LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Explicit Token Store

Computation Structures Group Memo 312
June 8, 1990

David E. Culler
Computer Science Division
University of California, Berkeley
Berkeley, CA

Gregory M. Papadopoulos
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA

To appear in the *Journal of Parallel and Distributed Computing*,
January 1991

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988. David Culler was supported in part by an NSF Presidential Young Investigator award and Motorola Inc.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The Explicit Token Store

David E. Culler
Computer Science Division
University of California, Berkeley

Gregory M. Papadopoulos
Laboratory for Computer Science
Masachusetts Institute of Technology

Abstract: This paper presents an unusually simple approach to dynamic dataflow execution, called the *explicit token store* architecture, and its current realization in *Monsoon*. The essence of dynamic dataflow execution is captured by a simple transition on state bits associated with storage local to a processor. Low-level storage management is performed by the compiler in assigning nodes to slots in an *activation frame*, rather than dynamically in hardware. The processor is simple, highly pipelined, and quite general. There is exactly one instruction executed for each action on the dataflow graph. Thus, the machine oriented ETS model provides new insight toward the real cost of direct execution of dataflow graphs.

1 Introduction

The *Explicit Token Store* (ETS) is a simplified model of dynamic dataflow execution, centered on the notion of an activation frame which provides explicit storage for operands and "presence bits" that guide instruction processing. It eliminates the associative matching and implicit storage allocation present in tagged-token dataflow models, relying on the compiler to determine the use of slots within a frame and to install frame allocation and deallocation operations in the dataflow graph. This report describes the ETS model and its realization in *Monsoon*, a large-scale dataflow multiprocessor[39]. A single-processor Monsoon prototype is operational at the MIT Laboratory for Computer Science, running large programs compiled from the dataflow language Id[37]. A full-scale multiprocessor system is under development in conjunction with Motorola Inc.[5] and should become operational during 1991. Formulation of the ETS began in 1986 as an outgrowth of work on the MIT Tagged-Token Dataflow Architecture (TTDA). We believe that it preserves the most valuable aspects of the TTDA, yet overcomes the most serious limitations of that design. The ETS is simpler, more powerful, and easily understood in its own terms.

The philosophic premise of the ETS/Monsoon effort is that progammability must accompany performance if highly parallel machines are to be useful where they are most needed — in solving problems significantly more complex than those addressed on current supercomputers. Applications possessing a static, uniform structure have demonstrated reasonable performance on current multiprocessors, but only if careful attention is paid to the detailed mapping of program activities and data onto the machine[12, 19, 27, 32]. Much like the use of overlays prior to virtual memory, this effectively lowers the level of programming[34] and introduces obscure and error-prone program constructs[35]. Moreover, many realistic applications exhibit neither static nor simple structure[36]. As long as synchronization and communication is expensive, it is unlikely that large amounts of useful parallelism can be exposed automatically in complex programs. On the other hand, ample parallelism is easily exposed in compiling high-level Id programs into dataflow graphs[6, 8, 41]. Realizing this potential parallelism requires that the underlying machine provide both a very large synchronization namespace and cheap synchronization. The ETS addresses this dual challenge.

1.1 Dynamic dataflow influence

Dynamic dataflow execution is characterized by three properties: the program representation is a partial order, *i.e.*, a graph, of essential dependences, instructions are scheduled based on availability

of operands, and iteration and recursion are supported in full generality. The latter property distinguishes the model from earlier static dataflow models[20, 22] and gives rise to the need for dynamic management of storage and "synchronization names." Dynamic dataflow execution is formalized as rules for propagating tagged data tokens through graphs[7]. A node fires when tokens with identical tags are present on the input arcs, producing result tokens on the output arcs. Tagged-token dataflow architectures, represented by machines developed at MIT[2], Manchester University[26], and the Electrotechnical Laboratory[29, 43], approximate this model quite closely. Data is conveyed within the machine as packets, *i.e.*, tokens, containing a fixed sized value and a tag, roughly equal in size to a value. In these machines, the dataflow firing rule is realized by a sophisticated hardware *matching-store*, essentially a large associative memory. When a token arrives at a processor, the tag it carries is checked against the tags on tokens in the matching-store. If no match is found, the incoming token is added to the store. If one is found, the matching token is extracted and the corresponding instruction is enabled for execution, eventually producing new tagged tokens. The tag serves as a name for synchronization point of two values destinated for the same instance of the same instruction. Each loop iteration or function invocation must be provided with a new set of synchronization names.

The tagged-token instruction scheduling paradigm supports a non-blocking processor pipeline that can overlap instructions from closely related or completely unrelated computations. Thus, parallelism can be exploited at all levels. It also provides a graceful means of integrating asynchronous and potentially out-of-order memory responses and synchronization events into the normal flow of execution, allowing communication latency to be masked by excess parallelism[9]. However, the matching operation places considerable complexity on the critical path of instruction processing. Thus, one goal of the ETS was to achieve the benefits of matching with a fundamentally simpler mechanism.

The classical associative matching paradigm would seem to be attractive from a resource management viewpoint as well. Work can be assigned to processors by simply applying a hash function consistently to the tag, in order to approximate a uniform, fine-grained distribution of activity. Storage for tokens is provided *as needed* by the matching store; failure to find a match implicitly allocates storage for an operand. However, this presents a subtle and very serious problem. In assigning a portion of the computation to a processor, an unspecified commitment is placed on the matching-store of that processor; when this resource becomes overcommitted the program may deadlock[1]. If the match is to be performed rapidly, we cannot assume this resource is so plentiful that it can be wasted.

One means of avoiding this problem is to "precompute" the worst-case token storage requirement of portions of the dataflow graph at compile-time and reserve sufficient matching-store resources at run-time[1]. Typically, a program is represented by a collection of disjoint graphs, called code-blocks, that correspond to individual loop or function bodies in the high-level program. Code-block invocations serve as a convenient unit of work distribution and of resource allocation. When a code-block is invoked, the task of perfoming the invocation is assigned to a processor, or a small subset of processors. The run-time system can record the load on each matching-store, using the worst-case estimates derived by the compiler, and avoid assigning work to an over-committed processor. The next logical step is to allow the compiler to completely manage the storage used within a code-block invocation and eliminate the implicit allocation in hardware. Thus, both engineering and management concerns led us to consider how to make token storage *explicit* in the dataflow model.

A third factor influencing the development of ETS was the I-structure concept[11]. An I-structure is an array of data slots with presence bits associated with each slot to indicate whether it is full or empty. Slots are permitted to be written only once and any reads of an empty slot are deferred until the corresponding write occurs. This provides very fine-grained synchronization

between the producers and consumers of a data structure and eliminates the overhead of functional arrays[20], while retaining determinacy. The ETS extends the use of presence bits to all forms of synchronization.

Finally, we believed it important that the execution model be general enough to allow construction of a stand-alone parallel machine. It should not require additional conventional processors to support resource management and operating system services. The current Monsoon implementation is a compromise: low-level run-time system services are supported *in situ*, demonstrating the generality of the ETS approach, yet, for expedience, a host processor provides file system services and the like.

1.2 ETS dataflow execution

The ETS approach shifts much of the low-level storage management burden associated with dataflow execution to the compiler in order to simplify the hardware and the run-time system. We only consider well-behaved dataflow graphs produced from a high-level language, such as Id. A program is comprised of a collection of code-blocks, *e.g.,* funciton and loop bodies, as mentioned above. When a code-block is invoked, the caller dynamically allocates an *activation frame*, thereby providing local storage for the activation. This bears obvious similarity to the use of stack frames in supporting imperative languages on conventional machines. The arcs in the code-block, (*i.e.*, the local variables for the function) are statically mapped onto slots in the frame by coloring the graph[14]. Each instruction specifies the location of its operands, as a simple effective address calculation, so no matching is required.

The basic structure of an executing program is illustrated in the lower portion of Figure 1. The sample code computes the expression `Z[i] = a*X[i] + Y[i]`. A *token* comprises a value, a pointer to the instruction to execute (IP), and a pointer to an activation frame (FP). The latter two form the *tag*. The instruction fetched from location IP specifies an opcode (*e.g.*, `ADD`), the offset r in the activation frame where the match will take place (*e.g.*, FP + 4), and one or more destination instructions, encoded as IP-relative diplacements, that will receive the result of the operation (*e.g.*, instruction IP + 2). An input port (left/right) is specified with each destination.

The unusual quality of ETS activation frames, is that each frame slot has associated *presence bits* specifying the disposition of the slot. The dynamic dataflow *firing rule* is realized by a simple state transition on these presence bits, as illustrated in Figure 2. At time t, the first token arrives and is processed. The slot is found empty, so the value on the token is deposited in the slot (making it full) and no further processing of the instruction takes place. At time $t + n, n \geq 1$, the second token arrives, the slot is found to be full, so the value is extracted (leaving the slot empty) and the corresponding instruction is executed, producing one or more new tokens. Observe, each token causes an instruction to be initiated, but when an operand is missing the instruction degenerates to a store of the one available operand. In general, the order of arrival of tokens is indeterminate, so the first token may be for either port. Initially, all slots in a frame are empty and upon completion of the activation they will have returned to that state. The graphs generated by the compiler include an explicit release of the activation frame, usually by the caller, upon completion of the invocation.

An executing program generates a *tree* of activation frames, rather than a stack, because a procedure may generate parallel calls where the parent and children execute concurrently. The concurrent children may themselves generate parallel calls, and so on. For loops, several frames are allocated, so that many iterations can execute concurrently[16].

Other matching rules are possible. In Figure 1, the slots marked with `c` indicate a constant operand which is *not* implicitly reset to empty after the match takes place. These constants play

