DESIGN OF CACHE MEMORIES FOR MULTI-THREADED DATAFLOW ARCHITECTURE

Krishna M. Kavi
Ali R. Hurson, †
Phenil Patadia
Elizabeth Abraham
and
Ponnarasu Shanmugam
The University of Texas at Arlington
†The Pennsylvania State University

Abstract

Cache memories have proven their effectiveness in the von Neumann architecture when localities of reference govern the execution loci of programs. A pure dataflow program, in contrast, contains no locality of reference since the execution sequence is enforced only by the availability of arguments. Instruction locality may be enhanced if, dataflow programs are reordered. Enhancing the locality of data references in the dataflow architecture is a more challenging problem. In this paper we report our approaches to the design of instruction, data (operand) and I-Structure cache memories using the Explicit Token Store (ETS) model of dataflow systems. We will present the performance results obtained using various benchmark programs.

Keywords. Dataflow Architecture, Explicit-Token-Store Model, Cache Memories, I-Structures

1. INTRODUCTION

It is an established fact, at least in the von Neumann arena, that the locality of reference in a program can be exploited using cache memories to achieve significant performance improvement. Until recently, dataflow architectures did not permit the use of traditional storage models, nor was it natural to consider localities in the execution sequence of a dataflow program. Of late, the trend has been to bring the dataflow computational model closer to the control-flow model. There have been a few designs of computer systems based on such hybrid execution models ([Arvind 89], [Culler 93], [Hicks 93], [Ianucci 88]). The reader is referred to numerous survey articles that have analyzed dataflow architectures (e.g., [Lee 94]).

In our research we use one such model known as Explicit-Token-Store [Papadopolous 90, 91], that permits the use of storage hierarchy within the context of dataflow.

Context-switching in dataflow architecture can occur on a per instruction basis since each datum carries a continuation (or a tag). The instruction-level context-switching capability combined with sequential scheduling provides a different perspective on dataflow architectures — **multithreading.** A thread is a sequence of statically-ordered instructions where once the first instruction in the thread is executed, the remaining instructions execute without interruption. The evolution from a pure self-scheduling paradigm of dataflow to multithreading requires locality and improved processor efficiency during remote memory accesses. In conventional architectures, the reduction in memory latencies is achieved by providing (explicit) programmable registers and (implicit) high-speed caches. Amalgamating the idea of caches or register-caches within the dataflow framework can result in a higher exploitation of parallelism and hardware utilization. In this paper we present various cache designs within the Explicit Token Store (ETS) dataflow model, and the performance resulting from the inclusion of cache memories in dataflow architecture.

In Section 2, we will briefly introduce the ETS architecture. In Section 3, we will describe instruction and operand cache memory designs with (uniprocessor) ETS architecture. In Section 4 we will describe a multi-processor ETS system and I-Structure cache memories. Results of our experiments are presented in Section 5.

2. EXPLICIT TOKEN STORE DATAFLOW

One of the important developments in the design of current dataflow proposals is the novel and simplified process of matching operands destined to an instruction based on matching tags. The basic

idea of this scheme (known as direct matching) is to eliminate the expensive and complex process of associative search used in previous dynamic dataflow architectures to match pairs of tokens comprising the operands for an instruction. In a direct matching scheme, storage (called an *activation frame*) is dynamically allocated for all the tokens generated by a code-block. The usage of locations within a code-block is determined at compile-time, although the actual allocation of activation frames is determined during run-time. 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.

An example of the ETS code-block invocation and its corresponding Instruction and Frame Memory is shown in Figure 1.

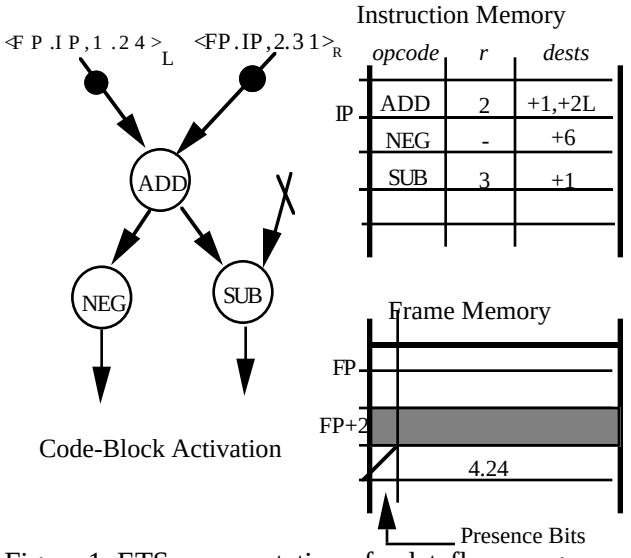


Figure 1. ETS representation of a dataflow program execution

When a token arrives at an actor (e.g., 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). The actual matching process 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 (e.g., execution of the ADD operation produces two result tokens <FP.IP+1, 3.55> and <FP.IP+2, $3.55>_L$).

A more detailed discussion of ETS and an implementation of the model (known as Monsoon) can be found in [Papadopolous 90, 91]. The fundamental design of the Monsoon is based on the mapping of activation frames among processors. Figure 2 shows an abstract view of the organization.

A Monsoon processor is an eight stage pipeline. On each processor cycle a token is entered in the pipe and after eight cycles, zero, one, or two tokens emerge from the pipeline. One of the output tokens can be readily circulated back into the pipe. Tokens that are not circulated back to the pipeline are either inserted into the token queue or sent to the destination processor via the interconnection network. In our simulation, we have used the abstract ETS model with a 4-stage pipe that includes an Instruction Fetch Unit, a Matching Unit, an ALU, and a Token Form Unit.

