In Proc. Int. Symp. on Microarchitecture, pages 271–278, Nov.'95, Ann Arbor, MI.

Design of Storage Hierarchy in Multithreaded Architectures *

Lucas Roh

Math. & Computer Science Division
Argonne National Laboratory
Argonne, IL 60439 USA
roh@mcs.anl.gov

Walid A. Najjar

Department of Computer Science
Colorado State University
Fort Collins, CO 80523 USA
najjar@cs.colostate.edu

Abstract

Multithreaded execution models attempt to combine some aspects of dataflow-like execution with von Neumann model execution. Their main objective is to mask the latency of inter-processor communications and remote memory accesses in large scale multiprocessors. An important issue in the analysis and evaluation of multithreaded execution is the design and performance of the storage hierarchy.

Because of the sequential execution of threads, the locality of access within *an executing thread can be exploited using registers and cache. At the* inter-thread *level, however, the locality of accesses to memory and its effect on the cache is not yet well understood. A storage model which can exploit this locality is developed and evaluated. The results indicate there is a large amount of inter-thread locality that can be exploited and that we can get an efficient storage system by exploiting the characteristics of nonblocking threads.*

1 Introduction

Multithreading provides an effective way to mask long latency operations. When operations such as inter-processor communications or remote data accesses are encountered, the processor switches to another ready thread thereby reducing the idle time. A variety of multithreaded execution models currently exist: some rely on large threads that can block during their execution while others opt for shorter threads that always execute to completion.

It has been argued that the benefit of multithreading comes at a cost of reduced locality arising from the frequent context switching. Each thread has its own working set, and threads compete against each other for a limited cache space. This competition can result in increased cache misses, possibly negating the benefit of multithreading. However, the working set of a thread between context switches should be relatively small regardless of the multithreading model chosen. If there is a sufficient *inter-thread locality*, then a good storage hierarchy may be able to accommodate many threads and still be effective.

In this paper, we develop and evaluate a storage hierarchy for a non-blocking multithreading system. The side-effect of this study is that we gain insight into the inter-thread locality also. The Pebbles execution model which is used as the basis of our experimentation is described in Section 2. In Section 3, the overall system performance is measured in terms of the synchronization cost in order to evaluate the target performance of a storage hierarchy. The Pebbles storage hierarchy system is then developed in Section 4. In Section 5, the analysis of inter-thread locality and its effect on the basic storage system are evaluated in terms of miss rates. Related work and concluding remarks are then given in Sections 6 and 7.

2 The Pebbles Multithreaded Model

In this section we briefly describe the Pebbles architectural model [RNSB94] used in our experiments. As shown in Figure 1, the Pebbles architecture consists of one or more more processing nodes connected by a general, high speed interconnection network. The local memory of each node consists of an *Instruction Memory* which is read by the *Execution Unit* and a *Data Memory* (or *Frame Store*) which is accessed by the *Synchronization Unit*. The *Ready Queue* contains the *continuations* representing those threads that are ready to execute. The *Structure Memory* stores data structures and is distributed among the nodes. The

*This work is supported in part by NSF Grant MIP-9113268

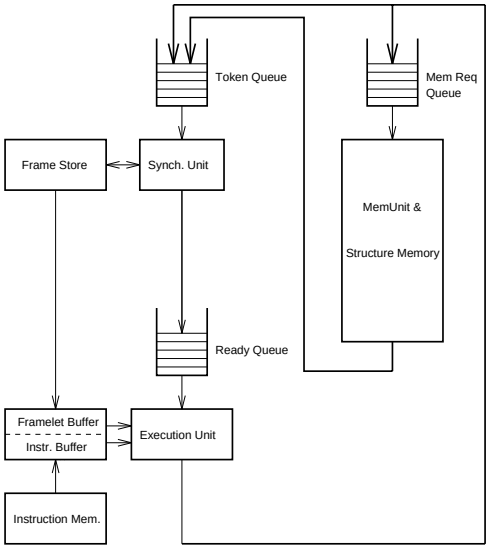

Figure 1: Abstract Model of a Processing Node.

