Proc. Int. Symp. on Parallel and Distributed Processing, October 1996, New Orleans, LA.

Comparison of Two Storage Models in Data-Driven Multithreaded Architectures *

Murali Annavaram and Walid A. Najjar
Department of Computer Science
Colorado State University
Fort Collins, CO 80523 USA
{*najjar,annavara*}*@cs.colostate.edu*

Abstract

Multithreaded execution models attempt to combine some aspects of dataflow-like execution with von Neumann model execution, with the objective of masking the latency of inter-processor communications and remote memory accesses in 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. Two storage hierarchy models, that attempt to capture and exploit this locality, are described and evaluated in this paper.*

1 Introduction

The foremost benefits of multithreading are the increased processor utilization realized by dynamically switching among ready threads at run-time and thereby its ability to mask memory and communication latencies. A consequence of this dynamic thread scheduling may be a loss of locality of access to the data storage, thus reducing the effectiveness of a traditional cache scheme.

Two mechanisms have been proposed to alleviate the loss of locality in the data accesses: The first method relies on static scheduling of threads to make the execution more predictable, the amount of locality that may be exploited by the cache is increased. This is the method adopted in TAM [CSS+91], whereby related threads are scheduled statically whenever feasible thus increasing the amount of locality available beyond a single thread. An orthogonal approach is to design efficient storage schemes that can make use of the locality available, both the intra-thread and the inter-thread locality. Such a mechanism is described in [RN95].

This paper examines the performance of two storage models for dynamically scheduled non-blocking threads: the *frame*-based and *framelet*-based models. The Frame model relies on a traditional cache design that is associated with an ETS-like (Explicit Token Store [CP90]) frame storage mechanism. Each frame consists of memory for the inputs and synchronization slots for a set of threads that belong to the same code block. The Framelet model relies also on traditional cache design except that the each framelet corresponds to storage required for only one thread. The framelet also has a synchronization slot corresponding to the thread. Hence we can consider a Framelet model as special case of Frame model where each frame holds only one thread.

The multithreaded execution model and the two memory models are described in Section 2. The experimental framework used in the evaluations is discussed in Section 3, with the results and their analysis given in Section 4. Related works are discussed in Section 5 and concluding remarks are given in Section 6.

2 Execution and Storage Models

The multithreading execution model used in this paper is based on a data-driven dynamic scheduling of threads. Threads are non-blocking: once a thread starts executing, it runs to completion without switching to another thread. A thread is enabled when *all* the inputs to the thread are available. Threads are

*This work is supported in part by NSF Grant MIP-9113268 and by DARPA Contract Number DABT63-95-0093

generated statically: to each thread is specified the number and type of the input data values as well as the thread(s) to which it sends result values, specified in terms of the thread number and offset. The input data set consists of the data values that the thread will need to complete the execution. Within a thread all temporary values are stored in registers and do not live across threads.

A code-block [Ian90] represents a semantically distinguishable code segment such as a loop or function body. There will usually be several static threads in a code-block. A *context* (often referred to as a "color") represent a dynamic instance of a code block and distinguishes multiple dynamic instances of a thread. Data values along with the context information are carried in *tokens*. The thread enabling condition is detected by the *Synchronization Unit* which matches all the input tokens to a particular instance of the thread. In the benchmark programs used (see Section 3) the thread size ranges from 10 to 30 instructions. Global scheduling and load balancing across processors is achieved by a simple hashing of the context information.

The Architectural Model. The basic organization of a processing node is shown in Figure 1, it is similar to a basic dataflow machine organization[1]. The *Execution Unit* executes ready threads which generate new data values that are forwarded by the *Synchronization Unit* to the destination threads. When all the data inputs to a thread are available, it is enabled to execute. The *Instruction Memory* stores thread instructions and the *Frame* or *Framelet Store* stores the data values that are inputs to pending (not ready) threads. The *Ready Queue* contains *continuations* representing those threads that are ready to execute. A continuation consists of a pointer to the starting address of the thread code and another to the frame or framelet containing the data values associated with that thread. A number of these processing nodes may be connected by an interconnection network to form a multiprocessor.