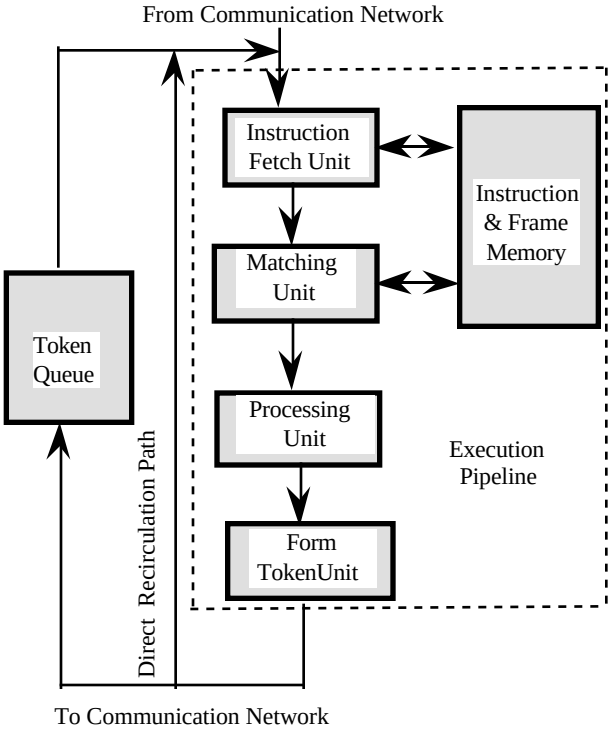


Figure 2. An organization of a pure-dataflow processing element

3. CACHE MEMORY DESIGNS WITH ETS

In general, the design of a cache is subject to more constraints and trade-offs than that of the main memory. Issues such as the *placement/replacement* policy**,** the *fetch/update* policy, *homogeneity*, the *addressing scheme*, *block size*, and the *cache bandwidth* are among those which should be taken into consideration ([Lebeck 94], [Przybylski 90], [Smith 82]). Optimizing the design of a cache memory generally has four aspects:

- Maximizing the probability of finding a memory reference's target in the cache (the hit ratio),
- Minimizing the time to access information that is residing in the cache (access time),
- Minimizing the delay due to a miss, and
- Minimizing the overheads of updating main memory, maintaining multi-cache consistency, etc.

3.1 Locality in a Dataflow Environment

The principle of locality of reference is the backbone of cache design. A data flow program in it's pure form is not amenable to a cache, primarily due to the self-scheduling of instructions for execution. However, reordering of instructions of such a program based on certain criteria [Tokoro 83] can produce synthetic localities. The recurrent use of instructions (in different activation frames) also causes the existence of temporal localities. For our initial studies, we have reordered the instructions on the basis of the time of availability of their operands. This can done by grouping instructions into execution levels (or E-levels [Thoreson 87]). Instructions that become ready (i.e., all inputs are available) at the same time unit are said to be in the same level. Instructions at level 0 for example, are ready for execution at time unit zero. Similarly, those at level 1 become ready for execution at time unit one and so on. Instruction locality can be achieved using the E-level ordering. Since the execution of an instruction may produce operands that may be destined to the instructions in the subsequent blocks, we need to prefetch more than one block of operand locations from the operand memory. We refer to these blocks as a w*orking set.* Block size and working set size are optimized for a given cache implementation to achieve a desired performance. While the optimum working set depends on the program, we have found that a block size of 2 instructions and working sets of 4 to 8 instructions yield significant performance improvements.

The locality for the operand cache is related to the ordering of the instructions in the instruction cache. When the first instruction in a block is referenced, the corresponding block is brought into the instruction cache. Simultaneously, operand locations for all the operands corresponding to the instructions in the *working set* of these instructions are prefetched into the operand cache. As a result of this, any subsequent references to the operand cache caused by the instructions within this block will be satisfied by the operand cache. Note that the operand cache block consists of a set of waiting operands or empty locations for storing the results. By prefetching, we ensure that future stores and matches caused by the execution of instructions in the block will take place in the operand cache

3.2 Instruction Cache Design

Figure 3 shows the detailed structure of the instruction cache. The structure is very similar to a conventional set associative cache, except for the additional information maintained. The low order bits of the instruction address (i.e., IP) are used to map instruction blocks into *N* sets; within each set, the blocks are searched associatively. Each block in the cache has a tag, a valid-bit and a process count associated with it. The tag and the valid bits serve the same purposes as those in conventional set-associative caches. The process count refers to the number of activation frames that refer to the instruction. This information is used in instruction cache replacement: an instruction block that is used by a large number of activation frames (i.e., loop iterations) is a poor candidate for replacement.

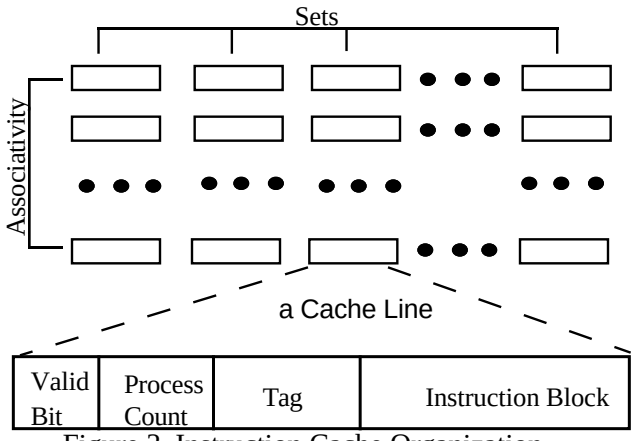


Figure 3. Instruction Cache Organization

3.3 Operand Cache Design.

Operand cache memory is used to store the activation frames associated with code-blocks, which are used for matching operands. Similar to DFM-II [Takesue 87, 92] we have examined the use of two-level set associativity to operand cache memories. At the first level of associativity, the operand cache is organized as a set of *superblocks*. Each active context (activation frame associated with a code-block) occupies a superblock. The

second level of associativity is used for accessing individual locations within a frame. Figure 4 shows the organization of the operand cache.

A superblock consists of the following information:

• *A cold bit* to indicate if the superblock is occupied or not. This information is used to eliminate misses due to cold start. In dataflow model, since 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 is used to allocate an entire frame (or context), and set when the first operand is written into the frame.

• *A Tag* which serves to identify the context (or frame) that occupies the superblock. This is based on the FP address obtained from a token tag.

• *Working set identifiers.* The memory locations within an activation frame (used for token matching) are divided into working sets as described in Section 3.1, A superblock representing an activation frame contains more than one working set, and these are accessed associatively (the second level of set associativity). Each working set of a superblock also contains a cold start bit. This bit is used to eliminate unnecessary fetches from memory when the operands are being stored in the activation frame.

The two level set associativity used in the operand cache design, presents several new issues in studying cache designs.