MemUnit handles the structure memory requests.

The execution is based on dynamic dataflow scheduling where each actor, or a node in the dataflow graphs, represents a thread. Once a thread starts executing, it runs to completion without blocking and with a bounded execution time. A thread is enabled to execute only when *all* the inputs to the thread are available. Multiple instances of a thread can be enabled at the same time and are distinguished from each other by a unique "color." The thread enabling condition is detected by a matching/synchronization mechanism which matches inputs to a particular instance of a thread. Data values are carried in *tokens*. Each token also carries a continuation and an operand number to the thread. A continuation uniquely identifies an activation of a thread and consists of a color and a pointer to the start of thread. A given thread activation can be executed on any processor, determined at run-time. Since each thread is relatively small in our code generation scheme (10 to 30 RISC-style instructions) [RNSB94], global scheduling and near perfect load balancing is achieved by a simple hashing of the tag.

For each instance of a thread, a small fixed size storage area (called a *framelet*) is allocated in the Frame Store to hold the incoming inputs to that thread instance. A framelet is large enough to hold inputs for most threads. When a particular thread's requirement exceeds the size of a single framelet, one or more additional overflow framelets are allocated as needed. Another method, which our compiler does not yet implement, would be to break up the thread into two or more threads so that inputs can fit within a given size.

When the first input of a thread activation (an instance) arrives to a processing node, the Synchronization Unit will first allocate a framelet and set the count of the total number of inputs in the framelet. If necessary, it will handle exceptional cases such as allocating additional framelets and setting the overflow framelet pointers. Each input token is stored in an appropriate slot within the framelet and the counter is decremented. When the count reaches zero, the thread is enabled to execute by making an entry in the Ready Queue. Also, the content of the framelet is migrated to the *Framelet Buffer*, and the framelet is deallocated in the Frame Store. In addition, instructions associated with the enabled thread can be prefetched into the Instruction Buffer[1]. The Execution Unit can therefore operate at full speed without load stalls.

Pebbles programs are represented in a form of dataflow graphs called MIDC. Each node of the graph represents a thread of machine independent instructions. MIDC code is generated from Sisal programs. The code generation is guided by the following objectives: (1) Minimize synchronization overhead, (2) Maximize intra-thread locality, (3) Assure non-blocking (and deadlock-free) threads, and (4) Preserve functional and loop parallelism in programs; it is described in [RNB93, NRB94].

3 Performance Impact of Storage Hierarchy

To give a motivation for designing an effective storage hierarchy, the performance impact, or rather, a penalty, for using an *ineffective* memory system are described in this section. Also, benchmark programs used for all our experiments are introduced here.

3.1 Benchmarks and Simulation Setup

The experiments are conducted using a set of seven Sisal programs: 1) *SGA* uses a genetic algorithm to find a local minima of a function; 2) *FFT* is a 1-D FFT routine; 3) *PSA* is a parallel scheduler code; 4) *SDD* solves an elliptic partial differential equation; 5) *SIMPLE* is a Lagrangian 2-D hydrodynamics code; 6) *AMR* is an unsplit integrator taken from an adaptive mesh refinement code; 7) *WEATHER* is a one level barotropic weather prediction code.

[1]Not all enabled threads' data and instructions need to be immediately brought into the buffers; only those threads near the head of the ready queue need to be in the buffers.

Benchmarks	No. of Framelets	No. of Inputs	Avg. Input Set Size (Bytes)	Input Set Size < 128 Bytes (%)
SGA	1,372,099	8,712,006	25.3	99.6
FFT	225,706	4,142,029	69.1	96.2
PSA	1,664,336	7,098,854	18.1	100.0
SDD	1,716,892	10,520,838	29.6	100.0
SIMPLE	1,328,271	9,269,106	34.4	98.1
AMR	205,774	3,207,898	85.1	84.2
WEATHER	828,983	8,045,883	35.9	99.8
Overall	7,342,061	50,996,614	30.5	99.0

Table 1: Benchmark Characteristics

The dynamic characteristics of the benchmarks are given in Table 1. The columns successively represent: the number of framelets that were allocated, the number of inputs to the framelet[2], the average size (in bytes) of the framelet, and the percentage of framelets that are smaller than 128 bytes. Except for AMR, a high percentage of the framelets can fit within a 128-byte physical framelet.

