impact of increasing latency upon overall performance of the multiprocessor system. By looking at speed-ups with a range of latency penalties we were able to make an assessment of the extent to which each processor was kept busy with useful work in light of longer and longer latencies. The code-block mapping policy selected was round-robin, with all domain sizes set uniformly to one. This of course means that a code-block instance was executed entirely on one processor and that iterations of a loop all unfold on the same processor. The I-structure allocation policy used the linear address space in a straight forward fashion and the address space is mapped horizontally across all the storage elements. For SIMPLE, under these policies, the ratio of network-bound to locally-circulated tokens is approximately one to four. Since the processor pipeline itself is 5τ for monadic instructions and 6τ for dyadic instructions and the ratio of monadic to dyadic instructions is approximately 1:1, we can thus determine the average latency incurred by every token to be (for large k)

$$\begin{aligned} L &= \frac{3}{4}5.5\tau + \frac{1}{4}(2k + 5.5)\tau \\ &= (5.5 + \frac{k}{2})\tau \\ &\approx \frac{k}{2}\tau \end{aligned}$$

The results of the simulation runs are presented as a set of speed-up curves in Figure 4. It is evident from the curves that the TTDA can indeed sustain an extraordinary amount of latency while still retaining much of its speed. The significance of this accomplishment must be viewed in light of the relatively small size of the run (200,000 instructions, or roughly enough work for one second of execution time on *one microcomputer*.) Based on these parameters for the simulator, a system with 16 dataflow processors only incurs a 50% loss in performance with an introduction of 200τ of delay on every network-bound token. A smaller system, which is much more realistic given the size of the run, fares even better.

As a second confirmation of this property, the same experiments were performed on a version of GITA modified to manipulate multiple queues, with each queue modeling a single processor. The code-block invocations are assigned queues according to a round-robin. It is obvious that this set of curves is very similar to that from the simulator. The slightly irregular results manifest in the intersection of the curves is due to the non-optimality of eager scheduling — sometimes by delay the firing of an enabled operator, we can actually speed up the execution of the program as a whole.

3.2 Comparison to von Neumann Computing

The speed-ups shown in Figure 4 demonstrate the ability of the TTDA to keep its processors relatively busy despite very long latencies. Even with more than a fifteen fold increase in average latency, the processors were able to keep busy enough with other threads of computation to retain 50% of their original performance. In comparison, on a single von Neumann processor, we would expect to see $3/4 + 200 \times 1/4 \approx 50$ times reduction in speed given a similar arrangement[4]. The

[4] We assume that a 25% mix of the instructions executed require a memory reference across the network, and that instructions are fetched locally in the processor.