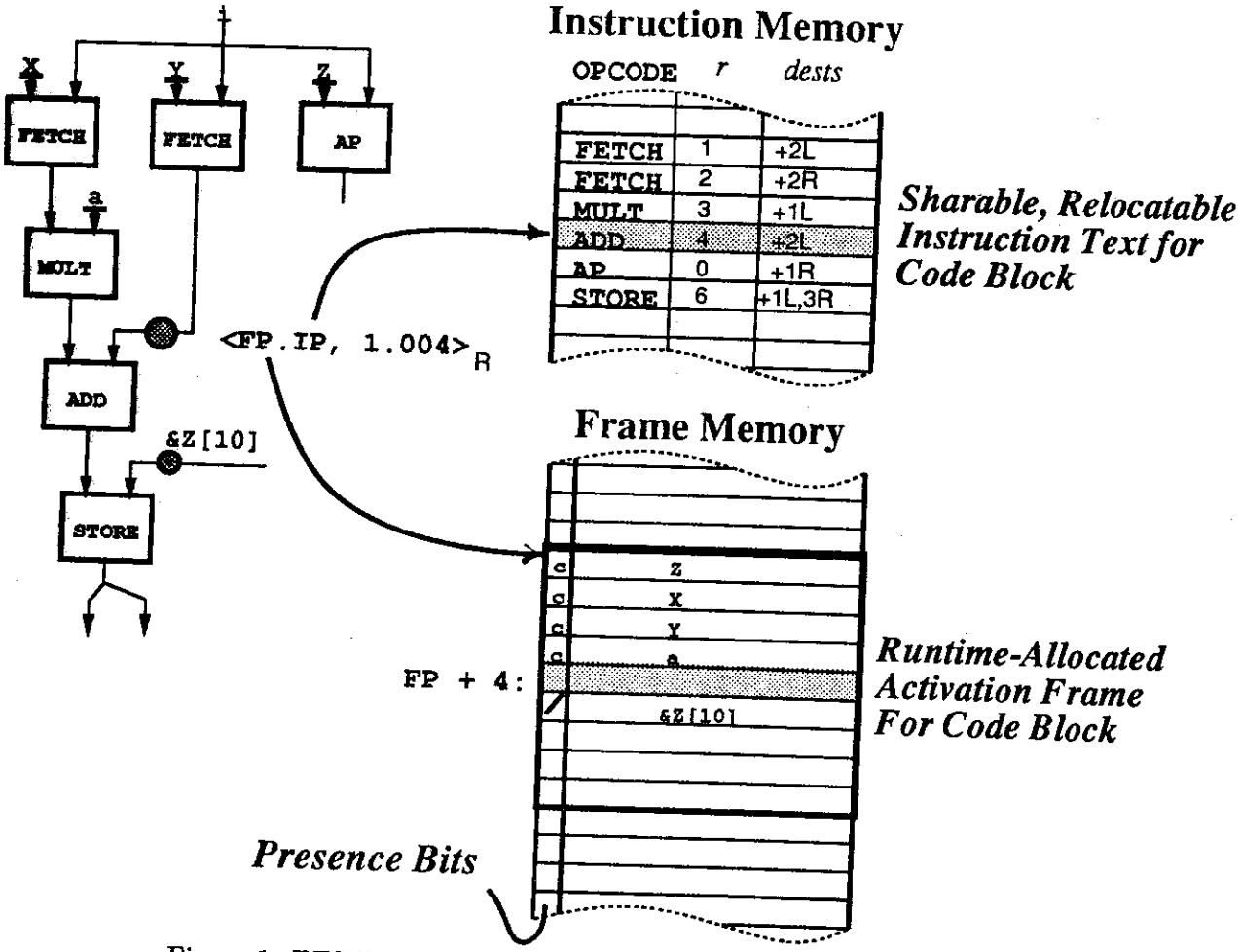

Figure 1: ETS Representation of an Executing Dataflow Program

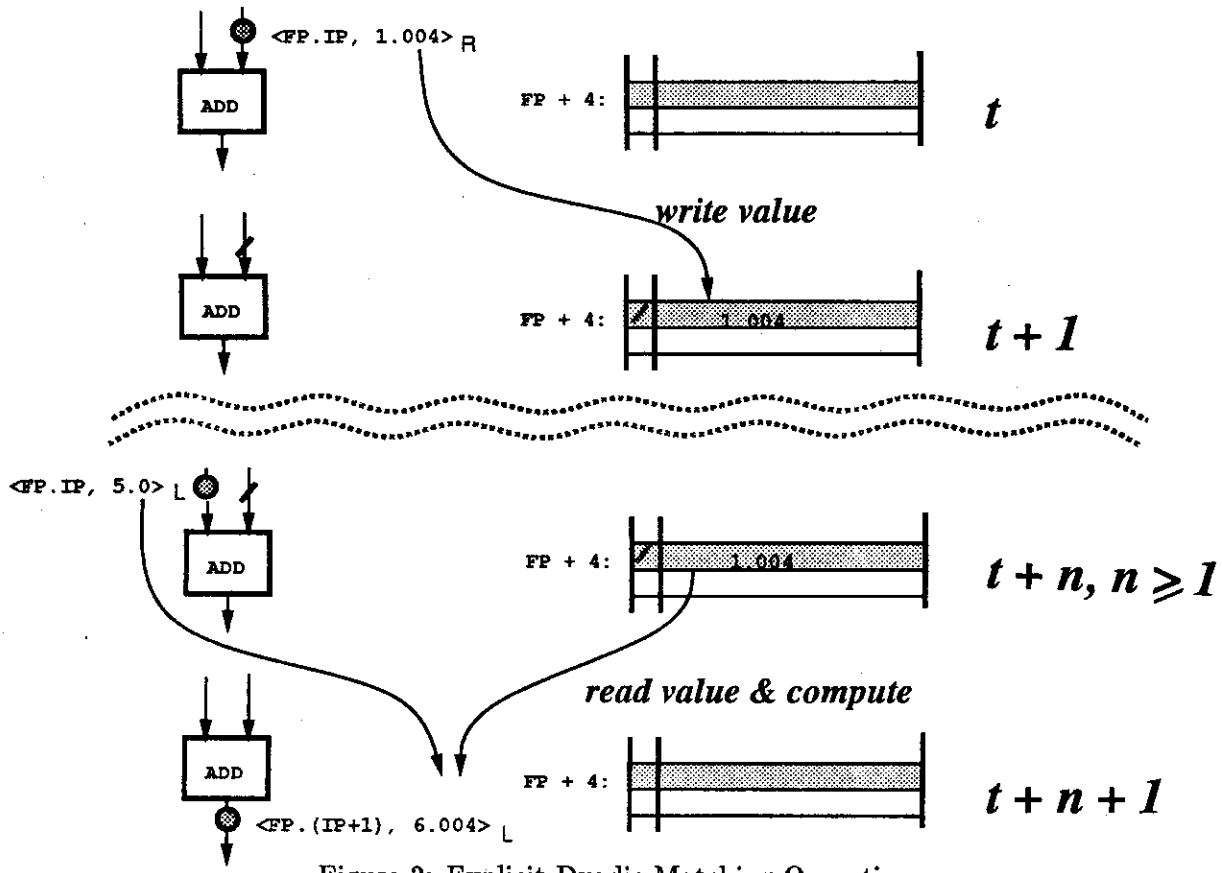

Figure 2: Explicit Dyadic Matching Operation

an important role in optimizing loops, where the constants are allowed to persist across a series of iterations.

In this way, an ETS architecture can achieve the power of tagged-token dataflow architectures with a leaner cycle and much less complexity. The graph schemas used for tagged-token dataflow programs can be used almost without change under ETS. (We assume the reader has some familiarity with dataflow graphs; a good overview can be found in the references[2, 10].) Exposing the operand store provides a natural means of supporting a variety of extensions to the dynamic dataflow model, including loop constants, I-structures, and accumulators. In addition, by supplying instructions which ignore the presence bits, *i.e.*, that directly read and write locations in the activation frame, the model supports traditional, imperative execution as well.

1.3 Overview

The informal discussion above is properly viewed as a strategy for using a specific set of mechanisms to support dynamic dataflow execution. The content of the ETS model is manifested in these mechanisms. To highlight the essential mechanisms Section 2 introduces them formally, independent of their realization in a machine. Section 3 demonstrates in greater detail how the mechanisms are used in the parallel execution of dynamic dataflow graphs including k-bounded loops and data structure operations. Section 4 describes how the ETS is realized in Monsoon. Finally, Section 5 gives preliminary performance measurements and reflects on the basic approach.

2 ETS Model of Computation

The explicit token store is a simple, inherently parallel model of computation. In this section, we develop a fairly precise view of the ETS model, apart from the particular implementation decisions and engineering concerns embodied in Monsoon. The ETS is not specifically bound to the dataflow model of execution. Rather, dataflow execution is an example of a set of compile-time and run-time disciplines that lie on top of the ETS model. An analogy is the difference between the von Neumann model and the compile-time and run-time conventions of a stack-oriented language. The stack discipline is just one way to rationalize the use of storage and control transfer primitives provided by the underlying von Neumann machine.

2.1 Storage, Tokens and Instructions

A basic feature of the ETS model is a global linear address space that holds instructions, heap objects and activation frames. Every addressable location is augmented with a small number of state bits to indicate the "presence" of data in the location.

Definition 1 *The ETS* **store** *is a linear array of locations, M. Each addressable location $M[i]$ contains $q.v$, where v is a fixed-size value and q is the* **presence state** *of location $M[i]$.*

An ETS machine state comprises the store and an arbitrarily large number of tokens. Each token is a compact computation descriptor encoding an instruction pointer, a pointer to an activation frame, and a single data value. The token may be viewed as an extremely lightweight "thread".

Definition 2 *An ETS* **token** *is a tuple* <FP.IP,DATA>, *where* DATA *is a fixed size data value,* IP *is the address of an instruction, and* FP *is the frame pointer, referring to the base of a contiguous block of locations in the store called an* **activation frame**. *The pair of pointers* FP.IP *is called a* **tag**. DATA *can encode a tag or a simple value, such as an integer, a floating point number or a storage address.*

The ETS machine state evolves as a result of processing tokens. For each token <FP.IP,DATA>, the instruction at location M[IP] is fetched and executed. During its execution, the instruction will typically refer to a location in the activation frame referenced by FP, perform an arithmetic operation, and then produce zero, one or two new tokens. An activation frame supplies the dynamic storage required for the local state of a procedure invocation, loop iteration, or more generally a code block. An activation frame corresponds roughly to a stack frame in conventional sequential languages. Two concurrent invocations of the same procedure will share the same code, but will use two distinct activation frames for local storage.

The novel feature the ETS is that every reference to storage by an instruction involves a *presence state transition* and the transition affects the subsequent behavior of the instruction. For clarity, we have decomposed ETS instructions into a specification of four separate actions performed on each token processed: computing an effective address into storage, accessing the computed location, performing an ALU operation and, finally, producing result tokens. This is akin to the actions specified by a von Neumann instruction: accessing register and memory operands, performing an ALU operation, storing result values, and either implicitly or explicitly updating the program counter.

Definition 3 *An ETS* **instruction** *is a tuple <E,S,A,C> where*

E *specifies the effective address calculation used to locate the storage operand:* $E \in \{\text{FP}+r, \text{IP}+r\}$, *where r is encoded as a literal in the instruction.*

S *specifies the presence state transition rule on the state-part of the storage location computed by E and, indirectly, control information:* $q \xrightarrow{S} (q', \text{M-OP}, \text{FLOW})$, *where q is current state of the location, q' is the new state of the location,* M-OP $\in$ {READ,WRITE,EXCHANGE} *is an action to perform on the value part store location computed by E and* FLOW $\in$ {COMPLETE, STOP} *determines whether further processing of the instruction will take place.*

A *specifies the arithmetic operation to perform on the* DATA *operand and, possibly, the value v read from the store, as determined by* M-OP. *This yields a result value:* $result = A(\text{DATA}, v)$.

C *specifies the continuation rule, when* FLOW = COMPLETE, *which dictates how result tokens are to be formed:* $C \in$ {*continue, fork, switch, extract, send*}. *The tokens produced by each rule are given by:*

continue:	<FP.IP + s_1, *result*>
fork:	<FP.IP + s_1, *result*> *and* <FP.IP + s_2, *result*>
switch:	*if* DATA = **true** *then* <FP.IP + s_1, *result*>
	if DATA = **false** *then* <FP.IP + s_2, *result*>
extract:	<FP.IP + s_2,FP.IP + s_1>
send:	<DATA,*result*> *and, optionally,* <FP.IP + s_2, *result*>

where s_1 *and* s_2 *are encoded as literals in the instruction. If* FLOW = STOP, *no tokens are produced.*

In Monsoon, E, S, A and C are encoded as a single instruction op-code, with separate r, s_1 and s_2 fields. In most cases, one of s_1 and s_2 is not explicitly encoded into the instruction, but implicitly set to one.

It is important to understand that the presence state is not just extra bits associated with each location; each operand access to a location specifies a possible transition of the presence state, $S(q)$,

followed by an operation on the value part. If the state transition yields M-OP = READ the value part of the location is read from storage and serves as an operand, v, for the ALU. If M-OP = WRITE then the token's DATA is written into the location and DATA is the only operand available to the ALU. If M-OP = EXCHANGE then the location is first read, yielding v, and then the token's DATA is written, and both v and DATA are ALU operands. The state transition on the location also dictates whether the instruction accessing the location can execute to completion (FLOW = COMPLETE) or is to be aborted (FLOW = STOP). Thus, the presence state dictates the dynamics of instruction execution.

Details of the presence state transition rules are given below, as required to implement particular synchronization paradigms. The simplest paradigm ignores the presence state bits entirely, specifies either a read or write of the value part, and always continues processing; this is traditional von Neumann impertative execution. More interesting rules can directly support the dataflow matching operation, I-structure operations, and even non-blocking semaphores and locks.

The ALU operation, A, is the full complement of arithmetic and logical operations, possibly extended with efficient operations on tags. As explained below in Section 3, dataflow execution associates an instruction with each *arc*. Thus, the "left" and "right" operands to an instruction like **subtract** actually specify two different instructions; the left operand will specify **subtract**, (DATA $- v$) while the the right operand will specify **subtract-reverse**, ($v -$ DATA). Since the two operations differ only in the orientation of the operands, the Monsoon implementation encodes left/right information as a separate bit in the tag, called the PORT, and has both operands specify the same instruction.

The continuation rule defines one or more successor activities. *Continue* simply passes control and data to the next instruction in a logical sequence. In a concrete implementation, it may be valuable to distinguish consecutive execution ($s_1 = 1$) and branching, as in most modern machines.

Fork allows multiple successor instructions to proceed in parallel by creating separate tokens for each successor, differing only in their IP fields. For instructions to fit in a fixed size word, the number of successors specified by the fork continuation must be limited. Fan-out instructions can be added to provide the additional successors, as needed.

Switch provides a form of conditional branch; it generates a result for one of two successors based on one of the input values. If the left operand is **true** then a token is generated with instruction pointer IP+ s_1, otherwise a token is generated with instruction pointer IP+ s_2.

Send supports efficient inter-frame transfer. It takes as operands an arbitrary value (DATA) and a tag (FP.IP) and merges these to form a result token (<FP.IP,DATA>). It is important to emphasize that *send* is nothing more than a simple concatentation of two operands to form a result token. The *send* operation is nonetheless fundamental to dynamic dataflow: a new tag can be dynamically computed by a program. By constructing a token with some DATA value and a new activation frame pointer and instruction pointer, activity is implicitly initiated within that activation. We will see that in the Monsoon different activation frames may map onto to different processing elements. Thus, the token that results from a *send* may need to be communicated to another processing element. This case is detected automatically and handled by the hardware. That is, interprocessor communication is transparent in the programming model and is a property of an implementation.

