# Cache Memories for Dataflow Systems

A.R. Hurson
*Pennsylvania State University*
Krishna M. Kavi and Behrooz Shirazi
*University of Texas*
Ben Lee
*Oregon State University*

*Cache memory—so effective in traditional control-flow architecture—has the potential to enhance dataflow system performance as well. The authors explore the recent trend in combining dataflow and control-flow processing, which offers new alternatives in computer architecture design, and analyze cache memory's application to the dataflow environment.*

Current microelectronics technology has enabled chip capacity to exceed 64 million transistors, and computer architects are facing the increasing challenge of ULSI (ultra large scale integration) technology. By the year 2000, technology advances are expected to make possible chips with more than 100 million transistors. With such a significant on-chip hardware capacity, concurrency is a way to reduce the computation gap between the computational power demanded by the applications and that demanded by the underlying computer platforms.

Designers can increase architectural support for instruction-level parallelism to absorb such a massive hardware capacity; examples are superscalar and superpipeline machines.[1] However, the single-instruction stream processing characteristic of the control-flow machine makes it inherently unsuitable to exploit superscalar and superpipeline architectures efficiently.[2] This is because the total ordering of the control-flow execution model is ill-equipped to tolerate long, unpredictable memory and communication latencies that are unavoidable in a multiprocessor system.

An alternative is to design parallel computers based on partial ordering of the execution. *Dataflow* machines are an example of this approach, where an instruction initiates (fires) only when all the required operands are available. Instructions impose no sequencing constraints except the one on the program's data dependencies. As a result, the program's dataflow graph representation exposes all forms of parallelism, eliminating the need to explicitly manage parallel program execution.

Research efforts have long focused on the dataflow computation model, simple and elegant in describing parallelism and data dependencies. Since


1063-6552/96/$4.00 © 1996 IEEE

the early 1970s, researchers have proposed, simulated, and prototyped dataflow designs. The consensus is that directly implementing the dataflow concept carries overhead costs, mainly due to its fine-grain approach to parallelism.[2]

In this article, we compare control-flow and dataflow architectures, examine cache as it relates to both, and describe our research experiments in adding cache to Monsoon, an example of a pure dataflow system.

## Dataflow revisited

Dataflow computation has received renewed attention lately, resulting from (1) a lack of developments in conventional parallel processing, and (2) a change in viewpoint on dataflow and its implementation (a shift from the exploitation of fine-grain to medium- and large-grain parallelism). To alleviate the inefficiencies associated with the pure dataflow model, designers have compromised, incorporating control-flow methods into the dataflow approach.

In dataflow architectures, *context switching* can occur on a per-instruction basis, which tolerates long, unpredictable latencies due to remote memory accesses. The instruction-level context-switching capability combined with sequential scheduling yields what we call *multithreading*. The evolution from a pure self-scheduling paradigm to multithreading requires locality and improved processor efficiency during remote memory accesses. Current dataflow research indicates multithreading as a means to build hybrid architectures that combine features of dataflow and von Neumann execution models.

Despite recent architectural advances that support fine-grain parallelism and latency tolerance, challenges such as thread scheduling still remain. Multithreading's success depends on how quickly and efficiently context switching can be supported. This is possible only if threads are resident in fast but small memories—cache, which limits the number of active threads and thus the amount of latency that can be tolerated. (See "The utility of cache" sidebar.) Dataflow scheduling's generality, however, makes it difficult to fetch and execute a logically related thread sequence through the processor pipeline, which means registers can't be used across thread boundaries. Relegating scheduling and storage-management responsibilities to the compiler alleviates this problem somewhat. In conventional architectures, reduced memory latencies are achieved through (explicit) programmable registers and (implicit) high-speed caches. Adding caches, or register-caches, to the dataflow framework could better exploit parallelism and hardware utilization.

### The utility of cache

Cache utility stems from locality of reference, which indicates that all programs favor a portion of their address space at any instant in time. There are two dimensions of locality: *Temporal locality* exists when an object referenced during the last time interval is referenced again in the next time interval. If $t$ is a point in time and the set $D$ was referenced during the time interval $(t - \Delta t, t)$, it is likely that $D$ will be referenced again during the interval $(t, t + \Delta t)$. For example, recurrent use of instructions, as in a loop, produces temporal locality. *Spatial locality* exists when the next reference is to a virtual space adjacent to the previous reference. If $m$ was referenced at time $t$, then the reference at time $t + 1$ would tend to be from $(m - 1, m + 1)$. Spatial locality occurs when instructions that execute consecutively are grouped together.

**SUGGESTED READINGS**


Dubois, M., and S. Thakkar, "Cache Architectures in Tightly Coupled Multiprocessors," *Computer*, special issue, Vol. 23, No. 6, June 1990.

Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, New York, 1993.

Smith, A.J., "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, 1982, pp. 473–530.

Tartalja, I., and V. Milutinovic, *The Cache Coherence Problem in Shared-Memory Multiprocessors, Software Solutions*, IEEE Computer Society Press, Los Alamitos, Calif., 1996.

Tomasevic, M., and V. Milutinovic, "Hardware Approaches to Cache Coherence in Shared Memory Multiprocessors," Parts 1 & 2, *IEEE Micro*, Oct.–Dec. 1994.


## Dataflow architecture

In the dataflow model, tokens carry data values and travel along the arcs connecting various instructions in the program graph. The arcs are assumed to be FIFO queues of unbounded capacity. Impractical for direct implementation,[2] the dataflow execution model instead is either static or dynamic.[3] To date, researchers categorize dataflow machine organizations as pure dataflow, macro dataflow, or hybrid[2] (see "Dataflow system features" sidebar). The major differences between pure dataflow and classical 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 result primarily from a direct matching scheme to reduce the hardware overhead (see "Token matching" sidebar).

## *Dataflow system features*

Table A lists key features and characteristics of six dataflow systems in three categories.

Table A. 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] | Direct matching of tokens using rendezvous slots in the frame memory (Explicit Token Store model). Associates three temporary registers with each computation thread. Sequential scheduling implemented by recalculating scheduling paradigm using 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[2] | A separate match memory maintains match counts for the rendezvous slots, and each operand is stored separately in the frame memory. Use of repeat unit reduces the overhead of copying tokens and represents a thread of computation (macro actor) as a linked list. Use of a register file temporarily stores values within a thread. |
| Macro-dataflow | Integration of a token-based circular pipeline and an advanced control pipeline. Direct matching of tokens. | EM-4[3] | Use of macro actors is based on the strongly connected arc model, and macro actors are executed using advanced control pipeline. Use of registers reduces the instruction cycle and the communication overhead of transferring tokens within a macro actor. Special thread library functions Fork and Null spawn and synchronize multiple threads. |
| Hybrid | Based on conventional control-flow processor, that is, 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 (Join). Message handlers implement interprocessor communication. Can use both conventional and dataflow compiling technologies. | P-RISC[4] | Multithreading is supported with a token queue and circulating continuations. Context switching can occur on every cycle or when a thread dies due to Loads or Joins. |
| | | *T[5] | Overhead is reduced by off-loading the burden of message handling and synchronization to separate coprocessors. |
| | | TAM[6] | Placing all synchronization, scheduling, and storage-management responsibility under compiler control—for example, exposes the token queue (continuation vector) for scheduling threads. The compiler produces specialized message handlers as inlets to each code block. |


*References*

1. G.M. Papadopoulos and K.R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1991, pp. 342–351.
2. V.G. Grafe and J.E. Hoch, "The Epsilon-2 Multiprocessor System," *J. Parallel & Distributed Computing*, Vol. 10, 1990, pp. 309–318.
3. S. Sakai et al., "An Architecture of a Dataflow Single Chip Processor," *Proc. 16th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1989, pp. 46–53.
4. R.S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?" *Proc. 16th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1989, pp. 262–272.
5. R.S. Nikhil, G.M. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1992, pp. 156–167.
6. 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*, ACM Press, 1991.


