Dataflow Architectures and Multithreading

Ben Lee, Oregon State University
A.R. Hurson, Pennsylvania State University

Contrary to initial expectations, implementing dataflow computers has presented a monumental challenge. Now, however, multithreading offers a viable alternative for building hybrid architectures that exploit parallelism.

The dataflow model of execution offers attractive properties for parallel processing. First, it is asynchronous: Because it bases instruction execution on operand availability, synchronization of parallel activities is implicit in the dataflow model. Second, it is self-scheduling: Except for data dependencies in the program, dataflow instructions do not constrain sequencing; hence, the dataflow graph representation of a program exposes all forms of parallelism, eliminating the need to explicitly manage parallel execution. For high-speed computations, the advantage of the dataflow approach over the control-flow method stems from the inherent parallelism embedded at the instruction level. This allows efficient exploitation of fine-grain parallelism in application programs.

Due to its simplicity and elegance in describing parallelism and data dependencies, the dataflow execution model has been the subject of many research efforts. Since the early 1970s, a number of hardware prototypes have been built and evaluated,[1] and different designs and compiling techniques have been simulated.[2] The experience gained from these efforts has led to progressive development in dataflow computing. However, a direct implementation of computers based on the dataflow model has been found to be an arduous task.

Studies from past dataflow projects revealed some inefficiencies in dataflow computing.[2] For example, compared to its control-flow counterpart, the dataflow model's fine-grained approach to parallelism incurs more overhead in instruction cycle execution. The overhead involved in detecting enabled instructions and constructing result tokens generally results in poor performance in applications with low degrees of parallelism. Another problem with the dataflow model is its inefficiency in handling data structures (for example, arrays of data). The execution of an instruction involves consuming tokens at the input arcs and generating result tokens at the output arcs. Since tokens represent scalar values, the representation of data structures (collections of many tokens) poses serious problems.

In spite of these shortcomings, we're seeing renewed interest in dataflow computing. This revival is facilitated by a lack of developments in the conventional parallel-processing arena, as well as by changes in the actual implementation of the dataflow model. One important development, the emergence of an efficient mechanism to detect enabled nodes, replaces the expensive and complex process of matching tags used in past projects. Another change is the convergence of the control-flow and dataflow models of execution. This shift in viewpoint allows incorporation of conventional control-flow thread execution into the dataflow approach, thus alleviating the inefficiencies associated with the pure-dataflow method. Finally, some researchers suggest supporting the dataflow concept with appropriate compiling technology and program representation instead of specific hardware. This view allows implementation on existing control-flow processors. These developments, coupled with experi-

0018-9162/94/ $4.00 © 1994 IEEE

mental evidence of the dataflow approach's success in exposing substantial parallelism in application programs, have motivated new research in dataflow computing.[3] However, issues such as program partitioning and scheduling and resource requirements must still be resolved before the dataflow model can provide the necessary computing power to meet today's and future demands.

Past dataflow architectures

Three classic dataflow machines were developed at MIT and the University of Manchester: the Static Dataflow Machine,[2] the Tagged-Token Dataflow Architecture (TTDA),[1] and the Manchester Machine.[2] These machines — including the problems encountered in their design and their major shortcomings — provided the foundation that has inspired many current dataflow projects.

Static versus dynamic architectures. In the abstract dataflow model, data values are carried by tokens. These tokens travel along the arcs connecting various instructions in the program graph. The arcs are assumed to be FIFO queues of unbounded capacity. However, a direct implementation of this model is impossible. Instead, the dataflow execution model has been traditionally classified as either static or dynamic.

Static. The static dataflow model was proposed by Dennis and his research group at MIT.[2] Figure 1 shows the general organization of their Static Dataflow Machine. The activity store contains instruction templates that represent the nodes in a dataflow graph. Each instruction template contains an operation code, slots for the operands, and destination addresses (Figure 2). To determine the availability of the operands, slots contain presence bits. The update unit is responsible for detecting instructions that are available to execute. When this condition is verified, the unit sends the address of the enabled instruction to the fetch unit via the instruction queue. The fetch unit fetches and sends a complete operation packet containing the corresponding opcode, data, and destination list to one of the operation units and clears the presence bits. The operation unit performs the operation, forms result tokens, and sends them to the update unit. The update unit stores each result in the appropriate operand slot and checks the presence bits to determine whether the activity is enabled.

Dynamic. The dynamic dataflow model was proposed by Arvind at MIT[1] and by Gurd and Watson at the University of Manchester.[2] Figure 3 shows the general organization of the dynamic dataflow model. Tokens are received by the matching unit, which is a memory containing a pool of waiting tokens. The unit's basic operation brings together tokens with identical tags. If a match exists, the corresponding token is extracted from the matching unit, and the matched token set is passed to the fetch unit. If no match is found, the token is stored in the matching unit to await a partner. In the fetch unit, the tags of the token pair uniquely identify an instruction to be fetched from the program memory. Figure 4 shows a typical instruction format for the dynamic dataflow model. It consists of an operational code, a literal/constant field, and destination fields. The instruction and the token pair form the enabled instruction, which is sent to the processing unit. The processing unit executes the enabled instructions and produces result tokens to be sent to the matching unit via the token queue.

Centralized or distributed. Dataflow architectures can also be classified as centralized or distributed, based on the organization of their instruction memories.[1] The Static Dataflow Machine and the Manchester Machine both have centralized memory organizations. MIT's dynamic dataflow organization is a multiprocessor system in which the instruction memory is distributed among the processing elements. The choice between centralized or distributed memory organization has a direct effect on program allocation.

Comparison. The major advantage of the static dataflow model is its simplified mechanism for detecting enabled nodes. Presence bits (or a counter) determine the availability of all required operands. However, the static dataflow model has a performance drawback when dealing with iterative constructs and reentrancy. The attractiveness of the dataflow con-

Definitions

Dataflow execution model. In the dataflow execution model, an instruction execution is triggered by the availability of data rather than by a program counter.

Static dataflow execution model. The static approach allows at most one instance of a node to be enabled for firing. A dataflow actor can be executed only when all of the tokens are available on its input arc and no tokens exist on any of its output arcs.

Dynamic dataflow execution model. The dynamic approach permits activation of several instances of a node at the same time during runtime. To distinguish between different instances of a node, a tag associated with each token identifies the context in which a particular token was generated. An actor is considered executable when its input arcs contain a set of tokens with identical tags.

Direct matching. In a direct matching scheme, any computation is completely described by a pointer to an instruction (IP) and a pointer to an activation frame (FP). A typical instruction pointed to by an IP specifies an opcode, an offset in the activation frame where the match will take place, and one or more displacements that define the destination instructions that will receive the result token(s). The actual matching process is achieved by checking the disposition of the slot in the frame. If the slot is empty, the value of the token is written in the slot and its presence bit is set to indicate that the slot is full. If the slot is already full, the value is extracted, leaving the slot empty, and the corresponding instruction is executed.

Thread. Within the scope of a dataflow environment, a thread is a sequence of statically ordered instructions such that once the first instruction in the thread is executed, the remaining instructions execute without interruption (synchronization occurs only at the beginning of the thread). **Multithreading** implies the interleaving of these threads. Interleaving can be done in many ways, that is, on every cycle, on remote reads, and so on.

cept stems from the possibility of concurrent execution of all independent nodes if sufficient resources are available, but reentrant graphs require strict enforcement of the static firing rule to avoid nondeterminate behavior.[1] (The static firing rule states that a node is enabled for firing when a token exists on each of its input arcs and *no token exists on its output arc.*) To guard against nondeterminacy, extra arcs carry acknowledge signals from consuming nodes to producing nodes. These acknowledge signals ensure that no arc will contain more than one token.

