DESIGN OF CACHE MEMORIES FOR DATAFLOW ARCHITECTURE

Krishna M. Kavi*
and
A. R. Hurson**

*Department of Computer Science & Engineering
The University of Texas at Arlington
Arlington, TX 76019
kavi @cse.uta.edu

and
**Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802
hurson@cse.psu.edu

Abstract

The recent advance in dataflow processing — to combine the dataflow paradigm with the control flow paradigm — has brought out many new challenging issues. This hybrid organization has made it possible to study and adapt familiar control flow concepts such as cache memories within the framework of the dataflow architecture.

The concept of cache memory has proven its effectiveness in the von Neumann architecture due to the **spatial** and **temporal** localities which govern the organization of the conventional programming execution. A dataflow paradigm, does not inormally support locality, since the execution sequence is enforced only by the availability of operands. However, dataflow programs can be reordered based on various criteria to enhance the locality of instruction references. This can be achieved by: i) careful partitioning of a dataflow program into vertical layers of data dependent instructions, and ii) proper distribution and allocation of the recurrence portions of the dataflow program. Enhancing the locality of data references in the dataflow architecture is a more challenging problem. This paper studies the design of instruction, data (operand), and I-Structure cache memories using the Explicit Token Store (ETS) model of dataflow system. The performance results obtained using various benchmark programs are presented and analyzed.

DESIGN OF CACHE MEMORIES FOR DATAFLOW ARCHITECTURE

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 improvements. 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 ([2], [3], [5], [7]). The reader is referred to numerous survey articles that have analyzed dataflow architectures (e.g., [11]). In our research we use one such model known as Explicit-Token-Store [12], [13], 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 context identifying information in the form of a continuation (or a tag). This permits the toleration of long and unpredictable latencies due to remote memory accesses, since the processor can switch to new contexts without having to wait for memory accesses. Interestingly, the instruction level parallelism leads to excessive overheads due to the dynamic scheduling of the instructions. A compromise between the instruction-level context-switching capability and sequential scheduling of instruction streams provides a different perspective on dataflow architectures — **multithreading**. A thread is a sequence of instructions where once the first instruction in the thread is executed, (for non-blocking threads) the remaining instructions execute without interruption. Thus, a thread defines the basic unit of work that requires synchronization only at the beginning of its execution. The evolution from a pure self-scheduling paradigm of dataflow to multithreading requires locality and improved processor utilization during remote memory accesses. Experiences from current dataflow projects show that there is a trend towards adopting multithreading as a viable method to build hybrid architectures that combine features of dataflow and von Neumann execution models. Multithreaded architectures can be viewed as either an evolution of (a) dataflow architectures in the direction of more explicit control over instruction execution order, or (b) von Neumann machines in the direction of better support for synchronization and tolerance of long latency operations.

The success of multithreaded systems depends on how quickly context switching can be supported. This is only possible if threads are resident in fast but small memories (such as instruction buffers and caches) which limits the number of active threads and thus the amount of latency that can be tolerated. The generality of dataflow scheduling makes it difficult to fetch

and execute a sequence of logically related sets of threads through the processor pipeline, thereby removing any opportunity to use registers across thread boundaries. Relegating the responsibilities of scheduling and storage management to the compiler alleviates this problem to some extent. 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. Section 6 outlines our approach to operand memory reuse in dataflow.

2. DATAFLOW ARCHITECTURE

In the data driven model of computation, operations are enabled only when all input operands are made available by predecessor instructions. Upon completion an operation makes its results available to its successor. This model makes it necessary for operands of instructions to wait for their matches. The *static dataflow model* was proposed by Dennis and his research group at MIT [11]. The general organization of the Static Dataflow Machine is depicted in Figure 1. The Activity Store contains *instruction templates* that represent the nodes in a dataflow graph. Each instruction template contains an operation code, slots for the operands, and destination addresses. The availability of the operands is determined by the contents of the presence bits (PBs) of the instruction template. The Update Unit detects the executability of instructions. The address of the enabled instruction is sent to the Fetch Unit via Instruction Queue. The Fetch Unit fetches and sends a complete operation packet to one of the Functional Units and clears the presence bits. The Functional Unit performs the operation, forms result tokens, and sends them to the Update Unit. The Update Unit stores each result in the appropriate operand slot and checks the presence bits to determine the active instruction(s).

The *dynamic dataflow model* was proposed by Arvind at MIT and by Gurd and Watson at the University of Manchester (Figure 2) [11]. Tokens are received by the Matching Unit and the Matching Unit tries to bring together tokens with identical tags. If a match exists, the corresponding token is extracted from the Matching Unit and the matched token set is passed on to the Fetch Unit. If no match is found, the token is stored in the Matching Unit to await a partner. In the Fetch Unit, the tags of the token pair uniquely identify an instruction to be fetched from the

Program Memory. The instruction together with the token pair form an enabled instruction packet that is sent to the Processing Unit (PE). The Processing Unit executes the enabled instructions and produces result tokens to be sent to the Matching Unit via the Token Queue.

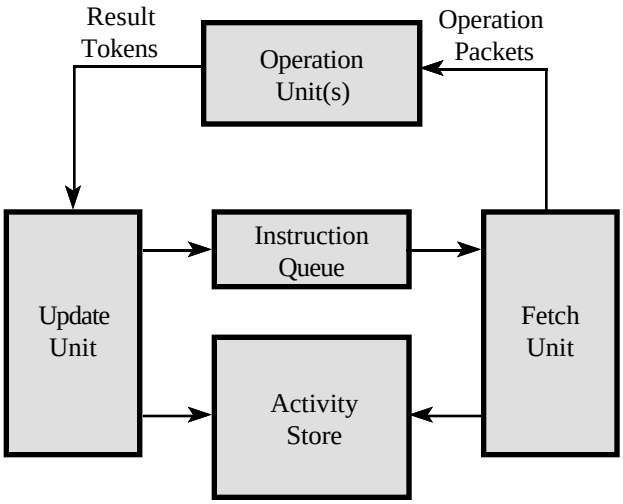


Figure 1: The basic organization of the static dataflow model.

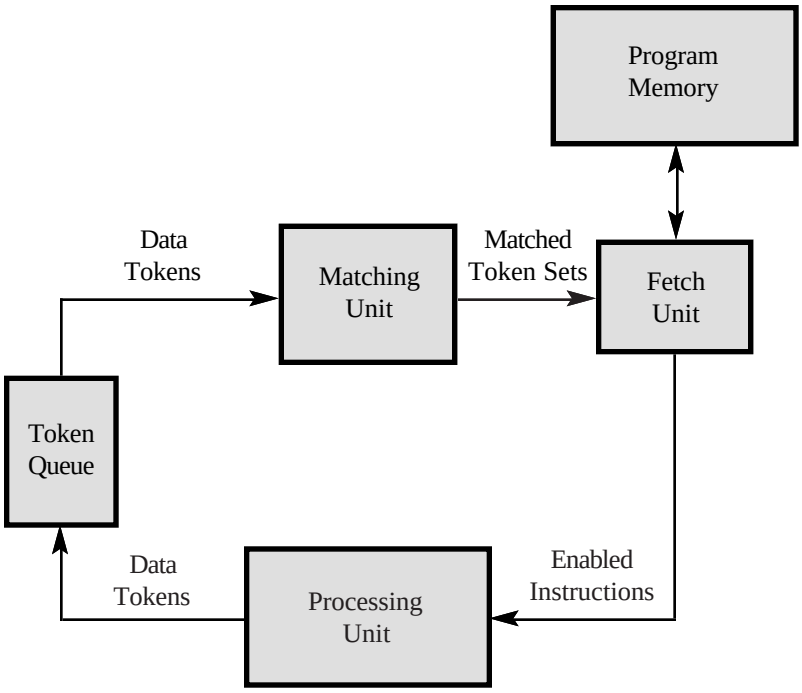


Figure 2: The general organization of the dynamic dataflow model.

2.1 Direct Matching

The complexity of the token matching process of dynamic dataflow architectures has been an obsticle to the success of dataflow machines. One of the important developments in the design of current dataflow proposals is the novel and simplified process of matching tags — **direct matching**. The basic idea of the direct matching scheme is to eliminate the expensive and complex process of matching tokens using associative memory. In a direct matching scheme, storage (**activation frame**) is dynamically allocated for all the tokens needed by the instructions in a code-block. A code-block can be viewed as a sequence of instructions comprising a loop body or a function. The actual disposition of locations within an activation frame is determined at compile-time; however, 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 of input operands for that instruction 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 instruction. To illustrate the operations of direct matching in more detail, consider the token matching scheme used in Explicit Token Store (ETS). An example of the ETS code-block invocation and its corresponding Instruction and Frame Memory are shown in Figure 3.

