Monsoon: an Explicit Token-Store Architecture

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

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

Abstract

Dataflow architectures tolerate long unpredictable communication delays and support generation and coordination of parallel activities directly in hardware, rather than assuming that program mapping will cause these issues to disappear. However, the proposed mechanisms are complex and introduce new mapping complications. This paper presents a greatly simplified approach to dataflow execution, called the *explicit token store* (ETS) 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. It may be viewed as a generalization of a fairly primitive von Neumann architecture. Although the addressing capability is restrictive, there is exactly one instruction executed for each action on the dataflow graph. Thus, the machine oriented ETS model provides new understanding of the merits and the real cost of direct execution of dataflow graphs.

1 Introduction

The Explicit Token Store (ETS) architecture is an unusually simple model of dynamic dataflow execution that is realized in *Monsoon*, a large-scale dataflow multiprocessor[27]. A Monsoon processor prototype is operational at the MIT Laboratory for Computer Science, running large programs compiled from the dataflow language Id. A full-scale multiprocessor system is under development[6] in conjunction with Motorola Inc. Formulation of the ETS architecture began in 1986 as an outgrowth of the MIT Tagged-Token Dataflow Architecture (TTDA) and was based on a family of design goals — some evolutionary, some revolutionary, and some reactionary.

The fundamental properties of the TTDA that we wanted to retain included a large synchronization namespace, inexpensive synchronization, and tolerance to memory and communication latency[9]. These properties do not improve peak performance, but they dictate how much of it is actually delivered on complex applications[5]. To obtain high performance on a parallel machine lacking these features, a program must be partitioned into a small number of processes that operate almost entirely on local data and rarely interact[10,21]. However, if highly parallel machines are to be used in solving problems more complex than what is addressed on current supercomputers, it is likely they will have to be programmed in a very high level language with little explicit programmer management of parallelism[7,8], and the behavior of these complex applications may be very dynamic. Together, these observations suggest that we cannot expect to achieve a perfect mapping for many applications that we will want to execute on a highly parallel machine, so we are studying the design of processors that tolerate an imperfect mapping and still obtain high performance.

Tagged-token dataflow machines achieve this tolerance through sophisticated matching hardware, which dynamically schedules operations with available operands[2,20,29]. When a token arrives at a processor, the tag it carries is checked against the tags present in the token-store. If a matching token is found, it is extracted and the corresponding instruction is enabled for execution; otherwise, the incoming token is added to the store. This allows for a simple non-blocking processor pipeline that can overlap instructions from closely related or completely unrelated computations. It also provides a graceful means of integrating asynchronous, perhaps misordered, memory responses into the normal flow of execution. However, the matching operation involves considerable complexity on the critical path of instruction scheduling[18]. Although progress has been made in matching hardware[20,29], our goal was to achieve the benefits of matching with a fundamentally simpler mechanism.

The more subtle problem with the matching paradigm is that failure to find a match *implicitly* allocates resources within the token store. Thus, in mapping a portion of the computation to a processor, an unspecified commitment is placed on the token store of that processor and, if 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. The worst-case token storage requirement can be determined on a per-code-block basis with sophisticated compiler analysis, but the "bottom line" is that this complex mechanism fails to simplify resource management. Thus, engineering and management concerns led us to consider how to make the token-store *explicit* in the dataflow model.

CH2887-8/90/0000/0082$01.00 © 1990 IEEE

The result is a simple architecture that directly executes dataflow graphs, yet can be understood as a variation on a (fairly primitive) von Neumann machine. It is simple enough to build and serves to clarify many aspects of dataflow execution.

The sequel describes the ETS architecture and its realization in Monsoon. Section 2 outlines the basic execution model. Section 3 describes the Monsoon implementation. Sections 4 and 5 provide preliminary measurements of programs on this machine and reflections on the evolution of dataflow architectures.