The acknowledge scheme can transform a reentrant code into an equivalent graph that allows pipelined execution of consecutive iterations, but this transformation increases the number of arcs and tokens. More important, it exploits only a limited amount of parallelism, since the execution of consecutive iterations can never fully overlap — even if no loop-carried dependencies exist. Although this inefficiency can be alleviated in case of loops by providing multiple copies of the program graph, the static dataflow model lacks the general support for programming constructs essential for any modern programming environment (for example, procedure calls and recursion).

The major advantage of the dynamic dataflow model is the higher performance it obtains by allowing multiple tokens on an arc. For example, a loop can be dynamically unfolded at runtime by creating multiple instances of the loop body and allowing concurrent execution of the instances. For this reason, current dataflow research efforts indicate a trend toward adopting the dynamic dataflow model. However, as we'll see in the next section, its implementation presents a number of difficult problems.

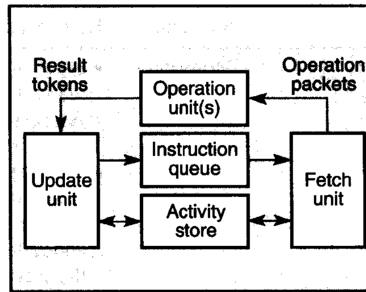


Figure 1. The basic organization of the static dataflow model.

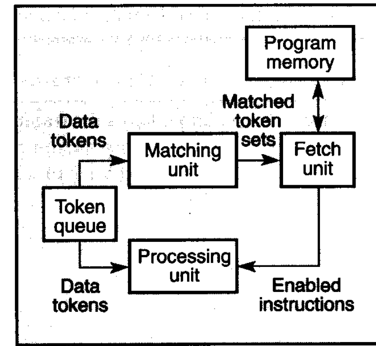


Figure 3. The general organization of the dynamic dataflow model.

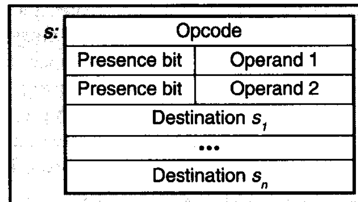


Figure 2. An instruction template for the static dataflow model.

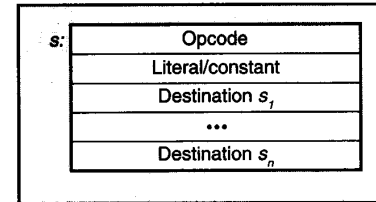


Figure 4. An instruction format for the dynamic dataflow model.

Lessons learned. Despite the dynamic dataflow model's potential for large-scale parallel computer systems, earlier experiences have identified a number of shortcomings:

- overhead involved in matching tokens is heavy,
- resource allocation is a complicated process,
- the dataflow instruction cycle is inefficient, and
- handling data structures is not trivial.

Detection of matching tokens is one of the most important aspects of the dynamic dataflow computation model. Previous experiences have shown that performance depends directly on the rate at which the matching mechanism processes tokens.[3] To facilitate matching while considering the cost and the availability of a large-capacity associative memory, Arvind proposed a pseudoassociative matching mechanism that typically requires several memory accesses,[1] but this increase in memory accesses severely degrades the performance and the efficiency of the underlying dataflow machines.

A more subtle problem with token matching is the complexity of allocating resources (memory cells). A failure to find a match implicitly allocates memory within the matching hardware. In other words, mapping a code-block to a processor places an unspecified commitment on the processor's matching unit. If this resource becomes overcommitted, the program may deadlock. In addition, due to hardware complexity and cost, one cannot assume this resource is so plentiful it can be wasted. (See Culler and Arvind[4] for a good discussion of the resource requirements issue.)

A more general criticism leveled at dataflow computing is instruction cycle inefficiency. A typical dataflow instruction cycle involves (1) detecting enabled nodes, (2) determining the operation to be performed, (3) computing the results, and (4) generating and communicating result tokens to appropriate target nodes. Even though the characteristics of this instruction cycle may be consistent with the pure-dataflow model, it incurs more overhead than its control-flow counterpart. For example, matching tokens is more complex than simply incrementing a program counter. The generation and communication of result tokens imposes inordinate overhead compared with simply writing the result to a memory location or a register. These inefficiencies in the pure-dataflow model tend to degrade performance under a low degree of parallelism.

Another formidable problem is the management of data structures. The dataflow functionality principle implies that all operations are side-effect free; that is, when a scalar operation is performed, new tokens are generated after the input tokens have been consumed. However, absence of side effects implies that if tokens are allowed to carry vectors, arrays, or other complex structures, an operation on a structure element must result in an entirely new structure. Although this solution is theoretically acceptable, in practice it creates excessive overhead at the level of system performance. A number of schemes have been proposed in the literature,[5] but the prob-

Table 1. Architectural features of current dataflow systems.

Category	General Characteristics	Machine	Key Features
Pure-dataflow	• Implements the traditional dataflow instruction cycle. • Direct matching of tokens	Monsoon[1,2]	• Direct matching of tokens using rendezvous slots in the frame memory (ETS model). • Associates three temporary registers with each thread of computation. • Sequential scheduling implemented by recirculating scheduling paradigm using a direct recirculation path (instructions are annotated with special marks to indicate that the successor instruction is IP+1). • Multiple threads supported by FORK and *implicit* JOIN.
		Epsilon-2[3]	• A separate match memory maintains match counts for the rendezvous slots, and each operand is stored separately in the frame memory. • Repeat unit is used to reduce the overhead of copying tokens and to represent a thread of computation (macroactor) as a linked-list. • Use of a register file to temporarily store values within a thread.
Macro-dataflow	• Integration of a token-based dataflow circular pipeline and an advanced control pipeline. • Direct matching of tokens.	EM-4[4]	• Use of macroactors based on the strongly connected arc model, and the execution of macroactors using advanced control pipeline. • Use of registers to reduce the instruction cycle and the communication overhead of transferring tokens within a macroactor. • Special thread library functions, FORK and NULL, to spawn and synchronize multiple threads.
Hybrid	• Based on conventional control-flow processor, i.e., sequential scheduling is implicit by the RISC-based architecture. • Tokens do not carry data, only continuations. • Provides limited token matching capability through special synchronization primitives (i.e., JOIN). • Message handlers implement inter-processor communication. • Can use both conventional and data-flow compiling technologies.	P-RISC[5]	• Support for multithreading using a token queue and circulating continuations. • Context switching can occur on every cycle or when a thread dies due to LOADS or JOINS.
		*T[6]	• Overhead is reduced by off-loading the burden of message handling and synchronization to separate coprocessors.
		Threaded Abstract Machine[7]	• Placing all synchronization, scheduling, and storage management responsibility under compiler control, e.g., exposes the token queue (i.e., continuation vector) for scheduling threads. • Having the compiler produce specialized message handlers as inlets to each code-block.

1. D.E. Culler and G.M. Papadopoulos, "The Explicit Token Store," *J. Parallel & Distributed Computing*, Vol. 10, 1990, pp. 289-308.

2. G.M. Papadopoulos and K.R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2146, 1991, pp. 342-351.

3. V.G. Grafe and J.E. Hoch, "The Epsilon-2 Multiprocessor System," *J. Parallel & Distributed Computing*, Vol. 10, 1990, pp. 309-318.

4. S. Sakai et al., "An Architecture of a Dataflow Single-Chip Processor," *Proc. 16th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 1948 (microfiche only), 1989, pp. 46-53.