Extract supports call and return by allowing an instruction to copy the FP.IP of its input into the DATA part of its result. A *send* operation can communicate this tag (as the DATA part of a token) to another activation which, in turn, can use the tag in a *send* operation to return values to the orignal activation. The optional second destination of the *send* stays within the context of the *send* instruction and is frequently used as a local acknowledgement or *signal* that the *send* operation has been performed.

2.2 Split-Phase Memory Operations

Conspicuously absent from effective address computation of an ETS instruction is the ability to directly specify an arbitrary storage location, say the location pointed to by DATA, M[DATA]. ETS is intentionally prejudiced towards implementations where activation frames are stored local to processing elements so that the activation frame references are fast and non-blocking. In contrast, a reference to a remote (or global) location might induce an arbitrary delay in the processing of an instruction.

Instead, ETS supports *split-phase* memory operations. A split-phase read involves two actions. First, the reading instruction issues a *request* for a location, and continues processing without waiting for a response. The request encodes the location to be read as well as the tag of the instruction that is to receive the contents of the location. Second, the request is processed by a read operation "local" to the accessed location, generating a new token that comprised of the response tag provided by the request and the contents of the accessed location.

Definition 4 *An ETS* **request token** *is a tuple* $<\alpha.op,$DATA$>$, *where* α *is an address of a storage location in M, op is the request operation, and* DATA *is a tag or a simple value.*

A read request is represented as $<\alpha.read,$ FP.IP$>$, where FP.IP is the tag of the instruction that is to receive the contents of location α. The response would be the token <FP.IP, M[α]>. An arbitrary amount of time may elapse between issuing the read request (*i.e.*, executing a `fetch` instruction) and receiving the response (*i.e.*, enabling the successor of the `fetch`). In particular, a read request might be *deferred* by the memory system until the requested location is written, as indicated by the associated presence state. This supports I-structure semantics.

A write request is represented as $<\alpha.write,$ DATA$>$, where DATA is the value to be deposited in location M[α]. Other request *ops* might also be necessary in an implementation, including ones that explictly write and read the presence state associated with a location.

2.3 Parallelism in ETS

Given a token, the basic machine mechanism involves fetching the instruction specified by IP, computing the operand address, performing a read-modify-write on the presence state bits, performing an operation on the data part of the operand location, computing a result value, possibly emitting a memory request, and generating zero, one, or two tokens for further processing.

The total number of tokens in the system grows whenever a *fork* is executed, and shrinks whenever a presence state transition dictates a STOP. At any time, the overall state of the machine is the store, M, and the set of existing tokens, each representing the local state of an execution thread.

Importantly, the ETS model *does not* place any restriction on the *order* of processing tokens from the set of existing tokens. Execution must only obey the atomicity of the transition rule applied to the presence state and value part of a location. An implementation may choose to process any subset of tokens concurrently, and this is the fundamental source of parallelism in the model[1].

The ETS model efficiently virtualizes the state of a very simple processor, permitting an essentially unbounded number of concurrent threads, represented by the set of unprocessed tokens. The only form of interaction amongst threads is through the presence state and value parts of words in the store. The presence state bits on each word of store support fine grain synchronization within

[1] An implementation might want to apply the requirement that the scheduling is **fair** — a given token will be processed within finite time.

activations, across activations, and for shared data objects. Although there is no explicit notion of processor or inter-processor communication in the model, the set of operations are carefully restricted so that the model can be implemented with only local operations and inter-processor messages. The basic mechanism allows arbitrary parallel constructs to be formulated, however, deterministic execution can be achieved through an appropriate choice of presence state transition rules and by following the compilation methodology embodied in dataflow graphs.

3 ETS Execution of Dataflow Graphs

In this section, we show in some detail how the ETS mechanism is employed to support dynamic dataflow execution. In addition to the basic scheduling paradigm, discussed briefly above, this includes management of activation frames, the realization of control structures, and the implementation of synchronizing data structures.

3.1 Firing rule

The dataflow *firing rule* states that a node may execute when a operand value is available on each of its input arcs. Upon firing, it consumes the input values and produces a value on each of its output arcs. Furthermore, values are tagged to distinguish between potentially concurrent firings of a node. Thus, a dataflow operation can be broken down into four components: a synchronization operation on the input arcs, a function computed on the input values, the generation of the result values, and finally a naming convention for identifying the operands, *i.e.*, the *tag*. These components are made explicit in the ETS formulation, as indicated by Figure 3. An ETS instruction describes how a single token is processed; thus, there is an ETS instruction for each arc in the graph. One of the operands is DATA on the token and the other is identified by the effective address calculation. The two ETS instructions representing a dyadic operation identify the same operand location as the synchronization point and specify the "match" transition rule, described in Figure 3. If the location is **empty**, the operand value is written into it and the execution thread is terminated. If the location is **full**, its contents are read, the state is reset to **empty**, and the operation is performed on the two operands. We note that the Monsoon implementation allows the two target instructions for dyadic operation to be combined into a single instruction that gets executed twice: once by the first arriving token, and again by the second token. This is accomplished by providing an extra IP bit, called the PORT, whose state does not affect the address of the instruction fetched, but serves to distinguish the left/right orientation of the operands for the ALU.

Arrival of a token corresponds to executing the instruction associated with the arc. Whichever of the two instructions executes first finds the slot to be empty and stores its DATA into the slot. The second one will find it to be full, empty it, and perform the operation. A token is produced for each destination arc, as represented by the continuation rule in the ETS instruction.

Unary operations do not require use of the activation frame, so the "ignore" status transition rule is specified, *i.e.*, no frame store operation is performed. Operations that are enabled by a single token, but that have a constant as a second operand, specify the "read" transition rule, which ignores the presence bits and reads the location, and an effective address calculation that locates the appropriate constant. The FP + r addressing mode may be used, in which case the constant operand is retrieved from the activation frame. This is particularly useful for loop constants, discussed below, and also provides a means of avoiding unnecessary synchronization.

Although the ETS is easily understood as a mechanism for propagating tokens through dataflow graphs, an alternative view is that it describes a very large number of very simple "virtual processors" working in concert. The DATA portion of the token is essentially an accumulator and the FP

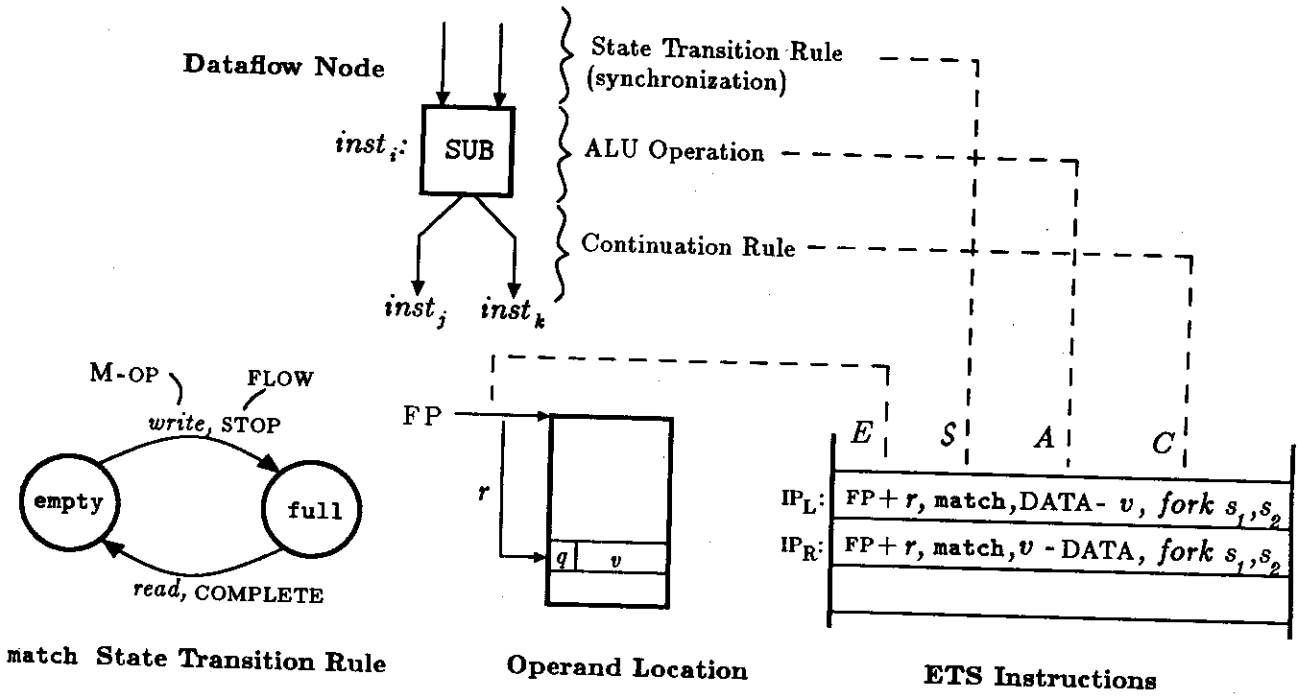

Figure 3: Correspondence of Dataflow Graphs and ETS Instructions

an index register. The instruction set provides 1+x addressing, but when the storage operand is not present the instruction can be transformed into "store accumulator" with no successor. Although this addressing capability is restrictive compared with most modern machines, it is precisely what is required for direct execution of dataflow graphs. This partly accounts for the observed difference in instruction counts on dataflow and conventional machines[3].

3.2 Procedure Call

The dynamic dataflow model allows many activations of a particular node to be enabled or partially enabled (*i.e.*, have tokens present on one arc) simultaneously. Different activations are distinguished by tag information carried on the token. In the ETS, this differentiation is provided by FP in specifying different activation frames. Thus, where tagged-token models name different synchronization points by their context in the program, under the ETS model the synchronization name identifies the resources used to perform the operation.

Invocation of an ETS procedure (or, more generally, a code block) involves allocating an activation frame and sending arguments to it, *i.e.*, generating tokens with FP referring to the base of the activation frame and IP referring to instructions within the procedure. Figure 4 shows an example ETS procedure call convention, in graph form, that invokes a function of two arguments.

The caller of must allocate an activation frame of the appropriate size and send the two arguments and the return adddress to the first three instructions in the callee. The return address carries the FP for the caller frame and the IP for the instruction within the caller code-block that is to receive the result. This call linkage can be seen in Figure 4. The **ALLOC-FRAME** node causes a fresh frame to be allocated and produces a tag as a result, comprising the address of the newly

allocated frame as FP_f and the address of the callee entry point as IP_f. (**ALLOC-FRAME** is shown as a single operation, but, in practice, would be implemented as a small collection of instructions, as discussed below.) The caller forms the return address by extracting the current tag as a value (**EXTRACT-TAG**) and adjusting the IP to refer to the instruction that is to receive the result, *e.g.*, IP_{ra}. A **SEND** instruction is used to communicate the values and return address to the the newly-allocated frame. **SEND** is parameterized by a small constant, which is used to adjust the IP on the result token. Note, **SEND** is a trivial operation that takes two values, a tag and an arbitrary value, and concatenates them to create a token. We illustrate the arc that the new token traverses with dashed lines, because the target instruction is computed *dynamically*. Just the act of creating the new token is sufficient to introduce it into the new context, as the hardware is responsible for automatically transporting a token to the processing element corresponding to a given FP. The I operators are simply identity instructions, used to establish conventional entry points. A less structured ETS calling convention would permit the compiler to pass the arguments directly into known offsets within the callee body, eliminating the identity operations.