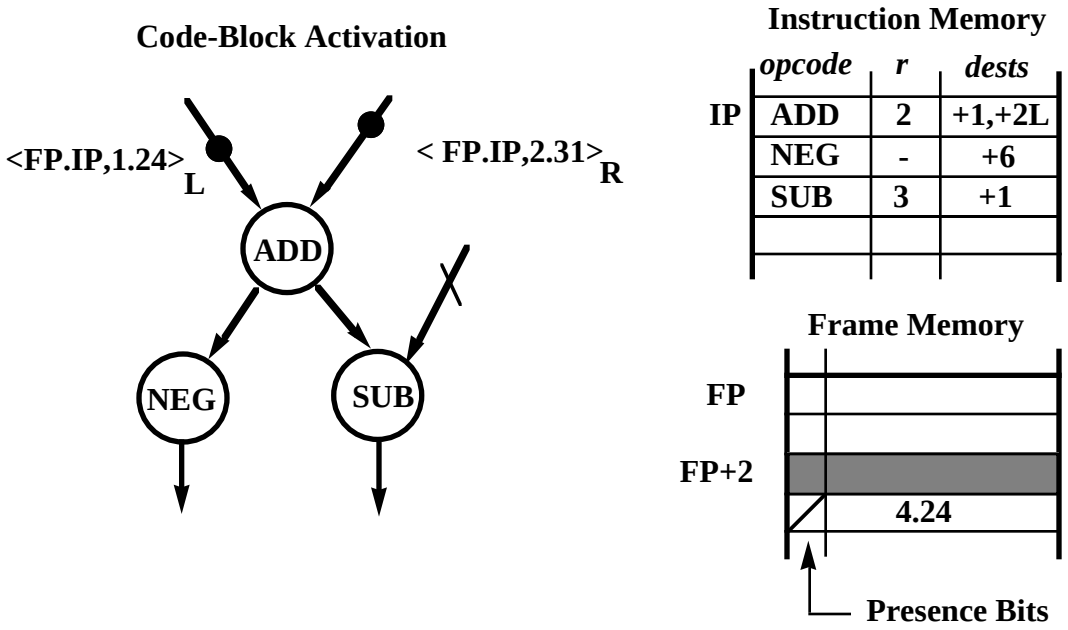


Figure 3: ETS representation of a dataflow program execution.

When a token arrives at an actor (e.g., ADD), the IP part of the tag 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>$_R$ and <FP.IP+2, 3.55>$_L$). Based on the discussion thus far, direct matching schemes used in the pure-dataflow organizations are implicit in the architecture. In other words, the token matching mechanism provides the full generality of the dataflow model of execution and therefore is supported by the hardware.

Figure 4 depicts the original hardware implementation of the ETS architecture extended with instruction, operand, and I-structure caches. For a more detailed description of ETS and Monsoon see ([8], [12], [13]).

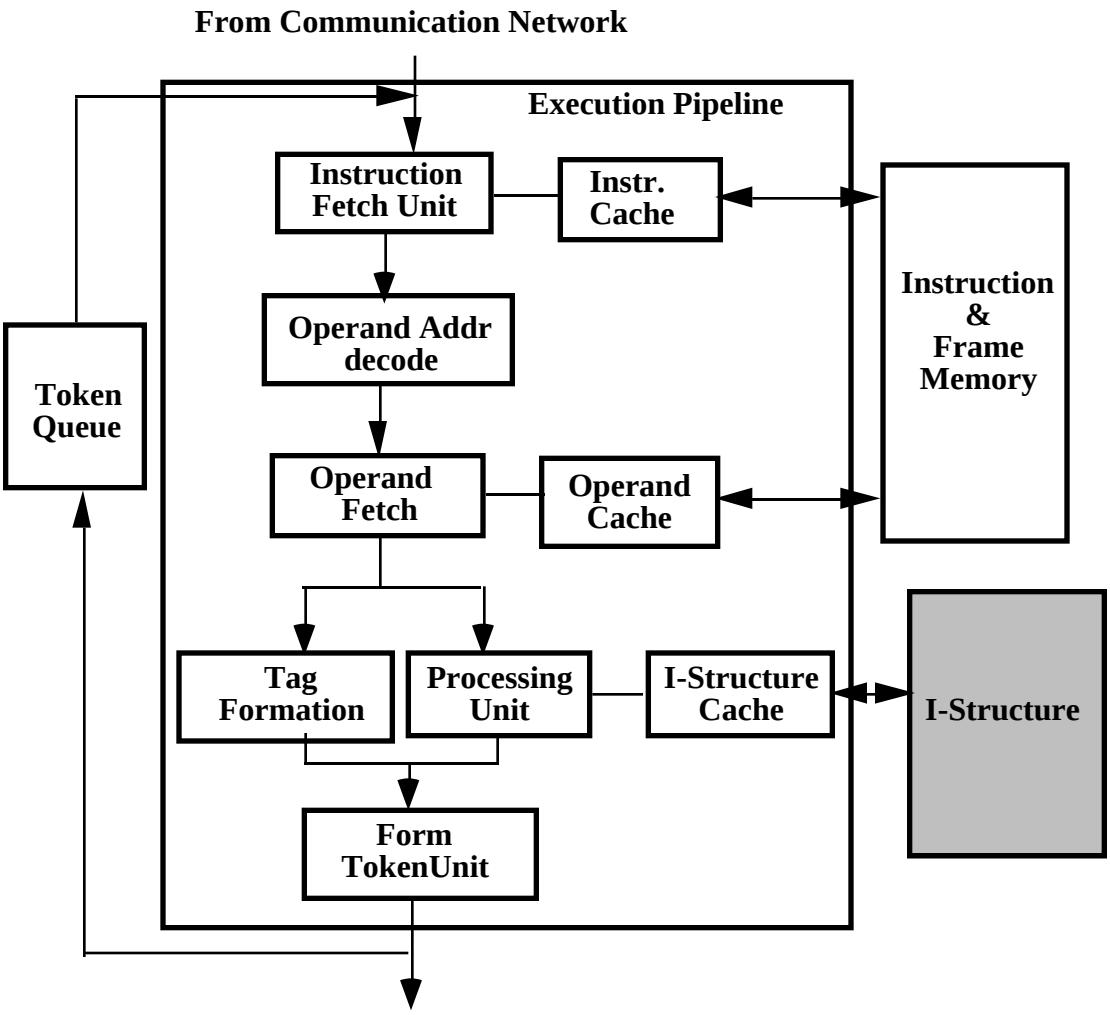


Figure 4: An organization of a pure-dataflow processing element.

Upon creation of a continuation, the ETS performs the following sequence of operations:

1. **Instruction Fetch**: The incoming token's instruction pointer (IP) is used to access an instruction in the local instruction memory.

2. **Address Decode**: The effective address(FP+r) of the operand memory location is computed (*r* is obtained from the instruction and FP from tag of the token).

3. **Operand Fetch**: The presence bit in the operand memory location is examined. The value in the operand memory location is read, written, exchanged or ignored depending on the value of the presence bit. If the bit is reset the data value is stored; otherwise a match occurs leading to a read.

4. **ALU**: On a match, one or more stages of the ALU executes the opcode, with the value retrieved from the operand memory and the value that is contained in the token. One or two tags are computed by concatenating the destination offsets with the instruction pointer.

5. **Token Form**: The result is packaged in tokens along with the tags and the tokens are written to a token queue.

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 ([9], [15], [17], [20]). 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 (miss penalty), and
- Minimizing the overhead 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 dataflow program in its pure form is not amenable to a cache, primarily due to the self-scheduling of instructions for execution. Reordering of instructions of such a program based on certain criteria [22] can produce synthetic localities justifying the presence of a cache. The recurrent use of instructions (in different activation frames) also causes the existence of temporal localities. Working set in a von Neumann environment refers to the smallest set of instructions and operands satisfying the current processor requests. Working set for a dataflow program can be defined as the minimum set of instructions that keep the execution unit busy [21]. The working set concept of a dynamic dataflow program could be based on both the principle of locality and the simultaneity of execution. The working set for a dataflow program is therefore determined by analyzing the dataflow graph.

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 [21]). 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 **working 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 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, the working set of operand locations corresponding to the instructions in the block of instuctions is prefetched into the operand cache. As a result of this, any subsequent references to the operand cache caused by the instructions 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 5 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.