5. R.S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?" *Proc. 16th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 1948 (microfiche only), 1989, pp. 262-272.

6. R. S. Nikhil, G.M. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *Proc. 19th Annual Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2941 (microfiche only), 1992, pp. 156-167.

7. D.E. Culler et al., "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstracted Machine," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1991.

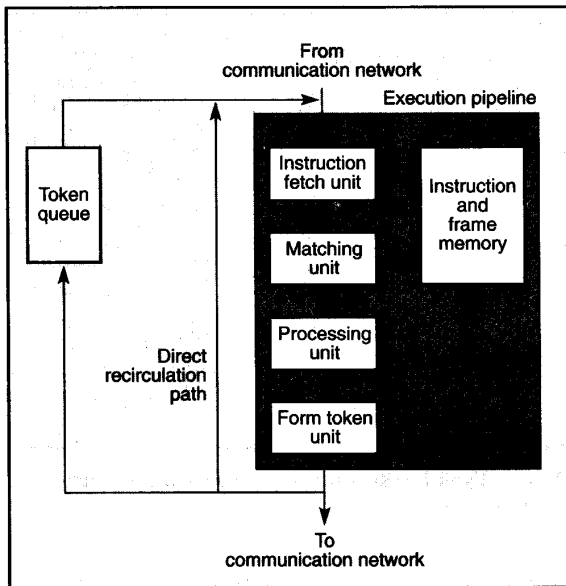


Figure 5. Organization of a pure-dataflow processing element.

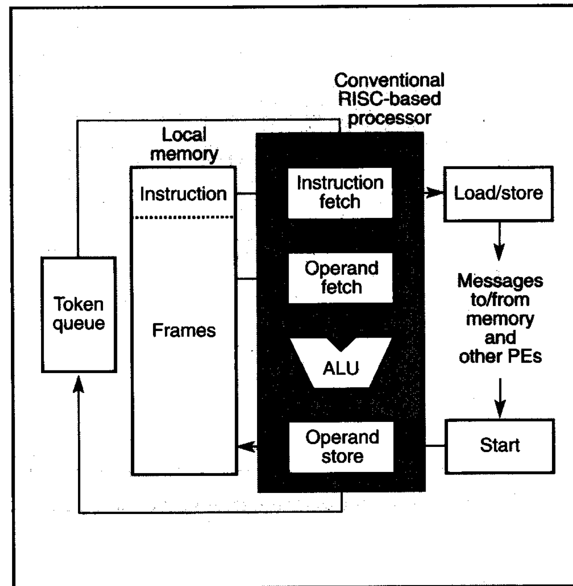


Figure 6. Organization of a hybrid processing element.

lem of efficiently representing and manipulating data structures remains a difficult challenge.

Recent architectural developments

Table 1 lists the features of six machines that represent the current trend in dataflow architecture: MIT's Monsoon, Sandia National Labs' Epsilon-2, Electrotechnical Labs' EM-4, MIT's P-RISC, MIT's *T, and UC Berkeley's TAM (Threaded Abstract Machine). Each proposal provides a different perspective on how parallel processing based on the dataflow concept can be realized; however, they share the goal of alleviating the inefficiencies associated with the dataflow model of computation. From these efforts, a number of innovative solutions have emerged.

Three categories. The dataflow machines currently advanced in the literature can be classified into three categories: pure-dataflow, macro-dataflow, and hybrid.

Figure 5 shows a typical processing element (PE) based on the pure-dataflow organization. It consists of an execution pipeline connected by a token queue. (The processing unit contains an ALU and a target address calculation unit for computing the destination address(es). It may also contain a set of registers to temporarily store operands between instructions.) The pure-dataflow organization is a slight modification of an architecture that implements the traditional dataflow instruction cycle. The major differences between the current organizations and the classic dynamic architectures are (1) the reversal of the instruction fetch unit and the matching unit and (2) the introduction of frames to represent contexts. These changes are mainly due to the implementation of a new matching scheme. Monsoon and Epsilon-2 are examples of machines based on this organization.

The hybrid organization departs more radically from the classic dynamic architectures. Tokens carry only tags, and the architecture is based on conventional control-flow sequencing (see Figure 6). Therefore, architectures based on this organization can be viewed as von Neumann machines that have been extended to support fine-grained dataflow capability. Moreover, unlike the pure-dataflow organization where token matching is implicit in the architecture, machines based on the hybrid organization provide a limited token-matching capability through special synchronization primitives. P-RISC, *T, and TAM can be categorized as hybrid organizations.

The macro-dataflow organization, shown in Figure 7, is a compromise between the other two approaches. It uses a token-based circular pipeline and an advanced control pipeline (a look-ahead control that implements instruction prefetching and token prematching to reduce idle times caused by unsuccessful matches). The basic idea is to shift to a coarser grain of parallelism by incorporating control-flow sequencing into the dataflow approach. EM-4 is an example of a macro-dataflow organization.

Token matching. One of the most important developments to emerge from current dataflow proposals is a novel and simplified process of matching tags — direct matching. The idea is to eliminate the expensive and complex process of associative search used in previous dynamic dataflow architectures. In a direct matching scheme, storage (called an activation frame) is dynamically allocated for all the tokens generated by a code-block. The actual location used within a code-block is determined at compile time; however, the allocation of activation frames is determined during runtime. For example, "unfolding" a loop body is achieved by allocating an activation frame for each loop iteration. The matching tokens generated within an iteration have a unique

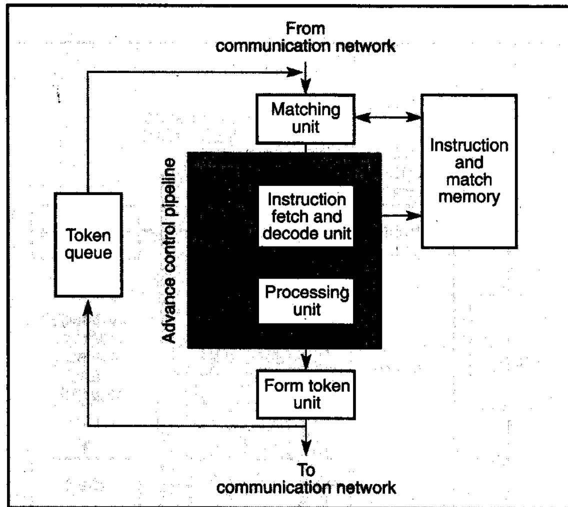


Figure 7. Organization of a macro-dataflow processing element.

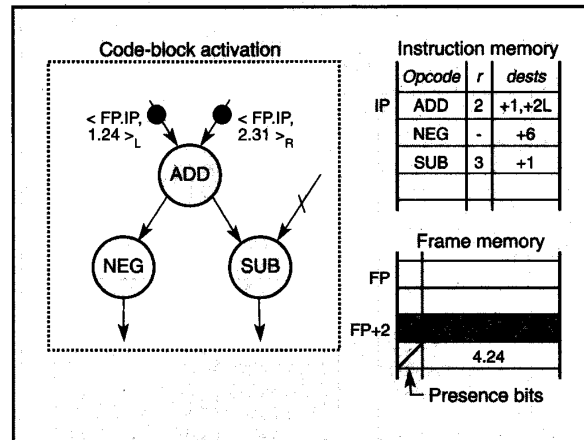


Figure 8. Explicit-token-store representation of a dataflow program execution.

slot in the activation frame in which they converge. The actual matching process simply checks the disposition of the slot in the frame memory.