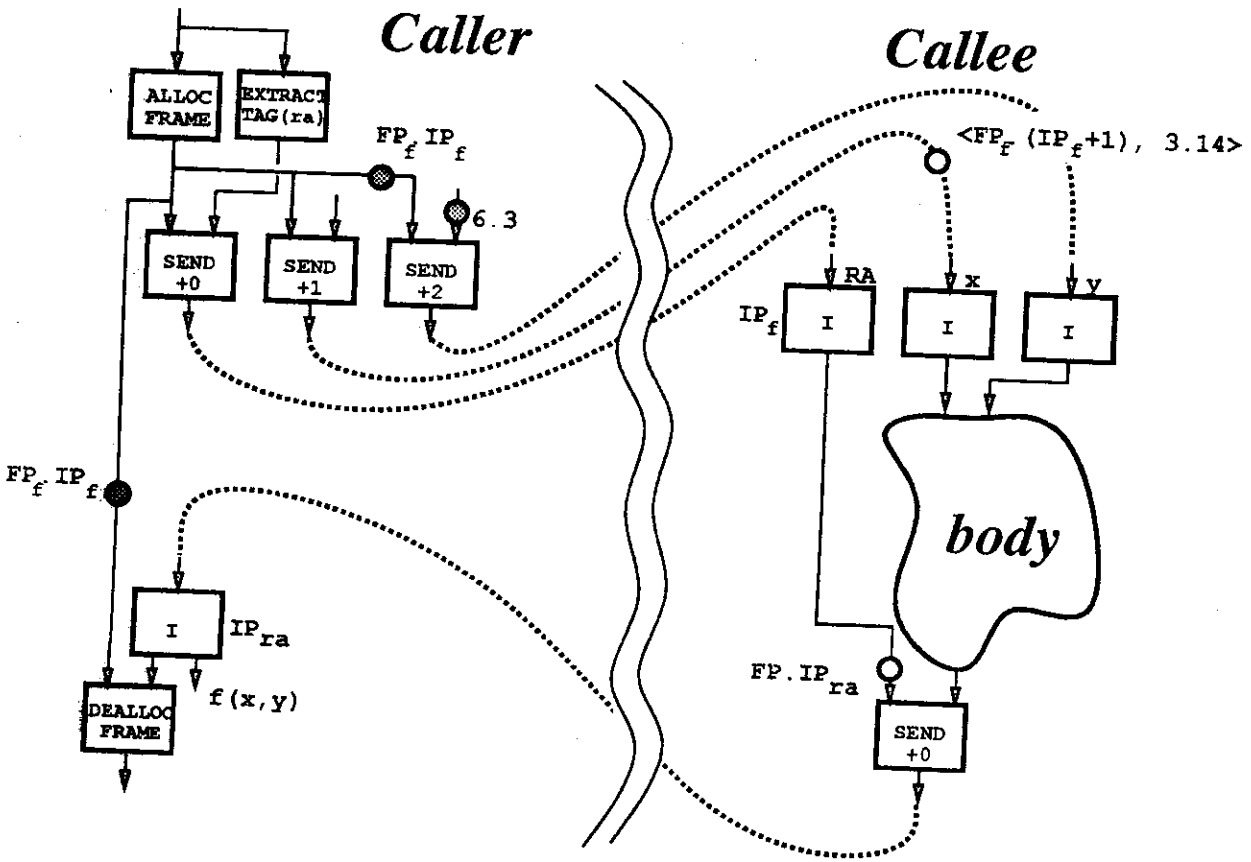

Figure 4: ETS Example Procedure Calling Convention

At first, the calling convention may look somewhat complex, but is surprising efficient. On

conventional sequential machines, **ALLOC-FRAME** corresponds to allocating stack space by advancing the stack pointer. The **EXTRACT-TAG/SEND+0** combination is equivalent to pushing a return address[2], and the **SEND+1** and **SEND+2** are like pushing the arguments. The **SEND+0** executed by the callee corresponds to a return value convention, and the final **DEALLOC-FRAME** executed by the caller corresponds to cleanup code. Of course, the ETS convention is completely *transparent* with respect to crossing processor boundaries, and should really be compared with a *remote* procedure call on a conventional machine. By these standards, the ETS mechanism is exceptionally efficient.

Frames are allocated dynamically as part of the call linkage, but the use of slots within the frame is determined statically by the compiler. The simplest means of assigning frame slots is to give each node in the graph a unique slot. However, this may result in poor utilization of the frame, since many of the slots will be empty most of the time. Each dyadic instruction specifies the frame slot (r) used as a rendezvous point for its operands, thus it is possible for two operations to specify the same slot. It is valid for two node to be mapped to the same slot as long as they cannot interfere, *i.e.*, if it is impossible for operand tokens to exist for both of them simultaneously. This is the case if all the inputs of one depend unconditionally on the result of the other or if the two nodes appear in different arms of a conditional. The compiler forms an *interference graph*, where edges represent potential coincident operands and colors this graph using simple heuristics. Nodes with a single input, including unary operations and binary operations with one constant operand, need not be assigned a frame slot.

Observe, the execution schedule of code-block Figure 4 depends on the arrival order of the arguments. Parts of the graph are enabled as each input arrives and concurrently enabled operations may be performed in parallel. The operation that sends the result to the caller has a "local" successor that releases the current frame (in practice, this too may require several instructions). The compiler systematically adds *signal arcs* to the dataflow graph to detect when an activation is complete and the frame can be released. In general, it is not sufficient to determine that the results have been sent, as suggested in the figure[45], but also that all instructions that do not produce results, *e.g.*, I-store, have completed.

We have not specified how frames should be mapped to processors or in what order threads (*i.e.*, tokens) are scheduled. The called procedure could execute on the same processor as its caller or on a different one. However, using dataflow graphs as a programming methodology, we are guaranteed that all execution schedules will produce the same result, regardless of the mapping. Clearly, activation frames provide a natural unit of distribution. In general, a code-block may generate many simultaneous invocations that can be assigned to different processors.

3.3 Loops

Tagged-token models of dynamic dataflow execution[7] provide an efficient means of expressing iteration as a cyclic graph. The tag on a token carries an iteration identifier to distinguish between different iterations and each cycle in a loop graph includes a single operator, that increments the iteration identifier. Loops may unfold, allowing an arbitrary number of concurrent iterations, constrained only by data dependences. Thus far, we have not demonstrated how to express the same behavior in an ETS framework. To allow an arbitrary number of concurrent iteration, the loop would have to be represented essentially as tail recursion, where each iteration allocates a frame for the next. The full generality of dataflow loops turns out to be difficult to support in tagged-token dataflow architectures as well, because no limits are placed on the number of concurrent iterations. Any fixed-size iteration identifier may overflow, requiring some additional mechanism to handle this case. Furthermore, the resource behavior of programs is extremely unpredictable

[2] A simple optimization, used in Monsoon, combines the **EXTRACT-TAG** and **SEND+0** into a single instruction.

with no constraints on loop unfolding[15, 16]. This led us to consider a more restricted form of loops, called k-bounded loops, where the maximum number of concurrent iterations is established dynamically at the time the loop is invoked.

k-bounded loops can be implemented quite naturally in the ETS framework. When a loop is invoked, k activation frames are allocated and formed into a ring by storing into each a pointer to the predecessor and successor frames. The compiler generates graphs in such a way that each iteration detects its own completion and signals its predecessor that the frame is available for reuse. A methodology for assigning the k-bounds is given elsewhere[16].

In examining the cyclic dataflow graphs generated for loops, it becomes apparent that certain values used in the loop are circulated from iteration to iteration, but never change[4, 23, 43]. These "loop constants" correspond to free variables of the loop in the program text or invariants hoisted out of the loop. Circulation of loop constants leads to considerable overhead. For example, the innermost loop of a straight-forward matrix-multiply routine references seven variables, five of which are loop constants. In the ETS framework, these need not be circulated. When the ring of frames is constructed, copies of the loop constants are deposited into the frames. They are referenced using FP-relative addressing, but the presence bits are not reset.

3.4 Synchronizing data structures

Since the ETS has an explicit notion of storage, data structures can easily be represented within the model — an array is simply a sequence of words, much like a frame. However, care must be exercised in how data structures are accessed. We have not specified how the memory space maps onto processors, but we cannot assume that the region occupied by a data structure will bear any relationship with that occupied by a frame supporting an operation that accesses the structure. We do not want to assert the existence of operations that reference remote locations, because these instructions could cause the processor that hosts them to suspend for large and unpredictable amount of time. For synchronizing operations, such as I-structure fetches, it is necessary that the processor be able to perform other tasks while the read is outstanding to avoid deadlock[46]. Thus, any potentially remote reference should be represented as a *split-transaction*, part is performed by the operation that makes the request and part is performed local to the element that is accessed. Refer to Figure 5.

I-structure semantics require that all reads against an element receive the (one) value written into the element. Thus, reads that arrive prior to the write must be deferred. This is easily implemented using the presence bits associated with the location itself. A read for a slot marked empty results in the return tag being deposited in the frame. The write causes a *send* of the the return tag with the value written. In order for subsequent reads to receive the same value, the write value on the token and the return tag in the slot must be exchanged when the write request token is processed. Multiple deferred readers can be accomodated by reserving a slot in the frame where the I-structure read was executed to provide room for the deferred-read list[39]. These techniques can be extended to very efficient event-driven locks[44].

3.5 Summary

The ETS model is essentially a multithreaded 1-address machine, yet it captures the essence of the dataflow execution model based on propagating values through graphs. Compilation of graphs from a high-level language is only slightly changed from tagged-token dataflow architectures to provide explicit allocations and deallocation of frames. Each operation on the graph is represented by the execution of a single instruction. A simple state transition rule dictates that an operation is

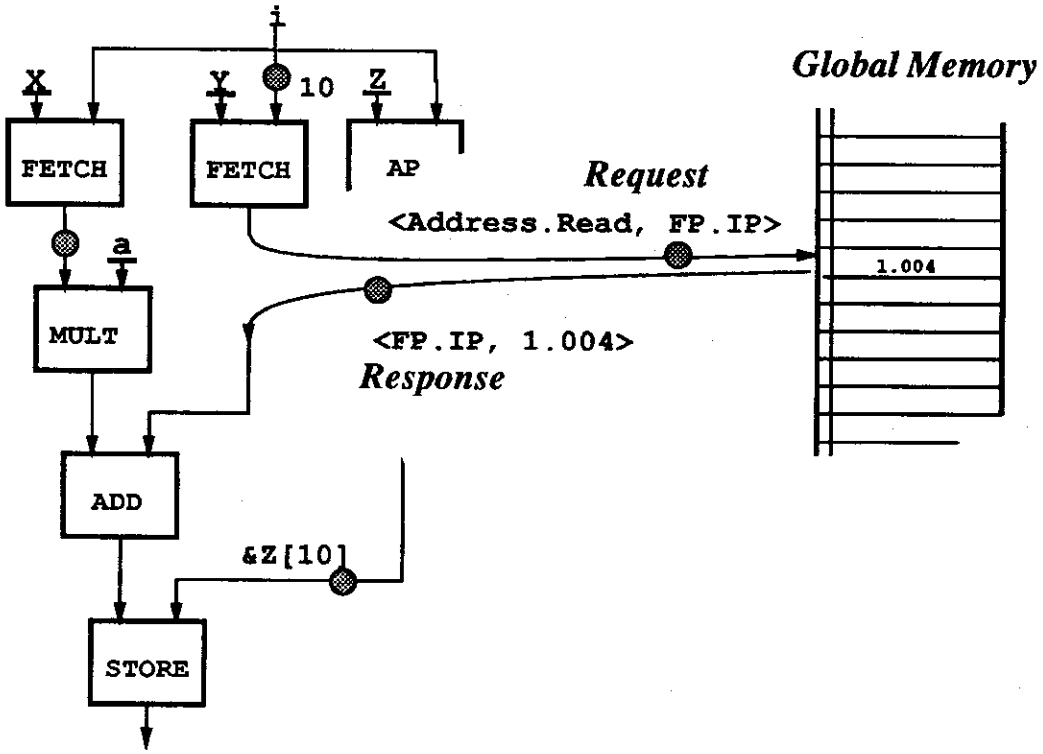

Figure 5: ETS Split-Transaction Memory Read

carried to completion only when all its operands are available. In fact, an action is taken for each value that is produced. The machine simply carries out the logical sequence of steps described by an execution thread, allowing a thread to be terminated if an operand it requires is not available.

4 Monsoon

Monsoon is a general purpose multiprocessor which incorporates an explicit token store. A Monsoon machine comprises a collection of highly pipelined processing elements (PE's), connected via a multistage packet switch network to each other and to a set of interleaved memory modules (IS's) that support I-structure storage as well as imperative storage. Please refer to Figure 6. Messages in the interprocessor network are tokens and request tokens — precisely the same format used within the PE and IS. Thus, the hardware makes no distinction between inter- and intra-processor communication.

4.1 Overview

A fundamental design decision in Monsoon concerns the mapping of activation frames across processors. Specifically, activation frames do not span processors. A frame required by a procedure is dynamically allocated by the caller on some processing element at invocation-time, and executes entirely on that processor. Iterations of a loop are distributed across multiple processing elements by assigning separate activation frames for each concurrent iteration. These strategies improve locality (*i.e.* reduce network traffic) without squandering the fine-grain parallelism — the parallelism within an activation is used to keep the processor pipeline full. Observe, we do not support load balancing algorithms that move activations, once allocated, amongst processors.

By insisting that frame references be local to a processing element, we are assured that instructions which perform frame accesses are non-blocking. In contrast, an instruction that reads an arbitrary storage location (say, an element in a shared array) might incur substantial latency as the read request and response traverse the interprocessor network. Instead, the system employs split-phase transactions for all data structure references and relies on the capability of the processor to generate multiple outstanding memory requests to fill the processor-memory "pipeline." In addition, data structures are word-by-word interleaved over memory modules in attempt to make structure memory traffic more uniform. Data structures are not cached.