2 ETS Architecture

The central idea in the ETS model is that storage for tokens is dynamically allocated in sizable blocks, with detailed usage of locations within a block determined at compile time. When a function is invoked, an *activation frame* is allocated explicitly, as part of the calling convention, to provide storage for all tokens generated by the invocation. Arcs in the graph for the function are statically mapped onto slots in the frame by coloring the graph, much like modern register assignment[12]. The basic structure of an executing program is illustrated in Figure 1. 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.*, SUB), the offset in the activation frame where the match will take place (*e.g.*, FP+3), and one or more destination instructions that will receive the result of the operation (*e.g.*, instructions IP+1 and IP+2). An input port (left/right) is specified with each destination.

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 1. If the slot is empty, the value on the token is deposited in the slot (making it full) and no further processing of the instruction takes place. If it is full, 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. 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 upon completion of the invocation.

The ETS activation frame is obviously similar to a conventional call frame. The differences are that presence-bits are associated with each slot and that an executing program generates a *tree* of activation frames, rather than a stack, because a procedure may generate parallel calls where the caller and the callees execute concurrently. The concurrent callees may themselves generate parallel calls, and so on. For loops, several frames are allocated, so that many iterations can execute concurrently[1,13], and reused efficiently. Graphs are compiled such that a frame is not reused until the previous uses of it are complete.

The ETS execution model is easily formalized in terms of a linear array of locations, M, such that the i^{th} location, M[i], contains $q.v$, where v is a fixed size value and q is the **status** of location M[i]. The status and value parts of a location may be manipulated independently. The one operation defined on the status part is an atomic read-modify-write, $M[i].q \leftarrow \mathcal{S}(M[i].q)$, where $\mathcal{S}$ is a simple transition function. Three atomic operations are defined on the value part: *read*, *write*, *exchange*. In addition to the store, the state of the machine includes a set of tokens. The pair of pointers FP.IP is a valid data value, so indirect references and control transfers are possible. Every token in the set of unprocessed tokens represents an operation that is ready to be executed. Thus, a parallel machine step involves selecting and processing some subset of the unprocessed tokens. This generates a set of updates to the store and new unprocessed tokens. The model is inherently parallel, as any number of operations may be performed in a step. Of course, in realizing the model, additional constraints will have to be imposed.

ETS instructions are essentially a 1-address form, in that one operand is the value carried on the token and the second is the contents of the location specified by a simple effective address calculation, *e.g.*, FP + r. The value part of the token functions as the accumulator, IP as the program counter, and FP as an index register. The unusual quality of the ETS instruction is that it may also specify a simple synchronization operation and multiple successors. The synchronization component is merely a state transition on the presence bits associated with the memory operand. However, the state transition affects the behavior of the instruction as a whole, possibly nullifying the continuation.

The simplest continuation is a single successor in the same code-block, specified relative to IP; this corresponds to a node with a single output arc. The fork continuation allows multiple tokens to be produced each carrying the result value and FP from the input token, but with different IPs, derived from that on the input token by a simple offset. To represent conditionals, the offset is selected based on one of the input values. To support the function call and return mechanism, the *Extract Tag* operation places the tag for a node (FP.IP+s), where s is a relative instruction offset, into the value part of the result token and *Send* uses the value part of one input as a result tag. Thus, program code is re-entrant.

The ETS is a dataflow architecture, in that it directly executes dynamic dataflow graphs. Operations in a tagged-token dataflow architecture correspond one-