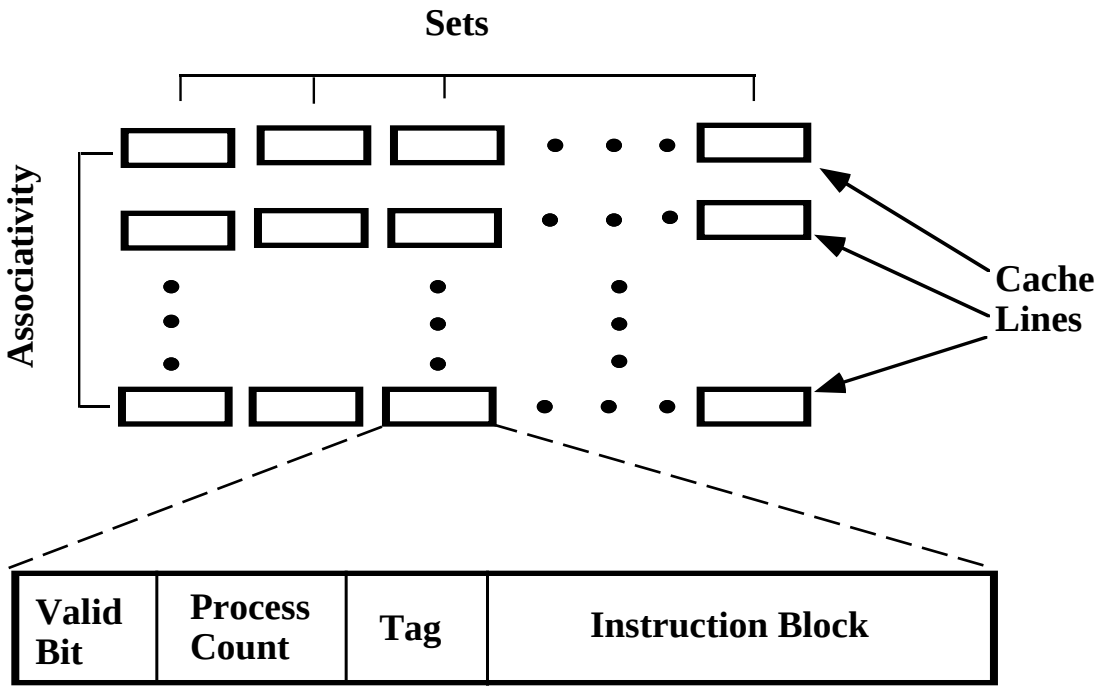


Figure 5: Instruction Cache Organization.

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 [18], [19] 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 or thread occupies a superblock.

The second level of associativity is used for accessing individual locations within a frame. Figure 6 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. This eliminates the compulsory misses [6] on writes.
- **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 blocks and working sets, paralleling the blocks and working sets of the instructions in a code-block. Thus, a superblock 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.

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

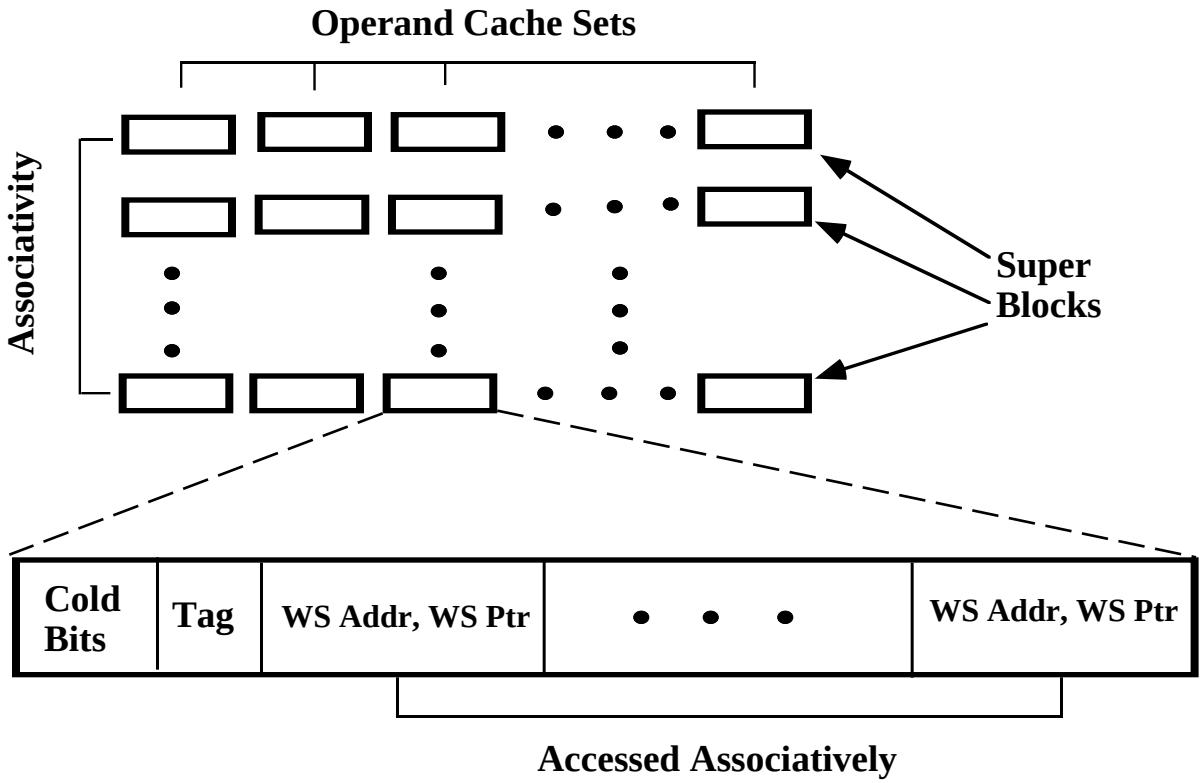


Figure 6: Operand Cache Organization.

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). We have also investigated the implication of compile-time analysis for reusing operand memory locations within an activation frame. These results are presented in Section 6. For superblock replacement, we have studied the **dead context replacement** policy that replaces a superblock representing a completed thread (or activation frame).

3.3.2 Process Control. The operand cache must accommodate several threads (activation frames) 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 [10]. 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 multithreaded 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. Figure 7 shows an I-Structure example with pending requests for unavailable array elements [1].

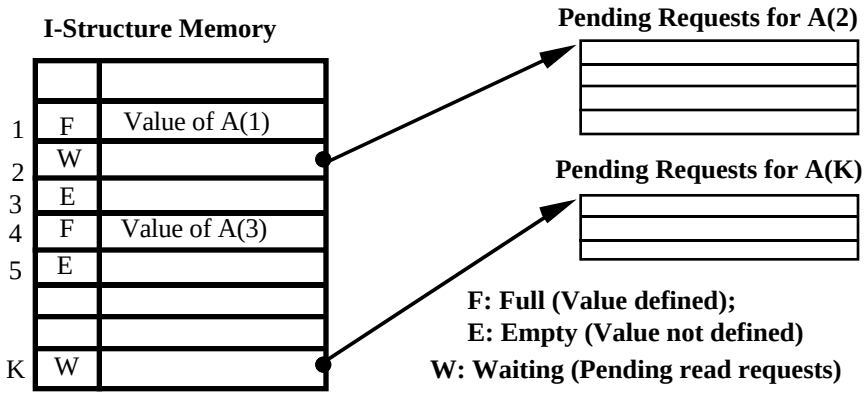


Figure 7: An I-Structure.

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 FP part of the tag identifies the receiving context and the processor containing the context. Although the single assignment property of dataflow 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 8 shows the general structure of our multiprocessor system.

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). No IS-Cache block is allocated until the (I-Structure) element is ready for definition. In other words, cache blocks are allocated only when the data elements are written to them. The following possibilities must be considered when a read request for an I-Structure element (or block) is received by the I-Structure memory controller.

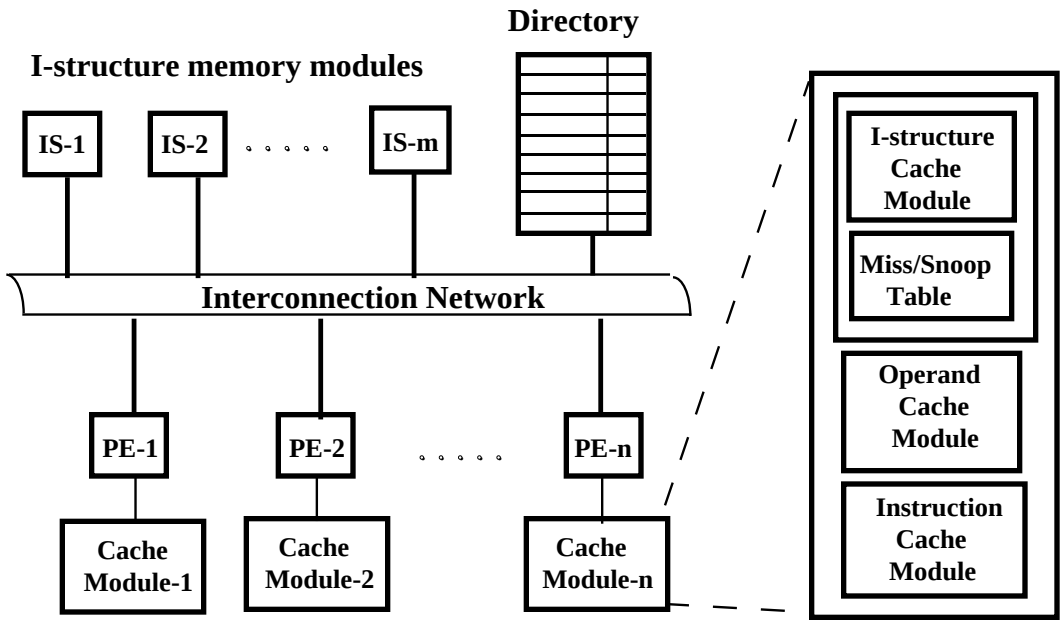


Figure 8: Multiprocessor ETS with I-Structure Cache Memories.

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

   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 a local table (known as the miss table) to reflect the pending requests with the I-Structure element. Deferred requests are also maintained with the I-Structure memory in the usual manner. Eventually, the data is 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. Note that once the data is written back to the I-Structure memory, further read requests are satisfied directly by the I-Structure.