In a direct matching scheme, any computation is completely described by a pointer to an instruction (IP) and a pointer to an activation frame (FP). The pair of pointers, <FP.IP>, is called a continuation and corresponds to the tag part of a token. A typical instruction pointed to by an IP specifies an opcode, an offset in the activation frame where the match will take place, and one or more displacements that define the destination instructions that will receive the result token(s). Each destination is also accompanied by an input port (left/right) indicator that specifies the appropriate input arc for a destination actor.

To illustrate the operations of direct matching in more detail, consider the token-matching scheme used in Monsoon. Direct matching of tokens in Monsoon is based on the Explicit Token Store (ETS) model.[1] Figure 8 shows an ETS code-block invocation and its corresponding instruction and frame memory. When a token arrives at an actor (for example, ADD), the IP part of the continuation points to the instruction that contains an offset *r* as well as displacement(s) for the destination instruction(s). Matching is achieved by checking the disposition of the slot in the frame memory pointed to by FP + *r*. If the slot is empty, the value of the token is written in the slot, and its presence bit is set to indicate that the slot is full. If the slot is already full, the value is extracted, leaving the slot empty, and the corresponding instruction is executed. The result token(s) generated from the operation is communicated to the destination instruction(s) by updating the IP according to the displacement(s) encoded in the instruction. For example, execution of the ADD operation produces two result tokens <FP.IP + 1, 3.55> and <FP.IP + 2, 3.55>$_L$.

In one variation to the matching scheme, EM-4 maintains a simple one-to-one correspondence between the address of an instruction and the address of its rendezvous slot (Figure 9). This is achieved by allocating an operand segment — analogous to an activation frame — that contains the same number of memory words as the template segment that contains the code-block. In addition, the operand segment is bound to a template segment by storing the corresponding segment number in the first word of the operand segment. The token's continuation contains only an FP and an offset. These values are used to determine the unique location in the operand segment (called an entry point) to match tokens and then to fetch the corresponding instruction word.

Direct matching schemes used in pure-dataflow and macro-dataflow organizations are implicit in the architecture.

In the Epsilon-2 dataflow multiprocessor, a separate storage (match memory) contains rendezvous slots for incoming tokens (see Table 1, reference 3). Similar to Monsoon, an offset encoded in the instruction word is used to determine the match-memory location to match tokens. However, unlike the Monsoon, each slot is associated with a match count that is initialized to zero. As tokens arrive, the match count is compared with the value encoded in the opcode. If the match count is less than the value encoded in the opcode, the match count is incremented and stored back in the match memory. Otherwise, the node is considered enabled, and the match count is reinitialized to zero. The instruction word also specifies offsets in the frame memory where the actual operands reside. Therefore, in contrast to the scheme used in Monsoon or EM-4, the opcode specifies two separate frame locations for the operands.

Direct matching schemes used in the pure-dataflow and macro-dataflow organizations are implicit in the architecture. In other words, the token-matching mechanism provides the full generality of the dataflow model of execution and therefore is supported by the hardware. Architectures based on the hybrid organization, on the other hand, provide a limited direct matching capability through software implementation using special JOIN instructions. P-RISC is an ex-

ample of such an architecture (Table 1, reference 5). P-RISC is based on a conventional RISC-type architecture, which is extended to support fine-grain dataflow capability. To synchronize two threads of computations, a JOIN x instruction toggles the contents of the frame location FP + x. If the frame location FP + x is empty, no continuation is produced, and the thread dies. If the frame location is full, it produces a continuation <FP.IP + 1>. Therefore, a JOIN instruction implements the direct matching scheme and provides a general mechanism for synchronizing two threads of computations. Note that the JOIN operation can be generalized to support n-way synchronization; that is, the frame location x is initialized to $n - 1$ and different JOIN operations decrement it until it reaches zero. JOIN instructions are used in *T and TAM to provide explicit synchronization.

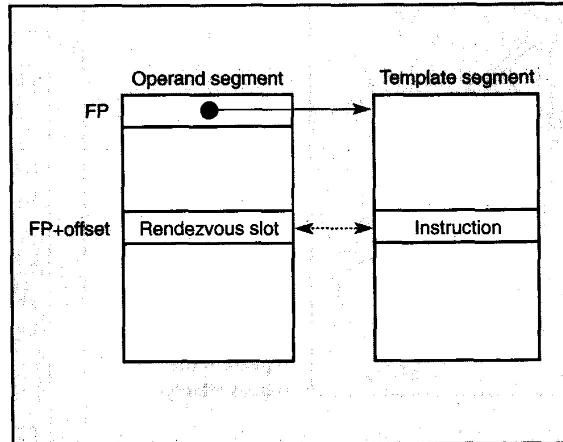


Figure 9. The EM-4's direct matching scheme.

Convergence of dataflow and von Neumann models. Dataflow architectures based on the original model provide well-integrated synchronization at a very basic level — the instruction level. The combined dataflow/von Neumann model groups instructions into larger grains so that instructions within a grain can be scheduled in a control-flow fashion and the grains themselves in a dataflow fashion. This convergence combines the power of the dataflow model for exposing parallelism with the execution efficiency of the control-flow model. Although the spectrum of dataflow/von Neumann hybrid is very broad, two key features supporting this shift are sequential scheduling and use of registers to temporarily buffer the results between instructions.

Sequential scheduling. Exploiting a coarser grain of parallelism (compared with instruction-level parallelism) allows use of a simple control-flow sequencing within the grain. This is in recognition of the fact that data-driven sequencing is unnecessarily general and such flexible instruction scheduling comes at a cost of overhead required to match tokens. Moreover, the self-scheduling paradigm fails to provide an adequate programming medium to encode imperative operations essential for execution of operating system functions (for example, resource management).[4] In addition, the instruction cycle can be further reduced by using a register file to temporarily store operands in a grain, which eliminates the overhead involved in constructing and transferring result tokens within a grain.

The combined model groups instructions into larger grains for control-flow scheduling of instructions but dataflow scheduling of grains.

There are contrasting views on merging the two conceptually different execution models. The first is to extend existing conventional multiprocessors to provide dataflow capability (hybrid organization). The idea here is to avoid a radical departure from existing programming methodology and architecture in favor of a smoother transition that provides incremental improvement as well as software compatibility with conventional machines. The second approach is to incorporate control-flow sequencing into existing dataflow architectures (pure- or macro-dataflow organization). For the sake of presentation, we sidestep discussion of the first approach, since sequential scheduling is implicit in hybrid architectures, and focus on the second.

Control-flow sequencing can be incorporated into the dataflow model in a couple ways. The first method is a simple recirculate scheduling paradigm. Since a continuation is completely described by a pair of pointers <FP.IP>, where IP represents a pointer to the current instruction, the successor instruction is simply <FP.IP + 1>. In addition to this simple manipulation, the hardware must support the immediate reinsertion of continuations into the execution pipeline. This is achieved by using a direct recirculation path (see Figure 5) to bypass the token queue.

One potential problem with the recirculation method is that successor continuations are not generated until the end of the pipeline ("Form token unit" in Figure 5). Therefore, execution of the next instruction in a computational thread will experience a delay equal to the number of stages in the pipe. This means that the total number of cycles required to execute a single thread of computation in a k-stage pipeline will be k times the number of instructions in the thread. On the other hand, the recirculation method allows interleaving up to k independent threads in the pipeline, effectively masking long and unpredictable latency due to remote loads.