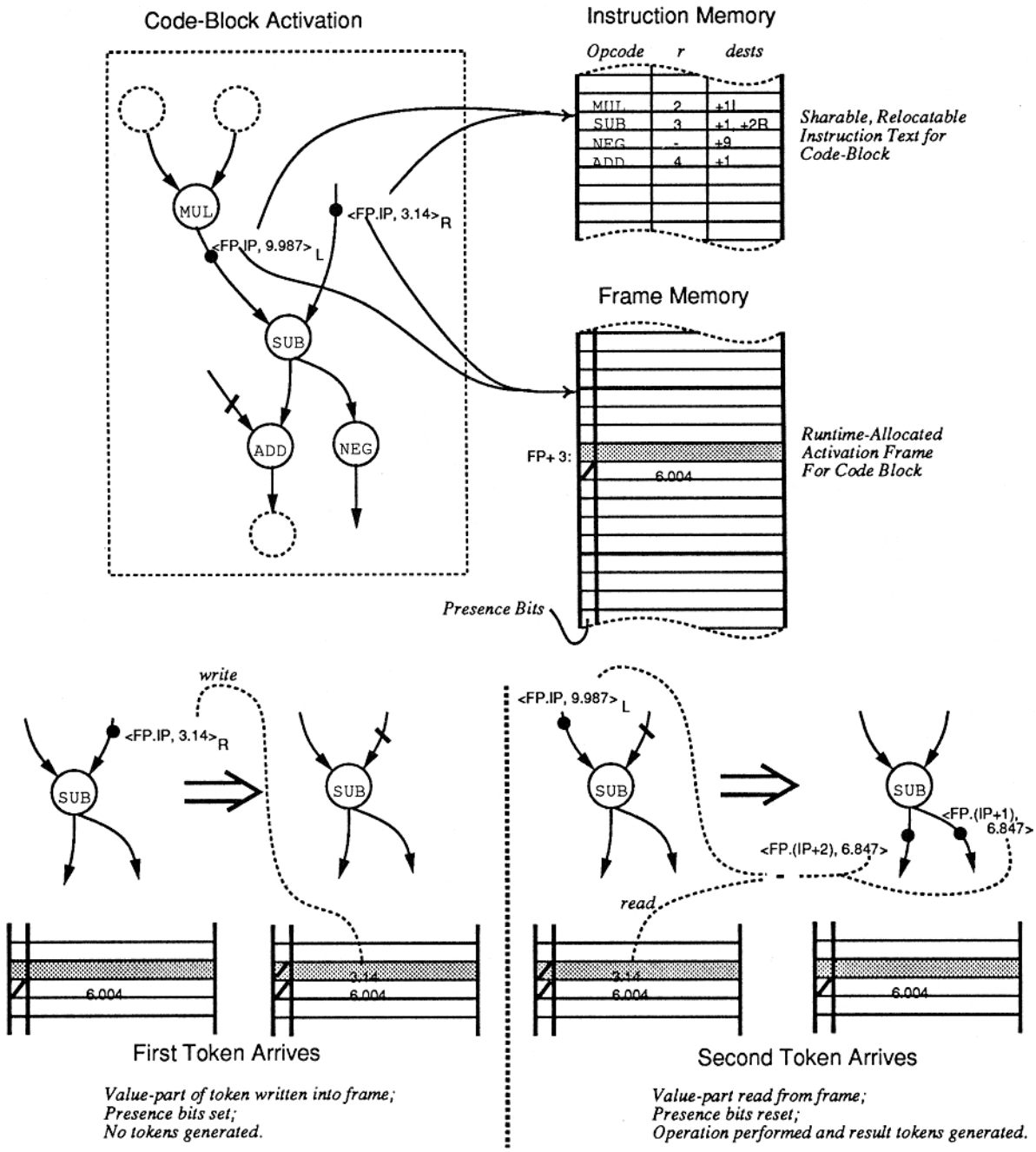

Figure 1: ETS Representation of an Executing Dataflow Program

to-one with operations in the ETS architecture. The data-driven scheduling mechanism is much simpler in the latter case, because the rendezvous point for the two operands is defined by a simple address calculation, rather than hash and match logic. However, making the token store explicit also makes the execution model more general. Using other state transitions on the presence-bits, it directly supports important extensions to the dynamic dataflow model, including loop constants, I-structures, and accumulators. By simply ignoring the presence bits, (multiple) imperative control threads are supported, as well. The overall execution schedule of an ETS program depends on a variety of run-time factors, however, by using dataflow graphs as a programming methodology, we are guaranteed that all execution schedules produce the same result.