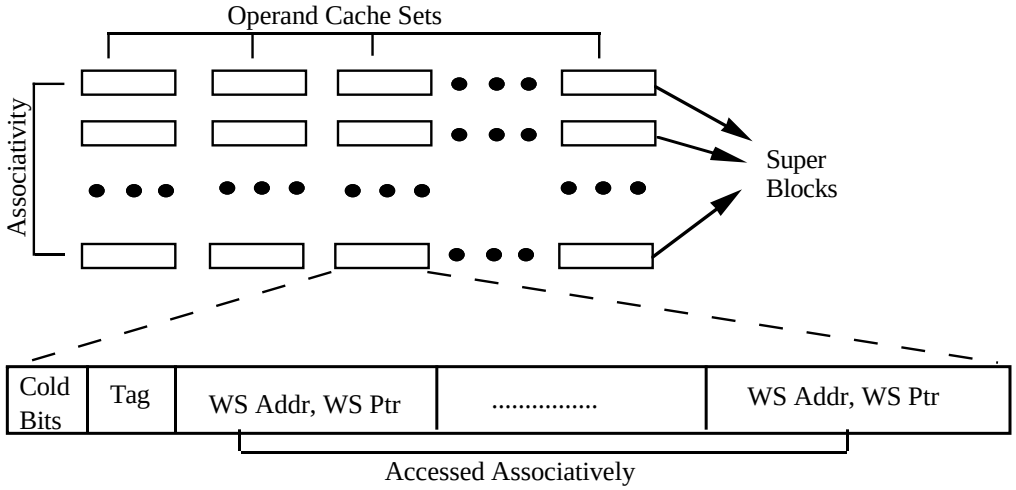


Figure 4. Operand Cache Organization

3.3.1 Cache Replacement Strategies. We have explored a few replacement algorithms with working sets within a superblock and for replacing superblocks themselves. For working set replacement, we have investigated a *used words* policy that replaces working sets containing memory locations already used for matching operands (hence will not be needed again in this activation). This has implications for our future research leading to the "reuse" of operand memory locations within an activation frame. As instructions complete their execution, the memory locations used for matching their input operands can be reused for matching operands of other instructions. We believe that such an optimization will significantly increase the performance of the operand cache memory. For superblock replacement, we have studied the *dead context replacement* policy that replaces a superblock representing a completed context (or frame) [Shanmugam 93].

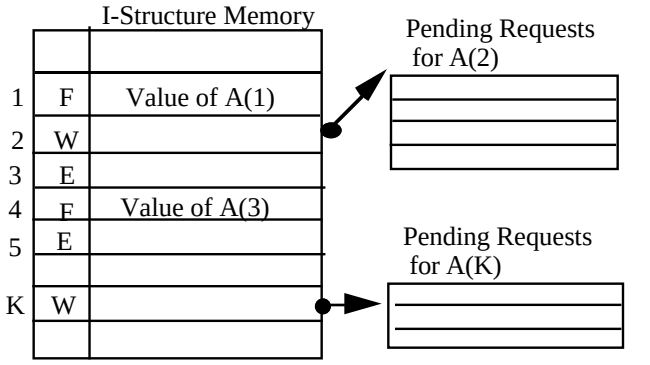
3.3.2 Process Control. The operand cache must accommodate several frames (contexts or threads) corresponding to different loop iterations, as well as frames belonging to other code-blocks. In order to minimize the possibility of thrashing, the number of "active" contexts (or threads) must be carefully managed. The number of active contexts will depend on the cache size and the size of an activation frame. It should be noted, however, for tolerating remote memory latencies, the processor must keep a larger number of contexts [Lee 93]. We believe that, by reusing locations within a frame, we can reduce the size of an activation frame and increase the process count. The concept of controlling the number of active threads can also be adopted for cache memories of conventional multi-threaded systems.

4. MULTIPROCESSOR ETS AND I-STRUCTURE CACHE MEMORIES

In this section we will describe how cache memories can be used with I-Structures in a multiprocessor environment. An I-Structure is a special kind of memory designed to handle arrays in dynamic dataflow computers. Three operations are defined with I-Structures: *allocate, i-store, i-fetch*. The *allocate (A, N)* returns an N-element empty array (i.e., each element of the structure is flagged as empty). An element of I-Structure can be assigned a value *V* no more than once using *i-store(A, I, V)*. The I-th element of the array A is now set to full. Any attempt to store values into a full element results in an error. An element of the array can be accessed using *i-fetch (A, I).* If the I-th element is already defined (indicated by the full status), the value of the element is returned. Otherwise, the request is deferred until the value is available. Fig. 5 shows an I-Structure example with pending requests for unavailable array elements [Arvind 86].

4.1. I-Structure Cache Memories.

We treat the I-Structure memory as the only shared memory in the multiprocessor environment. Processors communicate other types of data by sending and receiving tokens where the tag identifies the receiving context and the processor containing the context. Although the single assignment property of dataflow elements appears to eliminate all cache-coherence problems, caching I-Structure elements into processors does present some challenging design problems. We have investigated a directory-based protocol and a snoopy-protocol with I-Structure cache (IS-Cache) memories. Figure 6 shows the general structure of our multiprocessor system.



F: Full (Value defined); E: Empty (Value not defined)
W: Waiting (Pending read requests)

Figure 5. An I-Structure

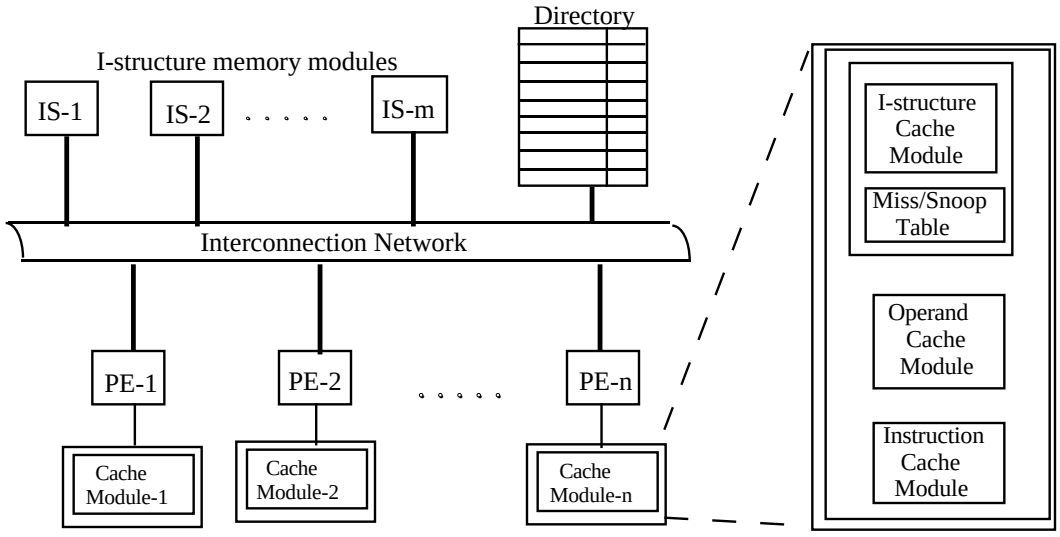


Figure 6. Multiprocessor ETS with I-Structure Cache Memories

4.1.1. Directory-Based Protocol. As with conventional directory-based methods, the I-Structure memory maintains a directory for each I-Structure block to identify the processor that is responsible for defining (or writing) the block. An I-Structure cache (IS-Cache) exists with each processor to store the I-Structure elements needed by that processor (including the elements that will be defined by the processor and the elements used by the processor but defined by a different processor). IS-Cache blocks are allocated only when the data elements are written to them. The following possibilities must be considered when a read request for a n I-Structure element is received by the I-Structure memory controller.