The second technique for implementing sequential scheduling uses the macroactor concept. This scheme groups the nodes in a dataflow graph into macroactors. The nodes within a macroactor are executed sequentially; the macroactors themselves are scheduled according to the dataflow model. The EM-4 dataflow multiprocessor implements macroactors based on the strongly connected arc model (Table 1, reference 4). This model categorizes the arcs of a dataflow graph as normal or strongly connected. A subgraph whose nodes are connected by strongly connected arcs is called a strongly connected block (SCB). An SCB is enabled (fired) when all its input tokens are available. Once an SCB fires, all the nodes in the

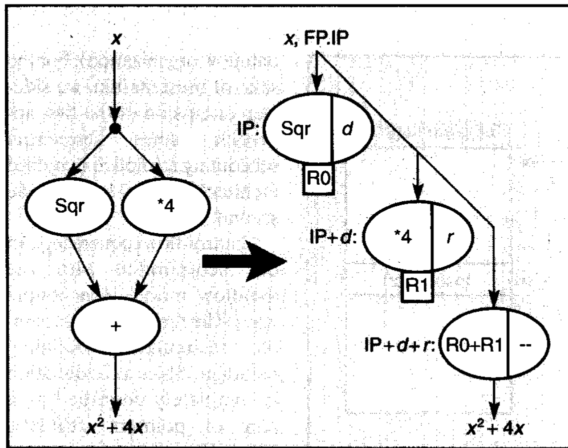


Figure 10. Linked-list representation of a dataflow graph using the repeat unit (*d* and *r* represent the repeat offsets).

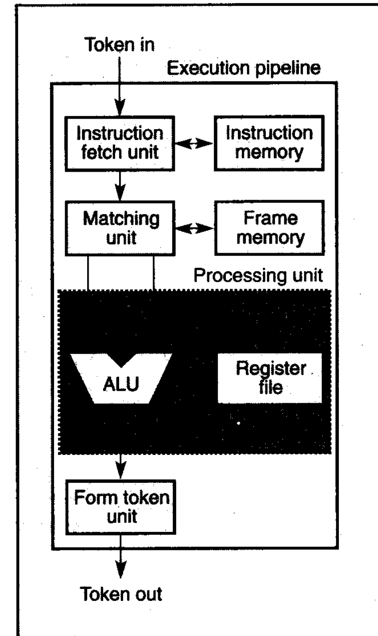


Figure 11. Use of a register file.

block are executed exclusively by means of the advanced control pipeline.

Epsilon-2 uses a unique architectural feature — the repeat unit — to implement sequential scheduling (Table 1, reference 3). The unit generates repeat tokens, which efficiently implement data fanouts in dataflow graphs and significantly reduce the overhead of copying tokens. The unit represents a thread of computation as a linked list and uses registers to buffer the results between instructions (see Figure 10). To generate a repeat token, it adds the repeat offset encoded in the instruction word to the current token's instruction pointer. Thus, the repeat unit can prioritize instruction execution within a grain.

Register use. Inefficient communication of tokens among nodes was a major problem in past dataflow architectures. The execution model requires combining result values with target addresses to form result tokens for communication to successor nodes. Sequential scheduling avoids this overhead; whenever locality can be exploited, registers can be used to temporarily buffer results.

The general method of incorporating registers in the dataflow execution pipeline is illustrated in Figure 11. A set of registers is associated with each computational grain, and the instructions in the grain are allowed to refer to these registers as operands. For example, Monsoon employs three temporary registers (called T registers) with each computational thread. The T registers, combined with a continuation and a value, are called computation descriptors (CDs). A CD completely describes the state of a computational thread. Since the number of active CDs can be large, only threads occupying the execution pipeline are associated with T registers. As long as a thread does not die, its instructions can freely use its registers. Once a thread dies, its registers may be committed to a new thread entering the pipeline; therefore, the register values are not necessarily preserved across grain boundaries. All current architectures use registers but, unlike Monsoon, do not fix the number of registers associated with each computational grain. Therefore, proper use of resources requires a compile-time analysis.

Multithreading

Architectures based on the dataflow model offer the advantage of instruction-level context switching. Since each datum carries context-identifying information in a continuation or tag, context switching can occur on a per-instruction basis. Thus, these architectures tolerate long, unpredictable latency resulting from split transactions. (This refers to message-based communication due to remote loads. A split transaction involves a read request from processor A to processor B containing the address of the location to be read and a return address, followed by a response from processor B to processor A containing the requested value.)

Combining instruction-level context switching with sequential scheduling leads to another perspective on dataflow architectures — multithreading. In the context of multithreading, a thread is a sequence of statically ordered instructions where once the first instruction is executed, the remaining instructions execute without interruption.[6] A thread defines a basic unit of work consistent with the dataflow model, and current dataflow projects are adopting multithreading as a viable method for combining the features of the dataflow and von Neumann execution models.

Supporting multiple threads. Basically, hybrid dataflow architectures can be viewed as von Neumann machines extended to support fine-grained interleaving of multiple threads. As an illustration of how this is accomplished, consider P-RISC, which is strongly influenced by Iannucci's dataflow/von Neumann hybrid architecture.[7] P-RISC is a RISC-based architecture in the sense that, except for load/store instructions, instructions are frame-to-frame (that is, register-to-register) operations that operate within the PE (Figure 6). The local memory contains instructions and frames. Notice that the P-RISC executes three address instructions on data stored in the frames; hence, "tokens" carry only continuations. The instruction-fetch and operand-fetch units fetch appropriate instructions and operands, respectively, pointed to by the continuation. The functional unit performs general RISC operations, and the operand store is responsible for storing results in the appropriate slots of the frame.

For the most part, P-RISC executes instructions in the same manner as a con-

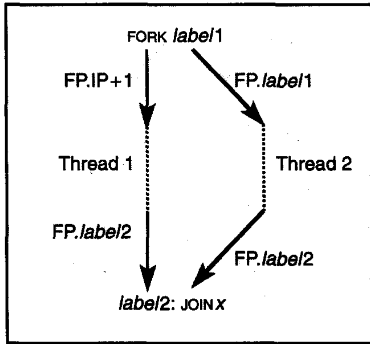


Figure 12. Application of FORK and JOIN constructs.

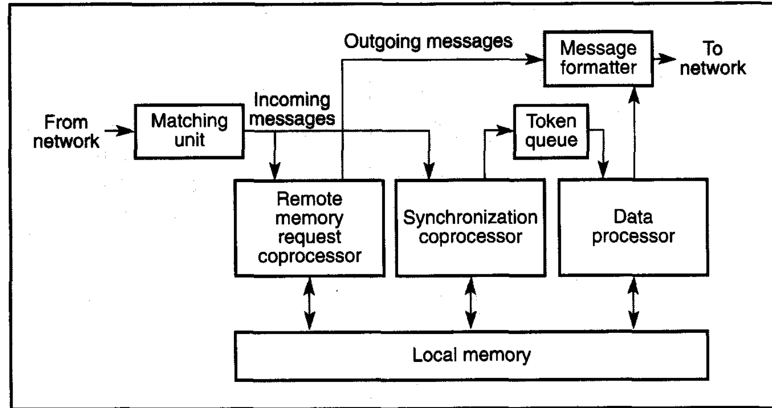


Figure 13. Organization of a *T processor node.

ventional RISC. An arithmetic/logic instruction generates a continuation that is simply <FP.IP + 1>. However, unlike architectures based on the pure-dataflow organization, IP represents a program counter in the conventional sense and is incremented in the instruction-fetch stage. For a JUMP *x* instruction, the continuation is simply <FP.*x*>. To provide fine-grained dataflow capability, the instruction set is extended with two special instructions — FORK and JOIN — used to spawn and synchronize independent threads. These are simple operations executed within the normal processor pipeline, not operating system calls. A FORK instruction is a combination of a JUMP and a fall-through to the next instruction. Executing a FORK *label* has two effects. First, the current thread continues to the next instruction by generating a continuation of the form <FP.IP + 1>. Second, a new thread is created with the same continuation as the current continuation except that the "IP" is replaced by "*label*" (that is, <FP.*label*>). JOIN, on the other hand, is an explicit synchronization primitive that provides the limited direct-matching capability used in other dataflow machines. FORK and JOIN operations are illustrated in Figure 12.