The hybrid organization is a radical departure from classical dynamic architectures in the sense that tokens only carry tags, and the architecture is based on conventional control-flow sequencing. Researchers regard such architectures as control-flow machines extended to support fine-grained dataflow capability. Unlike the pure dataflow organization, hybrid machines provide a limited token-matching capability through special synchronization primitives.

A token-based circular pipeline integrated with an

## *Token matching and the Monsoon system*

Direct matching—a simplified process of matching tags—eliminates associative search used in dynamic dataflow architectures to match pairs of tokens representing an instruction's operands. Storage (called *activation frames*) is dynamically allocated for all the tokens that can be generated by a code block. Location usage in a code block is determined at compile time; however, activation frame allocation is determined during runtime.

With this scheme, any computation is completely described by an instruction pointer (IP) and an activation frame pointer (FP). The pair of pointers, `<FP.IP>`, is called a continuation and corresponds to the tag part of a token. A typical instruction 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 displacement is also accompanied by an input port (left/right) indicator that specifies the appropriate input arc for a destination actor.

In Monsoon, the tokens-matching process is based on the Explicit Token Store model.[1] An example of the ETS code block invocation and its corresponding instruction and frame memory is shown in Figure A. 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 displacements for the destination instructions. The system achieves the actual matching process by checking the disposition of the slot in the frame memory pointed to by **FP + *r***. If the slot is empty, the system writes the token's value in the slot and sets its presence bit to indicate that the slot is full. If the slot is already full, the system extracts the value, leaving the slot empty, and executes the corresponding instruction. The system communicates the result tokens that the operation generates to the destination instructions by updating the IP according to the displacements 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 pure dataflow organizations, the token-matching mechanism provides the full generality of the dataflow model of execution, and therefore the hardware supports it.

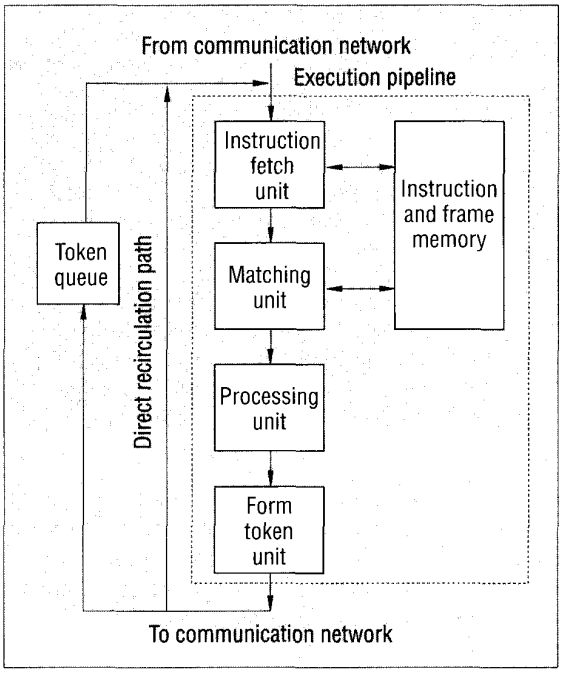



Figure B. An organization of a pure-dataflow processing element.

The fundamental Monsoon design concerns the mapping of activation frames among processors (see Figure B). A Monsoon processor is an eight-stage pipeline. On each processor cycle, a token enters the pipe and—after eight cycles—zero, one, or two tokens emerge from the pipeline. One output token can be readily circulated back into the pipe. Tokens that do not circulate back to the pipeline are either inserted into the token queue or sent to the destination processor.

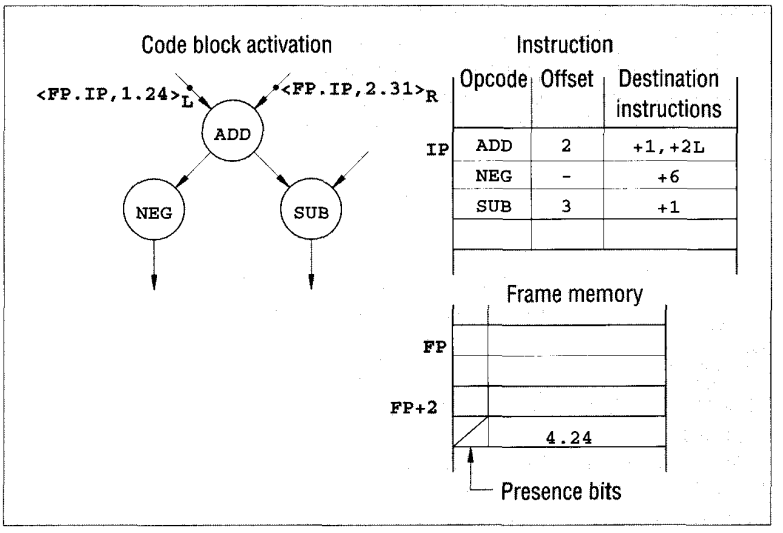



Figure A. ETS representation of a dataflow program execution.

*Reference*

1. G.M. Papadopoulos and K.R. Traub, "Multithreading: A Revisionist View of Dataflow Architectures," *Proc. 18th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1991, pp. 342–351.

### Classification of dataflow architectures

We categorize dataflow architectures as feedback and nonfeedback organizations. Feedback architectures require that the output arcs of an instruction be cleared of tokens before the instruction executes. This requirement is usually enforced by acknowledgment signals that prevent operands from being overwritten before they are consumed. Nonfeedback architectures instead provide space for multiple tokens on each arc, and they also provide a mechanism for identifying and matching tokens that must be used together. A static dataflow system is often implemented based on a feedback paradigm.

advanced control pipeline characterizes the macro dataflow organization, a compromise between pure dataflow and hybrid. This architecture exploits a more coarse-grained parallelism by incorporating control-flow sequencing into the dataflow approach.

## Cache in conventional multiprocessor systems

Shared-memory multiprocessors increase computing power and throughput cost-effectively, although memory contention, communication contention, and latency time problems can increase memory-access times and lower processor utilization and throughput. As a solution, cache memories have effectively reduced the average memory-access time. For a cache-based multiprocessor organization, the literature proposes *private-* and *shared-cache* multiprocessors. In a private cache, however, a drawback to the simplicity and reduced access time of shared code and data structures among the processes is that the sharing can result in a coherence problem.

Most multiprocessors are multiprogrammed such that each process uses the processor for a time slice, or quantum, in a round-robin fashion. Because processors are typically alternated in use, a significant fraction of cache misses result from task switching. The terms *warm-start* and *cold-start* refer to the miss ratio starting with a full cache and the miss ratio starting with an empty cache, respectively.

## Cache in dataflow systems

We knew that cache effectively uses the presence of locality in control-flow programs. The question underlying our research was, can it do likewise in dataflow programs?

A dataflow program is represented as a directed graph, $G = G(N, A)$, where nodes (or actors) in $N$ indicate operations, and directed arcs in $A$ indicate data dependencies among the nodes. Data packets called tokens convey the operands from one node to another.[3] Dataflow programs generally offer locality of effect and freedom from side effects. Moreover, dataflow procedures are not history-sensitive. These characteristics result from dataflow's asynchronous nature.

Dataflow executions are either static or dynamic (see "Classification of dataflow architectures" sidebar). A static dataflow program is a directed acyclic graph in which repetitively executed subgraphs must be unraveled before execution. A dynamic dataflow program (a directed cyclic graph) contains cycles and/or runtime procedure calls for loop constructs. In static programs, temporal locality for instructions does not fully apply, because each instruction is executed only once. Instead, locality can be established on the basis of simultaneity of execution by assigning a weight to each dataflow node, where a weight represents a node's distance from the root. The nodes with the same weight are then clustered on the same page. This strategy partitions the dataflow graph into $K$ horizontal layers such that the nodes in layer $K_i$ are data-independent from each other, meaning they can likely be executed in parallel, and are data-dependent on nodes in layer $K_i - 1$ $(1 < i \leq K)$. For a dynamic dataflow program, locality applies on the grounds of instruction recurrence (such as loops) and simultaneity of execution.