i). The element is absent in the I-Structure memory. Using the directory, the I-Structure controller will interrogate the processor that is responsible for defining the element. Two cases are possible.

a). The IS-Cache in the processor does not contain the requested element. Since the requested element is not defined yet, we will store the address of the element (tag) in its local table (known as the "miss" table) to reflect the pending status with the I-Structure element. Deferred requests are also maintained with the I-Structure memory in the usual manner. When the data is eventually defined in the local IS-Cache; the entry in the miss table will force a write-back to the I-Structure memory, causing the I-Structure memory controller to satisfy all deferred requests. Note that the I-Structure memory controller will need to interrogate the processor only on the first read request. We feel that maintaining a miss table is more efficient than requiring the directory to periodically poll the processor.

b). The IS-Cache in the processor already contains the requested element. This cache block is written to the I-Structure memory, causing the deferred request to be satisfied.

ii). The I-Structure memory contains the requested data element. This is possible since IS-Cache blocks will experience replacement on cache misses (i.e., the local processor is forced to write the IS-Cache block to global I-Structure memory to make room for other blocks). In this event, the I-Structure can satisfy the request directly.

We believe that compile-time analysis can be relied on to improve the performance of the directory protocol. If the read requests are scheduled sufficiently later than the writes so that the I-Structure elements from local IS-Caches are written back to the global I-Structure memory before any read requests arrive (case ii above), then there is little overhead with the directory protocol. If read requests arrive before the elements are defined, the directory protocol incurs overhead (in maintaining local "miss" tables and writing back the requested data immediately upon definition and the miss table can be excessively large).

4.1.2. Snoopy-Based Protocol. As with many snoopy protocols, each processor will snoop on a subset of I-Structure elements that are defined by the processor. A local table called "snoopy" table can be used to list the elements on which a processor snoops. As read-requests are sent to the global I-Structure memory, processors will snoop

for any requests that they can satisfy. The following two cases are possible.

a). The processor containing the requested data in its IS-Cache is successful in snooping on the request, and the request is satisfied from the processor's IS-Cache.

b). The processor containing the requested data in its IS-Cache is not successful in snooping on the request. In order to keep the snoop table small, a processor will not snoop on all elements contained in its IS-Cache. The request will be handled by the global I-Structure memory controller. We will assume that the I-Structure memory maintains a directory so that the request can be satisfied by interrogating the processor containing the data (similar to the directory protocol).

It is possible for the processor to snoop not only on the elements that are already defined in its IS-Cache, but also on the elements that will be defined in the future. This requires larger snoop tables. We believe that compile-time analysis can be used to minimize the possibility of a request for yet to be defined data, and to manage the snoop table more efficiently.

5. PERFORMANCE EVALUATION

Unlike with conventional cache experiments, benchmark programs and traces for dataflow architectures are not readily available. We have developed a translator that takes IF1 graphs from a Sisal compiler [Feo 90] and generates ETS instructions for our simulator[1]. This allowed us to use actual Sisal programs in our studies (although we could not find very large Sisal programs). The IF1 graphs are preprocessed to enhance locality as discussed earlier (Section 3.1). We have used an FFT (feo.fft) program, a matrix multiplication program, loop 5 of Livermore Loops and a random graph in our studies. The use of random graph is to study the effectiveness of our techniques for reordering instructions. We plan to extend the set of benchmarks by rewriting some standard C or Fortran programs in Sisal. Table 1 lists the characteristics of the programs used in our current experiment.

[1]We have not used IF-2 graphs since they incorporate optimizations for conventional architectures. Since our target is a dataflow instruction set, we wanted to maintain the dataflow purity in the source.

Name	#Instructions Referenced	# Operand References	# I-Structure References
FFT (feo.fft)	179,050	128,524	38,553
Livermore Loop 5	158,074	134,620	28,386
Matrix Mult	115,682	69,292	18,128
Random	281,960	196,204	36,786

Table 1 - Program Statistics

5.1. Experiments with Conventional Cache Parameters on Miss Ratios.

Initial experiments with the cache designs involved performance evaluation of various cache parameters like, cache size, working set and block size. The effects of these parameters on the miss ratio are similar to those obtained for a conventional caches.

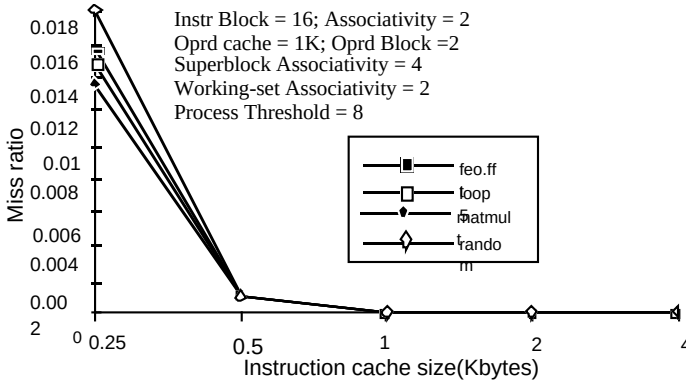


Figure 7. Instruction Cache Size vs. Miss Ratio

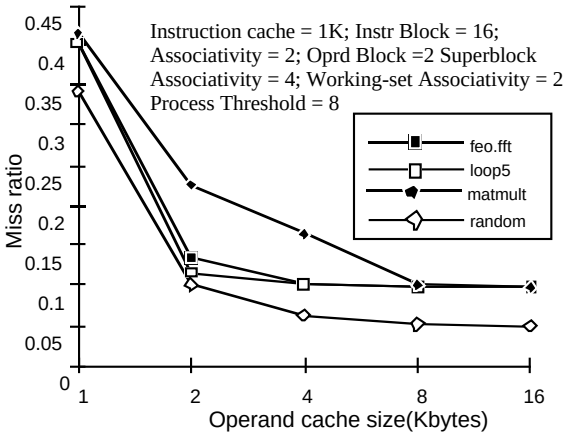


Figure 8. Effect of Operand Cache Size on Miss Ratio

This indicates that localities can be synthesized in a data flow environment. Increasing the operand and instruction cache sizes reduces the miss ratio

as can be seen in Figures 7 and 8[2] Nearly all instruction cache misses are due to "cold-start" misses, and these misses can be reduced by increasing block size as shown in Figure 9.