Simulation. All measurements are derived using a simulator. The following architectural parameters are specified to the Pebbles simulator: a 4-way issue super-scalar CPU execution unit per node with the instruction latencies of the Motorola 88110 microprocessor. An output network bandwidth of two tokens per node is used. For simplicity, all inter-node communications take 50 CPU cycles in network transit time. Every structure memory read takes the minimum of two network transits (one to send the request and another to send the reply). Unless otherwise noted, ten processing nodes are used. The number of processing nodes, ten, was chosen because it provides a realistic processor utilization for the problem sizes that can be run in a reasonable time on our simulator. These architectural parameter values are chosen to give a reasonable approximation of a real machine rather than exact measurements.

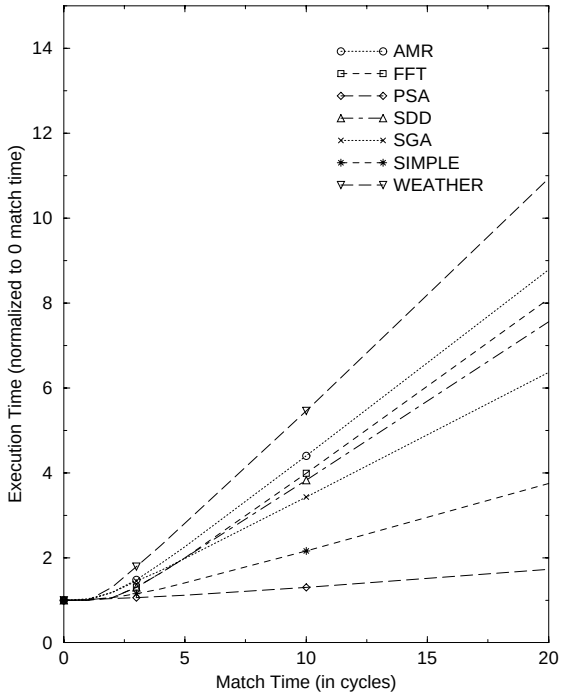

Figure 2: Execution Time versus Synchronization Time.

3.2 Impact

The performance of memory subsystem is modeled by varying the cost of synchronizing a token (an input). The resulting impact on the overall run-time performance is shown on Figure 2 as the cost is varied between 0 and 20 cycles[3]. The execution times shown are normalized with respect to the base execution time when the synchronization time is ideal (0 cycle).

The result shows that for the synchronization time beyond the first few cycles, the execution time grows linearly with respect to the synchronization time. In this linear regime, the synchronization operations are totally exposed to the multithreaded executions. For the match time less than a few cycles, especially less than 3 cycles, the performance degradation is relatively small compared to the ideal case. In this

[2]same as the number of tokens

[3]Synchronizing tokens on a mismatch (the first reference to a framelet) incurs 2 additional cycles.

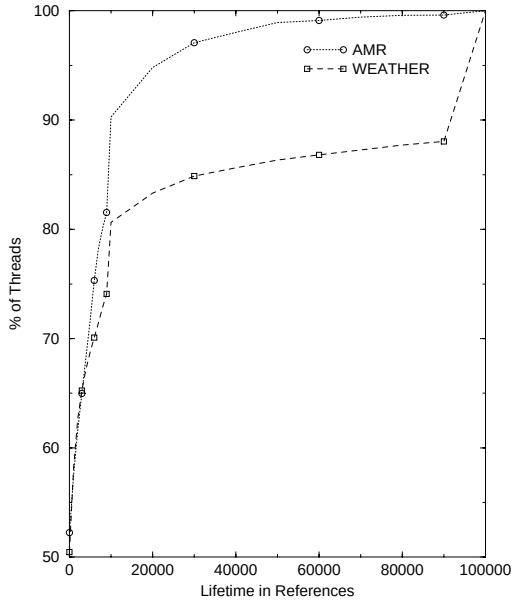

Figure 3: Distribution of Lifetime of Framelets in AMR and WEATHER.

regime, synchronization operations are well masked. Further experiments, not shown here, using 50 processing nodes also show a similar trend.