3 Monsoon

Monsoon is a general purpose multiprocessor that incorporates an explicit token store. A Monsoon machine includes a collection of pipelined processing elements (PE's), connected via a multistage packet switch network to each other and to a set of interleaved I-structure memory modules (IS's), as shown in Figure 2. Messages in the interprocessor network are simply tokens—precisely the format used within the PE and IS. Thus, the hardware makes no distinction between inter- and intra-processor communication.

The ETS model suggests a natural form of locality; a given activation frame resides entirely on one PE. A code-block activation is dynamically bound to a particular processing element at invocation-time and executes to completion on that PE. Each concurrent iteration of a loop is assigned a separate activation frame, and these frames may be on separate PEs. This strategy reduces network traffic without squandering fine-grain parallelism—the parallelism within an activation is used to keep the processor pipeline full. The policy of mapping an activation frame to a single PE implies that interprocessor token traffic is only generated by data structure reads and writes and transmission of procedure arguments and return values. The interprocessor network is therefore appropriately sized to handle this fraction (less than 30%) of the total number of tokens produced during the course of a computation.

A Monsoon PE is a highly pipelined processor. On each processor cycle a token may enter the top of the pipeline and, after eight cycles, zero, one or two tokens emerge from the bottom[1]. In the process, an instruction is fetched from instruction memory, which reads or writes a word in the data memory called *frame-store*. 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 be inserted into one of two token queues or delivered to the interprocessor network and automatically routed to the correct PE.

[1] A processor *cycle* usually corresponds to a single processor clock, but may extend to multiple clocks during certain operations that cause a pipeline stall, *e.g.*, a frame-store exchange or a floating point divide. Tokens advance from one stage to the next only at cycle boundaries.

3.1 Machine Formats

A Monsoon token is a compact descriptor of computation state comprising a tag and a value. 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 a tag.

As in the ETS, a Monsoon tag encodes two pointers: one to the instruction to execute and one to the activation frame that provides the context in which to attempt execution of that instruction. However, since activation frames do not span PEs, the frame pointer and instruction pointer are conveniently segmented by processing element, TAG = PE:(FP.IP), where PE is the processing element number and IP and FP are *local* addresses on processor PE. Tags and values are both 64-bit quantities and each is appended with eight additional bits of run-time type information. A token is therefore a 144-bit quantity. For tags, the size of the PE field is eight bits, and FP and IP are 24 bits each. The machine automatically routes tokens to the specifed PE, whereupon instruction IP is fetched and frame FP is accessed. 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. 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.

A data structure pointer encodes an address on a processing element or I-structure memory module. Pointers are represented in a "normalized" format as the segmented address PE:OFFSET, where PE denotes the processing element or I-structure module number and OFFSET is a local address on the PE or module. Additionally, pointers contain *interleave information*, which describes how the data structure is spread over a collection of modules. The interleave information describes a *subdomain*[4], *i.e.*, collection of 2^n processors or memory modules which starts on a modulo 2^n PE number boundary. If $n = 0$ then increments to the pointer will map onto the same PE. If $n = 1$ then increments to the pointer alternate between PE and PE+1, and so on.