In dataflow, it is not only necessary to assure the presence of input operands for instructions but also assure that the destination locations for results of instructions are available in the operand cache memory. Large block sizes lead to large working sets which in turn lead to more conflict misses since the cache is shared among several frames.

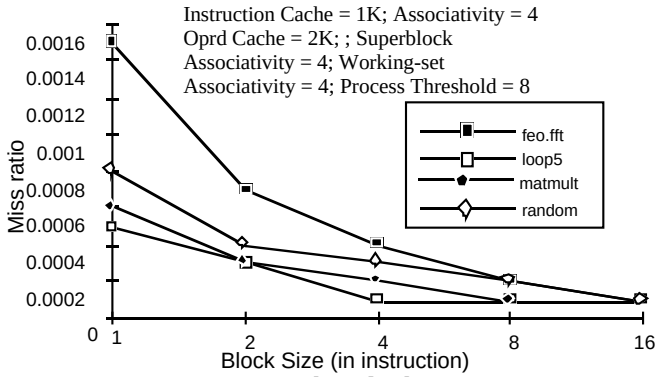


Figure 9. Instruction Cache Block Size Vs Miss Ratio

Figure 10 shows the effect of block size on the miss ratio for operand cache memories. This should be contrasted with the instruction miss ratios of Figure 9.

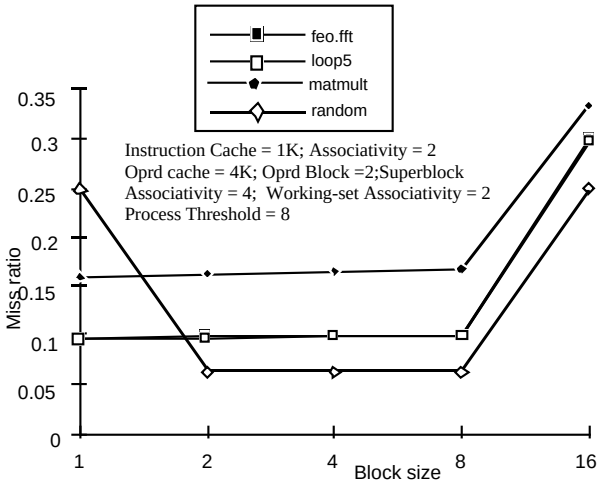


Figure 10. Effect of Block Size on Operand Cache Miss Ratio

[2]To compensate for the small sizes of the programs used in our experiments, the instruction and operand cache memory sizes are kept small.

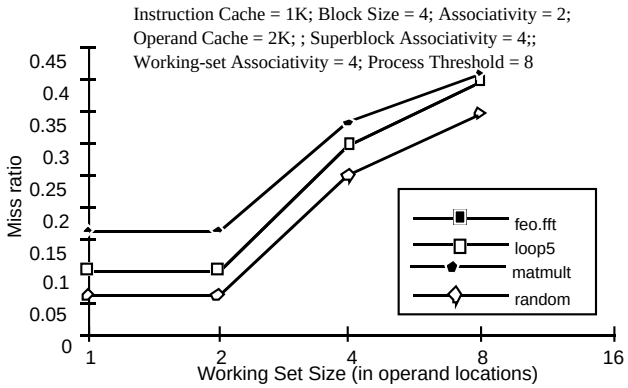


Figure 11. Operand Working Set Size vs. Miss Ratio

Figure 11 shows the effect of changing operand working set size on operand cache misses. Our results indicate that an optimal block size is 8, and working set size is 2 (we will use these sizes for the remaining experiments).

We then investigated the significance of associativity on instruction cache design. We found when the set associativity is increased beyond 2, the miss ratio increases (more details can be found in [Abraham 94], [Patadia 94]). This suggests that direct mapped caches perform well even for dataflow instructions. Since our operand cache contains two levels of associativity, we varied both associativities.

Increasing the super block associativity does not result in significant reduction in miss ratio (Figure 12). The optimal associativity depends on the block size used for the cache design. Figure 12 shows the benefit of addressing operands at two levels. The operand address space is divided into superblocks (threads/contexts/frames) and within a frame, operands are addressed using smaller addresses. We believe this gives more freedom to compilers in allocating threads (or loop iterations) to processors without losing localities.

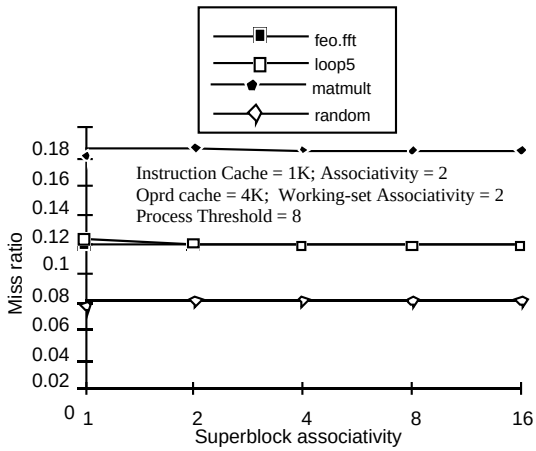


Figure 12. Superblock Set Associativity

In ETS, the significance of associativity within a context (i.e., associativity of working sets) behaves somewhat similar to that of conventional operand cache associativity. Increasing the working set associativity reduces the miss ratio (for fft and loop5) upto a point beyond which the miss ratio starts increasing (Figure 13). The initial increase is due to the elimination of conflict misses, while the increase at higher associativities is due to fewer sets (for a given cache size). Cold start misses in operand cache memories are eliminated since we allocate (not fetch) cache blocks on write (see Section 3.3).

5.2. The effect of unconventional cache parameters on miss ratio

Effect of Process Control. The motivation for introducing process control is to avoid too many active processes(or contexts) contending for the limited operand cache resources. An appropriate threshold value allows for disciplined use of the cache resources and hence better utilization and performance. This can be readily observed in Figure 14.