4 Pebbles Storage Hierarchy

The design goal of a Frame Store hierarchy is to minimize the average access time to Frame Store by the Synchronization Unit and thereby reduce the matching cost.

The crux of our design depends on the locality of tokens. The best scenario would be to have all the inputs for a given framelet arrive at once, followed by all the inputs for the next framelet, and so on; in this case, a single framelet sized cache would be sufficient to fully exploit locality. Note that the locality refers to the accesses of the Frame Store by the Synchronization Unit. It is not the locality of execution since the Execution Unit does not access the Frame Store: all the input data for each ready thread are, already, in the register-like Framelet Buffer before execution.

Locality of Framelets One direct approach to measuring the locality is to study the *lifetime* a framelet. The lifetime of a framelet is defined as the interval between the arrival of its first input and that of its last (when it is deallocated). This interval is measured in number of references to the framelet store (or token arrivals), and therefore gives an idea of how many other inputs will arrive during the lifetime of a framelet. Figure 3 shows the distribution of the lifetime for AMR and WEATHER which are representative of all the benchmarks. For the majority of threads, the lifetime is less than 50 cycles. The cumulative distributions rise quickly, reach their plateau at around 1000-2000 cycles, after which they rise rather slowly. The average result shows that the percentage of framelets whose lifetime is less than 2000 references: 57% to 94% of all framelets are in this category. Considering that the number of references per benchmarks is in the millions, a lifetime of under 2000 references is fairly short. The result indicates that a cache with an aging-based replacement policy, such as LRU or FIFO, might work well: With about 2000 cache blocks, by the time a cache block is replaced, it is highly probable that the associated framelet is already deallocated (i.e., the thread became ready).

4.1 Frame Store Model and Methodology

Figure 4 shows the cache-like design where *active* framelets are resident in the fast *Framelet Cache* (Cache) while the *inactive* ones are stored in the slower *Backup Framelet Store* (B-Store). The Cache and B-Store operate exclusively: any framelet in the cache will not be resident in the B-Store and vice versa.

Operations. A trace-driven simulation is used to model different cache designs. The traces of the incoming tokens to processors are generated by the Pebbles machine simulator. Figure 5 shows operations involved in storing an input to the Frame Store. The Cache is modeled as a fully associative cache where each block corresponds to a framelet and is logically addressed. The logical address of a framelet consists of a thread address and a color. The access to the cache is fully pipelined and is stalled on a miss. The B-Store is physically addressed. Translation from the logical to the physical address is performed by the *Backup Framelet TLB* (TLB) which has one entry per framelet. When a logical framelet address is presented to the Frame Store, the address is simultaneously sent to both the Cache and the TLB. In the case of a cache miss and a TLB hit, the framelet is brought from the B-Store, possibly replacing a cached framelet. In the case of a miss in both the Cache and TLB, implying the reference is to a new framelet (i.e a compulsory miss), a block is allocated in the Cache

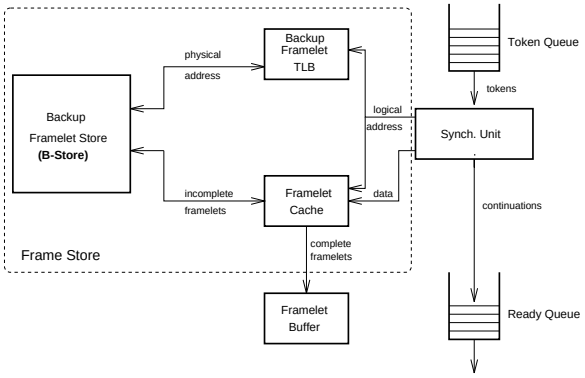

Figure 4: Storage Hierarchy Model.