### Temporal locality in dataflow programs

If a loop comprises a locality pattern, then the loop's complete execution appears as repetitions of that pattern.[4] These repetitions may be partially distinct (for example, `DoAcross`) or overlapping (for example, `DoAll`), depending on the loop's data dependencies and on the underlying dataflow architecture. In a sequential environment, loop instructions are reused in iterations. If instructions are similarly reused in a dataflow environment, temporal locality exists, as shown in Figure 1, where a loop is executed on a feedback processor. Figure 1c represents each instruction by a line number, operation code, position for operand values, and destination addresses.

A feedback processor will restrict or prohibit overlapped successive iterations to eliminate possible operand overwriting. A nonfeedback processor, however, lets a loop unwind naturally during execution time so that only data dependencies from one iteration to the next constrain successive iteration initiation. Figure 2 shows the loop execution, depicted in Figure 1b, on a nonfeedback processor. Here the complete loop execu-

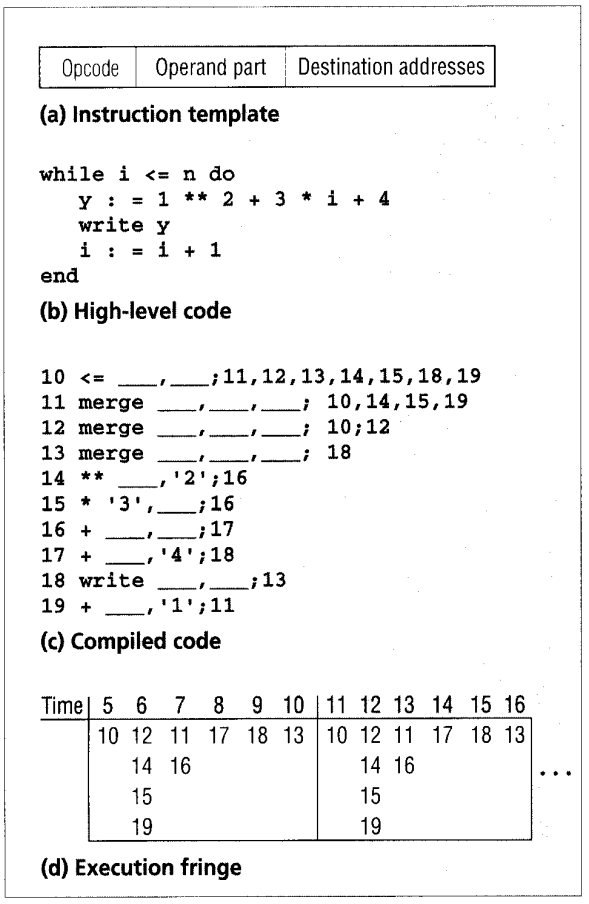



Figure 1. Execution of a loop on a feedback processor: (a) instruction template; (b) high-level code; (c) compiled code; (d) execution fringe.

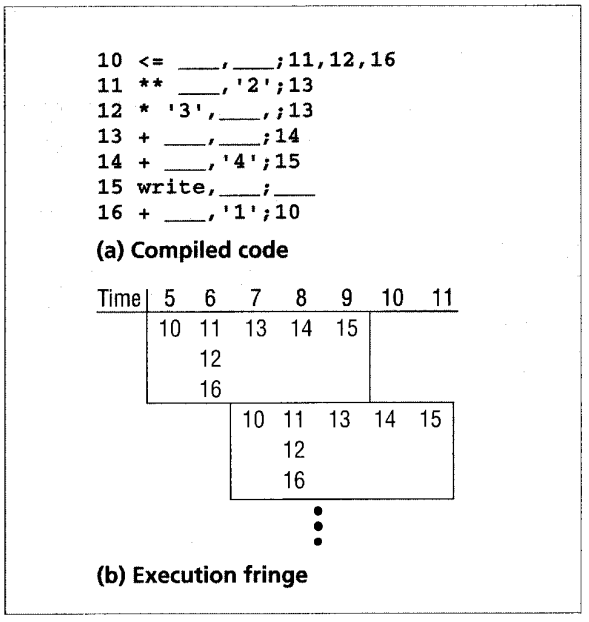



Figure 2. Execution of a loop on a nonfeedback processor: (a) compiled code; (b) execution fringe.

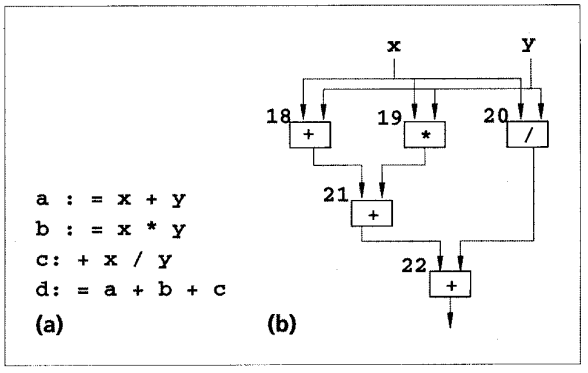



Figure 3. Spatial localities in a dataflow program: (a) high-level code; (b) dataflow graph.

tion appears as overlapped repetitions of the locality pattern. Temporal locality can be exploited if and only if the successive locality patterns use the same copy of instructions in the memory.

### Spatial locality in dataflow programs

Straight-line code—in fact, any section of the code—may produce exploitable spatial localities in a dataflow environment. (See "Execution fringe and reference fringe" sidebar). An exploitable spatial locality is a set of instructions that would constitute a spatial locality if grouped together in the virtual address space. An activity path, determined by the program's data dependencies, represents the locality. For example, the code in Figure 3a may be compiled and encoded into the graph of Figure 3b, which is summarized for this discussion. This code produces the activity paths `{(18, 21, 22), (19, 21, 22), (20, 22)}`. Each path will be an actual spatial locality if and only if the instructions in that path are grouped together in the virtual address space.

Spatial localities in a dataflow graph can be easily exploited. For example, the vertically layered allocation algorithm introduced by Lee and colleagues[5] can be modified to detect spatial locality in a dataflow graph. The modification is based on (1) assigning data-dependent nodes to the same vertical layer or frame and (2) combining heavily data-dependent frames together into a virtual address frame. Thus, clustering sequential nodes into a single frame exploits simultaneity of execution, and combining the frames into the same virtual frame based on their degree of connectivity and the physical characteristics of the underlying architecture enhances spatial locality.

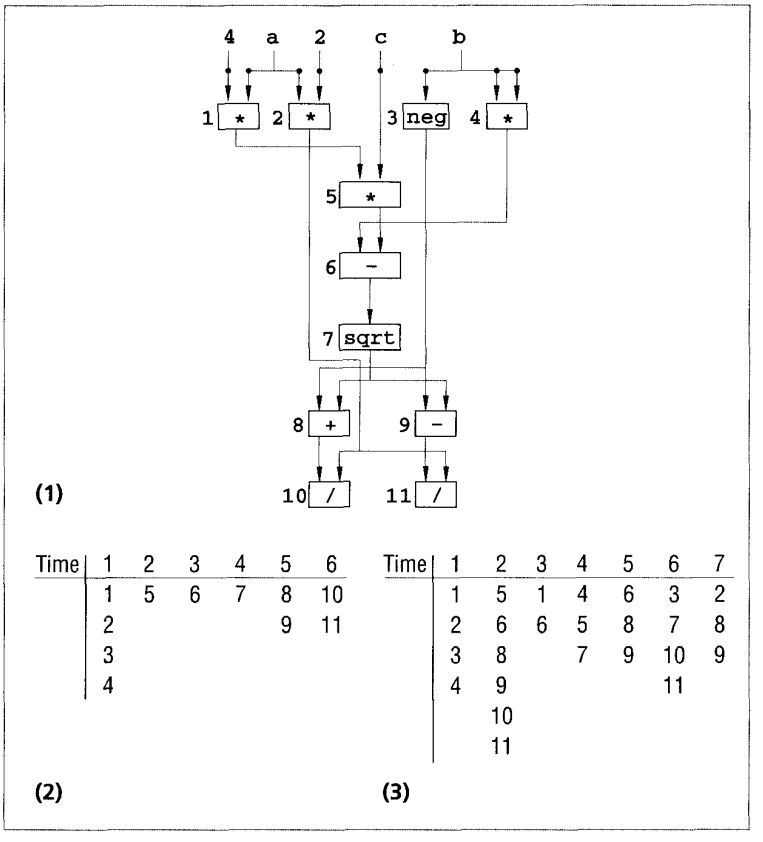
### Partitioning programs into threads

Partitioning programs into multiple sequential threads is important because a thread defines the granularity of a computation and thus the basic unit of work for scheduling. Each thread has an associated cost, which directly affects the amount of overhead required for synchro-

## *Execution fringe and reference fringe*

To show temporal and spatial localities in dataflow programs, we can employ two types of memory traces: execution fringe and reference fringe. The execution fringe records when an instruction begins execution; the reference fringe records when an instruction is referenced. Both fringes have two dimensions: time and degree of parallelism.