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, 32-

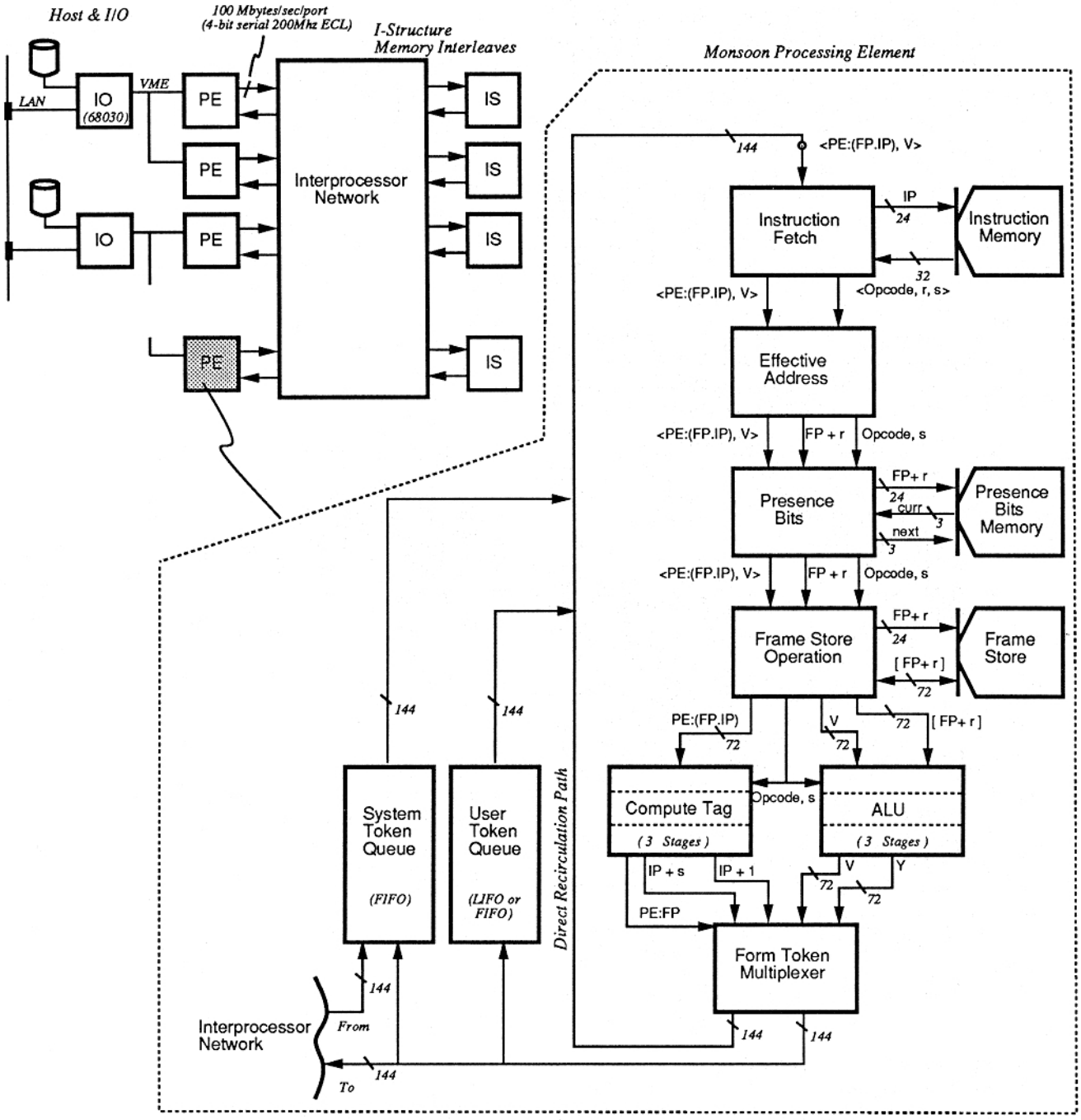

Figure 2: Monsoon Processing Element Pipeline

bit format, with a 12-bit opcode, a 10-bit operand, and a 10-bit destination field. The operand field, r, can be used as a frame-relative address in local frame store, FP + r, or as an absolute local address, to access literal constants and procedure linkage information kept in low memory by convention. The operand and destination fields can be combined to form a 20-bit address, as well. Every instruction can have up to two destinations, encoded as an adjustment to IP with an explicit PORT value. When an operand specifier is used, one of the destinations is IP+1. The opcode completely determines the interpretation of the other two fields. There are three presence (or status) bits associated with each word of memory to support data-driven scheduling.