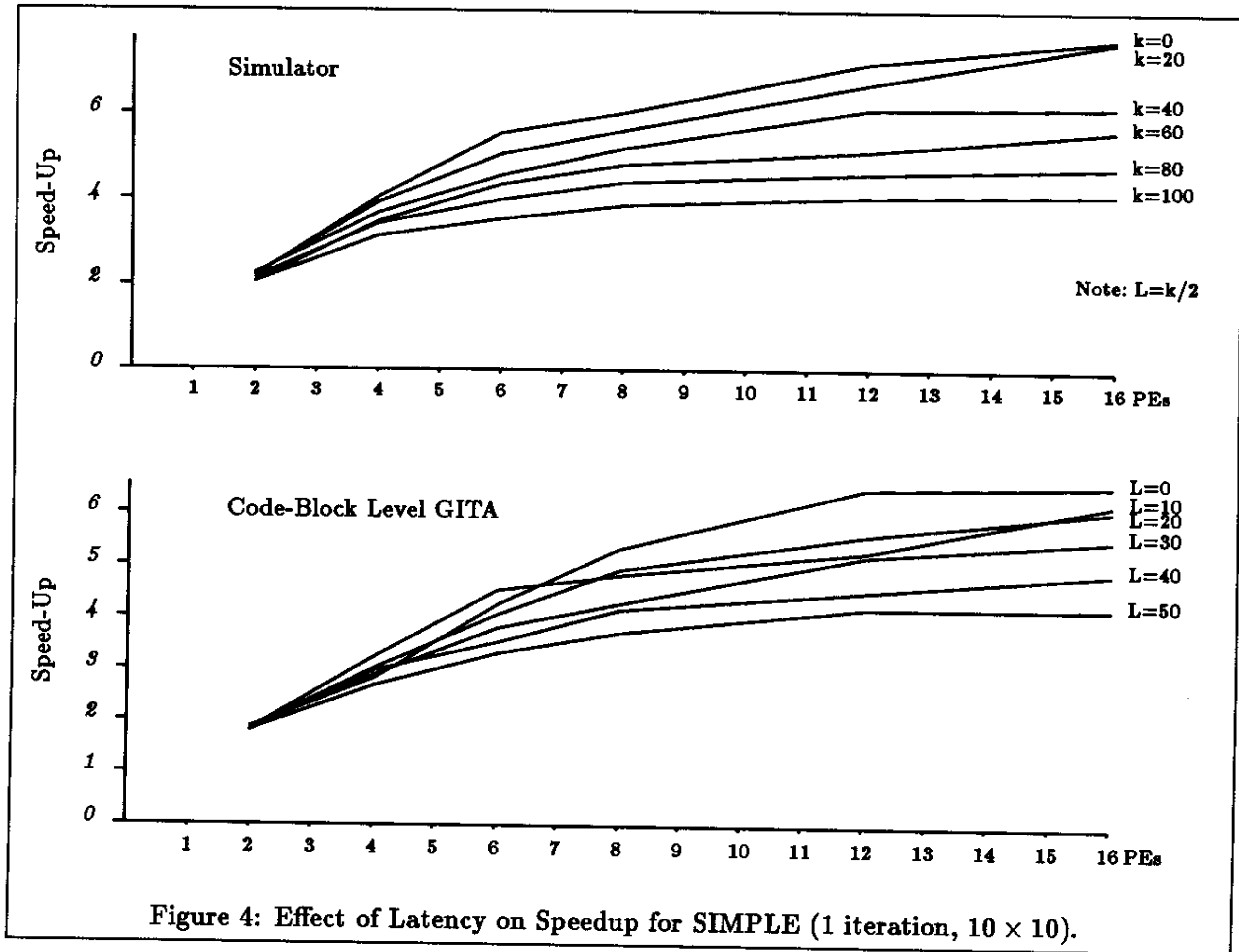


Figure 4: Effect of Latency on Speedup for SIMPLE (1 iteration, 10×10).

reason that von Neumann processors fare so poorly in the presence of memory latency is because such processors must idle during an entire memory reference. On the other hand, the dataflow instruction scheduling mechanism will immediately cause another instruction, from a parallel thread of computation, to be scheduled after issuing a memory request. Thus, the dataflow processor is busy doing useful work between the time when the request is issued and satisfied rather than idling uselessly as a von Neumann processor would. In this way dataflow processors can take full advantage of the pipelining of the network, whereas von Neumann processors can do so only to a limited extent.

Synchronization occurs in two places in the TTDA. In the Wait-Match unit we synchronize the arrival of operands that enable instructions for execution, and in the I-structure store we synchronize between writes and read requests. In both cases the dataflow instruction scheduling mechanism allows us to hide the synchronization waits in the same way that memory latency is hidden – by immediately scheduling other operations for execution rather than incurring induced processor idle time. A key point here is that we only pay a fixed cost for synchronization in the TTDA because the processor immediately "suspends" a thread of computation that requires a synchronization event. This is done by placing a token in the Wait-Match store until its partner arrives for synchronization of operand availability. In the I-structure store a read arriving before a write will merely be queued onto a deferred read list. Never is it the case that any pipeline blockage or other induced idle time is forced upon a dataflow processor; idle time will only occur if there is a lack of parallelism that can be exploited during a synchronization wait. As opposed to von Neumann processors which will generally idle while waiting for a synchronization event to occur, in the TTDA we put aside a task requiring synchronization and continue to process other threads of computation while operands or reads/writes catch up with each other. As long as there is sufficient parallelism in the program being executed there should be minimal induced processor idle time.

Note that in these simulations we are doing task distribution at the code-block level just as we would in most von Neumann multiprocessors. The major difference, then, between the von Neumann and dataflow multiprocessors is the instruction scheduling mechanism. The dataflow instruction execution mechanism allows us to interleave the execution of many code-block instances in the processor pipeline. Thus, suspending one of these threads does not have a severe impact on processor performance because it merely continues with other parallel threads of computation[5]. A von Neumann processor only schedules one thread of computation through its pipeline at a time, and requires an expensive context switch to turn its attention to a different thread[6]. Due to its ability to support concurrent pursuit of multiple comnpuational threads, and because synchonization events and memory requests do not cause pipeline blockage, the dataflow execution mechanism is inherently better suited to tolerate memory latency and synchronization events than the von Neumman model.