In addition to FORK and JOIN, a special START message initiates new threads and implements interprocessor communication. The message of the form

<START, *value*, FP.IP, *d*>

writes the value in the location FP + *d* and initiates a thread described by FP.IP. START messages are generated by LOAD and STORE instructions used to implement I-structure reads and procedure calls, respectively. For example, the general form of a synchronizing memory read, such as I-structure, is LOAD *x*, *d*, where an address *a* in frame location FP + *x* is used to load data onto frame location FP + *d*. Executing a LOAD instruction generates an outgoing message (via the load/store unit) of the form

<I-READ, *a*, FP.IP, *d*>

where *a* represents the location of the value to be read and *d* is the offset relative to FP. This causes the current thread to die. Therefore, a new continuation is extracted from the token queue, and the new thread will be initiated. On its return trip from the I-structure memory, an incoming START message (via the start unit) writes the value at location FP + *d* and continues execution of the thread. A procedure is invoked in a similar fashion when the caller writes the arguments into the callee's activation frame and initiates the threads. This is achieved by the instruction

START, *dv*, *d*FPIP, *dd*

which reads a value *v* from FP + *dv*, a continuation <FP.IP> from FP+*d*FPIP, and an offset *d* from FP + *dd*. Executing this instruction sends a START message to an appropriate processing element and initiates a thread.

P-RISC supports multithreading in one of two ways. In the first method, as long as a thread does not die due to LOADs or JOINs, it is executed using the von Neumann scheduling IP + 1. When a thread dies, a context switch is performed by extracting a token from the token queue. The second method is to extract a token from the token queue every pipeline cycle.

*T, a successor to P-RISC, provides similar extensions to the conventional instruction set (Table 1, reference 6). However, to improve overall performance, the thread-execution and the message-handling responsibilities are distributed among three asynchronous processors. The general organization of *T is shown in Figure 13. A *T node consists of the data processor, the remote-memory request coprocessor, and the synchronization coprocessor, which all share a local memory. The data processor executes threads by extracting continuations from the token queue using the NEXT instruction. It is optimized to provide excellent single-thread performance. The remote-memory request coprocessor handles incoming remote LOAD/STORE requests. For example, a LOAD request is processed by a message handler, which (1) accesses the local memory and (2) sends a START message as a direct response without disturbing the data processor. On the other hand, the synchronization coprocessor handles returning LOAD responses and JOIN operations. Its responsibility is to (1) continually queue messages from the network interface; (2) complete the unfinished remote LOADs by placing message values in destination locations and, if necessary, performing JOIN operations; and (3) place continuations in the token queue for later pickup and execution by the data processor.

In contrast to the two hybrid projects discussed thus far, TAM provides a conceptually different implementation of the dataflow execution model and multithreading (Table 1, reference 7). In TAM, the execution model for fine-grain interleaving of multiple threads is sup-

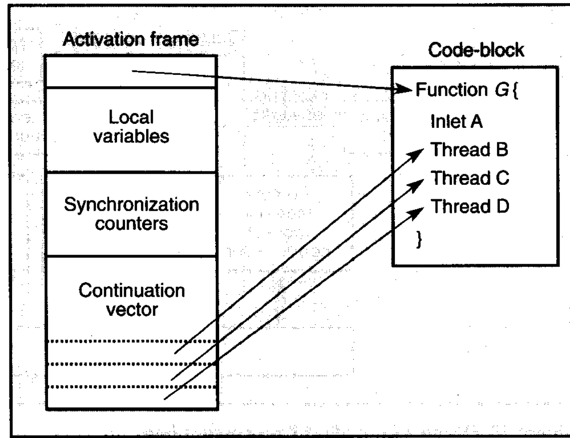


Figure 14. A Threaded Abstract Machine (TAM) activation.

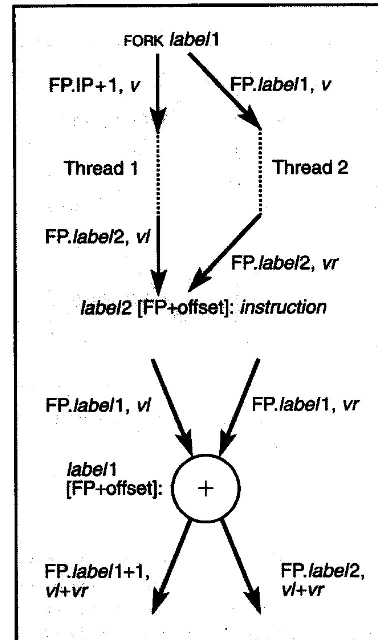


Figure 15. Application of FORK and JOIN constructs and equivalent dataflow actor in Monsoon.

ported by an appropriate compilation strategy and program representation, not by elaborate hardware. In other words, rather than viewing the execution model for fine-grain parallelism as a property of the machine, all synchronization, scheduling, and storage management is explicit and under compiler control.

Figure 14 shows an example of a TAM activation. Whenever a code-block is invoked, an activation frame is allocated. The frame provides storage for local variables, synchronization counters, and a continuation vector that contains addresses of enabled threads within the called code-block. Thus, when a frame is scheduled, threads are executed from its continuation vector, and the last thread schedules the next frame.

TAM supports the usual FORK operations that cause additional threads to be scheduled for execution. A thread can be synchronized using SYNC (same as JOIN) operations that decrement the entry count for the thread. A conditional flow of execution is supported by a SWITCH operation that forks one of two threads based on a Boolean input value. A STOP (same as NEXT) operation terminates the current thread and causes initiation of another thread. TAM also supports interframe messages, which arise in passing arguments to an activation, returning results, or split-phase transactions, by associating a set of inlets with each code-block. Inlets are basically message handlers that provide an external interface.

As can be seen, TAM's support for multithreading is similar to that of the other hybrid machines discussed. The major difference is that thread scheduling in P-RISC and *T is local and implicit through the token queue. In TAM, thread scheduling is explicit and under compiler control.

In contrast to P-RISC, *T, and TAM, Monsoon's multithreading capability is based on the pure-dataflow model in the sense that tokens not only schedule instructions but also carry data. Monsoon incorporates sequential scheduling, but with a shift in viewpoint about how the architecture works. It uses presence bits to synchronize predecessor/successor instructions through rendezvous points in the activation frame. To implement the recirculate scheduling paradigm, instructions that depend on such scheduling are annotated with a special mark to indicate that the successor instruction is IP + 1. This allows each instruction within a computation thread to enter the execution pipeline every k cycles, where k is the number of stages in the pipe. The actual thread interleaving is accomplished by extracting a token from the token queue and inserting it into the execution pipeline every clock cycle. Thus, up to k active threads can be interleaved through the execution pipeline.

New threads and synchronizing threads are introduced by primitives similar to those used in hybrid machines — FORK and JOIN. For example, a FORK *label*1 generates two continuations, <FP.*label*1> and <FP.IP + 1>. An implicit JOIN instruction of the form

*label*2 [FP + *offset*]: *instruction*

indicates the threads' rendezvous slot is at frame location [FP + *offset*]; therefore, the instruction will not execute until both continuations arrive. FORK and JOIN operations are illustrated in Figure 15.