The two storage models discussed in this paper rely on this same architecture, their differences lie in the organization of the Synchronization Unit and the Frame or Framelet Store.

The Frame Model and Operation. A frame represents a storage segment associated with each *invocation* of a code-block. All the threads within the code-block will use the associated frame to store data values. This model is used in several multithreaded models (e.g. TAM [CSS+91], StarT-NG [AAC94] and the EM-4 and EM-X [KSS+95]). Frames are of variable size and contiguously allocated in the virtual address space. The size of a frame is determined based on the maximum number of data values associated with the code block.

[1]For the sake of brevity, other components not germane to this study, such as the globally distributed data structure memory, are not shown.

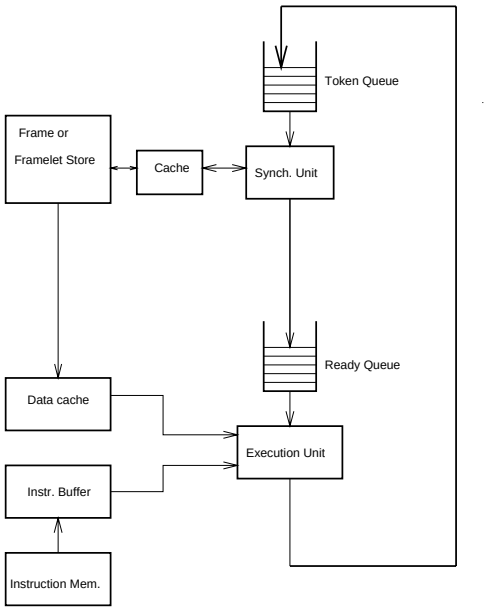

Figure 1: Abstract Model of a Processing Node.

When an instance of a particular code-block is invoked, a frame is first allocated in a given processor and all the tokens generated within that code-block instance will be stored in that frame. The virtual address carried by a token is of the form:

<*frame pointer, frame offset*>

The cache organization is that of a conventional set-associative write-through cache. Because of the variable frame size, a cache block could hold contents of more than one frame (or less than a frame). A synchronization slot in the frame is associated with each thread. Whenever an active thread sends a result to more than one thread belonging to the same frame only one copy of the value is stored in the frame. But the synchronization slots associated with each thread that utilizes this value is decremented. This process of decrementing the counters and storing the values in the frame is done either by the thread that is sending the token or by a special thread called *inlet* as used in TAM [CSS+91]. When the synchronization slot of a thread becomes zero the corresponding thread in the frame is enabled. When a thread is enabled a corresponding continuation is placed on the Ready Queue.

The content of the frame is accessed by the Execution Unit via a read-only cache. Tokens generated by the thread are sent to the Synchronization Unit to get written in the frame. The frame is deallocated when all the threads in the code-block have terminated.

A code-block consisting of three threads is shown in Figure 2. The corresponding frame memory model is shown in the Figure 3. The input z which is used by both threads A and B is stored at only one place in the frame memory. Each of the values in the frame memory is accessed by the frame base address and the offset into the frame. The first three slots are the counters for the three threads. Thus when the value y is stored only Counter A is decremented. But when z is stored both Counter A and Counter B are decremented but only one copy of z is stored in the frame.

The Framelet Model and Operation. The Framelet model also relies on a conventional set-associative write-through cache. A framelet is associated with each thread activation. Each framelet has one synchronization slot corresponding to the thread. This can be visualized as Frame model of execution where every frame represents a single thread, instead of representing a code block.