Figure C1 shows a dataflow graph for a quadratic equation. Assuming unit execution time for each instruction and a feedback processor, both the execution fringe and the reference fringe for the graph's complete execution are given in Figures C2 and C3, respectively. At time 3, a new reference to Instruction 1 is made because its results from a previous reference are consumed by Instructions 2 and 5. In a dataflow environment, an instruction may be referenced but not yet ready to execute. For example, comparing the execution fringe of Figure C2 to the reference fringe of Figure C3 shows that Instruction 6 is referenced by the arrival of an operand at time 2 but does not begin execution until time 3. This delay is caused by the necessity of Instruction 6 to wait for the result from Instruction 5.




| Time | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 1 | 5 | 6 | 7 | 8 | 10 |
| | 2 | | | | 9 | 11 |
| | 3 | | | | | |
| | 4 | | | | | |

| Time | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | 1 | 5 | 1 | 4 | 6 | 3 | 2 |
| | 2 | 6 | 6 | 5 | 8 | 7 | 8 |
| | 3 | 8 | | 7 | 9 | 10 | 9 |
| | 4 | 9 | | | | 11 | |
| | | 10 | | | | | |
| | | 11 | | | | | |

Figure C. Dataflow graph of a quadratic equation: (1) dataflow; (2) execution fringe; (3) reference fringe.

nization and context switching—parameters needed by a cache controller to enforce the prefetching and replacement policies. Therefore, a partitioning algorithm should maximize parallelism while minimizing the overhead required to support the threads. In dataflow architectures, context switching is free within the limits of the cache size because each token carries its context information.

Many control-flow designs use multithreading for tolerating high-latency memory operations.[3] The thread definition depends on the language and criteria used for context switching. For example, in the multiple contexts schemes of Weber and Gupta,[6] the partitioner divides a parallel loop into sequential processes, and context switching occurs when the system requires main memory access after a cache miss. Thread granularity in this model is coarse, thereby limiting the amount of parallelism that can be exposed. On the other hand, nonstrict functional languages for dataflow architectures, such as Id, complicate partitioning due to feedback dependencies that must be dynamically resolved. These situations arise because functions or arbitrary expressions can return results before all operands are computed (for example, I-structure semantics[2]). Therefore, the partitioner needs a more restrictive constraint to partition a nonstrict program into threads.

Another partitioning issue is maximizing the exploitable parallelism. The partitioner attempts to group into a thread only those instructions where little or no exploitable parallelism exists. Also, longer thread lengths result in longer intervals between context switches, which increases the spatial locality and the resource utilization. Data dependencies crossing thread boundaries should be used to improve the spatial locality and/or performance during execution.

An example of a partitioning algorithm that converts dataflow graph representation of programs into threads is that of Schauser and colleagues,[7] whose scheme used dual graphs: directed graphs with data, control, and dependence arcs. A data arc represents the data depen-

dency between producer and consumer nodes, a control arc represents the scheduling order between two nodes, and a dependence arc specifies long latency operation due to message handlers—sending/receiving messages across code block boundaries. The partitioning involves first grouping the nodes based on dependence sets. This results in a safe partition, with the following characteristics:

- Partition output is not produced before all inputs are available.
- When the inputs to the partition are available, all partition nodes execute.
- No arc connects a partition node to an input node of the same partition. The partitions merge into larger partitions based on rules that generate safe partitions.

When the general partitioning is complete, optimization reduces the synchronization cost. In short, the partitioner output is a set of threads where the nodes in each thread execute sequentially and the synchronization requirement, determined statically, occurs only at the beginning of a thread.

## *Design principles for dataflow caches*

In the static dataflow paradigm, an actor executes only when all the tokens are available on its input arcs and no tokens exist on any of its output arcs. Because only one instance of the node will be fired, regardless of the number of instructions referenced at any time, localities can be exploited and enhanced more effectively by concentrating on the dataflow program's execution fringe rather than its reference fringe. Cache implementation in the static model depends largely on the way we interpret programs' execution fringes.

The dynamic dataflow approach, however, permits multiple node activation during runtime. To distinguish between different nodes, each token has a tag that identifies the context in which a particular token was generated. An actor is executable when its input arcs contain a set of tokens with identical tags. In a dynamic environment, both reference and execution fringes can help us understand localities and hence caches. To exploit temporal and spatial localities in dynamic dataflow programs, we must separate instruction memory from the operand memory. However, asynchronous dataflow instructions means frequent context switching and a lack of temporal and spatial localities in accessing instruction and operand memories.[8]

To cope with these problems, a designer must control the load and manage resources in the processing elements. This involves partitioning the dataflow graphs into subgraphs and allocating subgraphs among processing elements, combined with controlling subgraph activation in a processing element. Asynchronous dataflow instructions mean that node addresses in a dataflow graph can be set as desired without affecting execution outcome.

To derive full benefit of the cache organization, we should study the effectiveness of traditional replacement algorithms (such as LRU) for instruction and operand memories apt to cause incorrect replacement. This means that a properly load-controlled PE requires a sophisticated deterministic algorithm to replace dataflow blocks.

Finally, because operand memory is crucial in achieving satisfactory dataflow machine performance, the operand cache must be effectively managed. In a dataflow machine, it's as necessary to maintain spatial locality for the input arguments of a code block (frame) as it is to maintain spatial locality for the result tokens of the code block. Cache management must keep track of several active frames to avoid cache misses in accessing arguments while storing the results.

### DATAFLOW CACHE EXAMPLES

Two examples of dataflow systems in which designers have introduced cache memory are the Dataflow Machine-II[8] and the Super-Actor Machine.[9]

#### *Dataflow Machine-II*

Designers introduced cache in the DFM-II (see "Dataflow Machine-II" sidebar) on the basis of four design principles: controlling the number of active processes, partitioning dataflow graphs, applying block-structured operand memory, and using a suitable replacement policy. With these principles, even small instruction and operand caches can achieve a sufficiently low miss ratio.

*Operand cache.* The instruction memory, the operand memory blocks, and the operand cache memory are each a group of $S$ sets (the number of stages in the instruction pipe determines the value of $S$; see Figure D in the "Dataflow Machine-II" sidebar). The system assigns each operand memory block to a process, and the active process controls the cache memory.

The system organizes OM cache into two levels of set-associative memories (see Figure 4): an operand

## *Dataflow Machine-II*

Figure D shows a DFM-II processing element block diagram. It consists of a circular pipeline of four stages: result packet unit (RPU), key matching unit (KMU), data memory unit (DMU), and functional unit (FUU). The system forwards a result packet (RP)—an operand composed of a value and a destination (combination of the instruction address and color)—from the FUU to an RP buffer register (RPB) or to the RP memory (RPM) depending on the KMU's status. The KMU attempts to match operands of a dyadic operation by comparing the destination in the RPB with locations in the key memory (KM).

With a successful match, the KMU erases the corresponding destination in the KM. Otherwise, it stores the result packet's destination in the KM. In either case, the destination fetches the operation code and the new destination of the corresponding instruction from the instruction memory (IM). The operation code, the destination, the key matching result (match/no match), and the operand value form an instruction packet. If the key matching result shows a match, the system activates the DMU to read out a mate operand from the data memory (DM) to complete the instruction packet. Otherwise, the system stores the operand value in the DM. For a monadic operation, the KM and DM are not used.