4 Code-Block Mapping Experiments on the Simulator

Discussions in [2] assume an execution vehicle with an infinite number of processors, so that every task invocation is allocated on a separate processor. In such a model there is never any contention for computation resources between activities of different tasks. However, in a more realistic scenario with a finite amount of computational resources we have to devise a policy for mapping a potentially

[5] These parallel threads of computation could be within the same code-block or in completely separate code-block than the suspended computation.

[6] With the notable exception of the Denelcor HEP which allows a static interleaving of computational threads [9].

unbounded set of tasks to a finite set of processors. As the invocation tree unfolds, decisions must be made as to the placement of code-block invocations across the processors of the machine. The objective is to balance the load so that all processors are effectively utilized.

To gain a better understanding of how the choice of mapping strategies impacts system performance we have conducted a sampling of code-block allocation policies for SIMPLE on the simulator for our machine. The following policies were implemented:

Round-robin: At run-time, successive procedure invocations are dispatched to processors based on a global round-robin processor pointer. This policy is fair, assuming procedure code-blocks are of approximately the same size, in the total amount of work allocated to each processor and also in the temporal distribution of work among the processors. A variation on this policy to rid ourselves of the global processor pointer would be to have each processing element implement its own round-robin with a local processor pointer.

Random: At run-time, procedure invocations are assigned to processors chosen by a pseudo-random number stream. This policy is fair only *statistically* but does not require global synchronization.

Hashing: Each procedure is allocated on every processor in the system. During run-time, a hashing function on the tag field of the tokens destined for an instruction determines the actual processor that will receive the tokens and thus execute the given instruction. This is the classic scenario for dataflow machine instruction mapping.

Load-leveling: At each procedure invocation time, the *load factor* of each processor is queried and the procedure is assigned to the processor with the lowest load factor. The load factor of a processor is determined by the amount of active code (*i.e.,* the sum of the sizes of all the active code-blocks on that processor, where a code-block is active if an invocation of it is outstanding.)

Static: The programmer specifies a semi-static assignment of procedures to processors. Presently there are four hierarchical procedure mapping catagories and three processor groups for the SIMPLE kernel:

1. Outer-level procedures and loops are allocated on group one processors on the round-robin basis.
2. Procedures and loops called by outer-level loops are distributed by the outer loops' context indices. Thus all activities spawned off by the first loop interation are dispatched to the first processor in group two, and so on.
3. Procedures and loops called by inner loops are distributed by the inner loops' context indices to group three processors.
4. Short-procedure calls are effectively "inlined" by invoking them on the same processor hosting the caller.

This partitioning of procedure groups was designed to give an equal distribution of the overall work in the program to processors in the system. Obviously, this policy maps a program well only onto a certain number of processors, depending on the problem size.

The following table shows system performance and variations in processor utilization (1 iteration, 10×10):

Policy	*Execution Rate (MOps/sec)*	*PE Utilization (%)*
Round-Robin	37.80	33 - 62
Random	37.83	30 - 65
Hashing	24.64	34 - 38
Load-Leveling	43.35	53 - 62
Static	33.56	38 - 58

The simulation results seem to show that the simple round-robin or random allocation schemes do almost as well or better than more complex schemes involving much more overhead. At first, we found it counter-inuitive that the static allocation policy with hand mapping of procedures to processing elements would do so poorly. However, further investigation showed that it was not good enough to merely balance the workload across processing elements, which in itself is a very difficult task. But also that the time domain behavior of a computation needs to be completely analyzed. We found that computation often concentrated in one group of the processors, thus leaving the other groups idle, and then moved on to other groups leaving the previously busy processor group idle. A very sophisticated analysis would be necessary to simultaneously balance workload and time domain distribution of even a straightforward computation such as the SIMPLE code. The unpredictablity of synchronization waits as well as variable latencies due to hardware contention and network congestion seem to indicate that development of static allocation policies is nearly intractable. Placing such a burden on the programmer is clearly unreasonable.

The load-leveling strategy is slightly more effective than random or round-robin, but also requires more intelligence on the part of the resource manager than the latter two choices. Implementation of this policy requires the resource manager either to probe the status of processors in the system to determine their load factor or to maintain knowledge of all outstanding task invocations. Either of these methods would force the system to do some global operations, which will be inherently non-scalable[7]. Furthermore, sensing load factor can produce undesirable results: An allocated task can actually be inactive, waiting for data from another task. It would nevertheless counted the same as an active task in our scheme. Or, if a more direct method to examine the processor token queue were implemented, an underutilized processor may be swamped with tasks before its load status starts to reflect that fact.