In combination with the policy of mapping a given activation frame to a single PE, the lack of global memory caches implies that interprocessor token traffic is generated by data structure requests (reads and writes) and transmission of procedure (code-block) argument and return values. The interprocessor network is therefore appropriately sized to handle this fraction, approximately 30%, of the total number of tokens produced during the course of a computation.

Another distinguishing feature of Monsoon is that the pool of unprocessed tokens is distributed across processing elements in the form of hardware managed *token queues*. A similar design decision was taken in the TTDA and the Sigma-1[43], in contrast to the centralized token queue of the Manchester machine[26]. In the present implementation, two queues are provided on each processing element. A high priority *system queue* and a low priority *user queue*. The system queue operates as a first-in first-out queue, whereas the user queue is a last-in first-out stack.

As illustrated in Figure 7, a Monsoon PE is an eight stage pipelined processor. On each processor cycle[3] a token is dropped into the top of the pipeline and, after eight cycles, zero, one or two tokens

[3] A processor *cycle* usually corresponds to a single processor clock. Sometimes, for instance during a frame store exchange or a floating point divide, the pipeline is stalled for one or more clocks. In this case, it may take several clocks to make a cycle. In any event, the difference in clock times only affects the performance, not logical operation,

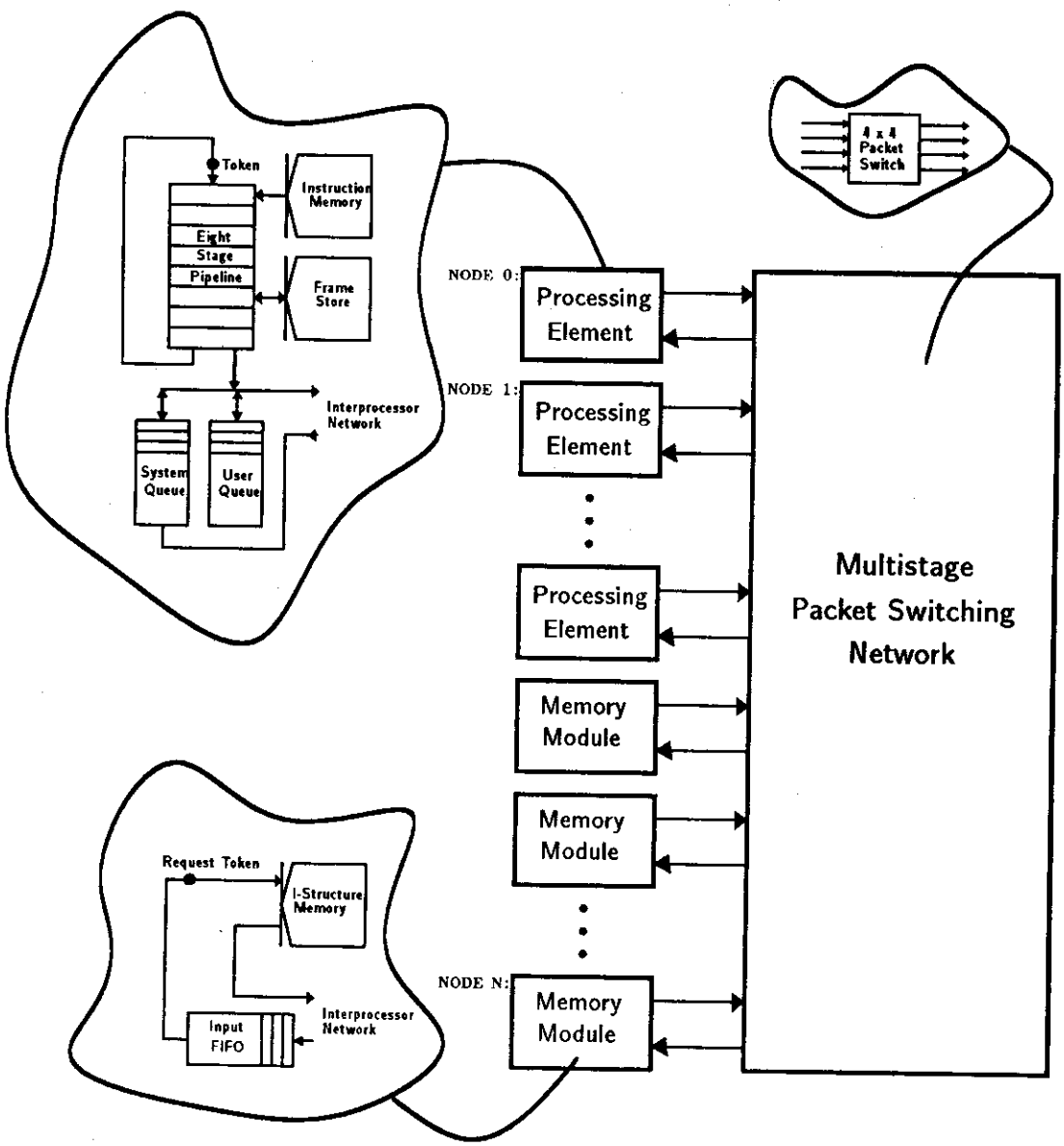

Figure 6: Top Level View of the Monsoon Multiprocessor

emerge from the bottom. Along the way, an instruction is fetched from instruction memory, an effective address in calculated into frame store, an read-modify-write occurs on the presence bits of the addressed location, followed by a read, write or exchange to the value part of the location. The incoming token's value and any operand read from frame store are processed by a three stage floating point ALU and, in parallel, two new tags are derived from the incoming token's tag. Finally, the result token(s) are composed by the form token section.

One of the output tokens can be *recirculated*, *i.e.*, immediately placed back into the top of the pipeline. Tokens produced by the pipeline that are not recirculated may either be inserted into one of two token queues or delivered to the interprocessor network and automatically routed to the correct PE. A more detailed description of the pipeline operation is presented below, after developing the basic programming model and instruction set.

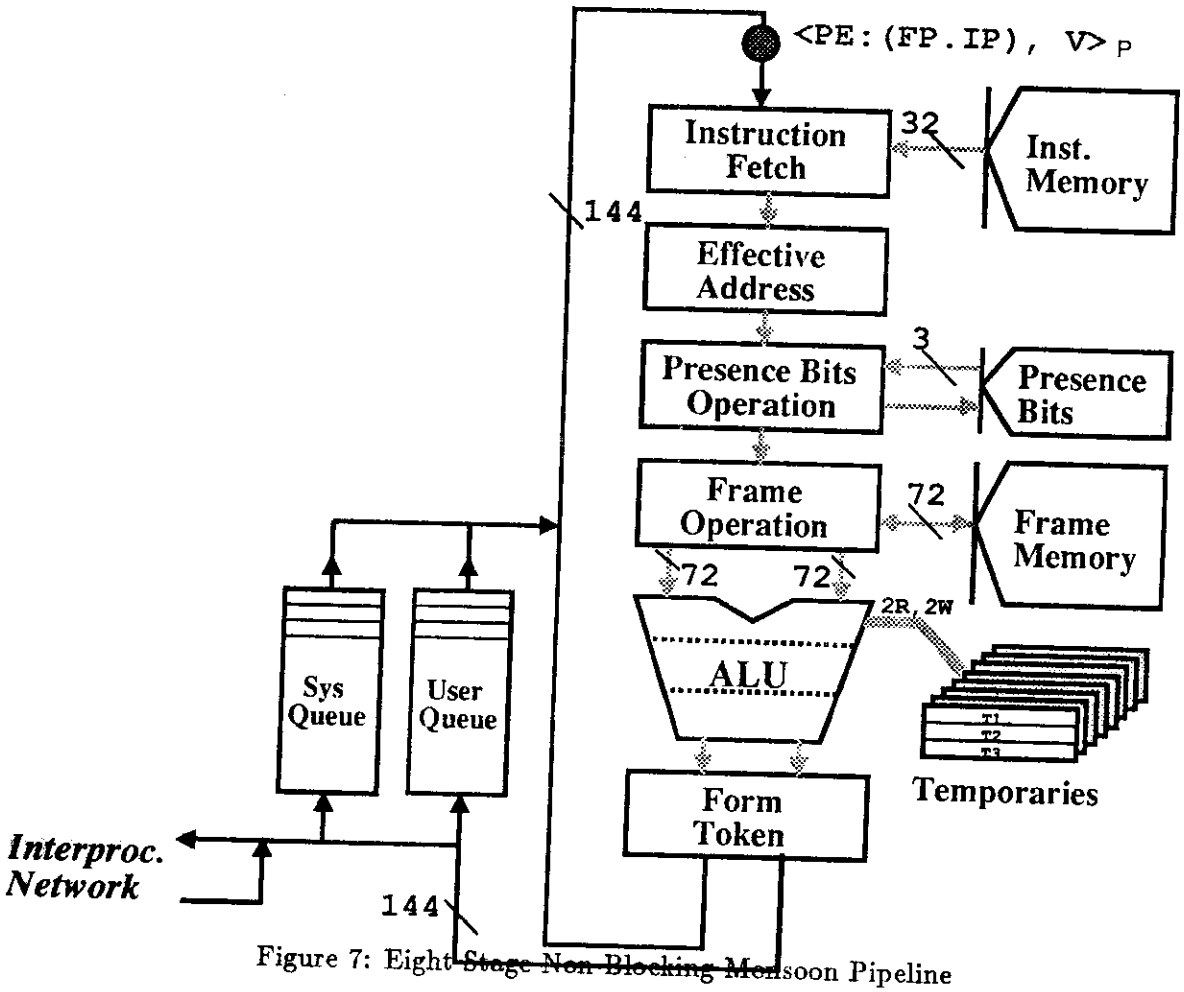

Figure 7: Eight-Stage Non-Blocking Monsoon Pipeline

because tokens advance from one stage to the next only at cycle boundaries.

4.2 The Token

Monsoon tags and values are 72 bit quantities comprising eight bits of hardware type information and 64 bits of data. A token is a tag-value pair, 144 bits in size.

Token			
TYPE	TAG	TYPE	VALUE
8	64	8	64

4.2.1 Values

The hardware provides direct support for five value formats. A value can be a 64-bit signed integer, an IEEE double precision floating point number, a bit field or boolean, a data memory pointer or, of course, a tag.

4.2.2 Tags

A tag encodes two pointers: a pointer to the next instruction to execute, IP, and a pointer to the activation frame, FP, that provides the context in which to attempt execution of the next instruction. On Monsoon, a given activation frame resides entirely on a single processing element. The frame pointer and instruction pointer are conveniently segmented by processing element as follows,

$$\text{TAG} = \text{PE}:(\text{FP}.\text{IP})$$

where PE is the processing element number and IP and FP are local addresses on processor PE. Adding a small offset to FP results in an address that maps to the local store of same processing element as FP. That is, all activation frame references are local and are considered non-blocking—activation frame reads and writes can take place within the processor pipeline without introducing arbitrary waits. PE, FP and IP are encoded into a 64-bit tag as follows:

TAG			
X	IP	PE	FP
8	24	8	24

Note that IP is a local reference as well. The next instruction is fetched on processor PE from instruction address IP. Furthermore, the most significant bit of the instruction pointer encodes a PORT bit which, for two-input operations, establishes the left/right orientation of the operands.

4.2.3 Pointers

A data structure pointer encodes an address on a processing element or I-structure memory module. Pointers are always represented in a "normalized" format as the segmented address NODE:OFFSET, where NODE denotes the processing element or I-structure module number and OFFSET is a local address on the NODE. Additionally, pointers contain *interleave information* in the MAP field, which describes how the data structure is spread over a collection of nodes. One option is for the entire data structure to be mapped to a single node. In this case, adding one to the pointer will yield the new pointer NODE:(OFFSET+1). Another option is for the data structure to be interleaved on a word-by-word basis across the entire machine. In this case adding one to the pointer will yield the new pointer (NODE+1):OFFSET.

The interleave pattern can be generalized into *subdomains*[4]. A subdomain is a collection of 2^n nodes which starts on a modulo 2^n NODE number boundary. If $n = 0$ then increments to the

pointer will map onto the same NODE. If $n = 1$ then increments to the pointer alternate between NODE and NODE+1. Figure 8 illustrates address interleaving as a function of n for subdomains of size one, two and four. Interleaved pointer arithmetic is performed by a special ALU called the pointer increment unit (or PIU).