We believe that compiler 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. At the other extreme, if read requests arrive before the elements are defined, the directory protocol incurs maximum overhead (in maintaining local miss tables and writing back the requested data immediately upon definition; 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.

i) 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.

ii) The processor containing the requested data in its IS-Cache is not successful in snooping on the request. 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). Alternatively, the request could have been deferred until the IS-Cache block is written back to the I-Structure memory.

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[1] a translator that takes IF1 graphs from a Sisal compiler [4] and generates ETS instructions for our simulator — We have not used IF-2 graphs since they incorporate optimizations for conventional architectures, our target is a

[1]At present the translator performs no optimizations. This caused some limitations on the size of the Sisal programs we could use for our experiments. We are in the process of eliminating some of the limitations imposed by our translator and hope to repeat our experiments on much larger Sisal programs.

dataflow instruction set, and we wanted to maintain the dataflow purity in the source. Our ETS instructions are abstract instructions designed to implement the ETS model (See Section 2) instead of any specific implementation. This has 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 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. All the other programs have been used by other dataflow researchers to evaluate the performance of Sisal or dataflow architectures. Table 1 lists the characteristics of the programs used in our current experiment.

Table 1: Program Statistics.

Name	#Instructions Referenced	# Operand References	# I-Structure References
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

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. It is observed that the effects of these parameters on the miss ratio are similar to those obtained for conventional caches. This indicates that localities can be synthesized in a dataflow environment. Increasing the operand and instruction cache sizes reduces the miss ratio as can be seen in Figures 9 and 10. 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 11.

It should, however, be noted that large block size adversely effects operand cache performance. 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 instruction block sizes lead to large operand working sets which in turn lead to more conflict misses since the cache is shared among several contexts (or frames).

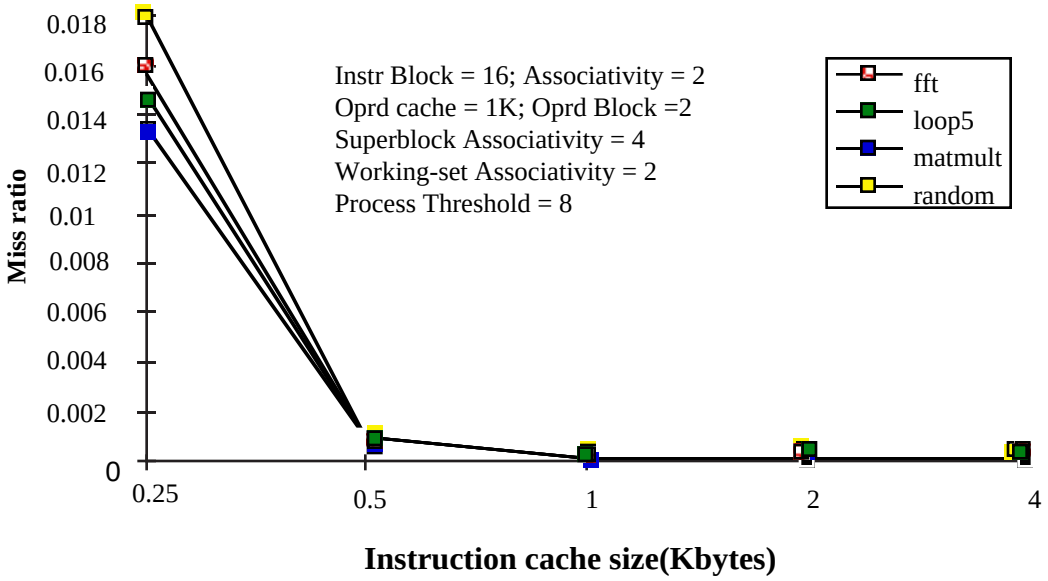


Figure 9: Effect of Instruction Cache Size on Miss Ratio.

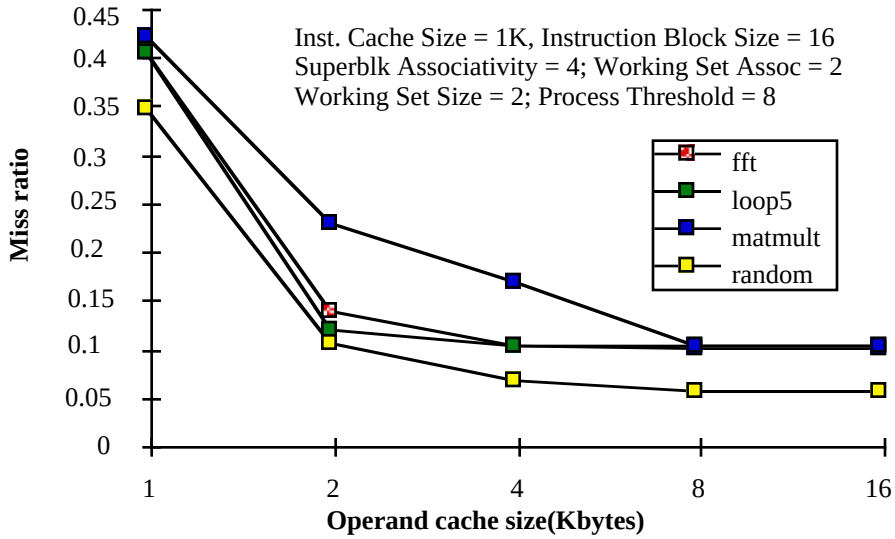


Figure 10: Effect of Operand Cache Size on Miss Ratio.

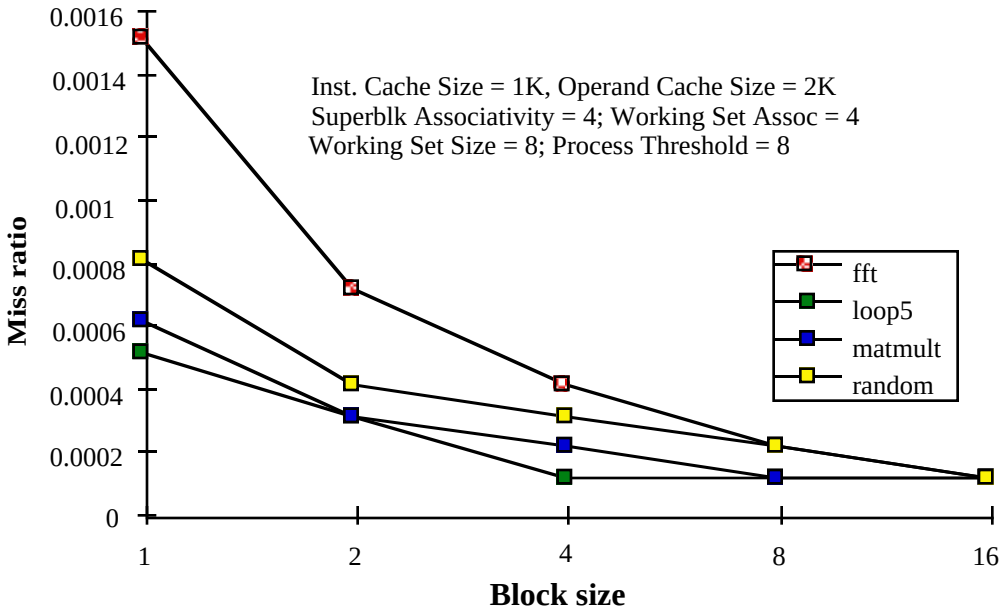


Figure 11: Miss Ratio *vs.* Instruction Cache Block Size.

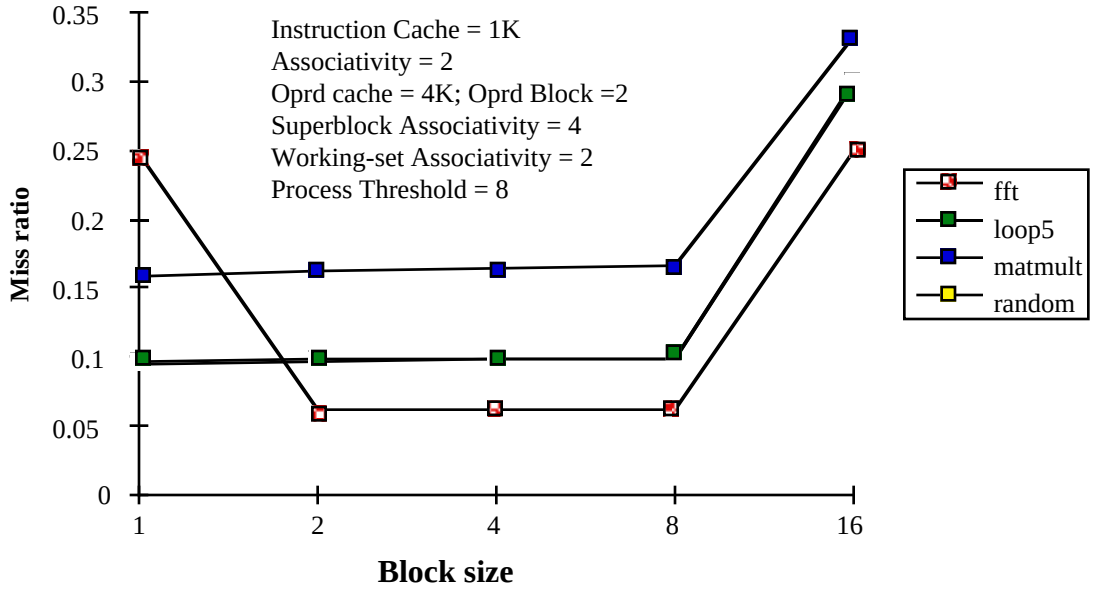


Figure 12: Effect of Instruction Block Size on Operand Cache Miss Ratio.