3.2 Pipeline operation

The right-hand portion of Figure 2 describes the internal pipeline of the Monsoon processor, which operates as follows. (1) The IP from the incoming token is used as an address into local instruction memory. (2) The effective address of a location in frame-store is computed (FP + r or r). (3) The three presence bits associated with this frame-store location are read, modified (by a table lookup), and written back to the same location. The state transition function represented by the lookup depends on the port bit on the incoming token and the instruction opcode. It produces the new presence bits and two control signals for subsequent pipeline stages: one dictates whether the operation on the value part of the associated frame-store location is a read, write, exchange or no-op, and the other suppresses the generation of result tokens. For example, when the first token for a two-input operator is processed, the lookup specifies a write to the frame-store and supression of results. (4) Depending on the result of the lookup, the value part of the specified frame-store location is ignored, read, written, or exchanged with the value on the token.

The ALU represents three stages and operates in parallel with tag generation. (5) The value on the token and the value extracted from the frame-store are sorted into left and right values using the port bit of the incoming token. It is also possible to introduce the incoming tag as one of the ALU operands. (6,7) The operands are processed by one of the function units: a floating point/integer unit, a specialized pointer/tag arithmetic unit, a machine control unit or a type extract/set unit. Concurrent with the final ALU processing, two new result tags are computed by the next address generators.

(8) Finally, the form-token stage creates result tokens by concatenating the computed tags with the ALU result. During inter-procedure communication (*i.e.* call and return values and structure memory operations) 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 value, a delayed version of the "right" value, is also available to the form-token multiplexor. This stage detects whether PE of a result token tag is equal to the present processing element number. If not, the token is forwarded to the network and routed to the correct processing element or I-structure module. One of the (local) result tokens may be recirculated directly to the instruction fetch stage. If two local tokens are created, one of the result tokens is placed onto either the system or user token queue. If no tokens are created then a token is dequeued from one of the token queues for processing.

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.

4 Evaluation

A single processor Monsoon prototype has been operational at the MIT Laboratory for Computer Science since October 1988 and a second prototype is due to be delivered to the Los Alamos National Laboratories for further evaluation. Except for an interprocessor network connection, the prototype employs the synchronous eight stage pipeline and 72-bit datapaths presented in Section 3. The memory sizes are fairly modest: 128KWords (72 bits) of frame-store and 128KWords (32 bits) of instruction memory. The prototype was designed to process six million tokens per second, although we typically clock at one-half this rate for reliability reasons. 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 execute directly on Monsoon.

Runtime management has been a particular challenge for large programs because, lacking an I-structure memory module, all activation frames and heap data structures are drawn from the same frame-store memory. We presently use 128 word activation frames. Free activation frames are kept on a shared free-list, so the frame *alloc* and *release* operators expand to three instructions each. Half of the frame-store memory is dedicated to the heap and managed by *allocate* and *deallocate* library routines. Two experimental memory managers have been developed for the prototype: a simple first-fit manager (with coalescing) and a 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 Monsoon, 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 executes 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 encouraged by these preliminary data and expect marked future improvements in the memory management system and the overall dynamic efficiency of compiled code.

One of the non-scientific codes we have experimented with is a simulated annealing approach to the traveling salesperson problem, written in Id, but exercising 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 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. Idling occurs 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[30] and 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.

We are now working with the Motorola Microcomputer Division and the Motorola Cambridge Research Center to develop multiprocessor Monsoon prototypes. 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) and, significantly, have dedicated I-structure memory modules (4MWords) and a high-speed multistage packet switch (100 Mbytes/sec/port). Versions comprising eight processors and eight memory 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.