In the Framelet model if a thread sends a token which will be used by more than one thread then multiple copies of this value(multiple tokens) are sent to different framelets associated with the threads. Every token also accesses the counter corresponding to the framelet and decrements it. When the count reaches zero, the thread is enabled to execute. The virtual address of a token in Framelet model is of the form:

<context #, thread #, framelet offset>

The Framelet model corresponding to the code block in Figure 2 is shown in Figure 4. There are three separate framelets for each of the threads A, B, C. Each framelet contains the counter for the corresponding thread. Each framelet contains a memory location for all the inputs to the corresponding thread. Hence framelet A corresponds to one particular activation of thread A. The z is stored in the framelets of both threads A and B and both counters are decremented. This accomplished as two separate store operations.

Discussion of the Models. The main difference between the Frame model and Framelet model of execution is the token duplication. The Framelet model does require that variables that are *shared* by several threads *within* a code block be replicated to all these threads while in the Frame model these variables are allocated only once in the frame.

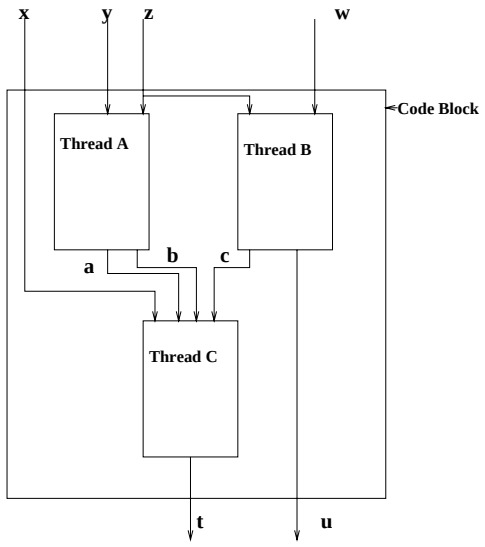

Figure 2: Code block with Three threads.

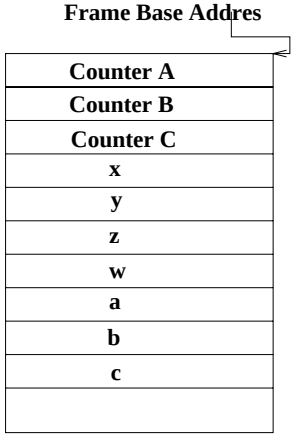

Figure 3: Frame Memory Representation.

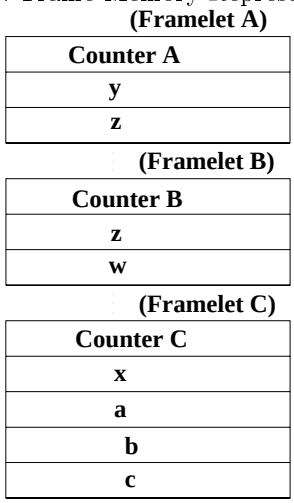

Figure 4: Framelet Memory Representation.

The advantage of Framelet model is that it is possible to design special storage schemes [RN95] that can take advantage of the locality of the inter-thread and intra-thread locality. But to make a fair comparison between Frame and Framelet model in this paper we restrict ourselves to using the traditional cache scheme for our simulations(see Section 3).

The amount of duplication between the Frame and Framelet models is shown in Table 1. The size of the traces in the frame or Framelet model differ by less than 4%. The main advantage of the Frame model is the reduction in the number of tokens sent. But from the table is is clear that such a decrease in the number of tokens is very small. This indicates there is not much sharing of tokens between the threads of the same frame.

3 Experimental Framework

The evaluation of the storage models is based on trace-based simulation. The traces are obtained from the simulated execution of the benchmarks using both frame and framelet based execution.

Trace Generation and Simulation. The traces are generated based on the following architectural parameters: a 4-way issue super-scalar CPU execution unit per processor.

The simulator output is a dynamic token trace, one per processor. Each processor trace is simulated separately and the results averaged. The average variance in the trace size among the ten processor traces for all benchmarks used is 1.5% which indicates a very reasonable load balance among the processing nodes. The storage hierarchy simulation uses DineroIII both for the frame based model and the Framelet model.