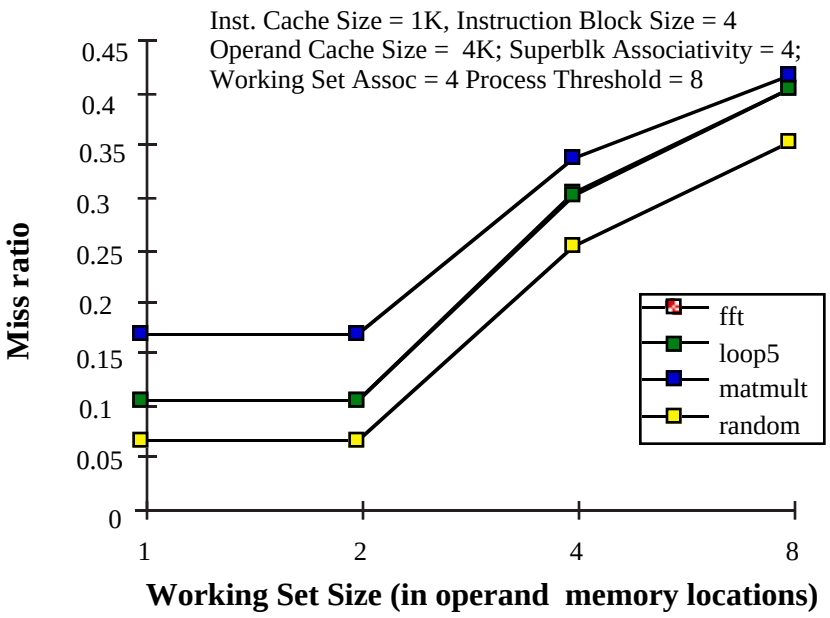


Figure 13: Operand Working Set Size vs. Miss Ratio.

Figure 12 shows the effect of instruction block size on the miss ratio for operand cache memories. This should be contrasted with the instruction miss ratios of Figure 11. Figure 13 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. This suggests that direct mapped caches perform well even for dataflow instructions. Since our operand cache contains two levels of associativity (superblock associativity and working set associativity), we varied both asociativities. Increasing the superblock associativity does not result in significant reduction in miss ratio (Figure 14). The optimal associativity depends on the block size used for the cache design. Figure 14 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. In ETS, the significance of associativity within a thread (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 (Figure 15). The increase in miss ratio when the associativity is increased beyond 4 is mainly due to the small size of the cache (i.e., fewer sets). Cold start misses in operand cache memories are eliminated since we allocate (not fetch) cache blocks on write (see Section 3.3).

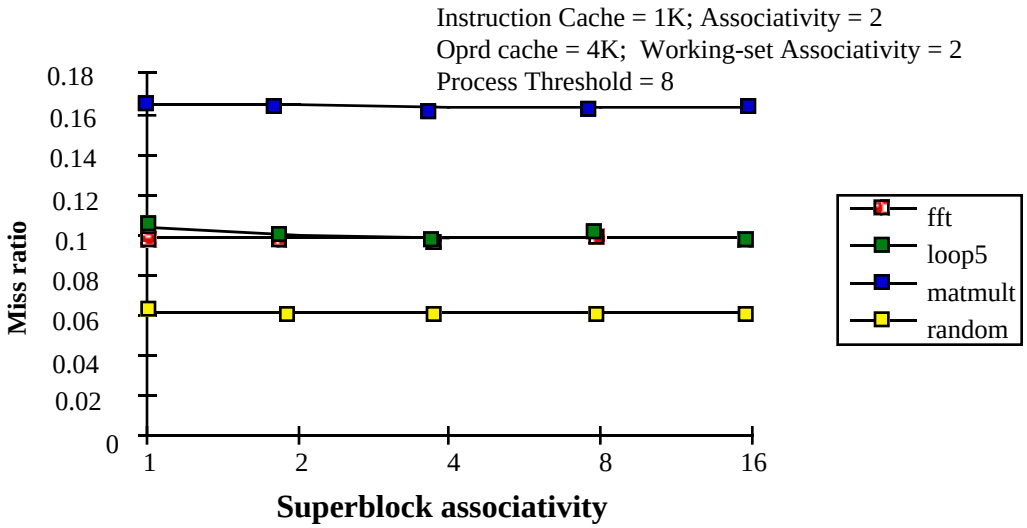


Figure 14: Superblock Set Associativity.

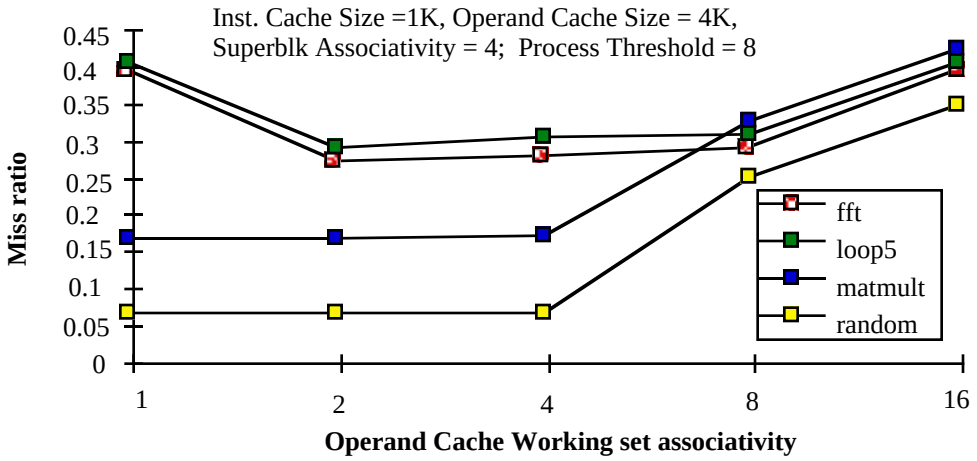


Figure 15: Working Set Associativity.

5.2 The effect of unconventional cache parameters on miss ratio

5.2.1. Effect of Process Control. The motivation for introducing process control is to avoid too many active threads(or contexts) contending for the limited operand cache resources. An appropriate threshold value allows for disciplined use of the cache resources and hence better performance. This can be readily observed in Figure 16. 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, from Figure 16(a), we could find that

the cut off value is 8 where the number of superblocks used was 8; while this is 4 in Figure 16(b). Increasing the number of active contexts (e.g. loop iterations or processes) beyond this threshold degrades the performance.

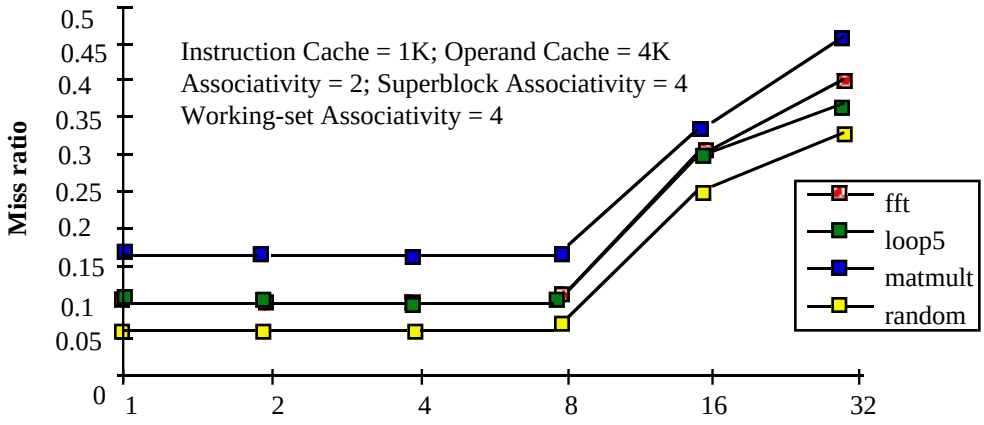
5.2.2. Effect of Replacement Strategies. As described in Section 3.3.1, we explored performance gains that can be achieved by using dead-context replacement for superblocks and used-words replacement for working sets. The dead-context replacement policy shows significant improvements for small caches (as much as 70% fewer superblock misses when compared to random replacement policy, for 2K or smaller caches). 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 17 shows the percentage of operand cache misses that can be satisfied by used-words. The improvement resulting from the used words policy led us to investigate the impact of operand memory reuse in dataflow systems. Section 6 will address the details of this investigation.