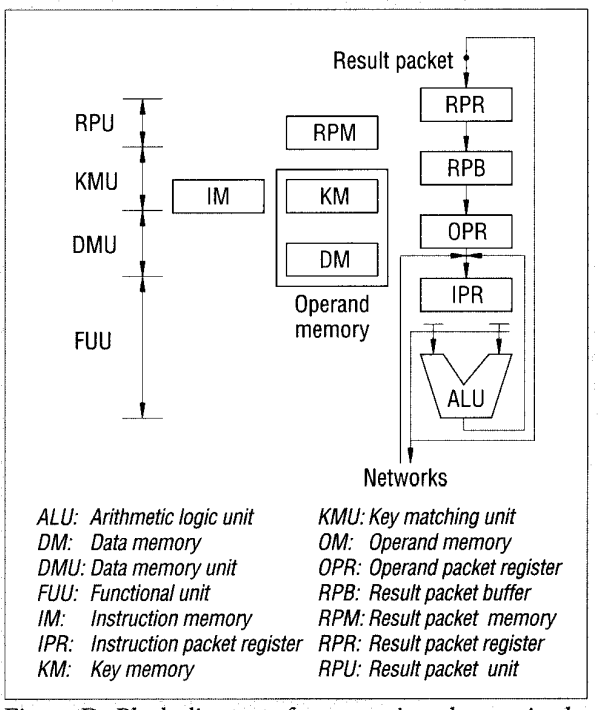



Figure D. Block diagram of a processing element in the Dataflow Machine-II.

address translation buffer and an operand cache memory. The OCM is composed of $p$ planes, each with the same structure as the OATB. An OATB entry corresponds to $p$ entries in the OCM blocks with the same relative addresses, and each OATB can be in one of the active, waiting, or idle states.

For address resolution, the token tag selects a set within an OATB. The system associatively matches the key portion of the operand token against key portions of the active or waiting entries in the designated set. If it finds no match, it allocates an idle entry in the set (if one exists), changes its validity bit, and loads $p$ blocks of the cache corresponding to that entry from the OM. If there is no available entry, the system selects one of the waiting entries in the set and swaps $p$ corresponding cache blocks.

*Instruction cache.* The instruction cache organization, its replacement policy, and its address resolution procedure are similar to those of the operand cache memory (see Figure 4). Instruction cache is composed of an instruction address translation buffer, an instruction-memory-block token count memory (ITCM), and an instruction cache memory. The IATB is a set-associative memory, and has the same structure as the ICM. The system assigns an entry in the IATB to a dataflow block, and allocates an entry of the ITCM to a process as defined by the dataflow blocks. The system uses a portion of the operand address to address a set of the IATB blocks. Again, the key portion of the result token associatively searches the selected blocks. In a successful match, the system accesses the instruction from the corresponding instruction cache block. Otherwise, it selects an idle (or waiting) cache block to house the new instruction block.

Takesau[8] simulated this cache design and analyzed its performance measures, then experimented with three different programs with varying degrees of parallelism. His model tried to overlap cache block swapping with normal cache accesses to allow multiple processes to share the same program blocks. He observed that if locality can be enforced in the programs, the IM cache of 1K words and the OM cache of 2K words suffice in offering low miss ratios. In addition, because of frequent interprocess switching in the dataflow environment, classical replacement algorithms could replace active blocks, which encourages development of more sophisticated replacement algorithms.

### *Super-Actor Machine*

(See the "Super-Actor Machine" sidebar). Hum and Gao[9] proposed the operand and instruction caches in this system on the basis that (1) in a multiprocessor system, the explicit use of programmable registers results in reduced memory latencies, and (2) conventional caches cannot handle multithreaded architectures. Therefore,

SAM features a set of high-speed memories for data and instructions, register caches (R-caches). These are organized both as a register file and a cache (see Figure 5). Similar to general register addressing in conventional CPUs, the execution unit accesses R-caches using relatively short addresses, and from the actor processing unit perspective, their contents are tagged just as in conventional caches. R-caches are transparent to the compiler, and R-cache line allocation occurs at runtime, aided by cache update and replacement algorithms. Afterward, the super-actor execution unit can directly access the locations in an R-cache line as if they were general registers.

In this scheme, a partitioner partitions a dataflow program into a number of super-actors (instruction threads). A super-actor is executable only if it satisfies the firing rule and the spatial locality conditions—that is, the input data resides in the R-

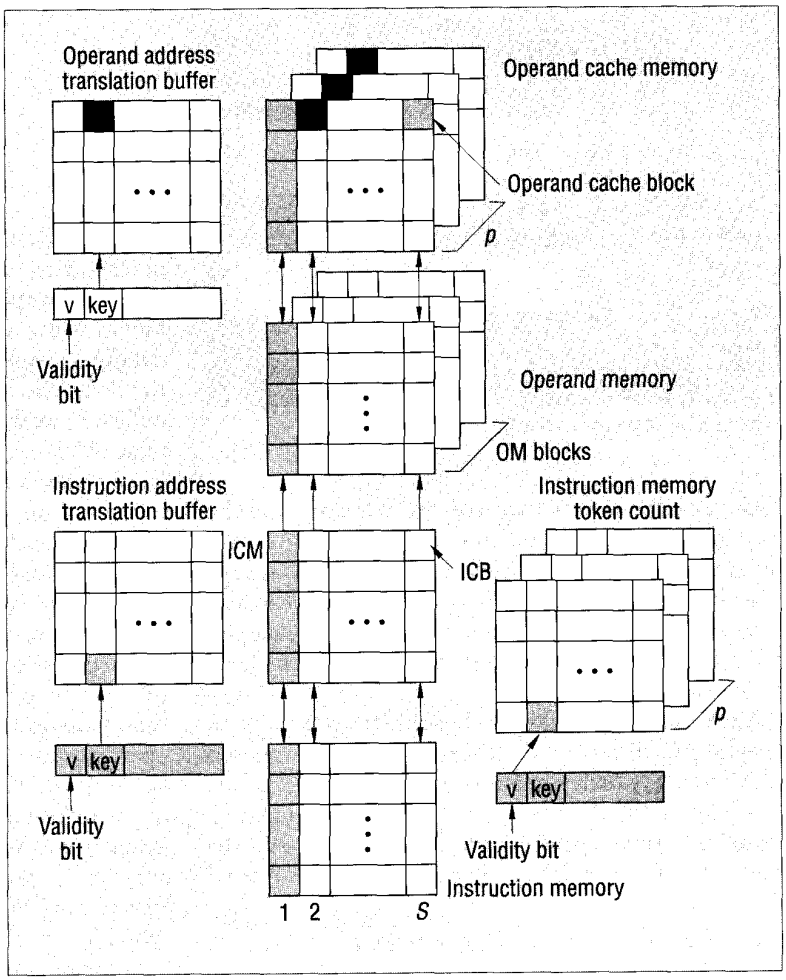



Figure 4. Operand and instruction caches for the Dataflow Machine-II.

## *Super-Actor Machine*

Super-Actor Machine (SAM) is a multiprocessor system with multithreading capabilities, in which each PE has five basic components (see Figure E): the actor preparation unit (APU), the super-actor execution unit (SEU), the long latency-actor execution unit (LEU), the actor scheduling unit (ASU), and the local main memory. The APU, coupled with an execution pipe, is responsible for ensuring that actors of the dataflow program have been fetched, processed, and are ready to be sent either to the SEU for local memory-access operations or to the LEU for nonlocal memory-access operations. The result from these units is fed to the ASU, which then sends it to the main memory or the APU for further processing.