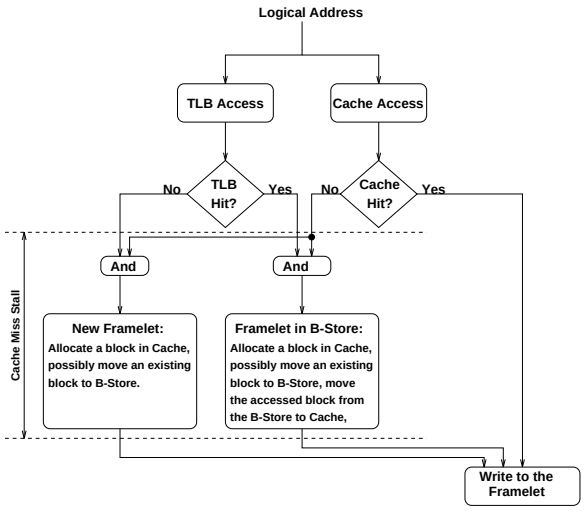

Figure 5: Processing of an input through the Frame Store.

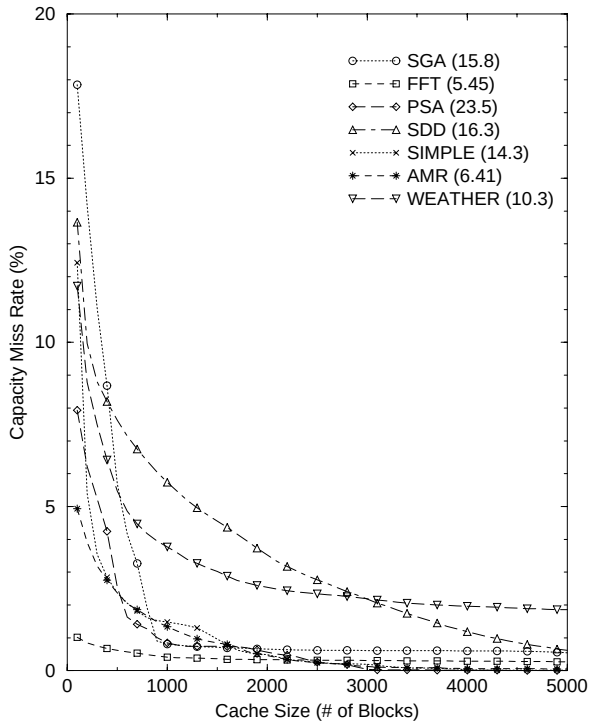

Figure 6: Capacity miss rates of *Basic* Cache.

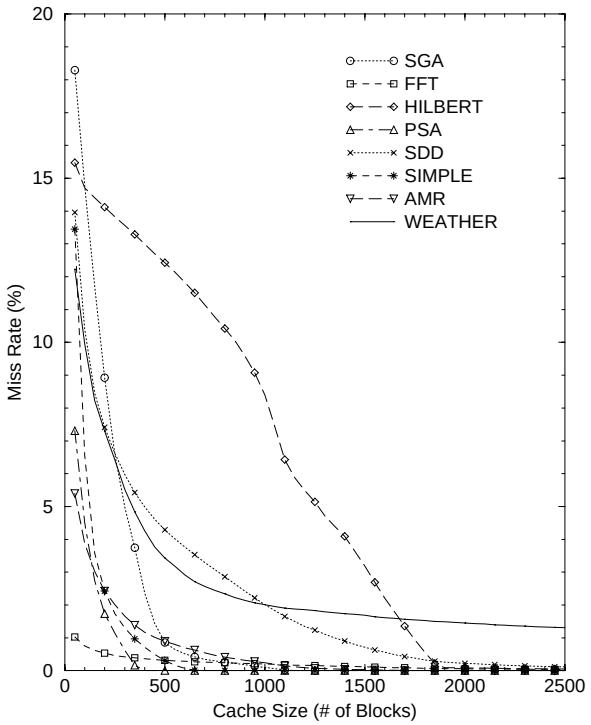

Figure 7: Compulsory miss rates of *Freelist* Cache.

again possibly replacing a cached framelet. Replacing a cached framelet involves allocating a framelet space in the B-Store, updating the TLB, and transferring the framelet from the cache. When a framelet is deallocated, the Synchronization Unit notifies the Cache so that the corresponding block can be marked invalid.