5 Effect of Distribution Granularity on Processor Utilization

In the previous section we saw that the policies for code-block mapping did not have a particularly significant impact upon processor utilization. The more important factor, it turns out, is the *granularity* at which tasks are distributed in the system. There is a tradeoff between the overhead in processor-to-processor communication costs and procesor utilization that we are making when a specific grain size is chosen for task distribution. When tasks granularity is small, much of the intra-processor communication that takes place within a task executing on a single processor becomes inter-processor communication between the smaller tasks executing on seperate processors. On the other hand, the smaller grain distribution of tasks has the potential for much better processor utilization because it is easier to evenly spread out the available program parallelism across processors.

[7]Note that SIGMA-1 has addressed this scalability problem by implementing a scheme in which distributed load-leveling is performed through the commuincations network [7].

The graphs in Figure ?? show the effects of grain size on overall performance of a multiprocessor system for a range of system sizes executing SIMPLE. **** Explain parameterizations of experiment **** We notice that for small numbers of processors, the code-block level distribution performs better than the smaller grained iteration level distribution. This is due to the fact that there is already plenty of parallelism at the code-block level to keep each of the processors busy. Thus, the smaller grain tasks only detract from system performance by imposing more inter-processor communication. However, as we increase the number of processors past ?? the iteration level distribution begins to out perform code-block distribution. This behavior is due to the fact that as the problem size stays the same and the number of processors increase, it is necessary to use smaller grain tasks to exploit the program parallelism and evenly distribute tasks to keep all processors busy.

6 Distribution and Mapping of I-Structures

The distribution and mapping of data structures in a multiprocessor architecture can have a significant impact upon performance in the machine. There is a full range of tradeoffs that can be made between exploiting locality in code-block to structure references and balancing workload across the structure storage elements. In order to assess the likely behavior of the various possibilities for distribution and mapping of structures we have studied the I-structure referencing patterns during execution of the SIMPLE [4].

The outcome of these studies is that implementation of clever distribution and mapping policies is likely to be very difficult. This difficulty is due to a skew in the structure referencing patterns as well as the large number of contexts that use these most heavily referenced structures. A study of the reference patterns for a range of problem sizes[8] for the SIMPLE code was performed. The table below shows a histogram of the structures according to the percentage of memory traffic received.

	Structure Histogram for SIMPLE		
% of Refs	10×10	20×20	40×40
0	310	632	1272
1	3	3	3
2	0	0	0
3	0	0	3
4	4	3	0
5	0	1	1
6	0	2	0
7	2	0	2
8	1	0	0
9	0	1	1
Total	320	642	1282

From the data it is apparent that there are ten structures that account for a disproportionately large amount of the overall traffic. It turns out that the three most heavily referenced structures are constant tables. These three most heavily referenced could therefore be replicated across structure storage elements to get around the contention problem. However, there still remain seven structures that each account for more than 2% of the structure memory requests. Moreover, the contention

[8] One iteration of the hydrodynamics algorithm over various size meshes.

problem is likely to be accentuated during program execution due to statistical fluctuations and because the lifetime of a data structure is less than the program execution time.

The studies also show that a fairly large number of I-structures are referenced by multiple code-block instances (see the table below). This behavior constrains the extent to which we can successfully take advantage of locality by mapping code-blocks which use a particular structure to a processing element that is "close" to structure element that stores it. Since there are many I-structures used by more than ten code-block instances it would severely limit the resource manager's ability to perform load-balancing if we tried to take advantage of structure locality in code-block mapping.

	Structure Histogram for SIMPLE		
# of Contexts	10×10	20×20	40×40
$1 - 10$	249	412	898
$10 - 100$	62	217	371
$101 - 11556$	9	13	13
Total	320	642	1282

Moreover, there are a small number of structures that are used by an extremely large number of contexts. The table below shows the fourteen "most popular" structures for SIMPLE where we define the "popularity" of a structure as the number of code-block instances which reference it. More importantly, however, is that these structures that are used by many code-block instances are also among the set of structures which account for the majority of traffic to the structure storage elements. Not surprisingly, the thirteen structures referenced by the largest number of code-block instances are also among the 50 structures which account for well over 50% of the structure element traffic. The most heavily referenced structures would be ideal candidates for localization in order to reduce memory traffic. However, since these also tend to be the structures used by many contexts this seems unlikely to yield beneficial performance because any attempts to enhance locality will interact strongly with the work distribution policies.