5.3 I-Structure Cache Performance.

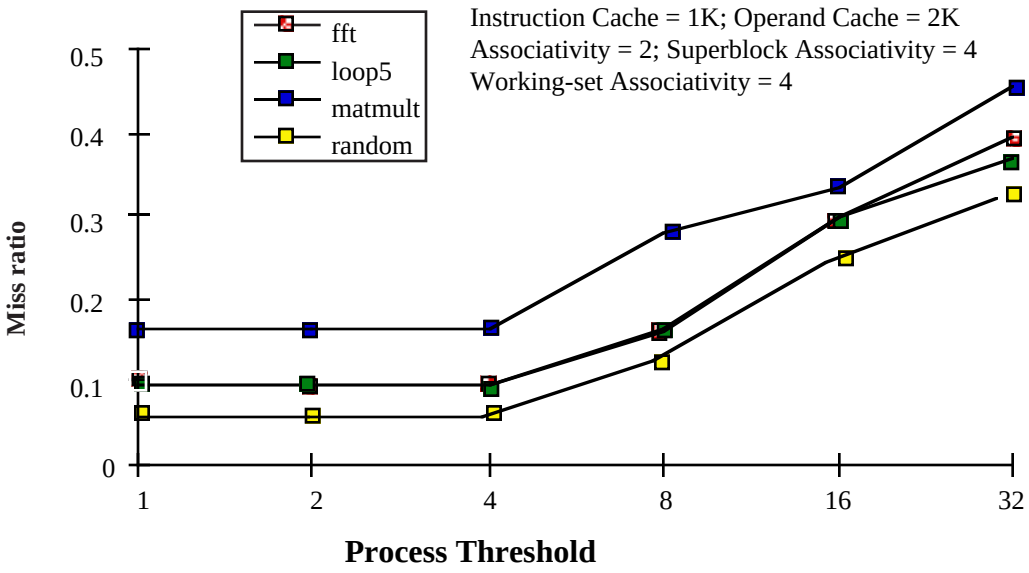
In order to investigate the significance of the I-Structure cache we have extended our experiments by implementing a 4 processor ETS system sharing the I-Structure memory. As described in Section 4, each processor contains an instruction cache, an operand cache and an I-structure cache. Figure 18 shows the miss ratios as the IS cache size is increased. 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 19 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.

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. 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



a) Operand cache size 4K.



b) Operand cache size 2K.

Figure 16: Significant of Process Threshold.

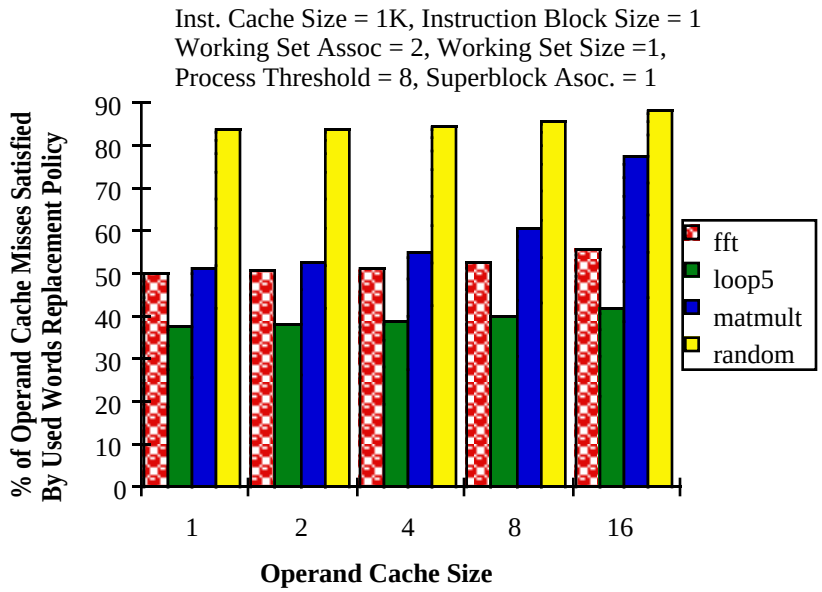


Figure 17: Significance of Used-Word Replacement Policy.

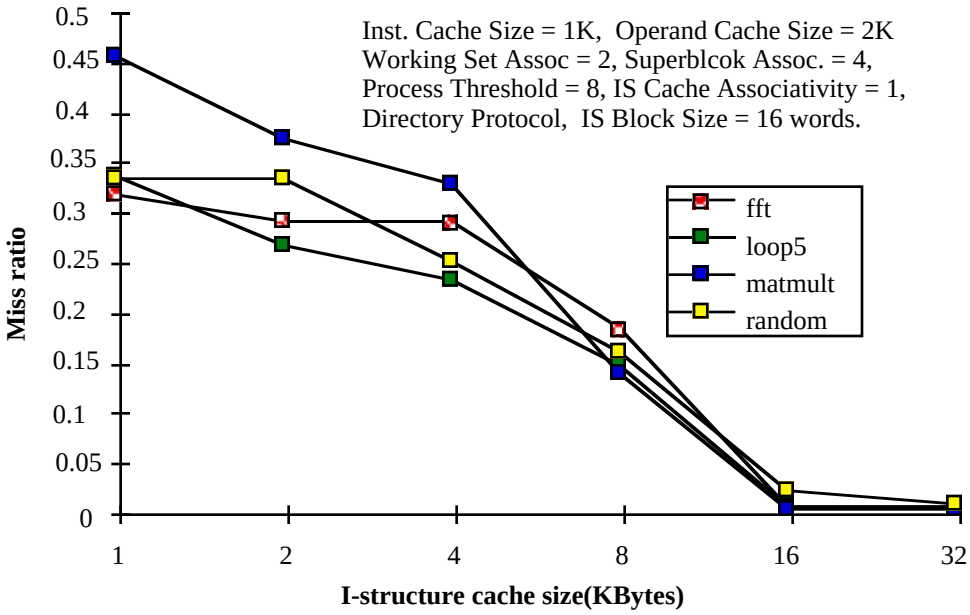


Figure 18: IS-Cache Size Vs. Miss Ratio.

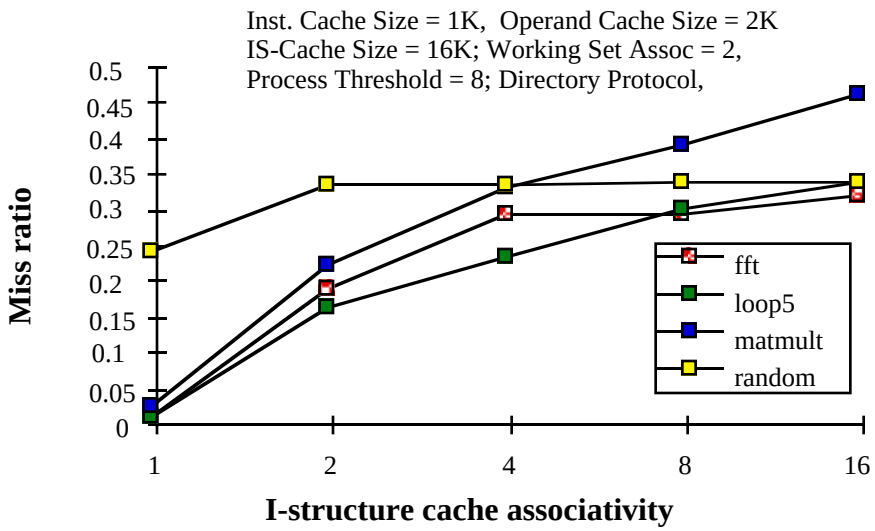


Figure 19: IS-Cache Associativity Vs. Miss Ratio.

are shown as a percentage gain when compared to a multiprocessor ETS system with no IS-Cache memory. We assumed 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).

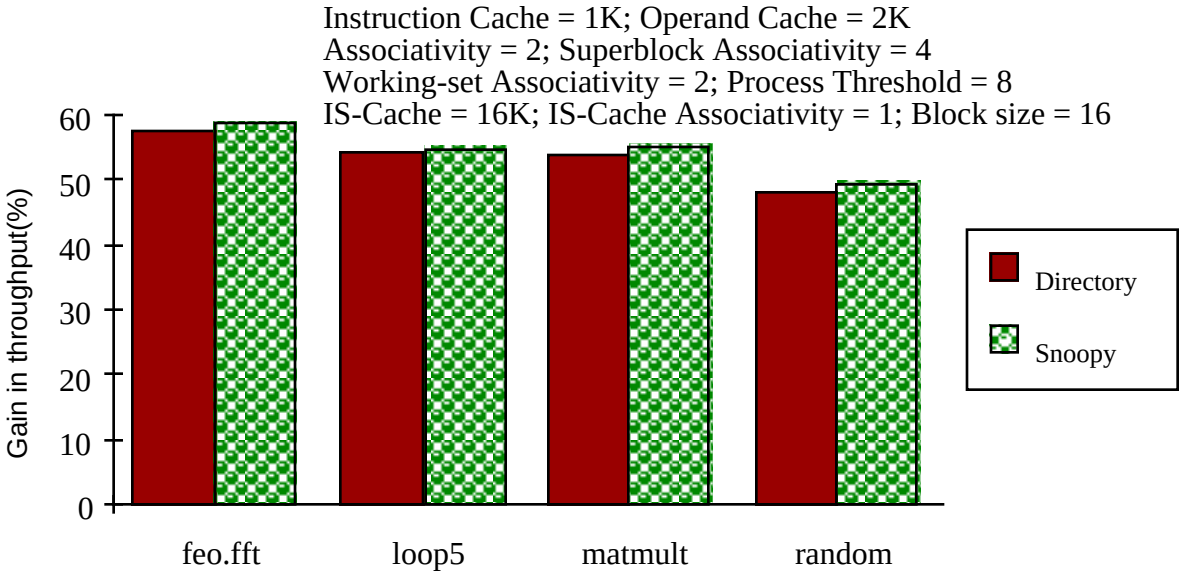


Figure 20: Throughput Gains from using a IS-Cache.