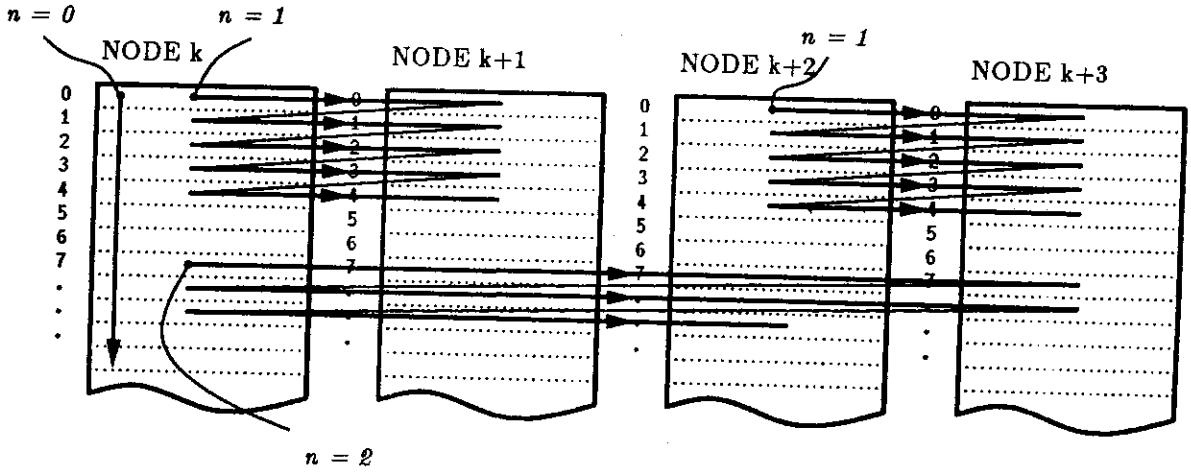

Figure 8: Data Structure Interleaving as a Function on n

A pointer and a tag share a similar set of field assignments:

POINTER			
MAP	INFO	NODE	OFFSET
8	24	8	24

The MAP simply encodes n, the size of the subdomain. Because MAP is used only when adding to a pointer, aliasing is not a problem. Two pointers refer to the same location if and only if they have identical NODE fields and identical OFFSET fields. The INFO field is not interpreted by the hardware and is available for use by the compiler and runtime system.

4.2.4 Requests

A request encodes a data structure operation, like read or write. A single data value is appended to a request to create a request token which is destined to a memory module. For example, writes are encoded as a write request and the value to write, while reads are encoded as a read request and the return tag. A processor wishing to perform a read or write operation does so by composing a request token and issuing the token to the interprocessor network, where it is routed to the appropriate NODE. Thus, a request plays a role very similar to a tag, and, not surprisingly, requests and tags have very similar formats:

REQUEST			
X	OP	NODE	OFFSET
8	24	8	24

The OP field encodes the request type (*e.g.* read, write)[4]. The address of the location is given by NODE:OFFSET. The neat correspondence of requests and tags permits a processing element to process a request token as if it were a regular token. In this case the OP field becomes the IP. So a processor that wants to emulate IS behavior need only provide a set of support instructions at a small number of "wired" locations. From another perspective, an IS can be viewed as a processing element which is specialized for the processing of request tokens.

4.3 Instructions

The IP field of the token is used as a direct address into the instruction memory of a PE. Following the ETS model, the instruction dictates the offset in the frame, the kind of matching operation that will take place (*i.e.* the state transition function on a word in frame store), the operation performed in the ALU and the way that new result tokens will be formed. All Monsoon instructions are of uniform size with the following format:

Instruction		
OPCODE	$f1$	$f2$
12	10	10

The OPCODE determines the meaning of the $f1$ and $f2$ fields. There are three basic combination of $f1$ and $f2$ assignments.

Instruction Formats		
CLASS	$f1$	$f2$
dyadic	DEST1	r
monadic	DEST1	DEST2
long	r_{msw}	r_{lsw}

The r field is used to compute the effective address of a location in local frame store. There are two possibilities. In *frame relative* addressing the frame store location is computed as FP + r. In *absolute* addressing the location is simply r — the first set of locations on *this* PE. An opcode can encode either absolute or relative addressing for any format in which r appears. Typically, r is a 10 bit (unsigned) displacement into an activation frame. Literal constants and procedure linkage information are conventionally kept in low memory and referenced by using a 20 bit absolute address in the **long** format. The absolute address mode is used in lieu of the IP +r address mode specified by the ETS.

Every instruction can have up to two destinations, as given by DEST1 and DEST2. These fields are encoded as an adjustment to IP and explicit PORT value (the msb of IP), as follows:

DEST Field Encoding	
PORT	s
1	9

The two's complement encoding of the displacement s is added to the current IP to form a target instruction address. An instruction may implicitly encode a destination as well. When a DEST does not explicitly appear in the instruction format, s defaults to +1 and PORT is set to zero. That is, one destination can be the next instruction in instruction memory, IP + 1.

[4]The I-structure hardware only decodes the least significant six bits, and the most significant bit, of OP.

4.4 Temporary Registers

Monsoon extends the explicit token store model by provided a degree of compiler control over scheduling as well. Recall, the processing of a token may result in zero, one or two new tokens. One of the result tokens may be automatically routed to another processing element[5], while a local result token may be *directly recirculated* back into the processor pipeline. If two local result tokens are created then one of them is inserted into a token queue while the other is directly recirculated. In the case of two local result tokens, the Monsoon instruction set allows the compiler to declare *which* token will be recirculated, thereby biasing the token scheduling along a sequence or "thread" of instructions. One should imagine eight such threads being interleaved by the pipeline. A thread is broken when a token has no local successor, either because it produced no tokens (say, being the first arriving token to a dyadic operator) or a token produced was routed to the interprocessor network. A new thread is started when a token is popped from a token queue an inserted into the pipeline.

Observe, if a token being processed in instruction fetch on cycle n emits a local token, then, by virtue of the direct recirculation path and the synchronous pipeline, the result token will be processed in instruction fetch precisely eight cycles later, cycle $n + 8$. Similarly, the successor of a token in the first stage of the ALU will also be in the first stage of the ALU precisely eight cycles later. It should be possible, therefore, for a token and its successor to share state associated with the ALU, as long as we are careful to have eight replicates of the state to prevent interference amongst the eight, possibly unrelated, threads in the pipeline.

The ALU state is introduced as a small set of *temporary registers* that can be imperatively read and written under instruction control. The current implementation provides eight sets of three temporary registers, one set for each pipeline thread. We have provided two new instruction formats for manipulating the temporary registers.

New Instruction Formats		
CLASS	*f1*	*f2*
register	REGS	*r*
register	REGS	DEST1

The REGS field specifies four register operations, two reads and two writes. The two writes permit both the incoming token value and a value from frame store to be recorded in temporary registers. The two read values are submitted as operands to the ALU. It is important to understand that the temporaries remain valid only as long as there is a local recirculating token. Thus, registers are not preserved across synchronization points (dyadic operations) or split phase memory reads. This restricts the typical run length of a thread to just a few instructions, but can nonetheless substantially improve the dynamic efficiency of certain instruction mixes.

Figure 9 illustrates the application of temporary registers to the expression `(x + y)*(x - y)`. In the standard dataflow graph, shown on the left, a total of six tokens are processed to compute the result of the expression. This is a cycles per instruction (CPI) rate of 2.0. Also note that three locations in frame memory are required. The threaded code is shown on the right. Now only one frame slot is used to perform the matching of `x` and `y`. Once matched, the values are written into two temporary registers, which are later used by the subtract operation *without* having to pay the cost of resynchronizing `x` and `y`. A total of four tokens are processed, yielding a CPI of 1.5. However, a small amount of potential parallelism (the add and the subtract being processed by

[5]This is automatically detected by the hardware by inspecting the PE field of the result token and comparing it against the local PE number. In the case of memory request tokens, the NODE field is compared against the local PE number.

the pipeline simultaneously) has been sacrificed. The use of temporary registers in Monsoon was inspired by the main register set Iannucci's hybrid machine[30], and are similar to the register set concept independently developed for the EM-4[42].

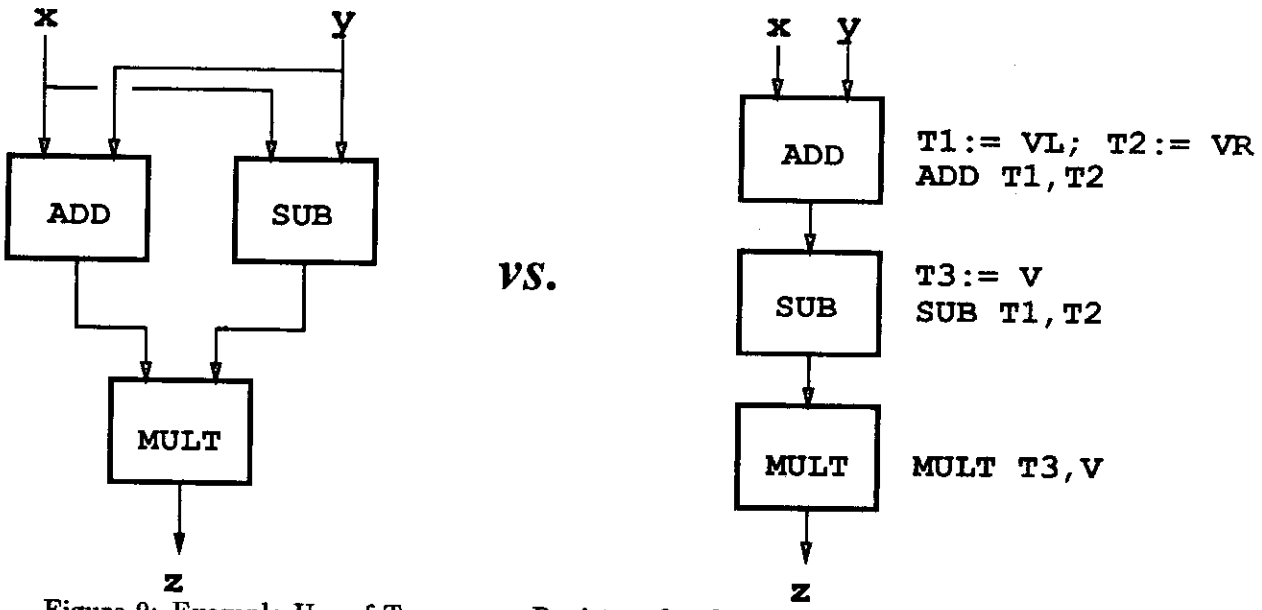

Figure 9: Example Use of Temporary Registers for the Expression `(x + y)*(x - y)`

4.5 Exceptions

The thread model used to reason about temporary state applies equally well to processor exceptions. We extend the eight sets of temporaries with two more registers, called **XA** and **XB**, and use them to always record the actual operands submitted to the ALU. Now, if these operands ultimately elicit an ALU exception the hardware suppresses the generation of any instruction-specified result tokens and, instead, recirculates an exception token into the pipeline.

An exception token is simply a token which has a predetermined tag, the IP pointing to the exception handler entry point and FP pointing to a reserved activation frame, and a value that is the tag of corresponding to the offending instruction. The exception handler can then query the saved **XA** and **XB** operand values[6].

As eight exception handlers may be active simultaneously, each interleaved thread is given a different exception FP. Furthermore, an exception handler can be vectored by changing the exception IP as a function of the kind of exception detected. Monsoon supports exceptions based on ALU conditions, operand type mismatches, and unconditional exceptions. In fact, the exception mechanism is so efficient that we use it to dynamically link into resource managers. For example, the compiler emits a **ALLOC-FRAME** and **DEALLOC-FRAME** instruction for procedure calls which are trapped

[6]The first few instructions of a handler can suppress the updating of **XA** and **XB**.

and vectored into a short sequence of instructions that perform the activation frame management operations.

4.6 Detailed Pipeline Operation

Figure 7 is a detailed view of the eight processor pipeline stages. A token can be inserted into the pipeline every cycle. After a delay of eight cycles, zero, one or two tokens emerge from the bottom. A stage-by-stage description of the eight processor cycles follow.

1. **Instruction Fetch.** The IP from the incoming token is used as an address into local instruction memory. A single 32-bit instruction is fetched.

2. **Effective Address Generation.** The effective address of the location in frame store is computed by adding the FP part of the tag to the r in the instruction (relative addressing), or by simply passing the r field (absolute addressing).

3. **Presence Bits.** There are three presence (or status) bits associated with each word in frame store. These presence bits associated with the frame store location computed during effective address generation are read, modified by a table lookup, and then written back to the same location. The lookup operation dictates a state transition on the presence bits. The transition function depends on the port bit on the incoming token and the instruction opcode. State machine outputs are used to control subsequent pipeline stages. One output dictates whether the operation on the value part of the associated frame store location should be a read, write, exchange or no-op. Another output suppresses the processing of result tokens. This will be asserted, for example, when the first token for a two-input operator is processed.