	Popularity		
Rank	10×10	20×20	40×40
1	516	2596	11556
2	323	1623	7223
3	323	1623	7223
4	194	974	4334
5	194	974	4334
6	131	651	2891
7	131	651	2891
8	130	650	2890
9	130	650	2890
10	91	381	1561
11	91	381	1561
12	68	328	1448
13	68	328	1448
14	24	44	84

The large percentage of traffic attracted by some structures coupled with the large variance in the amount of traffic to each structure seems to rule out any random vertical schemes which

completely localize individual structures. We would expect that some structure storage elements would be overloaded, and others greatly underutilized. Diagonal allocation schemes, in which "chunks" of each structure are localized, offer a reasonable compromise between localization and minimizing contention. They offer perhaps the most promising approach in the long run. However, the interconnection of the network in our multiprocessor system is likely to be very dense. Hence the most we could hope to gain with the diagonal approach is a logarithmic factor in latency. Given the TTDA's tolerence to memory latency, this is not likely to be a significant performance factor in the machine. Horizontal allocation appears to be the safest scheme. It distributes the load of heavily referenced structures over many structure storage elements, and thus increases the potential memory bandwidth of each structure. In some ways this scheme feels unsatisfactory because it relegates the memory latency seen by a processor to mediocrity. However, it is a fundamental characteristic of the TTDA that we are able to incur such mediocrity in memory latency and still provide high performance. Thus, this scheme seems to be taking advantage of the TTDA's greatest strength.

7 An Efficient Instruction Scheduling and Token Storage Unit

The Wait-Match unit is a critical component of a dataflow processing element. This unit in the processing element architecture has two roles fundamental to a dynamic implementation of data driven computation in the machine: to store intermediate results, and scheduling of instructions once their operands become available. Unfortunately, these two responsibilities introduce conflicting design requirements for the Wait-Match hardware. To support storage of the large number of intermediate results necessary during pursuit of many concurrent threads of computation, a large token store is required. On the other hand, a relatively small content-addressible store is desirable to facilitate efficient tag matching and subsequent scheduling of instructions. In this section we will discuss alternatives for organization of the Wait-Match store as well as opportunities for caching tokens based on their high volatility in the WM.

7.1 Organization of the Wait-Match Unit

Current implementations of the tagged-token dataflow approach in the Manchester and Sigma-1 machines have made use of hashing memories for the associative store necessary to perform the tag-matching operation in the Wait-Match unit. Because of the large number of tokens that need to be stored in the Wait-Match unit this is the only feasible implementation of an associative store based on tag-matching. The problem with this approach is that due to collisions accesses can take several cycles; thus resulting in degraded performance of the machine.

However, another alternative exists for organization of the Wait-Match unit. Rather than using a self-managed store implemented as a hash-type of associative memory, we can use an *explicitly* managed store in which individual frames, one for each code-block activation, are allocated to store waiting tokens. A major attraction of this approach is that tokens can be *directly addressed* based on their tags in such an approach. This suggests a very simple implementation with very fast access. The only problem is that with uncontrolled program unfolding, only a few of the locations of a frame hold active tokens. Thus, much space is wasted due to fragmentation, and the total memory required is too large to be feasible.

Loop bounding techniques suggest a way out. By controlling the unfolding of the program, we can limit the number of active frames to a reasonable amount, and decrease fragmentation because

activities are concentrated in fewer frames. Thus we can implement the Wait-Match store as a fast, directly-addressed memory. Each time a code block is invoked, a frame is allocated containing storage for all the "matching events" that take place in that code block. The tag (or context) on each token is now simply a frame pointer. Thus, given a token, we can use the destination instruction and the frame pointer to compute the exact address where it will rendezvous with its partner. Because two matching tokens may arrive in any order, the frame location where they rendezvous needs "presence bits" indicating whether it is empty or full. The detailed elaboration of these ideas is given in [11].

This directly addressed implementation of the Wait-Match store now has remarkable similarities with I-structure storage. Both have presence bits, and in both cases tokens arrive at a location and wait there for another token (in the I-structure case, the first is a read-token that gets deferred, the second is a write-token). These similarities are not surprising— both Wait-Match and I-structure stores implement *synchronization* in the TTDA. These similarities suggest a way to unify the two stores, so that a single mechanism can be used for both. The details of such a unification are beyond the scope of this paper and are given in [11].