The ETS activation frame functions much like a conventional register set and, by ignoring the presence-bits, can be accessed as such. Of course, a *single* instruction of a three-address von Neumann processor could read two registers, perform an operation and write the result register, whereas Monsoon takes three instructions to accomplish the same action. Monsoon permits only a single frame-store operation per cycle. In a very real sense, the value part of a token corresponds to an accumulator—it can be loaded, stored, or operated upon, in combination with the local frame. However, from a hardware engineering viewpoint, the single port access to frame-store is an important restriction, since the frame-store simultaneously holds *thousands* of activation frames; three-port access would be prohibitively expensive. Competitive implementations of a Monsoon-like processor would certainly employ a cache of local frame memory; nonetheless, the single port frame-store suggests what might be an inherent inefficiency in the ETS model.

The future architectural development of Monsoon will continue to explore fundamental improvements in dynamic instruction efficiency. Part of this work addresses a basic mismatch in the Monsoon pipeline, that is 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). Observe, that a sequence of instructions that produce one local result at each step follows the direct

recirculation path, thus occupying one of eight processor interleaves. The new version of Monsoon provides a 3 × 72-bit four-port (two read, two write) temporary register set for each interleave. For simple arithmetic expressions, the temporary set can improve the dynamic instruction efficiency (the number cycles required to compute the expression) by a factor of two. Note, the temporaries are valid only as long as a thread has a recirculating token; when a token is first popped from a queue, the values of the temporaries are indeterminate. The temporaries are also invalidated when performing a split-phase read. These temporaries are very similar to the register set in Iannucci's hybrid architecture [23].

5 Related Work

In our view, the beauty of the ETS model and its realization in Monsoon lies in its simplicity, not its novelty. It draws heavily on developments in dynamic and static dataflow architectures, yet demystifies the dataflow execution model be providing a simple, concrete, machine-oriented formulation — one simple enough to build. Activation frames are certainly not a new idea. The use of presence-bits to detect enabled operations is represented in the earliest static dataflow architectures[15,16,28]. In those designs, instructions and operand slots were combined into an instruction template, which was delivered to a function unit when it was determined that the operand slots were full. Presence detection was performed by an autonomous unit, functioning asynchronously with the rest of the system, rather than simply treated as a stage in an instruction pipeline. Also, the utilization of the activity store was poor, because storage was preallocated for every operand slot in the entire program, even though the fraction of these containing data at any time is generally small. Other drawbacks included the complex firing rule of the merge operator, the need for acknowledgement arcs to ensure one token per arc, loss of parallelism due to artificial dependences, and the inability to support general recursion.

Tagged-token architectures addressed these problems by naming each activity in terms of its role in the computation, rather than by the resources used to perform the activity. Iteration and recursion is easily implemented by assigning new names for each activation of the loop or function. This eliminated the troublesome merge operator, the acknowledgement arcs, and the artificial dependences. Storage for operands was allocated "as needed" via the matching mechanism. In our own efforts to refine the MIT Tagged-Token Dataflow Architecture, the association between the name for an activity and the resources devoted to performing the activity became ever more immediate. Once state information was directly associated with each activation, it was a small step to eliminate the matching store. However, before it made sense to represent storage for operands directly, it was necessary to ensure that the utilization would be reasonable. This involved developments in compilation of loops[13], as well as frame-slot mapping.

A separate line of development generalized the static model by dynamically splicing the graph to support recursion[32]. VIM[17] advances these ideas by separating the program and data portions of the instruction template, so the splicing operations could be implemented by allocating an operand frame and providing a form of token indirection. Representation of iteration in this context presents problems and is generally eliminated in favor of tail-recursion.

The ETS model pulls these three areas together in an elegant fashion. The power of the tagged-token approach is provided with a simple mechanism, expressed in familiar terms. The mechanism is quite close to that which is used to support I-structures and provides a uniform means fo representing synchronizing data structure operations. Since the instruction determines how the operand store is accessed, it is straightforward to realize imperative control threads as well.