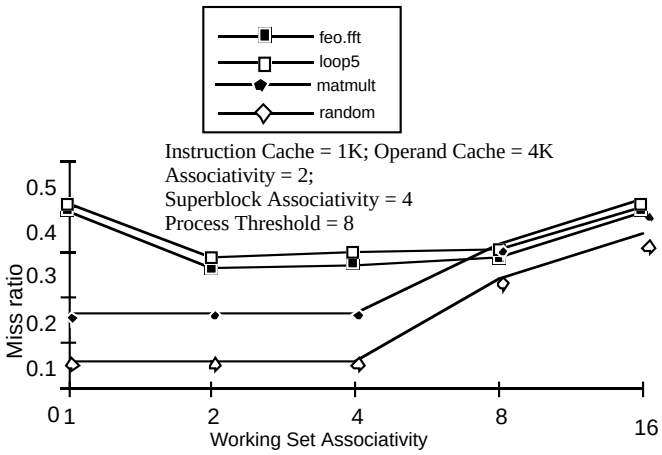


Figure 13. Working Set Associativity

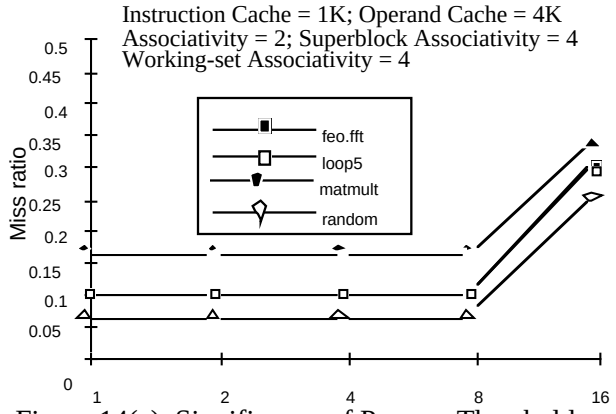


Figure 14(a). Significance of Process Threshold

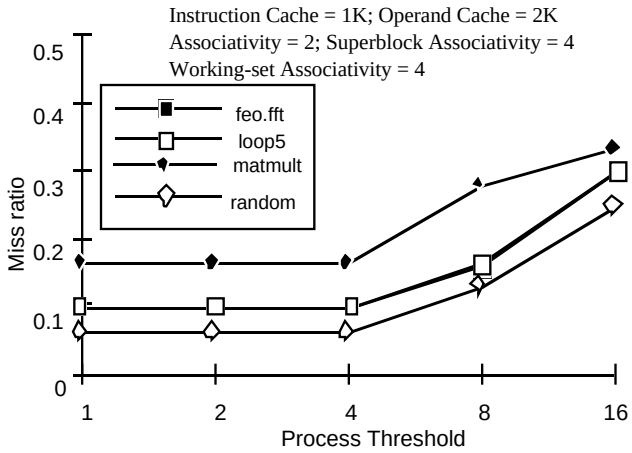


Figure 14(b). Significance of Process Threshold

The best value for the threshold depends on the number of superblocks that can be held in the operand cache; for a k-way, N set cache, the process threshold should be N*k. For example, in Figure 14(a), we find that the cut off value is 8 where the number of super blocks used was 8; while this is 4 in Figure 14(b). Increasing the number of active contexts (e.g. loop iterations or processes) beyond this threshold degrades the performance.

Effect of Replacement Strategies. As described in Section 3.3.1, we explored performance gains that can be achieved by using dead-context replacement for superblocks and used-words replacement for working sets. The dead-context replacement policy shows significant improvements for small caches (as much as 70% fewer superblock misses when compared to random replacement policy, for 2K or smaller caches) [Shanmugam 93].

For working set replacement (within a superblock) we experimented with a "used-words" policy. Here, a working set (if one exists) that contains operand locations that have already been used by instructions are replaced. Figure 15 shows the percentage of operand cache misses that can be satisfied by used-words.

The improvement resulting from the used words policy leads us to believe that dataflow systems can be made to reuse operand cache memory locations for matching operands of more than one instruction within a frame. In other words, in stead of replacing used words in a frame, they can be reused for storing and matching other operands. A significant number of operand cache misses can then be eliminated. Reusing operand locations is akin to the use of registers to keep temporary variables during a computation, bringing the dataflow processor even closer to von Neumann architecture.

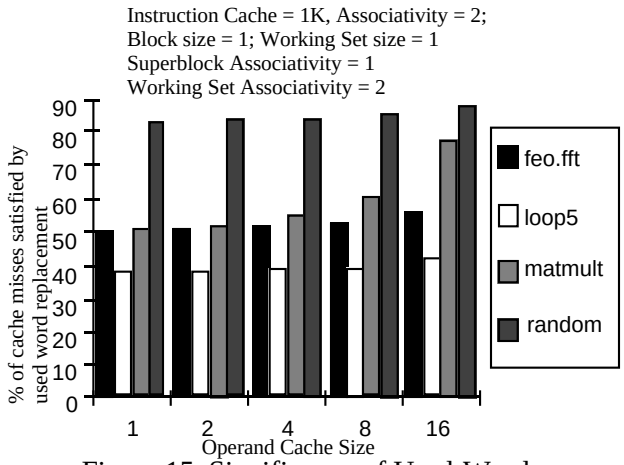


Figure 15. Significance of Used-Word Replacement Policy

5.3. Throughput improvement with cache. Next, we experimented with throughput improvement resulting from various operand cache sizes. Since code size is small and since the instruction cache misses are very rare, we did not experiment with various instruction cache sizes. In our experiments we assumed that memory accesses require 6 cycles as compared 2 cycle access to cache memories. Figure 16 shows the gains (reduction in execution times) that can be achieved using cache memories when compared to a system with no operand cache memory. We believe that performance can be further improved with reusing operand locations.

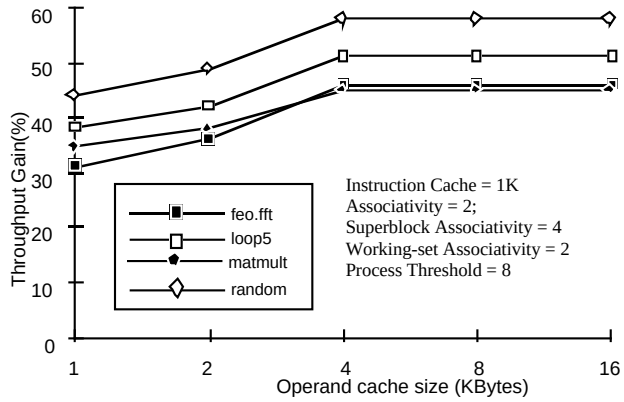


Figure 16. Uniprocessor Throughput Gains Vs. Operand Cache Size

5.4. I-Structure Cache Performance.

In order to investigate the significance of the I-Structure cache we have extended our experiments by implementing a 4 processor ETS system sharing the I-Structure memory. As described in Section 4, each processor contains an instruction cache, an operand cache and an I-structure cache. Figure 17 shows the miss ratios as the IS cache size is increased.