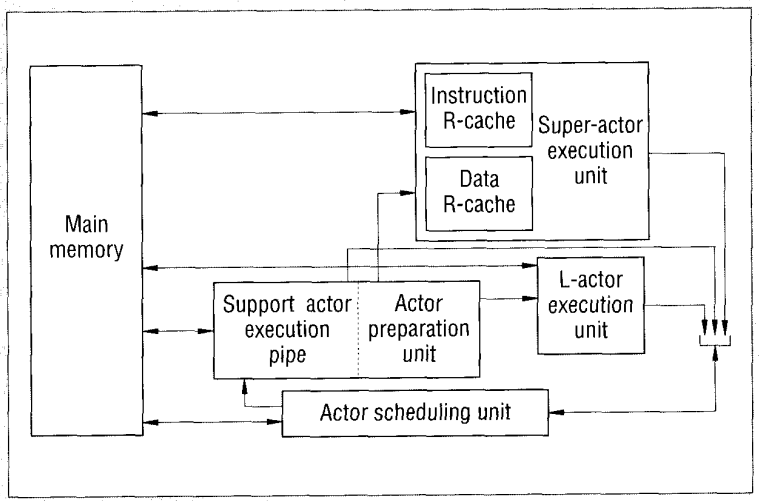



Figure E. Block diagram of a processing element in the Super-Actor Machine.

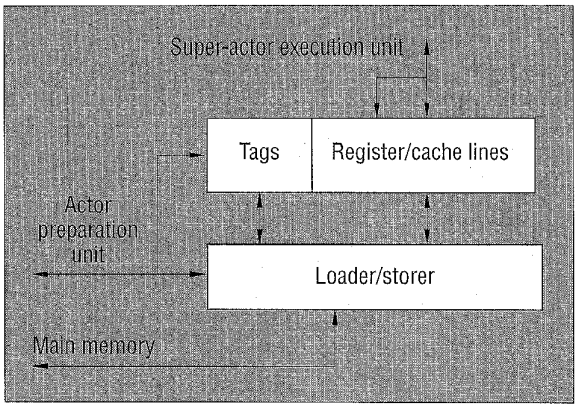


Figure 5. A register cache.

cache and the system had reserved space to hold the results. Therefore, the execution unit of the SAM will never stall when accessing instructions or data, as happens in conventional systems due to cache misses. A loader attached to the R-caches detects and schedules the ready super-actors for execution in the high-speed buffer memory.

Hum and Gao used three benchmark programs to test the effectiveness and the feasibility of the proposed architecture, which had an instruction R-cache of 1K words and a data R-cache of 1K words. The preliminary simulation results have shown the effectiveness of the register cache in hiding local memory latencies.[9]

## *Adding cache to Monsoon*

Our research focused on adding cache to Monsoon, an example of a pure dataflow system based on the concept of the Explicit Token Store (ETS). The following discussion details, from the perspective of our experiments, the architectural changes that would need to be made to Monsoon in order for cache to be effective in improving system performance.

Because the self-scheduling of instructions in a pure dataflow program precludes the use of cache, reordering those program instructions can produce synthetic localities that justify a cache. Executed instructions can produce operands relating to instructions in subsequent blocks; thus, we must consider multiple blocks of operand locations from the operand memory. These blocks are a working set. (The working set for a dataflow program is the minimum set of instructions that keep the execution unit busy, and we can determine it by analyzing the dataflow graph.[8]) We can optimize block size and working set size, for a given cache configuration, to achieve a desired performance. We found that for instruction blocks of two instructions, working sets of 4 to 8 instructions yield significant performance improvements.

The locality for the operand cache relates to the ordering of instructions in the instruction cache. When the system references a block's first instruction, it brings the corresponding block into the instruction cache. Simultaneously, the system allocates locations in the operand cache for all the operands corresponding to the working set. The operand cache will satisfy any subsequent references to the operand cache caused by the instructions in this block. (The operand cache block consists of waiting operands or empty locations.) Prefetching ensures that future stores and matches caused by executing the block instructions will occur in the operand cache.

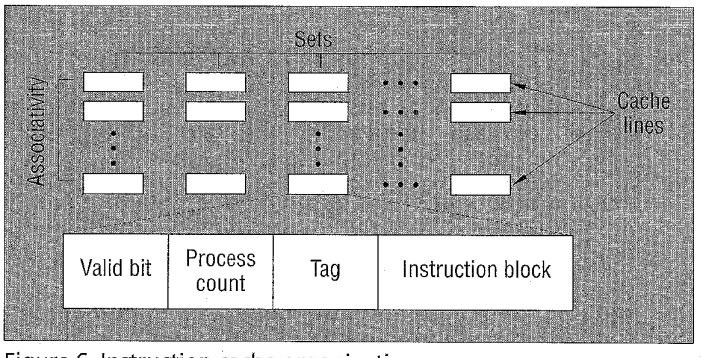


Figure 6. Instruction cache organization.

### Instruction cache design

Figure 6 shows the instruction cache structure, which resembles a conventional set-associative cache, except for the additional information. The instruction address's low-order bits map instruction blocks into $N$ sets; in each set, the blocks are associatively searched. Each block has a tag, a valid bit, and a process count. The tag and the valid bits function as in conventional caches. The process count refers to the number of activation frames referring to this block's instructions. This information could be the basis for a context-sensitive cache replacement policy: an instruction block used by many activation frames (loop iterations) is a poor candidate for replacement.

### Operand cache design

We examined the use of two-level set associativity for operand cache memories, which is similar to the DFM-II design.[8] At the first level, we partitioned the operand cache into superblocks. (At the second level the system accesses individual locations in a frame.) A superblock is composed of three parts:

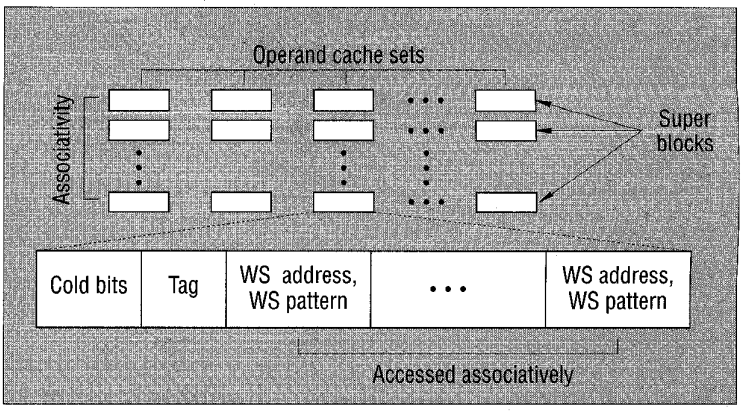


Figure 7. Operand cache organization.

- *A cold bit* that indicates whether or not the superblock is occupied. This information eliminates misses resulting from cold starts. In the dataflow model, because the first operand to arrive will be stored (written), there is no need to fetch an empty location from memory. The cold bit with a superblock allocates an entire frame (or context) and is set when the first operand is written into the frame. This eliminates the compulsory misses[12] on writes.
- *A tag* that identifies the context (frame) occupying the superblock. This is based on the frame pointer address obtained from a token tag.
- *Working set identifiers*. The system divides memory locations within an activation frame into blocks and working sets, associating an instruction block with its corresponding working set. Thus, a superblock contains multiple working sets, which are accessed at Level 2 of set associativity.

Figure 7 depicts the operand cache organization. Our simulation results indicated that no significant benefit will be gained by using two-level set associativity.

The two-level set associativity we used presented several new issues:

*Cache replacement strategies*

For working set replacement, we investigated a *used-words* policy that replaces working sets containing memory locations already used for matching operands. This policy will lead to the reuse of operand memory locations within an activation frame. At the termination of an instruction, the memory locations used for matching the input operands can be reused for matching operands of other instructions. Such an optimization could improve the operand cache memory's performance. For superblock replacement, we studied the *dead-context* replacement policy that replaces a superblock representing a completed context (or frame).

*Process control*

The operand cache must accommodate several frames (contexts or threads) corresponding to different loop iterations, as well as frames belonging to other code blocks. To minimize the possibility of thrashing, the system must carefully manage the number of active contexts (threads). The number depends on cache size and activation frame size. For tolerating remote memory latencies, however, the processor must keep a larger number of contexts. By reusing locations within a frame, we can reduce the size of an activation frame and increase the process count. We can also exploit this concept for cache memories within the scope of conventional multithreaded systems.