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).

6. REUSE OF OPERAND MEMORY

As indicated in the previous section (Figure 17), our experiments indicated that operand memory locations used to match the inputs of an instruction (we will call this the matching-location) can be released for reuse when the instruction completes execution. In order to reuse a memory location for matching the operands of more than one instruction, we must analyze the program for dependencies. It should be observed that the matching-location of instruction *i* can be reused as the matching-location of instruction *j*, if and only if instruction *i* completes its execution before any inputs of instruction *j* are available. Consider the dataflow graph segment shown in Figure 22. We will assume that each node represents an ETS instruction; each node has at most two inputs and at most two outputs. A matching-location is associated with each instruction and it is used for storing operands awaiting a match. The dependencies among the instructions are completely specified by the data dependencies represented by the directed edges.

A node *i* is called the left (right) ancestor of a node *j* if a directed path exists from node *i* to the left (right) input of node *j*. For example, node 0 is a left ancestor of node 7. A node *i* is a common-ancestor of node *j* if node *i* is both the left and the right ancestor of node *j*. For example, node 0 is a common ancestor of node 7. Likewise, we can define left, right and common descendants of a node *i*. Node 7 is a common descendent of both nodes 0 and 1. As can be readily observed, the memory location used for matching the operands of a node *i* can be reused to match the operands of one of its common descendants. In Figure 22, the matching-location of instruction 0 (or 1) can be used to match the operands of instruction 7.

We have implemented an algorithm to find the common descendants of dataflow graph nodes so that matching memory locations can be reused. Using the same benchmark programs, we have repeated our experiments with operand cache memories to evaluate the performance gained by reusing the matching locations. Figures 23-26 show our results for each of the benchmark programs. The graphs compare the cache miss ratios obtained by reusing operand memory locations (after curves) with those that did not reuse (before). As can be seen, with small caches, the reuse of memory locations reduces the working sets needed by a code-block, In order to fully benefit from the reuse of operand memory locations, cache replacement policies must be modified. The cache block with operand memory locations that can be reused should **not** be replaced. Our experiments enforced such a replacement policy. It is also necessary to

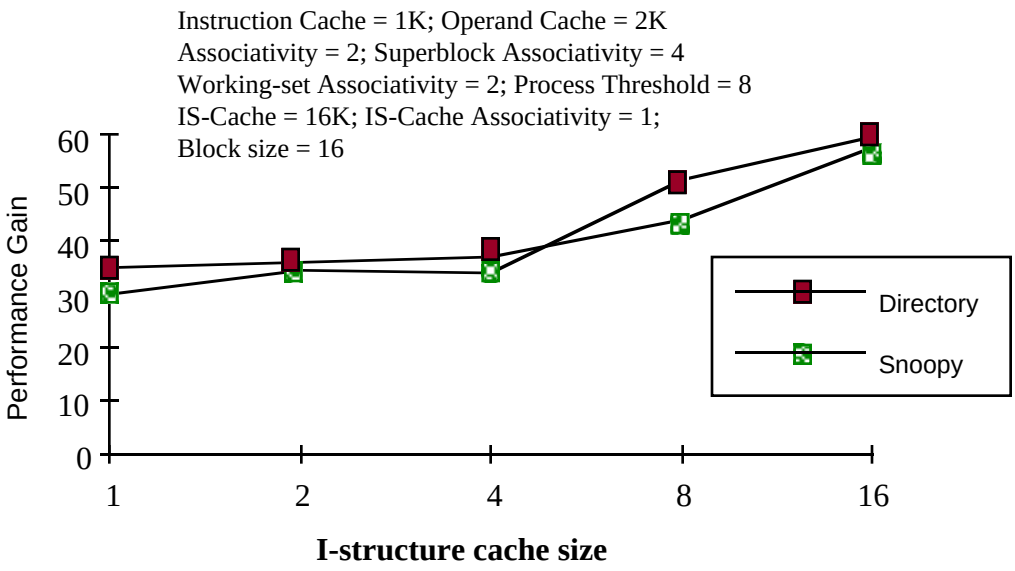


Figure 21: Snoopy Vs Directory.

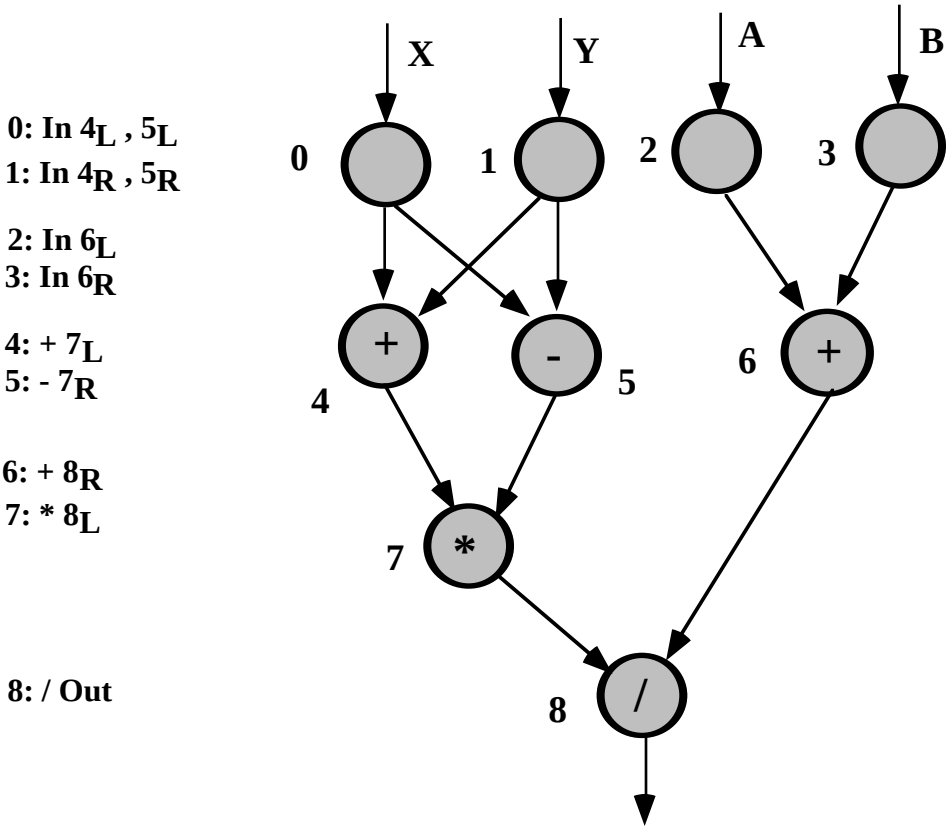


Figure 22: Data Dependencies and Operand Memory Reuse.

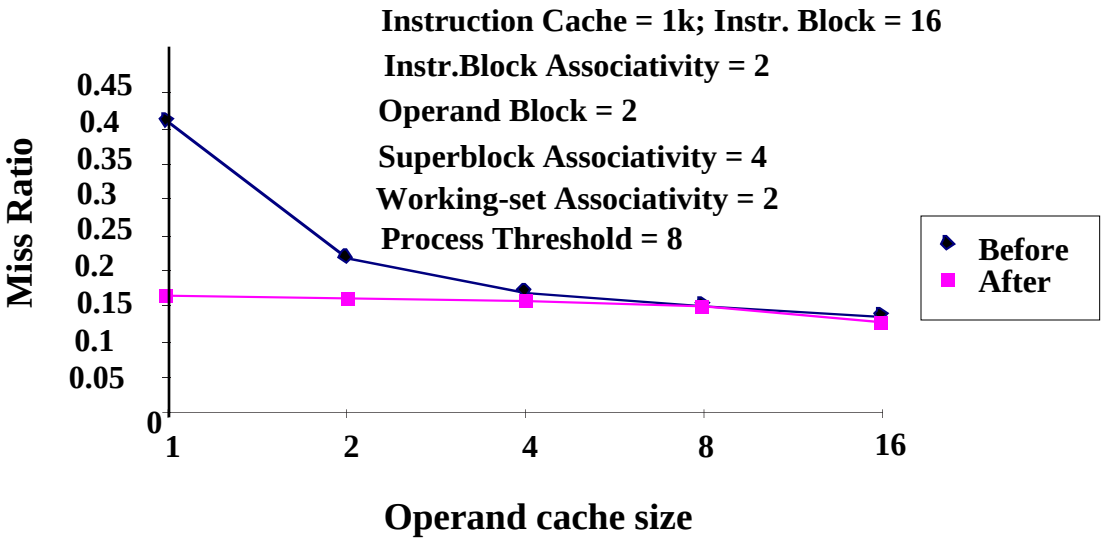


Figure 23: Significance of Resue for Matrix Multiplication.