7.2 Token Volatility in the Wait-Match Unit

As was mentioned, it would be extremely valuable to implement a small content-addressible store as an efficient mechanism for matching tokens in the Wait-Match unit. Yet the overall storage requirements of the WM make this an unrealistic approach for storing all tokens entering the unit. Fortunately, however, we have observed a substantial amount of volatility in the majority of tokens passing through the Wait-Match unit. As was pointed out in [5], token storage in the Wait-Match unit of a tagged-token dataflow machine is analagous to the activation frames in a von Neumann architecture. In sequential machines, even though the total size of the stack may be very large, most of the activity (reads and writes) is concentrated in a small region, usually in the currently *active* stack frame. This temporal locality makes it possible to improve performance significantly by caching data in high-speed memory. Similar techniques are applicable in the design of a dataflow machine, although the structure of the "activation frames" is slightly more complex because it corresponds to a *tree* of procedure invocations rather than a simple stack.

Thus, the key to realizing an effective caching policy is to recognize that there are two classes of tokens stored in the Wait-Match unit. The first class of tokens are those that correspond to data values in use by some *active* thread of computation. In a conventional von Neumann computing environment these tokens would be stored as data values in the general purpose registers of the machine. The second class of tokens to be distinguished are those that maintain context for suspended procedures. This token class is analogous to entries in old activation frames on a von Neumann machine designed to support block-structured programming languages.

We can estimate the volatility of tokens in the TTDA by studying their lifetimes— the time they spend in the Wait-Match store. Tokens destined for monadic operators bypass the Wait-Match store completely and have zero lifetimes. Our experiments show that typically 50-60% of the dynamic instruction mix for scientific codes is monadic. For each dyadic operator, only the first operand spends any time in the Wait-Match store.

The following tables are for two runs of 3 iterations of a 20×20 SIMPLE on our abstract graph interpreter in idealized mode. One run is executed with no loop bounds, and one with the outer loop bounded with a loop bound of one. The table below gives the histogram for the fify percent

most volatile tokens. The results show that most of the tokens are very volatile, so that caching the Wait-Match store is likely to be effective in improving performance.

	$k = 1$		$k = \infty$	
Lifetime	Number	%	Number	%
0	71,103	9	70,829	9
1	159,436	20	158,164	20
2	75,199	9	74,031	9
3	60,360	7	60,292	7
4	19,485	2	19,215	2
5	31,849	4	31,707	4
> 5	396,168	49	399,362	49
Total	813,600	100%	813,600	100%

Although these results are very encouraging, it is important to note that they are overly optimistic. In the idealized execution mode of GITA we assume an infinite amount of computational resources. Therefore, there is no contention among computations for such resources as the Wait-Match unit, Instruction Fetch unit, etc. In a realistic machine implementation there would be many computational threads interleaved in the pipeline of a processor, and these threads would necessarily compete for hardware resources. Unlike the idealized model, only a finite number of tokens would be able to find a match in a given time step. Thus, we would expect the average lifetime of tokens in the Wait-Match store to increase due to the finite amount of hardware resources in a real machine. Moreover, in associative Wait-Match implementations using a hash memory scheme such as in the Manchester and Sigma-1 machines we would expect bounded loops to have a *positive* effect in increasing token volatility because fewer collisions would occur. By bounding program loops we decrease the token flooding effects of unbounded parallelism upon the Wait-Match store as demonstrated in [5]. The result is a smaller pool of tokens and a more efficient hashing performance when performing a matching operation.

To quantify the effects of finite resources and resource management policies on token volatility we have studied the behavior of SIMPLE on the TTDA simulator [3]. In these simulations, the Wait-Match unit is modeled as a hash memory (open hash with chaining) with a cost of 0.5τ for the initial hash, 1.5τ for each collision, and then an additional 2τ for the store or fetch of a token. Inserting and removing packets from the buffers between pipeline stages costs 1τ. Typical ALU costs are 1.5τ for integer and boolean operations, and 5τ for floating-point add class instructions. Instruction/constant/destination list fetches cost 1.5τ for the initial access plus 0.5τ for each word retrieved. The Input and Output stations, as well as buffers between stations, each cost 0.5τ. A round-robin distribution of code-blocks and I-structures is used. The detailed timing model and the finite resource limitations put forth give a much more realistic execution scenario than the idealized mode of GITA.

One run of the 10 x 10 SIMPLE is executed with no loop bounds, and the other with a loop bound of five that applies to *all* loop iterations in the SIMPLE program. The table below lists the histogram for token lifetimes (units of τ) in the Wait-Match unit.