Although FORK and JOIN can be viewed as thread-spawning and thread-synchronizing primitives, these operations can also be recognized as instructions in the dataflow execution model. For example, a FORK instruction is similar to but less general than a COPY operation used in the pure-dataflow model. JOIN operations are implemented through the direct matching scheme. For example, consider the dataflow actor shown in Figure 15, which receives two input tokens, evaluates the sum, and generates two output tokens. The operation of this actor can be realized by a combination of instructions of the form

*label*1[FP + *offset*]:
ADD *vl*, *vr* || FORK *label*2,

where || represents the combination of an ADD and a FORK. In a pure-dataflow organization (including Monsoon), the execution pipeline handles token matching, arithmetic operation, and token forming. Therefore, instructions that appear to be distinctively multithreaded in fact can be viewed as part of the more traditional dataflow operations.

In EM-4, an SCB is thought of as a sequential, uninterruptable thread of control. Therefore, the execution of multiple threads can be implemented by passing tokens between SCBs. This is

achieved by having thread library functions that allow parallelism to be expressed explicitly.[8] For example, a master thread spawns and terminates a slave thread using functions FORK and NULL, respectively. The general format for the FORK function is given as

FORK(PE, func, n, arg1, . . . , argn),

where PE specifies the processing element where the thread was created, and *n* represents the number of arguments in the thread. This function causes the following operations: (1) allocate a new operand segment on the processing element and link it to the template segment specified by the token's address portion, (2) write the arguments into the operand segment, (3) send the NULL routine's address as a return address for the newly created thread, and (4) continue the execution of the current thread. Once execution completes, the new thread terminates by executing a NULL function.

Using the FORK and NULL functions, a master thread can distribute the work over a number of slave threads to compute partial results. The final result is collected when each slave thread sends its partial result to the master thread. Multithreading is achieved by switching among threads whenever a remote LOAD operation occurs. However, since each thread is an SCB (that is, an uninterruptable sequence of instructions), interleaving of threads on each cycle is not allowed.

Partitioning programs to threads. An important issue in multithreading is the partitioning of programs to multiple sequential threads. A thread defines the basic unit of work for scheduling — and thus a computation's granularity. Since each thread has an associated cost, it directly affects the amount of overhead required for synchronization and context switching. Therefore, the main goal in partitioning is maximizing parallelism while minimizing the overhead required to support the threads.

A number of proposals based on the control-flow model use multithreading as a means of tolerating high-latency memory operations, but thread definitions vary according to language characteristics and context-switching criteria. For example, the multiple-context schemes used in Weber and Gupta[9] obtain threads by subdividing a parallel loop into a number of sequential processes, and context switching occurs when a main memory access is required (due to a cache miss). As a consequence, the thread granularity in these models tends to be coarse, thereby limiting the amount of parallelism that can be exposed. On the other hand, non-strict functional languages for dataflow architectures, such as Id, complicate partitioning due to feedback dependencies that may only be resolved dynamically. These situations arise because of the possibility of functions or arbitrary expressions returning results before all operands are computed (for example, I-structure semantics). Therefore, a more restrictive constraint is placed on partitioning programs written in non-strict languages.

Iannucci[7] has outlined several important issues to consider in partitioning programs: First, a partitioning method should maximize the exploitable parallelism. In other words, the attempt to aggregate instructions does not imply restricting or limiting parallelism. Instructions that can be grouped into a thread should be the parts of a code where little or no exploitable parallelism exists. Second, the longer the thread length, the longer the interval between context switches. This also increases the locality for better utilization of the processor's resources. Third, any arc (that is, data dependency) crossing thread boundaries implies dynamic synchronization. Since synchronization operations introduce hardware overhead and/or increase processor cycles, they should be minimized. Finally, the self-scheduling paradigm of program graphs implies that execution ordering cannot be independent of program inputs. In other words, this dynamic ordering behavior in the code must be understood and considered as a constraint on partitioning.

Thread definitions vary according to language characteristics and context-switching criteria.

A number of thread partitioning algorithms convert dataflow graph representation of programs into threads based on the criteria outlined above. Schauser et al.[6] proposed a partitioning scheme based on dual graphs. A dual graph is a directed graph with three types of arcs: data, control, and dependence. A data arc represents the data dependency between producer and consumer nodes. A control arc represents the scheduling order between two nodes, and a dependence arc specifies long latency operation from message handlers (that is, inlets and outlets) sending/receiving messages across code-block boundaries.

The actual partitioning uses only the control and dependence edges. First, the nodes are grouped as dependence sets that guarantee a safe partition with no cyclic dependencies. A safe partition has the following characteristics[6]: (1) no output of the partition needs to be produced before all inputs are available; (2) when the inputs to the partition are available, all the nodes in the partition are executed; and (3) no arc connects a node in the partition to an input node of the same partition. The partitions are merged into larger partitions based on rules that generate safe partitions. Once the general partitioning has been completed, a number of optimizations are performed in an attempt to reduce the synchronization cost. Finally, the output of the partitioner is a set of threads wherein the nodes in each thread are executed sequentially and the synchronization requirement is determined statically and only occurs at the beginning of a thread.

Future prospects

To predict whether dataflow computing will have a legitimate place in a world dominated by massively parallel von Neumann machines, consider the suitability of current commercial parallel machines for fine-grain parallel programming. Studies on TAM show that it is possible to implement the dataflow execution model on conventional architectures and obtain reasonable performance (Table 1, reference 7). This has been demonstrated by compiling Id90 programs to TAM and translating them, first to TL0, the TAM assembly language, and finally to native machine code for a variety of platforms, mainly CM-5.[10] However, TAM's translation of dataflow graph program representation to control-flow execution also shows a basic mismatch between the requirements for fine-grain parallelism and the underlying architecture. Fine-grain parallel pro-

gramming models dynamically create parallel threads of control that execute on data structures distributed among processors. Therefore, efficient support for synchronization, communication, and dynamic scheduling becomes crucial to overall performance.

One major problem of supporting fine-grain parallelism on commercial parallel machines is communication overhead. In recent years, the communication performance of commercial parallel machines has improved significantly.[11] One of the first commercial message-passing multicomputers, Intel's iPSC, incurs a communication overhead of several milliseconds, but current parallel machines, such as KSR1, Paragon, and CM-5, incur a one-way communication overhead of just 25 to 86 microseconds.[11] Despite this improvement, the communication overhead in these machines is too high to efficiently support dataflow execution. This can be attributed to the lack of integration of the network interface as part of hardware functionality. That is, individual processors offer high execution performance on sequential streams of computations, but communication and synchronization among processors have substantial overhead.[10,11]

In comparison, processors in current dataflow machines communicate by executing message handlers that directly move data in and out of preallocated storage. Message handlers are short threads that handle messages entirely in user mode, with no transfer of control to the operating system. For example, in Monsoon, message handlers are supported by hardware: A sender node can format and send a message in exactly one cycle, and a receiver node can process an incoming message by storing its value and performing a JOIN operation in one or two cycles. In hybrid-class architectures, message handlers are implemented either by specialized hardware, as in P-RISC and *T, or through software (for example, interrupt or polling) as part of the processor pipeline execution. The most recent hybrid-class prototype under development at MIT and Motorola, inspired by the conceptual *T design at MIT, uses processor-integrated networking that directly integrates communication into the MC88110 superscalar RISC microprocessor.[11] Processor-integrated networking efficiently implements the message-passing mechanism. It provides a low communication overhead of 100 nanoseconds, and overlapping computation with communication — for example, by off-loading message handling to coprocessors, as in *T — may reduce this even more.