Replacement Algorithm. A simple FIFO scheme is used whenever a free block is needed. We have also studied the effect of other algorithms, including LRU, however their results are not shown in this paper due to the lack of space. We note that FIFO did almost as well as the best performing LRU.

Cache Organizations. There are many ways to design the cache that take a particular advantage of the way nonblocking threads operate. We develop three schemes of increasing complexity and compare the performances of each:

- *Basic Cache* is the garden variety cache that always invokes the replacement policy to allocate a block for nonresident framelets. If the chosen block is not invalid, then the block is replaced to the B-Store. This is our default policy.

- *Freelist Cache* maintains a list of free (invalid) blocks. Whenever possible, the Cache allocates from the free block list. The replacement policy is invoked if the list is empty. The advantage of this policy over the *Basic* is that it prevents an active framelet from being unnecessarily replaced, at the cost of additional complexity.

- *Reserve Block Cache* attempts to reduce the cost of the compulsory misses by making use of the free-blocks and marking one of them as *reserved* specifically to handle the *next* compulsory miss.

The block size of Cache is set to 128 bytes. Since Cache is fully associative[4], only compulsory or capacity misses occur. A compulsory miss occurs when the first input of a thread activation arrives. The compulsory miss rate is inversely proportional to the average number of inputs to threads, and is independent of the cache size[5]

[4] We note that our block size is fairly large, a 256K byte cache has only 2000 blocks to be associatively searched.

[5] Invalidation misses do not occur since the data values in each framelet are unique to that framelet.

5 Analysis and Evaluation

Performance of *Basic* Cache. Figure 6 shows the plot of capacity miss rates of various benchmarks as the number blocks in the cache is varied. The compulsory misses are shown within parentheses next to each legend. The figure shows that there is a high degree of locality of references to the framelets. With a 256K cache (2000 blocks of 128 bytes each), the capacity miss rates are below 1% for all the benchmarks except for SDD and WEATHER. With a 5000 block cache, the miss rates drop below 2% for WEATHER and well below 1% for the others. The WEATHER benchmark works with a large data set and hence its inherent locality is rather poor. FFT has a very low miss rate of around 1% with only 100 block cache, but the miss rate does not drop much after a 1000 block. We can attribute the low miss rates to the following: 1) only a fraction of parallel loop iterations are likely to be active concurrently for an extended period of time and therefore present a higher locality; and, 2) there is a high locality among neighboring threads (e.g. within a loop body). Note that the compulsory miss rates of benchmarks are relatively high, 6.4% to 23.4%.

Performance of the *Freelist* Cache. Figure 7 shows the plot of capacity miss rates versus the number of blocks. It is not too surprising that the miss rates are much lower. With 2000 blocks, miss rates are well below 1% for most benchmarks. We note again that the improvement comes from not replacing resident framelet if possible, thus saving the cost of bringing that framelet back at a later time.

5.1 Compulsory Misses

Results so far indicate that capacity misses can be made very small with a reasonable sized cache at which point compulsory misses become the dominant type of misses. For example, with a 2000 block *Basic* Cache, the capacity misses range from 0.34 to 3.52%, whereas the compulsory misses range from 5.45 to 23.4%. The cost of a compulsory miss consist of the allocation of a cache block, and possibly replacing a cached framelet.