	$k = 5$		$k = \infty$	
Lifetime	Number	%	Number	%
$1 - 100$	34,181	33	20,818	20
$101 - 200$	18,443	18	6,759	6
$201 - 300$	9,255	9	6,196	6
$301 - 400$	6,366	6	6,671	6
$401 - 500$	4,400	4	4,400	4
> 500	31,486	30	59,287	58
Total	104,131	100%	104,131	100%

The results show that although the lifetime of tokens has increased due to the finite resources available for token matching, there is still a substantial skew in the lifetimes of tokens passing through the Wait-Match unit. We see that the majority of tokens have a volatility of a few hundred pipeline beats. Thus, caching is a feasible mechanism for improving performance in the Wait-Match unit. Notice that bounded loop unfolding does indeed have a significant effect on token volatility in a realistic machine. By preventing the Wait-Match unit from being flooded with tokens that do not contribute to useful parallelism (see [5]) the number of collisions in the hash memory is reduced and overall performance of the machine as well as token volatility is improved.

8 The Cost of Executing Dataflow Instructions

Empirical data presented in [1] demonstrates the relative equality of dataflow MIPs to von Neumann style MIPs. However, the analysis presented did not address the comparative complexity of the hardware machinery required to support execution in the two models. The von neumann framework provides a simple, elegant model for the execution of a sequential instruction stream. Supporting the dynamic scheduling and pipeline interleaving of instructions in the MIT Tagged-Token Dataflow Architecture necessitates more complex execution mechanisms than is required for von Neumann computing. Some of this complexity, we will argue, is inherent in any machine that supports multiple, active code-block contexts for maximal parallelism. Other aspects of the complexity of our machine is specific to our implementation of a *data-driven* computing model. In this section we will attempt to characterize the complexity of dataflow computation in terms of the mechanisms required for instruction execution.

In a straightforward implementation of the von Neumann machinery the basic execution cycle involves an instruction fetch, decode, operand fetch, execution, and finally an update of the program counter, etc. in preparation for the next instruction to be executed. In the data-driven execution model of the TTDA we start off the execution cycle with an operand fetch. This operand fetch may or may not be successful depending on whether both operands are yet available. The arrival of one operand will cause the second operand for a dyadic instruction to be fetched, if present, or will result in storage of the first arriving operand otherwise. The Wait-Match unit provides the hardware mechanism to store or fetch an operand, as appropriate. The fetching of operands based on an associative matching of tags is fundamentally more complex than a simple operand fetch in a von Neumann computer. However, the temporal locality of operands passing through the Wait-Match unit, as described in Section 3 indicate that a hierarchical storage facility with token caching could be an effective way of building an efficient Wait-Match unit. Such an implementation, except for the synchronization of operand arrivals, is equivalent in complexity to a data cache in a von Neumann computer. Moreover, the resource-bounded program graphs presented in [5] allow

a Wait-Match unit implementation with explicitly managed token storage based on the allocation of activation frames for code-block invocations. The advantage of this is that a directly addressed token store can be implemented for faster access than an associative hash-type of memory.

Part of the complexity of the Wait-Match unit is that when a token arrives we do not know a priori whether we will be storing the operand or fetching its partner. This is due to the implicit synchronization that is being performed in the Wait-Match unit. Since the dataflow model supports asynchronous, parallel execution of operators it is necessary to synchronize the arrival of operands in a facility such as the the Wait-Match unit. There is certainly an extra cost in performing this synchronization above and beyond that of a simple operand fetch in the von Neumann model. However, it is clear that in a parallel execution model, there must be some form of synchronization among parallel activities. Our model incurs this cost directly in terms of the hardware complexity of the instruction scheduling unit[9]. However, it is important to recall that the characteristics of the dataflow instruction scheduling mechanism make this synchronization *non-blocking*. In the Tagged-Token Dataflow Architecture an operand will step aside into the Wait-Match store and let other computations proceed while it waits for synchronization with its partner, as opposed to most von Neumann machines in which a synchronization event will cause a processor to idle until the event is satisfied.

The instruction fetch in a dataflow execution engine is slightly more complex than that of a von Neumann computer because there are multiple threads of computation that are concurrently active in a single processing element. This requires us to maintain multiple contexts in the machine via a set of CBRs and DBRs for pointing to the code-block instructions and constant (data) area, respectively, for each active code-block. In a traditional von Neumann machine one would require only a single program counter (PC) and a single frame-base register (FBR) for pointing to the currently active instruction and stack frame, respectively. The existence of multiple contexts in a dataflow processing element introduces added hardware cost for additional registers as well as complexity in resource management for managing the finite set of context registers. Note, however, that the cost over that of the von Neumann model is explicitly due to the fact that we are maintaining multiple contexts that can execute in parallel, and *not* because dataflow contexts are inherently more complex than von Neumann contexts (of which there is only one per processor).