4. **Frame Operation.** This stage operates on the value part of the frame store location computed during the effective address stage. The presence bits state transition decides whether the addressed location should be read, written with the value carried on incoming token, exchanged with the incoming value, or whether no operation should be performed.

5. **ALU-1.** If the frame store was read or exchanged during stage 4, the ALU has two values on which to operate: the value read and the incoming value. These are sorted into left and right values using the port bit of the incoming token. If frame store is written or if no frame store operation is performed, then only a single new value (the incoming token) is available for the ALU. During this stage, the incoming value and the frame store value can be written into temporary registers, and two registers can be read to supply the ALU operands. The actual ALU operands are recorded into the XA and XB temporaries. It is also possible to introduce the incoming tag as one of the ALU operands.

6. **ALU-2.** During this stage the left and right operands are processed by one of the ALUs: a floating point/integer unit (FALU), a specialized pointer/tag arithmetic unit (PIU), a machine control unit (MCU) or a type extract/set unit (TPU). The PIU can also treat the $f1$ field from the instruction as a literal value.

7. **ALU-3** Concurrent with the final ALU processing, two new result tags are computed by the next address generators. These parallel units create new destination tags by passing the incoming PE and FP while incrementing the incoming IP as dictated by DEST1 and DEST2. Either DEST can implicitly refer to IP + 1.

8. **Form Token/Token Queue** The form token stage creates up to two result tokens by concatenating the tags computed by the next address generation with the ALU result. During inter-procedure communication and data structure requests the result tag is actually *computed by* the ALU. The form token multiplexor therefore allows the ALU result to be the tag of one of the tokens. An extra result token value, a delayed version of the "right" value, is also available to the form token multiplexor. This stage detects whether the PE or NODE part of a result token tag is equal to the present processing element number. If not, the token is forwarded to the network, where it is automatically routed to the correct processing element or I-structure module. One of the new tokens may be recirculated directly to the instruction fetch stage (if the PE matches, of course). If two tokens are created, one of the result tokens can be placed onto either the system or user token queue. If no tokens are created then a token can be dequeued from one of the token queues.

Consider the processing of a two-input operator. Either the left or right token may be processed first. The first token to be processed enters the pipeline and fetches the instruction pointed to by IP. During the effective address stage the location in frame store where the match will take place is computed. The associated set of presence bits are examined and found to be in the `empty` state. The presence state is thus set to `full` and the incoming value is written into the frame store location during the frame store stage. Further processing of the token is suppressed because the other operand has yet to arrive. This "bubbles" the pipeline for the remaining ALU stages; no tokens are produced during form-token, permitting a token to be removed from one of the token queues for processing.

The second token to be processed enters the pipeline and fetches the same instruction. It therefore computes the same effective address. This time, however, the presence state is found to be `full` so the frame store location (which now contains the value of the first token) is read and both values are processed by the ALU. Finally, one or two result tokens are created during the form token stage.

5 Evaluation

A single processor Monsoon prototype has been operational at the MIT Laboratory for Computer Science since October 1988. Except for the interprocessor network connection and temporary registers, the prototype employs the synchronous eight stage pipeline and 72-bit datapaths presented in Section 4. The memory sizes are fairly modest: 128KWords (72 bits) of frame store and 128KWords (32 bits) of instruction memory. The prototype cycle time is 250ns., so it processes four million tokens per second. The processor is hosted via a NuBus adapter in a Texas Instruments Explorer lisp machine. The compiler and loader are written in Common Lisp and run on the host lisp machine, whereas the runtime activation and heap memory management kernels are written in Id and directly execute on Monsoon.

We are working with the Motorola Microcomputer Division and the Motorola Cambridge Research Center to develop several Monsoon multiprocessor research systems. The new processors are similar to the first prototype but are faster (10 million tokens per second), have somewhat larger frame storage (256KWords to 1MWord), have dedicated I-structure memory modules (4MWords), and are connected via a high-speed multistage packet switch (100 Mbytes/sec/port). A photograph of the processor board is shown in Figure 10. A PE is implemented on a single 9U×400mm surface-mounted printed circuit card which supports a VME port for diagnostics and input/output, and two unidirectional network links. The processor core is byte sliced into eight 10,000 gate CMOS

To contain photograph of Monsoon processor

Figure 10: Monsoon Processor Board

arrays. The floating point unit is fully pipelined, yielding up to 10 million double precision floating point operations per second. Presently, we are implementing a 256KWord (32 bit) instruction memory, 256Kword (72 bit) frame store, and a 64KWord (144 bit) token queue. The token queue memory is divided equally between system and user queues.

The network is based on a 4×4 packet switching integrated circuit called PaRC (Packet Routing Chip)[31]. Each fixed-size PaRC packet comprises 16 bits of routing header, 144 bits of data (a token), and 32 bits of cyclical redundancy check for error detection. The PaRC datapaths are 16 bits wide and cycle at 20ns. This yields a per link bandwidth of 100 Megabytes/second, or a total PaRC bandwidth of 800 Megabytes/second. PaRC employs streaming (or virtual cut-through routing) and input packet buffers on each input link capable of storing four packets each. The packet buffers are intelligently managed as virtual channels, permitting sustained utilization of the switching fabric in the vicinity of 90%.

Versions comprising eight processors, eight memory modules, eight switch modules and four Unix-based I/O processors should be operational in the Spring of 1991. Motorola will also be supporting a Unix-based single processor/single memory module workstation for Id program development.

Given the prototype nature of the existing hardware platform, the evaluation of the approach at this stage is mostly qualitative. Runtime management has been a particular challenge for large programs because, lacking specialized I-structure memory, all activation frames and heap data structures are drawn from the same frame store memory. As mentioned above, the Monsoon processor pipeline is general enough to emulate an I-structure memory module. All fetches are still split-phase, but rather than issuing a read or write request to a remote memory module, the request is recirculated into the pipeline and processed locally. We presently use 128 word activation frames. Free activation frames are kept on a shared free-list, so the frame `ALLOC-FRAME` and `DEALLOC-FRAME` operators expand to three instructions each that respectively pop from and push onto the free-list. The compiling discipline ensures that an activation frame being deallocated has all of its presence

bits set to **empty**. Half of the frame store memory is dedicated to the heap and accessed through *allocate* and *deallocate* library routines. Two experimental memory managers have been developed for the prototype: a simple first-fit manager with coalescing and a somewhat more sophisticated buddy system that permits simultaneous allocations and deallocations against the various free-lists.

In spite of the serious memory limitations, some surprisingly large codes have been executed on the first Monsoon prototype, including GAMTEB, a monte carlo histogramming simulation of photon transport and scattering in carbon cylinders. This code is heavily recursive and relatively difficult to vectorize. On Monsoon, a 40,000 particle simulation executed a little over one billion instructions. For comparison purposes, a scalar Fortran version of GAMTEB simulates 40,000 particles in 250 million instructions on a CRAY-XMP. We have found that about 50% of Monsoon instructions were incurred by the memory management system (the Fortran version uses static memory allocation). The remaining overhead of about a factor of two when compared with Fortran is consistent with our experience with other codes on the MIT tagged token dataflow architecture [3]. We are very encouraged by these preliminary data and expect marked future improvements in the memory management system and the overall dynamic efficiency of compiled code.

TSP is a simulated annealing approach to the traveling salesperson problem. The code performs implicit control over storage through user-defined object managers. The following statistics are typical of an iteration from a tour of fifty cities.

Fifty City TSP Tour on Monsoon		
Instruction Class	Total Cycles	Percentages
Fanouts and Identities	27,507,282	39.25
Arithmetic Operations	6,148,860	8.77
ALU Bubbles	20,148,890	28.75
I-Fetch Operations	3,590,992	5.12
I-Store Operations	285,790	0.41
Other Operations	8,902,202	12.70
Idles	3,503,494	5.00

Fanout and identities are the basic set of operations used for replicating data values and termination detection. These are roughly equivalent to **move** instructions in von Neumann machines. Arithmetic operations include both integer and floating point operations. ALU bubbles occur when the first-arriving operand of a two-input operator is written into a frame slot and and further processing of instruction is suppressed. Idles occur during a cycle where no tokens are produced and the token queues are empty.

The current Monsoon compiler is a retargeted version of the Id to TTDA graph compiler. This "first cut" essentially follows a transliteration of TTDA instructions into the Monsoon instruction set. It performs the static assignment of nodes to frame slots, but takes little advantage of the additional power of the ETS model. As such, we view the current application base as a proof of principle more than as a statement of potential performance.

Preliminary studies of frame slot assignment show the amount of reuse within a function body or loop iteration to be considerably smaller than expected. Roughly 60% of the instructions take a single input token and, therefore, need not be assigned a frame slot for matching. For the remaining dyadic instructions, we find that, on average, there are roughly 1.3 instructions per slot. This number is heavily dependent on the current compilation paradigm, especially the call linkage, which is unnecessarily general in many cases. We expect considerable improvement in this area with additional compiler work. There is, however, extensive reuse of entire frames in k-bounded loops.

The future architectural development of Monsoon will continue to explore fundamental improvements in dynamic instruction efficiency. It is important to note a basic mismatch in the Monsoon pipeline, characteristic of dataflow architectures. Each two-input instruction requires two operations against frame store, and thus two processor cycles, but only utilizes the ALU with the arrival of the second token. As suggested by the statistics above, approximately 30% of the ALU cycles are consumed by this mismatch (ALU bubbles). The introduction eight sets of temporary registers — one associated with each recirculating token in the pipeline — appears to offer significant efficiency improvement for short instruction sequences, possibly eliminating a large fraction of ALU bubbles and identities. These should also reduce the overall frame size by eliminating many of the match operations.

6 Related Work and Conclusions

The development of ETS and Monsoon rests heavily on previous dataflow architectures and on advancements in compilation techniques for functional languages. In this section, we trace some of the roots of this development, noting key similarities and differences. In addition, comparisons are drawn with contemporary dataflow machines.

The use of a directly addressed token store with presence bits is represented by the earliest static dataflow architectures[18, 22, 21, 40]. In the static approach, each instruction is a template containing the opcode, a list of destination addresses and slots to hold operands. While the work established the fundamentals of the dataflow execution paradigm, the static model has several shortcomings. The entire graph must be expanded at compile-time, thus general recursion is disallowed, and at most one token can exist on each arc at any time. To meet the latter requirement, the dataflow graph is embellished with *acknowledgement arcs*. Processing of acknowledgements complicates the scheduling mechanism, increases the computational overhead, and decreases the available parallelism. A merge operator is required for conditionals and loops, which has a complex firing rule. At any point in the execution of a program, the fraction of operand slots occupied by data values is likely to be quite small, so the utilization of the template store is expected to be low. Since, each instruction implicitly addresses the operand slots in its template, distinct instruction cannot share operand storage.

Key differences in the ETS include collecting the operand storage associated with a code-block into an activation frame, which is allocated dynamically, and providing a means of token indirection, `SEND` and `EXTRACT-TAG`. This provides full generality, while simplifying the machine and improving storage utilization. Having the instruction specify the frame slot allows slot reuse, but also supports traditional imperative execution. There is no machine-level concept of acknowledgement; proper reuse of operand storage is assured at a higher-level in compiling graphs. This compilation discipline evolved directly out of work on tagged-token dataflow architectures.

Tagged-token architectures sought to overcome the shortcomings of static architectures by *naming* tokens and accessing them by name, rather than by address. The name (or tag) includes a static part, identifying the instruction that is to use the value, and a dynamic part, identifying a particular instance of that instruction. Code is separated from data, so the program is re-entrant. The dynamic component of the tag supports recursion and iteration, with arbitrary overlap of iterations. The single token per arc restriction is removed, allowing greater parallelism and avoiding acknowledgement processing. Instructions are scheduled by matching tags. Result tags are generated from input tags by a set of simple rules. Perhaps the most important contribution of the tagged-token work is the development of a set of simple graph schemas representing the basic constructs in a high-level language that allow consistent generation of tags. This includes a clear

notion of code-block and code-block invocation. In theory, the model could be realized by simply treating the tag as an operand address, but the tag space is much too large for this to be practical. Instead, tagged-token architectures use an associative matching store to detect enabled operations; this need only be large enough to hold the tokens in existence, including their tags.