The *Reserve Block* Cache attempts to reduce the cost of compulsory misses to that of a cache hit by making use of the free-blocks and marking one of them as *reserved* specifically to handle the *next* compulsory miss. The reserved block *cannot* be used to satisfy the capacity miss, i.e., it will not be used to allocate a framelet from the B-Store. The index of the reserved block in the cache is kept in a special register. We

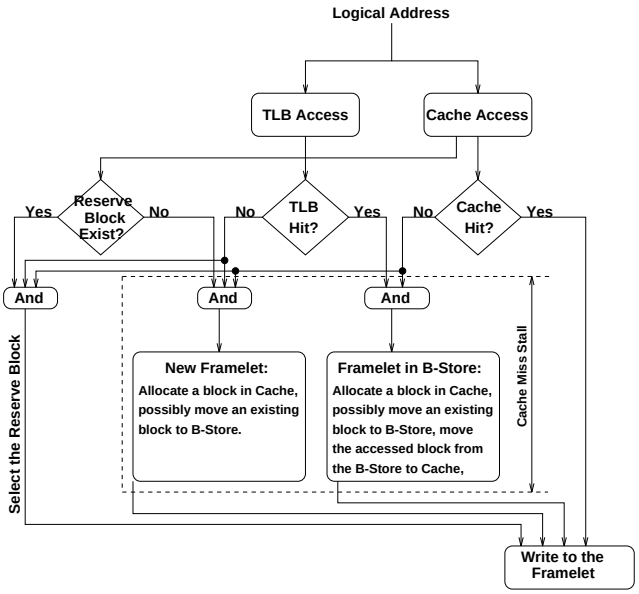

Figure 8: Processing of an input through the Reserve Block Cache.

recall that a compulsory miss is detected by a miss in both the Cache and TLB. When such a miss is detected, the reserved block immediately becomes the substitute block associated with the logical framelet with a very small delay. The management of the free list in the cache and the designation of the reserved block is done in the "background" and would not affect the cache access time. A stall occurs when there is no reserve block (i.e., the free-block list is empty). In this case a cache block must be replaced. We also note that a stall occurs whenever a compulsory miss is *immediately* (i.e in the next cycle) followed by one or more compulsory misses. Figure 8 shows the operations involved in the Reserve Block scheme.

Overall, the compulsory miss rate is now dependent on the probability of encountering an empty free block list, since the cost of accessing a new framelet is near that of the cache hit if the allocation occurs out of the reserve block.

Figure 9 shows the effect of the Reserve Block scheme: the effective compulsory misses are significantly reduced even with a very small cache size. With a reasonably sized cache (i.e. 2000 blocks), the compulsory misses are smaller than the capacity misses for all the benchmarks. The total miss rates with a 2000 block Cache is shown in Table 2. The reduction in the total miss rate ranges from a factor of six to 2400.

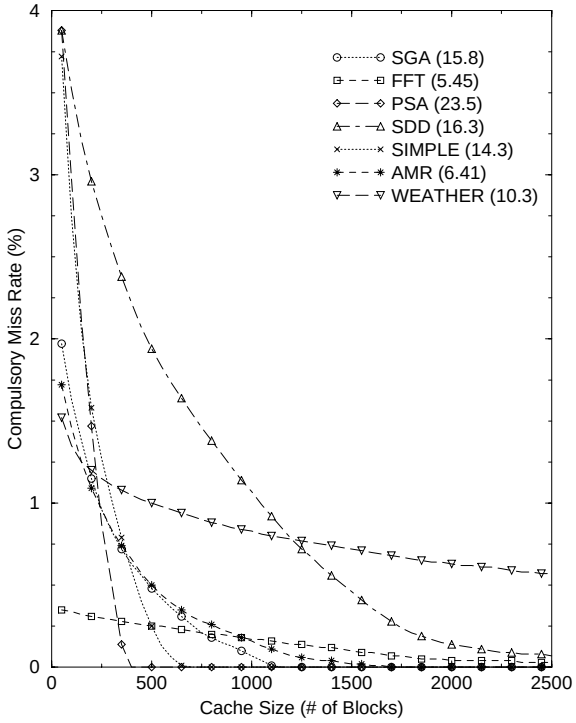

Figure 9: Compulsory miss rates using the Reserve Block scheme (Original miss rate in parentheses next to legends).