With the advent of multithreading, future dataflow machines will no doubt adopt a hybrid flavor, with emphasis on improving the execution efficiency of multiple threads of computations. Experiments on TAM have already shown how implementation of the dataflow execution model can be approached as a compilation issue. These studies also indicate that considerable improvement is possible through hardware support, which was the original goal of dataflow computer designers — to build a specialized architecture that allows direct mapping of dataflow graph programs onto the hardware. Therefore, the next generation of dataflow machines will rely less on very specialized processors and emphasize incorporating general mechanisms to support fine-grain parallelism into existing sequential processors. The major challenge, however, will be to strike a balance between hardware complexity and performance.

The success of multithreading depends on rapid support of context switching.

Despite recent advances toward developing effective architectures that support fine-grain parallelism and tolerate latency, some challenges remain. One of these challenges is dynamic scheduling. The success of multithreading depends on rapid support of context switching. This is possible only if threads are resident at fast but small memories (that is, at the top level of the storage hierarchy), which limits the number of active threads and thus the amount of latency that can be tolerated. All the architectures described — Monsoon, Epsilon-2, EM-4, P-RISC, and *T — rely on a simple dataflow scheduling strategy based on hardware token queues. The generality of the dataflow scheduling makes it difficult to execute a logically related set of threads through the processor pipeline, thereby removing any opportunity to utilize registers across thread boundaries. TAM alleviates this problem to some extent by relegating the responsibilities of scheduling and storage management to the compiler. For example, continuation vectors that hold active threads are implemented as stacks, and all frames holding enabled threads are linked in a ready queue. However, both hardware and software methods discussed are based on a naive, local scheduling discipline with no global strategy. Therefore, appropriate means of directing scheduling based on some global-level understanding of program execution will be crucial to the success of future dataflow architectures.[12]

Another related problem is the allocation of frames and data structures. Whenever a function is invoked, a frame must be allocated; therefore, how frames are allocated among processors is a major issue. Two extreme examples are a random allocation on any processor or local allocation on the processor invoking the function. The proper selection of an allocation strategy will greatly affect the balance of the computational load. The distribution of data structures among the processors is closely linked to the allocation of frames. For example, an allocation policy that distributes a large data structure among the processors may experience a large number of remote messages, which will require more threads to mask the delays. On the other hand, allocating data structures locally may cause one processor to serve a large number of accesses, limiting the entire system's performance. Therefore, the computation and data must be studied in unison to develop an effective allocation methodology.

The eventual success of dataflow computers will depend on their programmability. Traditionally, they've been programmed in languages such as Id and SISAL (Streams and Iterations in a Single Assignment Language)[2] that use functional semantics. These languages reveal high levels of concurrency and translate onto dataflow machines and conventional parallel machines via TAM. However, because their syntax and semantics differ from the imperative counterparts such as Fortran and C, they have been slow to gain acceptance in the programming community. An alternative is to explore the use of established imperative languages to program dataflow machines. However, the difficulty will be analyzing data dependencies and extracting parallelism

from source code that contains side effects. Therefore, more research is still needed to develop compilers for conventional languages that can produce parallel code comparable to that of parallel functional languages. ■

References

1. Arvind and D.E. Culler, "Dataflow Architectures," *Ann. Review in Computer Science*, Vol. 1, 1986, pp. 225-253.

2. J.-L. Gaudiot and L. Bic, *Advanced Topics in Dataflow Computing*, Prentice Hall, Englewood Cliffs, N.J., 1991.

3. Arvind, D.E. Culler, and K. Ekanadham, "The Price of Fine-Grain Asynchronous Parallelism: An Analysis of Dataflow Methods," *Proc. CONPAR 88*, Sept. 1988, pp. 541-555.

4. D.E. Culler and Arvind, "Resource Requirements of Dataflow Programs," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 861 (microfiche only), 1988, pp. 141-150.

5. B. Lee, A.R. Hurson, and B. Shirazi, "A Hybrid Scheme for Processing Data Structures in a Dataflow Environment," *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, No. 1, Jan. 1992, pp. 83-96.

6. K.E. Schauser et al., "Compiler-Controlled Multithreading for Lenient Parallel Languages," *Proc. Fifth ACM Conf. Functional Programming Languages and Computer Architecture*, ACM, New York, 1991, pp. 50-72.

7. R.A. Iannucci, "Towards a Dataflow/von Neumann Hybrid Architecture," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 861 (microfiche only), 1988, pp. 131-140.

8. M. Sato et al., "Thread-Based Programming for EM-4 Hybrid Dataflow Machine," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2941 (microfiche only), 1992, pp. 146-155.

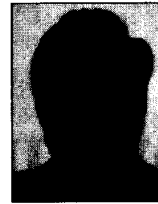
9. W.D. Weber and A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results," *Proc. 16th Ann. Int'l Symp, Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 1948 (microfiche only), 1989, pp. 273-280.

10. E. Spertus et al., "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 3811 (microfiche only), 1993.

11. G.M. Papadopoulos et al., "*T: Integrated Building Blocks for Parallel Computing," *Proc. Supercomputing 93*, IEEE CS Press, Los Alamitos, Calif., Order No. 4340, 1993, pp. 624-635.

12. D.E. Culler, K.E. Schauser, and T. von Eicken, "Two Fundamental Limits on Dataflow Multiprocessing," *Proc. IFIP Working Group 10.3 (Concurrent Systems) Working Conf. on Architecture and Compilation Techniques for Fine- and Medium-Grain Parallelism*, Jan. 1993.

Ben Lee is an assistant professor in the Department of Electrical and Computer Engineering at Oregon State University. His interests include computer architectures, parallel and distributed systems, program allocation, and dataflow computing. Lee received the BEEE degree from the State University of New York at Stony Brook in 1984 and the PhD degree in computer engineering from Pennsylvania State University in 1991. He is a member of ACM and the IEEE Computer Society.



A.R. Hurson is on the computer engineering faculty at Pennsylvania State University. His research for the past 12 years has been directed toward the design and analysis of general as well as special-purpose computers. He has published over 120 technical papers and has served as guest coeditor of special issues of the *IEEE Proceedings*, the *Journal of Parallel and Distributed Computing*, and the *Journal of Integrated Computer-Aided Engineering*. He is the coauthor of *Parallel Architectures for Data and Knowledge Base Systems* (IEEE CS Press, to be published in October 1994), a member of the IEEE Computer Society Press Editorial Board, and a speaker for the Distinguished Visitors Program.

Readers can contact Lee at Oregon State University, Dept. of Electrical and Computer Engineering, Corvallis, OR 97331-3211. His Internet address is benl@ece.orst.edu.

IEEE Parallel & Distributed Technology

A quarterly magazine published by the IEEE Computer Society

Subscribe before August 31.
Receive the final 1994 issues for just $12.

- **FALL 1994 — High-Performance Fortran**
- **WINTER 1994 — Real-Time Computing**

❑ YES, SIGN ME UP! As a member of the Computer Society or another IEEE Society, I qualify for the member rate of $12 for a half-year subscription (two issues).

Society__________ IEEE Membership No.__________

Since IEEE subscriptions are annualized, orders received from September 1994 through February 1994 will be entered as full-year subscriptions for the 1995 calendar year. Pay the full-year rate of $24 for four quarterly issues. For membership information, see pages 16A-B.

FULL SIGNATURE DATE

NAME

STREET ADDRESS

CITY

STATE/COUNTRY ZIP/POSTAL CODE

❑ **Payment enclosed** Residents of CA, DC, Canada, and Belgium, add applicable tax.

❑ **Charge to** ❑ Visa ❑ MasterCard ❑ American Express

CHARGE-CARD NUMBER EXPIRATION DATE

MONTH YEAR

MAIL TO: CIRCULATION DEPT., 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264