## Performance evaluation

Our preliminary experiments on cache memories were very encouraging.[12] We believe that dataflow machines can derive performance benefits by using cache memories in a manner similar to control-flow machines. In addition, the results for process control can be easily extended to optimal use of cache memories in superscalar (multithreaded) implementations.

To analyze cache memory's effectiveness in a dataflow environment, we developed a simulator to mimic the behavior of the ETS model enhanced by the instruction and data caches. We also developed a translator that takes IF1 graphs from a Sisal compiler and generates ETS instructions for our simulator. The IF1 graphs are further preprocessed to enhance locality.[12]

In our studies, we used a fast Fourier transform program, a matrix multiplication program, Loop 5 of Livermore Loops, and a random graph. We used the random graph to study the effectiveness of our techniques for reordering instructions. Table 1 lists the characteristics of these programs. Unlike conventional cache experiments, benchmark programs and traces for dataflow architectures are not readily available. We plan

Table 1. Characteristics (number of references) in the benchmark programs.

| Name | Instructions referenced | Operand references | I-structure references |
|---|---|---|---|
| Fast Fourier transform | 179,050 | 128,524 | 38,553 |
| Livermore Loop 5 | 158,074 | 134,620 | 28,386 |
| Matrix multiplication | 115,682 | 69,292 | 18,128 |
| Random | 281,960 | 196,204 | 36,786 |

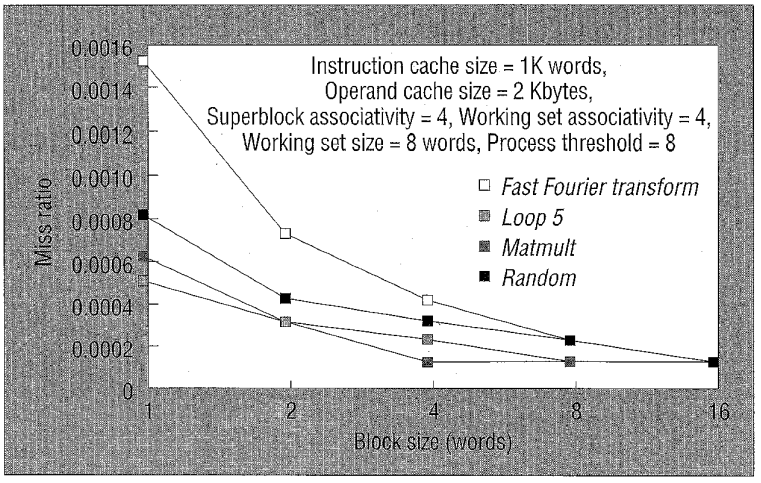



Figure 8. Miss ratio versus instruction cache block size.

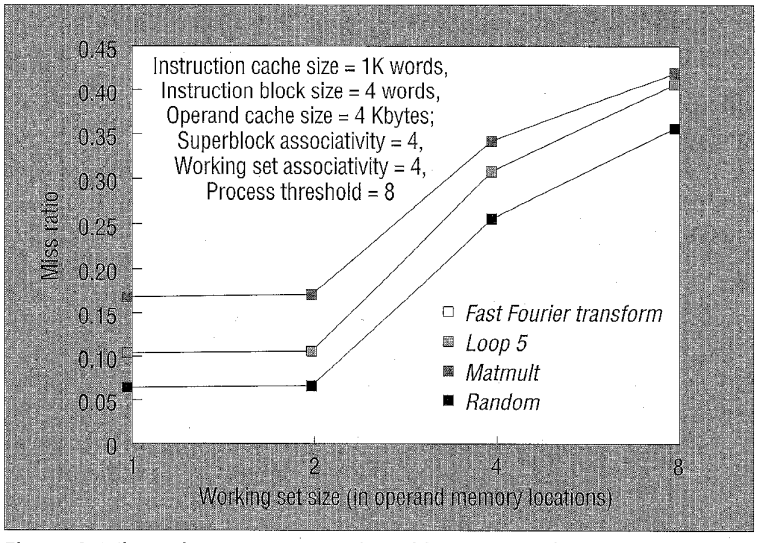



Figure 9. Miss ratio versus operand working set size. (FFT and Loop 5 coincide.)

to extend the benchmarks by rewriting some standard C or Fortran programs in Sisal.

*Experiments with cache parameters*

In our initial experiments, we evaluated the cache size, the working set, and the block size. The effects of these parameters on the miss ratio are similar to those obtained with a conventional cache. For example, as we increase instruction cache size, the miss ratio decreases. Set associativity adversely affects the instruction cache. The miss ratio increases almost linearly with instruction cache set associativity, indicating that a direct-mapping scheme is better for instruction caches.

Nearly all instruction cache misses result from cold-start misses, and increasing block size can reduce these misses (see Figure 8). Increasing block size, however, negatively affects the operand cache memories, as will be discussed shortly. Similarly, the miss ratio drops as we increase the operand cache memory size. But unlike the instruction cache, the operand cache miss ratio increases as we increase the block size (and working set size) (see Figure 9). We expected this behavior: in dataflow environments, efficient cache utilization implies that both input operands and memory for result tokens be kept in the cache. With larger working sets, it's difficult to assure that operands and memory for results are in the operand cache.

In one of our experiments, we measured the miss ratio against the associativity degree of both the superblock and the working set. Superblocks are somewhat similar to base addresses and paging of conventional virtual memory systems. Increasing the working set associativity (second level of operand cache), however, reduces the miss ratio (for fast Fourier transform and Loop 5) up to a point beyond which the miss ratio starts to increase: the initial decrease occurs when conflict misses are eliminated, while the increase at higher associativities results from fewer sets in the cache (for a given cache size). Cold-start misses in operand cache memories are eliminated because cache blocks are allocated on writes.

*Effect of process control*

Process control prevents too many active processes (contexts) from contending for the limited operand cache resources. An appropriate threshold value permits disciplined use of the cache resources and better utilization and performance. The best value for the threshold depends on the number of superblocks that can be held in the operand cache.

*Effect of replacement strategies*

We explored performance gains achieved through dead-context replacement for superblocks and used-words replacement for working sets. The dead-context replacement policy showed significant improvements for

small caches (as much as 70% fewer superblock misses when compared to random replacement policy, for 2K or smaller caches[12]). We applied the used-words scheme for the working set replacement (within a superblock). Here, the system replaces a working set (if one exists) containing operand locations that have already been used by instructions.

Figure 10 shows the percentage of operand cache misses that can be satisfied by the used-words policy. The improvement resulting from this policy indicates that dataflow systems can be designed to reuse operand cache memory locations for matching operands of more than one instruction within a frame. In other words, instead of replacing used words in a frame, the system can reuse them for storing and matching other operands. Many operand cache misses can then be eliminated. Reusing operand locations is akin to using registers to keep temporary variables during a computation, bringing the dataflow processing even closer to control-flow architecture.

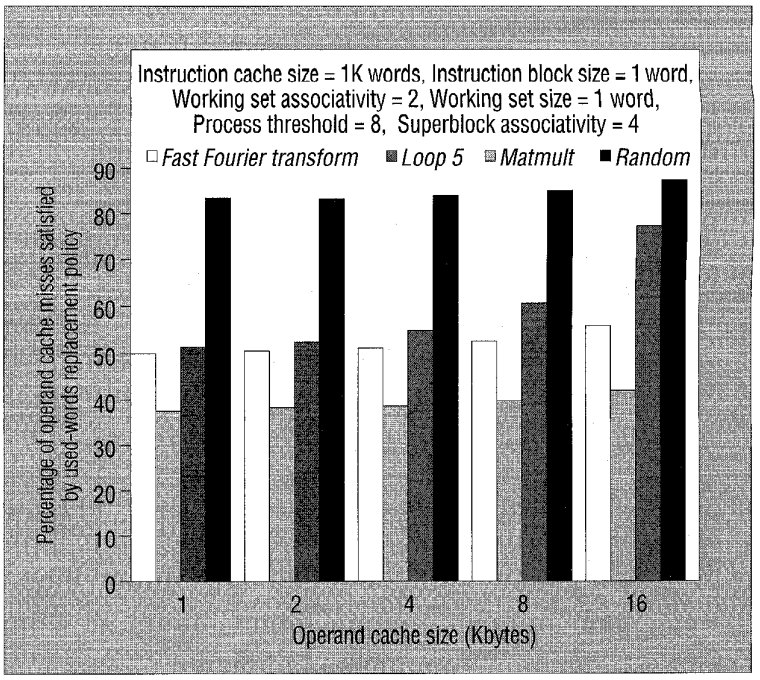



Figure 10. Significance of used-words replacement policy.