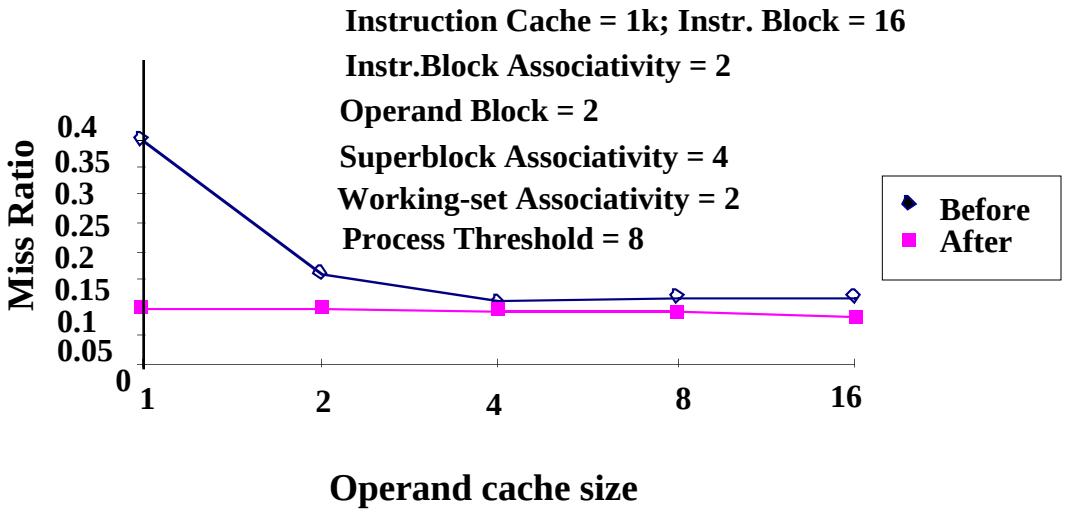


Figure 24: Significance of Resue for FFT.

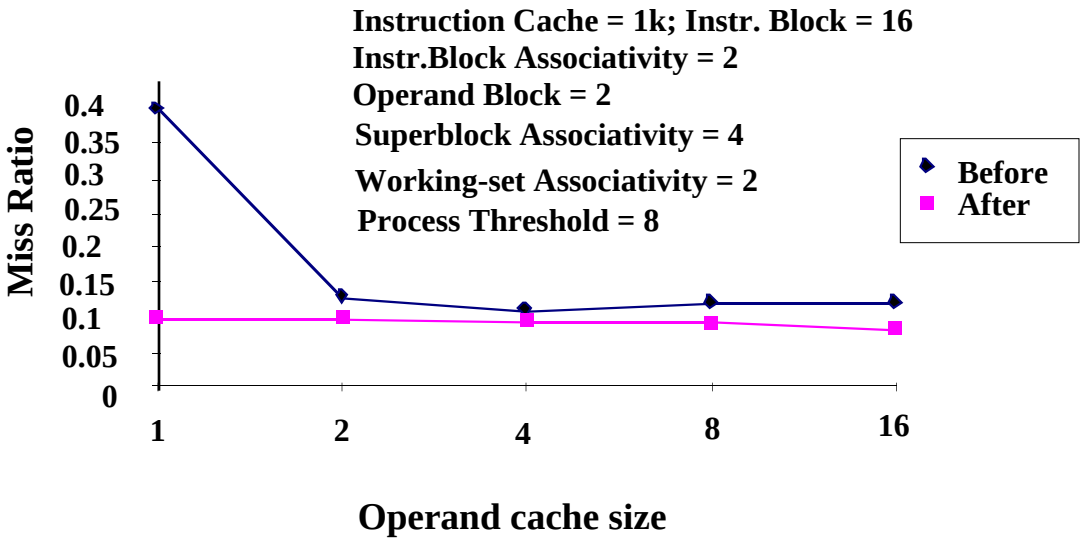


Figure 25: Significance of Resue for Loop5.

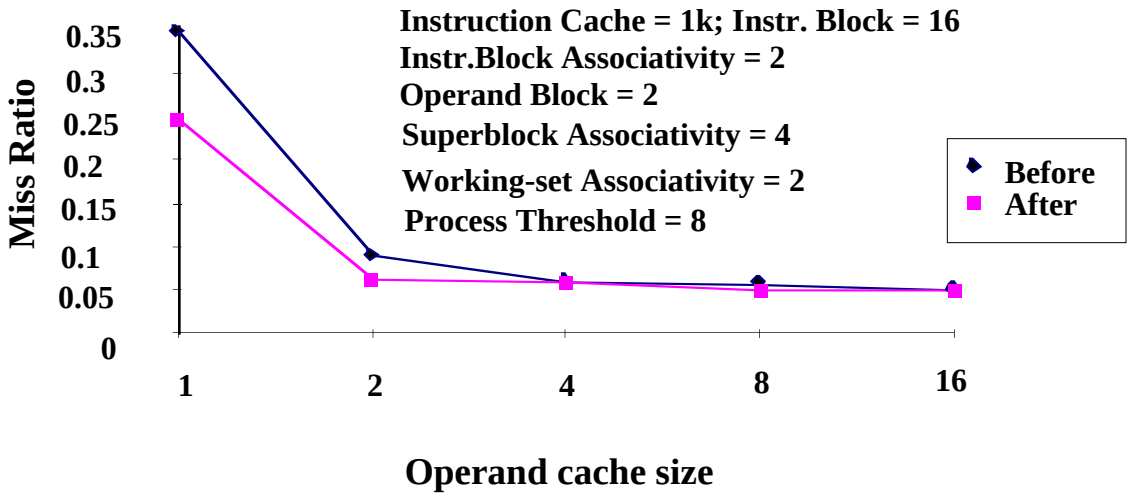


Figure 26: Significance of Resue for Random Graph.

modify ETS matching rules; the present bit should be reset after a match takes place, to indicate that the location can be reused. We hope to study the significance of program partitioning and scheduling on the reuse. Consider for example the dataflow graph shown in Figure 22. If instruction 6 is delayed until after execution of instructions 4 and 5, then we can reuse the matching location of instruction 0 (or instruction 1) for matching the operands of instruction 6. This, however, necessitates that instructions 2 and 3 also be delayed. It will be interesting to investigate the trade-off between the higher reuse (with concomitant improvement in cache performance) and the loss of instruction level parallelism.

7. SUMMARY AND FUTURE DIRECTIONS

In this paper we have shown how the performance of dataflow machines can be enhanced by the use of cache memories. In addition, we have demonstrated that the operand memory locations within a frame can be reused for the matching of the operands of multiple instructions. We believe that the amount of operand memory reuse can be increased by forcing sequential execution of instructions within a thread (bring the model closer to conventional control flow). However, this necessitates architectural modifications to the ETS. In addition, this may reduce the amount of parallelism, thus limiting the performance of the processor. The performance can be improved by interleaving instructions from several sequential threads [23]. A clear understanding of the issues in supporting multiple threads within the dataflow framework will permit us to adapt them to hybrid architectures. Hybrid systems present the most interesting opportunities in the area of multiprocessing — they directly address problems that will be faced by future superscalar processors such as, long memory latencies, context switching overhead, multiple active instruction streams, fast and efficient support for task synchronization.

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 multiprocessor systems. We hope to continue our studies by expanding the benchmark suite 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.

8. REFERENCES

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

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

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

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

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

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

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

[8] K.M. Kavi, A.R. Hurson, P. Patadia, E. Abraham and P. Shanmugam. (1995). "Design of cache memories for multi-threaded dataflow architecture", *Proceedings of the 22nd Intl. Symp. on Computer Architecture*, pp. 253-264.

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

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

[11] B. Lee and A.R. Hurson. (1994). "Dataflow architectures and multithreading", IEEE Computer, pp. 27-39.

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

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

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

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

[16] P. Shanmugam, S. Andhare, K. Kavi, B. Shirazi and A.R. Hurson. (1993). "Cache memory for an explicit token store dataflow architecture", *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pp 45-50.

[17] A.J. Smith. (1982). "Cache Memories". *ACM Computing Surveys*, pp. 473-530.

[18] M. Takesue. (1987). "A unified resource management and execution control mechanism for Dataflow Machines". *Proc. 14th Annl. Intl. Symp. on Computer Architecture*, pp. 90-97.

[19] M. Takesue. (1992). "Cache Memories for Data Flow Architectures". *IEEE Transactions on Computers*, 41(6), pp. 667-687

[20] I. Tartalja and V. Milutinovic. (1996). The Cache Coherence Problem in Shared-Memory Multiprocessors, Software Solutions, IEEE Computer Science Press, 1996.

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

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

[23] D.M. Tullsen, S.J. Eggers and H.M. Levy. (1995). “Simultaneous multithreading: Maximizing on chip parallelism”, *Proceeding of the 22nd International Symposium on Computer Architecture*, pp 392-403.