System	Total Miss Rates for 2000 block cache: SGA	FFT	PSA	SDD	SIMPLE	AMR	WEATHER
Basic	16.4	5.79	24.0	19.8	14.8	6.88	12.8
Reserve Block	0.01	0.14	0.01	0.45	0.01	0.01	2.31

Table 2: Comparison of the Effective Miss Rates (%) between *Basic* and *Reserve Block* Caches

6 Related Work

A number of published papers have discussed issues related to thread level locality and the design of storage hierarchies to exploit it. Due to the space limitation, it is not possible in this paper to present a detailed comparison between our work and other similar ones, especially since the issues sometimes depend on a specific multithreaded execution model and a code generation strategy. We will limit the discussion therefore to a simple description of these.

Culler *et al.* [CSvE93] argue that the performance of the storage hierarchy fundamentally limits the amount of multithreading within a processor, thus limiting the latency that can be tolerated. The idea of storage hierarchy rests on the principle that fast memories are small and expensive while slow memories are large and inexpensive. The observation is that switching is cheap only for those threads residing in the top part of hierarchy. Hence, only a limited number of threads may be switched inexpensively. A scheduling policy that favors threads that already lie in the top part of hierarchy would be preferred.

Research on incorporating caches into multithreaded executions and measuring their effectiveness is ongoing. Cache designs in a dataflow model is discussed in [Tak92] for DFM-II [Tak87]. This model is designed for a fine-grained dataflow machine and must therefore take into account the explosion of parallelism that is typical in these machines [Cul89]. The model is evaluated using a relatively small set of hand-coded benchmarks. The cacheing mechanism attempts to preserve the working set of the program in the cache.

7 Conclusion

Multithreading provides an efficient mechanism to overlap communication with computation resulting in an improved processor utilization. In order to consider the overall system performance, the issue of inter-thread locality in a multithreaded execution in the presence of memory hierarchy must be addressed. In this paper, this issue is measured quantitatively (indirectly via cache misses) and a storage hierarchy is developed to take advantage of the locality and to reduce the cost of synchronizations.

The results show that the reference stream contains a great deal of locality: a cache size of 256K is sufficient to bring the capacity miss rates well below 1% for most benchmarks using a simple FIFO replacement policy and keeping a list of free-blocks.

To deal with compulsory misses, a scheme was devised where one free cache block, called a *reserve block*, is kept on reserve and used when a compulsory miss is detected. Hence, the cost is reduced to nearly that of a cache hit when there exists a reserve block. The resulting scheme produces *total* miss rates well below 1% for all but one benchmark.

The upshot of our storage hierarchy scheme is that the average synchronization time is reduced to nearly 1 cycle, even when the miss penalty is 10 cycles. The drawback is that the design relies on a fully associative cache. Future research will focus on more technologically feasible cache designs.

References

[CSvE93] D. E. Culler, K. E. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In Cosnard, Ebcioglu, and Gaudiot, editors, *Proc. IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, Orlando, FL, 1993. North-Holland.

[Cul89] D. E. Culler. *Managing parallelism and resources in scientific dataflow program.* PhD thesis, MIT, June 1989.

[NRB94] W. Najjar, L. Roh, and W. Böhm. An evaluation of medium-grain dataflow code. *Int. J. of Parallel Programming*, 22(3):209–242, June 1994.

[RNB93] L. Roh, W. A. Najjar, and W. Böhm. Generation and Quantitative Evaluation of Dataflow Clusters. In *Proc. Symposium on Functional*

Programming Languages and Computer Architecture, pages 159–168, Copenhagen, Denmark, 1993.

[RNSB94] L. Roh, W. A. Najjar, B. Shankar, and A. P. W. Böhm. An evaluation of optimized threaded code generation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques(PACT'94)*, Montreal, Canada, 1994.

[Tak87] M. Takesue. A unified resource management and execution control mechanism for data flow machine. In *Int. Ann. Symp. on Computer Architecture*, pages 90–97. ACM, 1987.

[Tak92] M. Takesu. Cache memories for data flow machines. *IEEE Trans. on Computers*, 41(6):677–687, June 1992.