Graph reduction architectures provide an additional reference point, contemporary with the development of dataflow architectures and addressing a similar class of languages[14,25]. The function application mechanism under a reduction model closely resembles graph splicing, in that a copy of the function body is produced and arguments substituted where formal parameters appear. The copy of the function body can be viewed as an activation frame, the slots of which contain references to chunks of computation that will eventually be reduced to a value. In this sense, state information is associated with each slot in the frame to indicate whether it is reduced or not. Parallel graph reduction architectures require additional mechanisms for recording requests made for a value while it is being reduced. By demanding values before they are actually needed, data-driven scheduling can be simulated[24]. The rather primitive ETS mechanism can be used to support demand-driven execution as well[22], although we have not pursued that direction extensively. A detailed comparison between the two execution models is beyond the scope of this paper.

Several researchers have suggested that dataflow and von Neumann machines lie at two ends of an architectural spectrum[11,19,23,26]. In reflecting upon the development of Monsoon, our view is somewhat 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. Dataflow graphs essentially represent 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 complexity in synchronization. Recently, attention has been paid to multithreaded variants of a full 3-address load/store architecture to tolerate latency on a cache miss[31]. The proposed techniques range from a four-port register file to complete replication of the data path. Thus, considerable complexity is contemplated to address only the latency aspect of parallel computing. 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 combine the two major directions of architectural evolution, but may not be extreme in either direction.

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, Richard Soley, Bob Iannucci, Andy Shaw, 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.

The research was performed primarily at the MIT Laboratory for Computer Science and partly at the University of California, Berkeley. Funding for the 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.

References

[1] Arvind and D. E. Culler. Managing Resources in a Parallel Machine. In *Proc. 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 *Proc. of CONPAR 88*, Univ. of Manchester, September 1988. British Computer Society — Parallel Processing Specialists. (also CSG Memo No. 278, MIT Lab 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 Lab for Computer Science, 545 Tech. Sq, Cambridge, MA, August 1983. Revised October, 1984.

[5] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. *The Int'l Journal of Supercomputer Applications*, 2(3), November 1988.

[6] 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 Lab for Computer Science, 545 Tech. Sq, Cambridge, MA, 1988.

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

[8] Arvind, S. K. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Proc. of the Fourth Int'l Symp. 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 *Proc. of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 1987.

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

[11] L. Bic. A Process-Oriented Model for Efficient Execution of Dataflow Programs. In *Proc. of the 7th Int'l Conference on Distributed Computing*, Berlin, West Germany, September 1987.

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

[13] 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 Lab for Computer Science TR446.

[14] 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.

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

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

[17] J. B. Dennis, J. E. Stoy, and B. Guharoy. VIM: An Experimental Multi-User System Supporting Functional Programming. In *Proc. of the 1984 Int'l Workshop on High-Level Computer Architecture*, pages 1.1–1.9, Los Angeles, CA, May 1984.

[18] D.D. Gajski, D.A. Padua, David J. Kuck, and R.H. Kuhn. A Second Opinion of Data Flow Machines and Languages. *IEEE Computer*, 15(2):58–69, February 1982.

[19] V. G. Grafe, J. E. Hoch, and Davidson G.S. Eps'88: Combining the Best Features of von Neumann and Dataflow Computing. Technical Report SAND88-3128, Sandia National Laboratories, January 1989.

[20] 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.

[21] 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.

[22] S. K. Heller. Efficient lazy data-structures on a dataflow machine. Technical Report LCS/MIT/TR-438, MIT Lab for Computer Science, 545 Tech. Sq, Cambridge, MA, 1988. (PhD Thesis, Dept. of EECS, MIT).

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

[24] R. M. Keller and F. C. Lin. Simulated Performance of a Reduction-Based Multiprocessor. *IEEE Computer*, pages 70–82, July 1984.

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

[26] R. S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proc. of the 16th Annual Int'l Symp. on Computer Architecture*, Jerusalem, Israel, May 1989. To appear.

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

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

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

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

[31] W. Weber and A. Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proc. of the 1989 Int'l Symp. on Computer Architecture*, pages 273–280, Jerusalem, Israel, May 1989.

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