The decode and execute portions of an intstruction execution will essentially be equivalent for the two models of computation. However, in preparing for subsequent instructions to be executed we have some subtle differences. In a dataflow implementation a destination list fetch mechanism is used which is similar in concept to the update of a program counter in the von Neumann model, but has some additional cost due to the fact that many instructions can be enabled via the firing of a single dataflow operator. In updating the program counter in a von Neumann machine we merely increment it to point to the next instruction in the sequential thread of computation, or construct a completely new value by using the target of a branch instruction. In the Compute-Tag unit of the TTDA we construct a tag for each destination of a result from the CBR/DBR numbers and destination list specification of the instruction that produced it. Essentially, the work that is being done here is to calculate the next instructions to be executed along the parallel threads of computation emanating from an instruction. In a von Neumann computer we have only a single "next instruction" to calculate in a sequentially stored stream of instructions, and therefore the hardware is simpler because an instruction is only an increment away from its predecessor (in most cases). In dataflow instructions we specify subsequent instructions to be executed by building new instruction offsets into the otherwise identical tag of an operand whenever the next instruction(s)

[9]Similar observations hold true for the implementation of I-structure storage.

are within the same codeblock invocation[10]; these instruction offsets obtained from the destination list of the instruction we are executing. There may be many destinations for the result of a particular instruction. By using a fan-out graph to distribute results we can restrict the number of destinations per instruction to some small constant, however this approach requires the execution of additional *Identity* instructions to implement the fan-out (see [1]).

9 Conclusions

Through empirical studies we have shown the TTDA to be capable of incurring large latencies while still maintaining acceptable levels of performance. The dataflow instruction scheduling mechanism allows the machine to suspend a thread of computation while it awaits completion of a memory request and work on other tasks. Synchronization waits are tolerated in the same way by suspending a task requiring synchronization until it is ready to resume. Direct execution of dataflow graphs provides the implicit synchronization between operands destined for single operator in the Wait-Match unit, and allows us to interleave parallel threads of computation in the pipeline for efficient processor utilization. Moreover, we have a compiler that produces highly parallel dataflow graphs from the high-level functional language ID [12]. Straighforward code-block and I-structure mapping policies appear to be acceptable in the TTDA, although we continue to investigate tradeoffs between the available strategies.

Architecturally, there is still room for improvement in the TTDA. We are investigating ways to improve performance of the Wait-Match unit through use of caching and directly-addressed store. Unification of the Wait-Match and I-structure stores will provide further simplification of the hardware. The overhead associated with multiple contexts seems unavoidable if we want to effectively exploit parallelism in the program by interleaving the execution of different code-block instances in the pipeline. The variability in destination list lengths is also a complication associated with effectively exploiting program parallelism; this overhead relates to pursuing multiple threads internal to a code-block instance. It is the presence of this parallelism and the ability of the TTDA to exploit it during program execution that allows tolerence to latencies and synchronization waits.

References

[1] Arvind, David Ethan Culler, and Kattamuri Ekanadham. Parallelism: what is the cost of going all the way? In *submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, North-Holland Publishing Company, May 30-June 2 1988.

[2] Arvind, David Ethan Culler, and Gino K. Maa. Parallelism in dataflow programs. In *submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii*, North-Holland Publishing Company, May 30-June 2 1988.

[3] S.A. Brobst. *Instruction Scheduling and Token Storage Requirements in a Dataflow Supercomputer.* Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

[10]There is also a case in which a new tag is constructed by incrementing an iteration identifier when proceeding from one loop iteration to the next.

[4] Andrew A. Chien. *Structure Referencing in the Tagged Token Dataflow Architecture.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, October 1986.

[5] David E. Culler and Arvind. Resource requirements of dataflow programs. In *submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii,* North-Holland Publishing Company, May 30-June 2 1988.

[6] John R. Gurd, C.C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the Association for Computing Machinery,* 28(1):34–52, January 1985.

[7] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. *Load Scheduling Schemes Using Inter-PE Network.* Technical Report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[8] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. *System Architecture of a Dataflow Supercomputer.* Technical Report, Computer Systems Division, Electrotechnical Laboratory, 1-1-4 Umezono, Sakura-mura, Niihari-gun, Ibaraki, 305, Japan, 1987.

[9] Harry F. Jordan. Performance measurement on hep - a pipelined mimd computer. In *Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden,* North-Holland Publishing Company, June 1983.

[10] Gino Maa. *Code-Mapping Policies for the TTDA.* Technical Report, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987 (expected). (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).

[11] Gregory M. Papadopoulos88. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, June 1988 (expected).

[12] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture.* Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).