In developing the MIT Tagged-Token Dataflow Architecture (TTDA), several factors motivated further refinement of the compilation paradigm, in order to reduce the size of the tag and allow tags and resources to be more closely related[4]. One was the obvious reduction in storage overhead and data path width. A second factor was the importance of loop constants; circulating constant values produced considerable overhead that was easily avoided by associating a small amount of state with each loop invocation. A third was the use of mapping parameters describing how a code-block invocation was distributed over a collection of processors. The TTDA provided code-block registers containing the base addresses for the code-block and the loop-constant area, along with mapping information. Sigma-1[28, 29] provides similar features. In associating state with each invocation, it became necessary to embellish the graph schema to detect when an invocation was complete and the resources associated with the invocation could be released. Extending this to detect the completion of iterations allowed small iteration identifiers to be used and, more importantly, provided a means of avoiding excessive use of matching-store resources. In compiling programs to dataflow graphs in this manner, the associative matching store was no longer required, since the tag space was used more densely. It was natural to determine frame-slot assignments at compile-time, as well. ETS carries this line of development slightly further in placing the frame offset in the instruction, rather than on the token, so that execution of imperative threads is easily represented.

The EM-4[42], a successor of Sigma-1, developed at the Electrotechnical Laboratory in Japan, is contemporary with Monsoon. It is a smaller scale, but more highly integrated, design and represents a similar path of architectural evolution. It eliminates the hash-based matching store of the Sigma-1 in favor of direct access into an activation frame. The current EM-4 implementation consists of 80 processors with on-chip routers in fifteen groups of five. Each group includes a conventional maintainance processor for system support. The processor is a 38-bit tagged architecture, with 20-bit local addresses. Floating-point operations and integer multiply are performed in software. Instructions, activation frames, data structures, and buffered network packets are held in a common local memory with no caches, thus the match and initiation of an operation may require several DRAM accesses, whereas Monsoon can attempt a match every cycle.

In EM-4, each match initiates the execution of a sequence of instructions — the nodes in the dataflow graph whose inputs are completely determined by the two matched values. These are processed by a fairly conventional execution unit. Instructions within a sequence may reference either of the two input values or local registers, holding temporaries within the sequence. Although it employs direct matching, EM-4 is incomplete as an ETS architecture, because instructions cannot explicitly refer to slots within the activation frame. There is no provision for reusing frame slots. However, the macro operations with local registers are expected to reduce the overall frame size. The Monsoon temporaries and direct recirculation provide a similar function, but several Monsoon sequences are interleaved in the pipeline.

Epsilon2[25], currently under development at Sandia National Laboratory, is an extention of the Epsilon static dataflow machine[24] and early ETS work at MIT. An instruction can refer to two distinct operand frame slots and a third synchronization slot. A "repeat on input" mechanism allows a sequence of instructions to be initiated and ALU registers may hold temporaries within a sequence. There is no conditional branching within a sequence. Instructions cannot explicitly write to the activation frame, a separate token must be generated to deposit a value. The processor supports 64-bit operations with tag and pointer formats similar to Monsoon.

A separate line of research generalized the static dataflow model by dynamically splicing the

graph to support recursion[48]. VIM advances these ideas by separating the program and data portions of the instruction template, so that splicing is implemented by allocating an operand frame. Representing iteration by cyclic graphs presents difficulties under this approach and is generally eliminated in favor of tail-recursion.

Graph reduction models closely resemble graph splicing; in invoking a function, a copy of the graph representing the function is produced with arguments values substituted in place of formal parameters[17, 33]. The copy of the function body, represented as a linearized graph, may be viewed as an activation frame. Each slot contains an instruction that will eventually be reduced to a value and state bits that record the disposition of the slot. Instructions do not explicitly update the frame.

Several researchers have suggested that dataflow and von Neumann machines lie at two ends of an architectural spectrum[13, 30, 38]. In reflecting upon the development of Monsoon, our view is slightly different. Dataflow architectures and modern RISC machines represent orthogonal generalizations of the single accumulator "von Neumann" machine. The mainstream architectural trend enhances the power of a single execution thread with multiple addresses per operation. As we have shown here, dataflow execution essentially represents multiple 1-address execution threads, with a very simple synchronization paradigm. Having made the transition from propagating values through graphs to "virtual" processors, we can begin to address the question of what is the best processor organization to "virtualize." Certainly there are gains to be made by incorporating more powerful operand specification, but this must be weighed against additional cost and complexity in the synchronization paradigm. Recently, attention has been paid to multithreaded variants of a full 3-address load/store architecture to tolerate latency on a cache miss[47]. Considerable complexity is contemplated to address only the latency aspect of parallel computing and it is not obvious that a simple, inexpensive synchronization mechanism can be provided in this context.

It is likely that the optimal building block for scalable, general purpose parallel computers will represent a compromise between the two major directions of architectural evolution — trading some single thread efficiency for more efficient synchronization and vice versa. We must also address the question of how virtual processors are scheduled onto physical processors. Monsoon represents a viewpoint in which scheduling is directly supported by hardware, in the form of token queues. Perhaps another dimension of architectural variation is the manner in which the communication is integrated with the processor instruction set. For example, in many shared-memory architectures communication is invisble, with the cache lying between the processor and the network. Under ETS, operations that may potentially involve communication with remote processor or memory modules are represented by explicit sends to appropriate logical entities, however, the physical transport is transparent. Although various architectural alternatives can be weighed on engineering grounds, the final judgement can only be made with considerable experience on large scale parallel programs and the management and programming challenges they represent.

Acknowledgements

This work reflects considerable contributions of many members and past members of the Computation Structures Group, led by Prof. Arvind, including R. S. Nikhil, Andy Boughton, Ken Traub, Jonathan Young, Paul Barth, Stephen Brobst, Steve Heller, Jamey Hicks, Bob Iannucci, Andy Shaw, Richard Soley, Ken Steele, Jack Costanza, and Ralph Tiberio. Special thanks to our growing user community, including Olaf Lubeck of LANL, and to Motorola Inc. for their continuing support. We are grateful to the reviewers for many helpful comments.

The research was performed primarily at the MIT Laboratory for Computer Science and partly

at the University of California, Berkeley. Funding for the Monsoon project is provided in part by the Advanced Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099. D. Culler was supported in part by an NSF Presidential Young Investigator award and Motorola Inc.

References

[1] Arvind and D. E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester, England.* North-Holland Publishing Company, July 1985.

[2] Arvind and D. E. Culler. Dataflow Architectures. In *Annual Reviews in Computer Science*, volume 1, pages 225–253. Annual Reviews Inc., Palo Alto, CA, 1986. Reprinted in Dataflow and Reduction Architectures, S. S. Thakkar, editor, IEEE Computer Society Press, 1987.

[3] Arvind, D. E. Culler, and K. Ekanadham. The Price of Asynchronous Parallelism: an Analysis of Dataflow Architectures. In *Proceedings of CONPAR 88*, Univ. of Manchester, September 1988. British Computer Society — Parallel Processing Specialists. (also CSG Memo No. 278, MIT Laboratory for Computer Science).

[4] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The Tagged Token Dataflow Architecture. Technical Report FLA memo, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, August 1983. Revised October, 1984.

[5] Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos. PROJECT DATAFLOW, a Parallel Computing System Based on the Monsoon Architecture and the Id Programming Language. Technical Report CSG Memo 285, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA, 1988.

[6] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *The Journal of Parallel and Distributed Computing*, 5(5):460–493, October 1988.

[7] Arvind and K. P. Gostelow. The U-interpreter. *IEEE Computer*, 15(2), February 1982.

[8] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proceedings of the Fourth International Symposium on Biological and Artificial Intelligence Systems*, pages 255–286, Trento, Italy, September 1988. ESCOM (Leider).

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

[10] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259).* Springer-Verlag, June 1987. To appear in IEEE Transactions on Computers, revised June, 1988.

[11] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, February 1987. (Also appears in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[12] R. G. Babb II, editor. *Programming Parallel Processors.* Addison-Wesley Pub. Co., Reading, Mass., 1988.

[13] L. Bic. A Process-Oriented Model for Efficient Execution of Dataflow Programs. In *Proceedings of the 7th International Conference on Distributed Computing*, Berlin, West Germany, September 1987.

[14] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation via Coloring. *Computer Languages*, 6:47–57, 1981.

[15] D. E. Culler. Resource Management for the Tagged-Token Dataflow Architecture. Technical Report TR-332, MIT Laboratory for Computer Science, January 1985. (MS Thesis, Dept. of EECS, MIT).

[16] D. E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs.* PhD thesis, MIT Dept. of Electrical Engineering and Computer Science, Cambridge, MA, June 1989. To appear as MIT Laboratory for Computer Science technical report, TR446.

[17] J. Darlington and M. Reeve. ALICE - A Multi-Processor Reduction Machine for Parallel Evaluation of Applicative Languages. In *Proc. of the 1981 Conference on Functional Programming and Computer Architecture*, pages 65–76, 1981.

[18] A. L. Davis. The Architecture and System Methodology of DDM1 : A Recursively Structured Data Driven Machine. In *Proceedings of the 5th Annual International Symposium on Computer Architecture*, pages 210–215, April 1978.

[19] J. Deminet. Experience in Multiprocessor Algorithms. *IEEE Transactions on Computers*, C-31, April 1982.

[20] J. B. Dennis. First Version of a Data Flow Procedure Language. In G. Goos and J. Hartmanis, editors, *Proc. Programming Symposium, Paris 1974 (Lecture Notes in Computer Science 19)*, New York, 1974. Spinger-Verlag. (Revised: MAC TM61, May 1975, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[21] J. B. Dennis. Data Flow Supercomputers. *IEEE Computer*, pages 48–56, November 1980.

[22] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Dataflow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, page 126. IEEE, January 1975.

[23] J. Gaudiot and Y.H. Wei. Token Relabeling in a Tagged Dataflow Architecture. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[24] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon Dataflow Processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[25] V. G. Grafe and J. E. Hoch. The Epsilon-2 Hybrid Dataflow Architecture. In *Proceedings of Compcon90*, pages 88–93, March 1990.

[26] J. 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.

[27] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4), July 1988.

[28] K. Hiraki, K. Nishida, S. Sekiguchi, and T. Shimada. Maintainence Architecture and its LSI Implementation of a Dataflow Computer with a Large Number of Processors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 584–591, 1986.

[29] K. Hiraki, S. Sekiguchi, and T. 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.

[30] R. A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, May 1988. (PhD Thesis, Dept. of EECS, MIT).

[31] C. F.Joerg Design and Implementation of a Packet Switched Routing Chip. MS Thesis, MIT Department of Electrical Engineering and Computer Science, 77 Massachusetts Avenue, Cambridge, MA, May 1990.

[32] A. H. Karp. Programming for Parallelism. *IEEE Computer*, 20(5):43–56, May 1987.

[33] R. M. Keller, G. Lindstrom, and S. Patil. A Loosely-Coupled Applicative Multi-Processing System. In *Proceedings of the National Computer Conference*, volume 48, pages 613–622, New York, NY, June 1979.

[34] O. M. Lubeck. A User's View of Dataflow Architectures. In *Proceedings of COMPCON90*, pages 84–87, April 1990.

[35] D. A. Mandell and H. E. Trease. Parallel Processing a Real Code — A Case History. Technical Report LA-UR 88-1836, Los Alamos National Laboratory, 1988. In Proceedings of the LANL Workshop on Instrumentation for Future Parallel Systems.

[36] D. Nicol and J. Saltz. Dynamic Remapping of Parallel Computations with Varying Resource Demands. *IEEE Transactions on Computers*, 37(9):1073–1087, September 1988.

[37] R. S. Nikhil. Id (Version 88.0) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, March 1988.

[38] R. S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, May 1989. To appear.

[39] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report TR432, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[40] J. Rumbaugh. A Data Flow Multiprocessor. *IEEE Transactions on Computers*, C-26(2):138–146, February 1977.

[41] Michel M. Sadoune. *Terminal Area Flight Path Generation Using Parallel Constraint Propagation.* PhD thesis, Massachusetts Institute of Technology, Department of Areonautics and Astronautics, 1989.

[42] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, Jerusalem, Israel, June 1989.

[43] T. Shimada, K. Hiraki, and K. Nishida. An Architecture of a Data Flow Machine and its Evaluation. In *Proceedings of CompCon 84*, pages 486–490. IEEE, 1984.

[44] K. M. Steele Implementation of an I-Structure Memory Controller. Technical Report TR471, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, March 1990. (MS Thesis, Dept. of EECS, MIT).

[45] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, August 1986. (MS Thesis, Dept. of EECS, MIT).

[46] K. R. Traub. Sequential Implementation of Lenient Programming Languages. Technical Report TR-417, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, September 1988. (PhD Thesis, Dept. of EECS, MIT).

[47] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 1989 International Symposium on Computer Architecture*, pages 273–280, Jerusalem, Israel, May 1989.

[48] K. Weng. An Abstract Implementation for a Generalized Data Flow Language. Technical Report MIT/LCS/TR-228, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 1979. (PhD Thesis, Dept. of EECS, MIT).