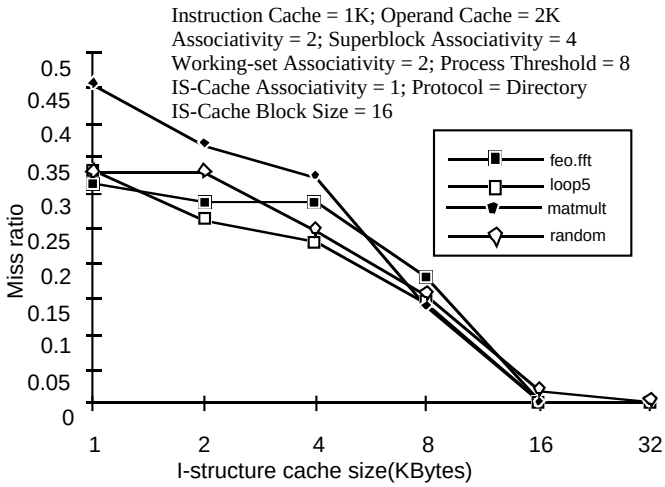


Figure 17. IS-Cache Size Vs. Miss Ratio

The sizes shown are per processor cache. The miss ratio for IS-Cache does not depend on the protocol used (viz., directory vs. snoopy); only the throughput depends on the protocol.

Figure 18 shows the results obtained by varying the associativity of IS-Cache. As can be seen, direct mapped caches are better suited for IS-Structures. We believe that separate direct mapped caches for arrays are beneficial even in conventional architectures.

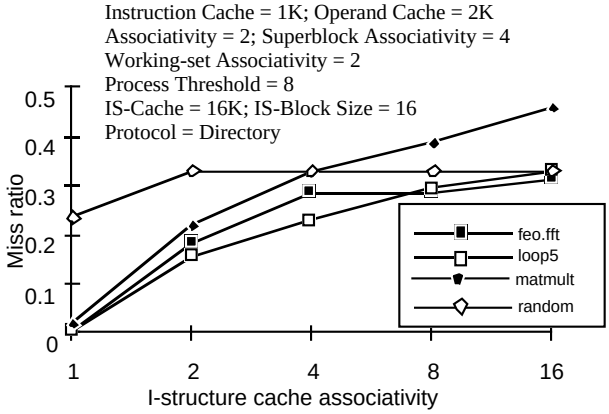


Figure 18. IS-Cache Associativity Vs Miss Ratio

Figure 19 shows that the block size of an I-Structure cache has very little impact on the miss ratio..

Intuition makes us believe that the block size should depend on how loop iterations are allocated to processors. For example if several consecutive iterations are assigned to the same processor, larger block size may produce better hit ratio. At present our results do not bear this primarily because the benchmarks chosen have very simple dependencies (iteration i depends on iteration i-c for some constant). It should be noted that increasing the block size leads to higher miss penalties

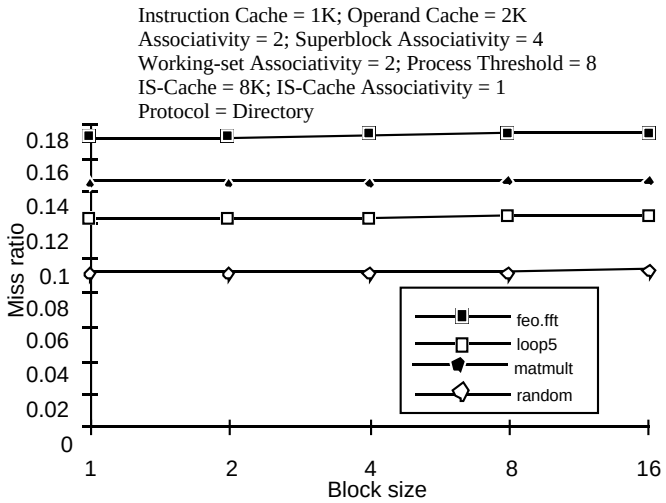


Figure 19. IS-Cache Block Size Vs Miss Ratio

Figure 20 shows the throughput gains (reduction in execution times) obtained from using IS-Cache memories in both the directory-based and snoopy-based approaches. The improvements are shown as a percentage gain when compared to a multiprocessor ETS system with no IS-Cache memory. We assume that it takes 12 cycle to access the shared (remote) I-Structure memory. Snoopy protocol consistently behaves better because of the smaller latency required as compared to directory protocol. In directory approach, the latency is at least a round trip delay to the remote memory. In snoopy protocol, the latency can be much smaller when the snooping is successful (see Section 4.1.2).

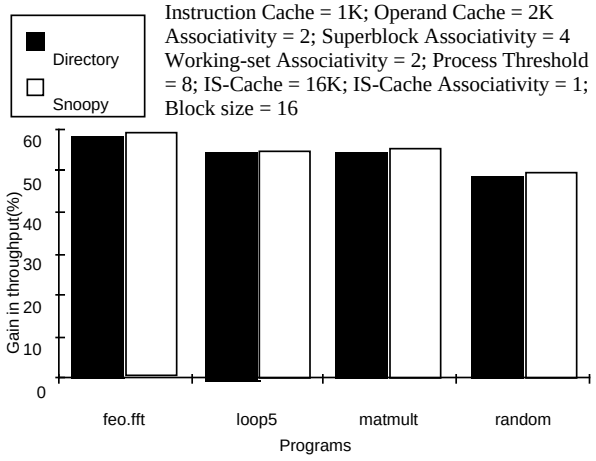


Figure 20. Throughput Gains from using a IS-Cache

Figure 21 shows the significance of IS-Cache size on performance gains for one of the benchmarks. This graph shows that snoopy protocol performs better than directory protocol for small cache memories (this is expected since caches snoop only on a small subset of entries in their cache memories).

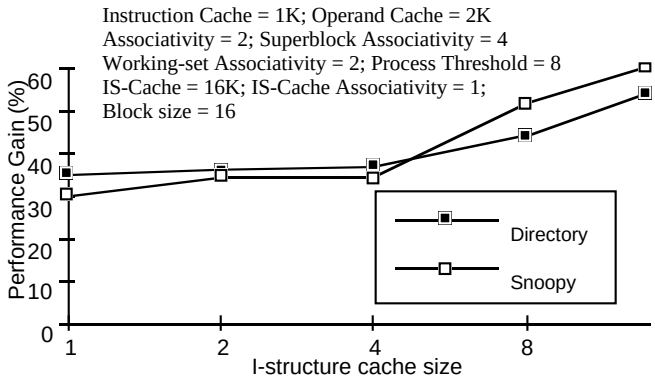


Figure 21. Snoopy Vs Directory

While all cache memories (instruction, operand and I-Structure) improve performance, we feel that the IS-Cache is the most significant contributor to the performance gain in. This is primarily because of the improvements in latencies while accessing remote I-Structure elements. We have already shown the throughput gains obtained by using operand cache memories, in uniprocessor environment (Figure 16). These performance gains will scale with the number of processors.