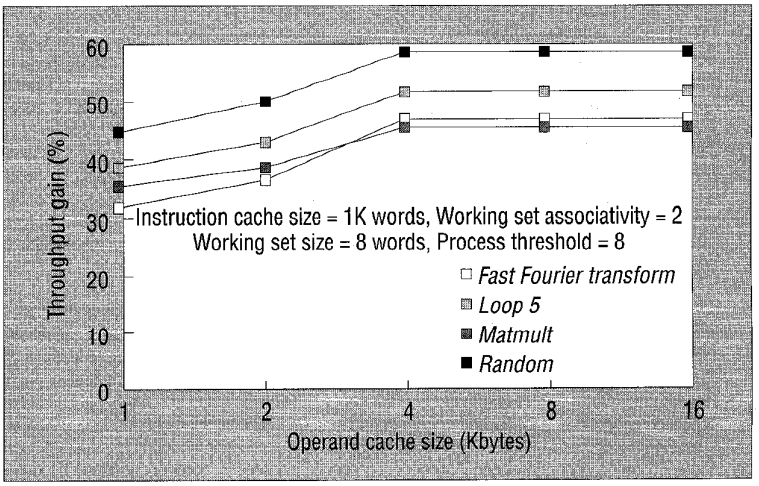



Figure 11. Uniprocessor throughput gains versus operand cache size.

### *Throughput improvement with cache*

We measured the throughput improvement against various operand cache sizes. Because of the small code size of the benchmark programs and the fact that instruction cache misses are rare, we didn't experiment with the effect of various instruction cache sizes. Figure 11 shows the gains (reduced execution times) that can be achieved relative to a platform with no cache memory.

Cache memories have proved their usefulness in conventional systems, while in dataflow models cache has shown unequaled effectiveness in improving system performance. Intuitive ideas, gained through experience with control-flow systems, generally apply to the dataflow environment, especially in a hybrid machine in which a control-driven model executes intraprocess instructions and a data-driven model executes interprocess communication and synchronization. The ultimate goal is to develop a computational environment that competes with today's RISC processors in performance without the shortcomings of control-flow organization. We can achieve this by bringing the dataflow computation model closer to the control-flow model of computation, and by using conventional technological innovations such as hierarchical memories, branch prediction, and superscalars.

Cache, if properly designed, generally shows more promise in dataflow organization when we can rely on compiler optimization to exploit locality. In a dataflow environment, designers can implement a working set concept for an active process to devise sophisticated prefetching and replacement policies. As an example, because the program can reference an instruction block in more than one context in a dataflow environment, a simple process count attached to each block can help effectively select suitable blocks for replacement.

The impact on dataflow architecture performance by memory hierarchy and caches with mixed data and instructions needs to be explored further. How a particular partitioning algorithm can be used efficiently to establish or to enhance the localities within dataflow programs is another question for today's computer architects to explore. Finally, there are few algorithms for accomplishing various tasks in dataflow caches, and researchers must direct their efforts toward this issue as well.


## Acknowledgments

This work has been supported in part by the National Science Foundation under Grants MIP-9622836 and MIP-9622593.


## References


1. N.P. Jouppi and D.W. Wall, "Available Instruction-Level Parallelism for Super-Scalar and Super-Pipelined Machines," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, 1988, pp. 272–282.
2. B. Lee and A.R. Hurson, "Dataflow Architectures and Multithreading," *Computer*, Vol. 27, No. 8, Aug. 1994, pp. 27–39.
3. G.R. Gao, L. Bic, and J.L. Gaudiot, *Advanced Topics in Dataflow Computing and Multithreading*, IEEE Computer Society Press, Los Alamitos, Calif., 1995.
4. S.A. Thoreson and A.N. Long, "Locality, a Memory Hierarchy, and Program Restructuring in a Dataflow Environment," *J. Systems and Software*, Vol. 9, No. 4, 1989, pp. 245–252.
5. B. Lee, A.R. Hurson, and T.Y. Feng, "A Vertically Layered Allocation Scheme for Dataflow Systems," *J. Parallel and Distributed Computing*, Vol. 11, No. 3, Mar. 1991, pp. 175–187.
6. 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*, ACM Press, 1989, pp. 273–280.
7. K.E. Schauser et al., "Compiler-Controlled Multithreading for Lenient Parallel Languages," *Proc. Fifth ACM Conf. Functional Programming Languages and Computer Architecture*, ACM Press, 1991, pp. 50–72.
8. M. Takesau, "Cache Memories for Dataflow Machines," *IEEE Trans. Computers*, Vol. 41, No. 6, June 1992, pp. 677–687.
9. H.H.J. Hum and G.R. Gao, "A High-Speed Memory Organization for Hybrid/von Neumann Computing," *Future Generation Computer Systems*, Vol. 8, No. 4, 1992, pp. 287–301.
10. M. Sato et al., "Thread-Based Programming for EM-4 Hybrid Dataflow Machine," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM Press, 1992, pp. 146–155.
11. Arvind, B.S. Ang, and D. Chiou, "StarT the Next Generation: Integrating Global Caches and Dataflow Architecture," *Proc. ISCA Dataflow Workshop*, ACM Press, 1992.
12. K.M. Kavi et al., "Design of Cache Memories for Multithreaded Dataflow Architecture," *Int'l Symp. Computer Architecture*, ACM Press, 1995, pp. 253–264.



**A.R. Hurson** is a computer science and engineering faculty member at Pennsylvania State University. His research interests include computer architecture, dataflow architecture, multidatabases, object-oriented databases, and VLSI algorithms. He cofounded the IEEE Symposium on Parallel and Distributed Processing. He was a member of the IEEE Computer Society Press Editorial Board and an IEEE Distinguished Speaker and now serves on the IEEE/ACM Computer Sciences Accreditation Board. Readers can contact Hurson at the Computer Science & Engineering Dept., Pennsylvania State Univ., 202 Pound Laboratory, University Park, PA 16802; hurson@cse.psu.edu; http://www.cse.psu.edu/gradbroc/gbFhurson.html.

**Krishna M. Kavi** is a professor of computer science and engineering at the University of Texas at Arlington. Previously, he was a program manager at the NSF. His research interests include computer systems architecture (dataflow systems, cache memories, multithreading, microkernels), formal specification of concurrent processing systems, performance modeling and evaluation, load balancing, and scheduling of parallel programs. He was an IEEE Computer Society Distinguished Visitor and now serves on the editorial board of the *IEEE Transactions on Computers*.

**Ben Lee** is an assistant professor in the Department of Electrical and Computer Engineering at Oregon State University. His research interests include computer architecture, parallel and distributed systems, program partitioning and scheduling, and multithreaded systems. Lee received a BE in electrical engineering from the State University of New York at Stony Brook in 1984, and a PhD in computer engineering from Pennsylvania State University in 1991. He is a member of the IEEE Computer Society and the ACM.

**Behrooz Shirazi** is a professor of computer science and engineering at the University of Texas at Arlington. Previously, he taught computer science and engineering at Southern Methodist University. Shirazi's research interests include parallel and distributed systems, dataflow computing, cache designs, task partitioning and scheduling, and computer architecture. He is on the editorial board of the *Journal of Parallel and Distributed Computing*. He is an IEEE Distinguished Visitor and an ACM Lecturer.