6. SUMMARY AND FUTURE DIRECTIONS.

In this paper we have described how instruction, operand and I-Structure cache memories can be designed with multi-threaded dataflow systems. We have studied cache design issues that are applicable to both conventional and dataflow architecture and design issues that are specific to dataflow model. As new generation architectures are combining dataflow and control flow paradigms to achieve higher performance, we believe that our study can play a role in designing cache memories for such systems. We feel that conventional architecture can also benefit from two level addressing for operands whereby the address space can be divided into threads (or contexts) and smaller addresses can be used for accessing data within a context.

We have shown how directory and snoopy protocols can be used to improve the performance of I-Structure cache memories in multiprocessor environment. The idea of including a separate cache memory for storing arrays (and other data structures) may be interesting to consider even in conventional architectures, since the locality patterns of such data objects are more predictable than other operands. As shown in our experiments,

I-Structure caches behave more like instruction caches and hence direct mapping (with possibly small miss or victim caches) may be more appropriate for such data objects than set-associative caches. Software methods for improving cache performance in conventional systems have produced very promising results ([Lam 91], [Porterfield 89]). We feel that compile-time analysis can also improve the cache performance in dataflow architecture. We plan to investigate issues related to "reusing" operand locations, scheduling loop iterations on multiple processors to optimize the IS-Cache performance, and to optimize the number of active contexts in a processor and minimize trashing of cache memory blocks.

It is not our objective to claim that our experiments are either exhaustive or conclusive; only that they are a start. There are several inter-related parameters that together influence the overall performance of multi-processor systems. We hope to continue our studies by expanding the benchmarks to extrapolate our results to large scale systems. This will then allow us to investigate compiler optimizations that can extract optimum performance for a given set of cache designs.

7. REFERENCES.

[Abraham 94] E. Abraham. "Cache memories for multiprocessor dataflow architecture", *MS Thesis,* Dept. of CSE, UTA, Dec. 1994.

[Arvind 86] Arvind and D. E. Culler. ``Dataflow Architectures", in *Annual reviews in computer science,* (1986). vol. 1, pp. 225-253.

[Arvind 89] Arvind and R.S. Nikhil. "Can Dataflow subsume von Neumann Computing?". *Proc. 16th Annl. Intl. Symp. on Computer Architecture,* May 1989, pp. 262-272.

[Culler 93] D.E. Culler et. al. "TAM - a compiler controlled threaded abstract machine", *Journal of Parallel and Distributed Computing,* 18 (3), pp. 347-370 (1993).

[Feo 90] J.T. Feo, D.C. Cann and R.R. Oldehoeft. "A report on Sisal language project", *Journal of Parallel and Distributed Computing,* Oct. 1990, pp. 349-366.

[Hicks 93] J. Hicks, D. Chiou, B.S. Ang and Arvind. "Performance studies of the Monsoon dataflow processor", *Journal of Parallel and Distributed Computing*, 18(3) pp. 273-300 (1993).

[Hill 89] M.D. Hill and A.J. Smith. "Evaluating associativity of CPU caches", IEEE Transactions on Computers, Dec. 1989, pp. 1612-1630.

[Ianucci 88] R.A. Ianucci. " Toward a dataflow/von Nuemann Hybrid Architecture", *Proc. 15th Annul. Intl. Symp. on Computer Architecture*, May 1988, pp. 131-140.

[Lam 91] M.S. Lam, E.E. Rothberg and M.E. Wolf. "The cache performance and optimizations of blocked algorithms", *Proceedings of ASPLOS 4,* 1991, pp. 63-74.

[Lebeck 94] A.R. Lebeck and D.A. Wood. "Cache profiling and the SPEC benchmarks: A case study", *IEEE Computer*, Oct. 1994, pp. 15-26.

[Lee 93] B. Lee and K.M. Kavi. ""Program partitioning for multithreaded dataflow computers", *Proc. of 26th Hawaii International Conference on System Sciences (HICSS-26)*, Jan. 5-8, 1993, pp. II 487-495.

[Lee 94] B. Lee. and A.R. Hurson. "Dataflow architectures and multithreading", *IEEE Computer*, Aug. 1994, pp. 27-39.

[Papadopolous 90] G.M. Papadopolous and D.E. Culler. "Monsoon: an Explicit Token-Store Architecture", *The 17th Annl Intl. Symp. on Computer Architecture,* June 1990, pp. 82-90.

[Papadopolous 91]G.M. Papadopolous. *Implementation of a General Purpose Dataflow Multiprocessor*. MIT Press, 1991.

[Patadia 94] P. Patadia. "Design and evaluation of I-structure cache memory for dataflow multiprocessor environment", *MS Thesis,* Dept of CSE, UTA, Dec. 1994

[Porterfield 89] A. Porterfield. "Software methods for improvement of cache performance on supercomputer applications", PhD thesis, Dept. of Computer Science, Rice University, 1989.

[Przybylski 90] S. Przybylski. *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, San Mateo,CA, 1990.

[Shanmugam 93] P. Shanmugam, S. Andhare, K. Kavi, B. Shirazi and A. Hurson. "Cache design for an ETS dataflow architecture", *Proc. of the 5th IEEE Symp. on Parallel and Distr. Processing,* pp 45-50, Dec. 1993.

[Smith 82] A.J. Smith. "Cache Memories". *ACM Computing Surveys,* September 1982, pp. 473-530.

[Takesue 87] M. Takesue. "A unified resource management and execution control mechanism for Dataflow Machines". *Proc.*

14th Annl. Intl. Symp. on Computer Architecture, June 1987, pp. 90-97.

[Takesue 92] M. Takesue. "Cache Memories for Data Flow Architectures". *IEEE Transactions on Computers,* vol. 41, June 1992, pp. 667-687

[Thoreson 87] S.A. Thoreson and A.N. Long. "A Feasibility study of a Memory Hierarchy in Data Flow Environment". *Proc. Intl. Conference on Parallel Conference*, June 1987, pp. 356-360.

[Tokoro 83] M. Tokoro, J.R. Jagannathan and H. Sunahara. "On the working set concept for data-flow machines", *Proc. 10th Annul. Intl. Symp. on Computer Architecture*, July 1983, pp. 90-97.