A wide range of cache associativities (direct-mapped, 2-way, 4-way, 8-way and fully-associative) is used for both trace simulators. The cache sizes range from 4 KB to 256 KB per processor. All results are measured separately for each processor and averaged across all processors. The variation in the miss rates between the processors is less than 2%. Hence the mean value represent closely the actual miss rates of each processor. The framelet-based and the frame-based storage models uses the block sizes of *32 bytes, 64 bytes, 128 bytes and 256 bytes*. Even though some of these sizes are beyond what is commonly found in today's cache designs, they are used for the sake of experimental investigation. A simple FIFO scheme is used for block replacement in both models.

Benchmarks	Trace Size		Replication(%)
	Frame	Framelet	
AMR	641648	645321	0.57
FFT	829702	859232	3.43
PSA	1419798	1459898	2.82
SDD	2104168	2104822	0.03
SGA	1742532	1802562	3.44
SIMPLE	1853822	1853962	0.01
WEATHER	1609176	1619876	0.66

Table 1: Trace Sizes

Benchmarks. The seven benchmark programs used in these experiments are written in Sisal[2]. The multi-threaded 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. The benchmark programs consist of: 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.

4 Results and Analysis

The results of the trace simulation and their analysis are presented in this section. The effects of the cache size, set-associativity and block size are examined for each of the two models.

4.1 Frame-based Simulation

Cache Size. The miss rates, as the cache size is varied from 4K bytes to 256K bytes per processor for a 4-way set-associative are shown in Figure 5 for block size from 64 Bytes to 256 Bytes[3]. The incremental improvement in the miss rates beyond a size of 32 KB is small for all the benchmarks. As the size of cache increases the compulsory misses tend to dominate the

[2]Sisal [MSA+85] is a strict functional programming language with arrays and loops.

[3]To make the graphs readable the graphs are drawn for the weighted average of the miss rates of all the seven benchmarks, the weights being the size of the traces.

overall miss rates. Since compulsory misses are unaffected by the cache size, the improvement in the miss rates diminish as the cache size increases. Given the trace size, we chose to restrict our analysis to a cache size of less than 64 KB. The ratio of cache size to the trace size is in the range of 0.5% to 2% for a cache size of 16 KB which is a valid ratio. Unless otherwise specified all our further results are given for a cache size of 16 KB.

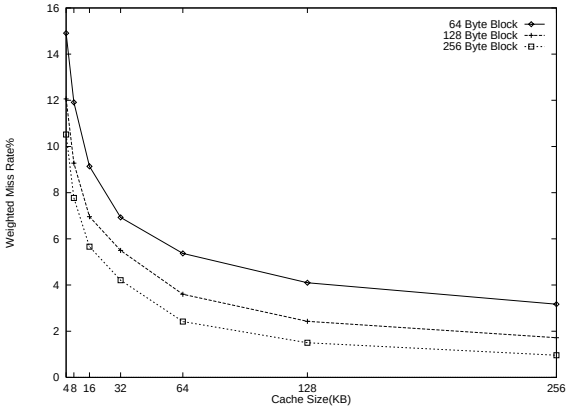

Figure 5: Effect of Cache Size : Frame Based 4-way associative cache.

Set-Associativity. The miss rates for direct-mapped, 2-way, 4-way, 8-way set-associative, and fully-associative 16K cache per processor with 64-byte blocks are shown in Table 2. The associativity has no appreciable effect on the miss rates. This indicates that a the percentage of conflict misses is small which is an indication of the high reference locality in the traces. This high locality means that all the references to a given frame are clustered in relatively small time span. Hence as long as the cache size is reasonably large enough to accommodate the locality the conflict misses will be negligible.

Block Size. When there exists sufficient locality in the address trace the compulsory misses can be reduced by increasing the block size. Table 3 shows the effect of block size on the miss rates. An 8-way associative 16K bytes cache is used with block sizes set to 32, 64, 128 and 256 bytes. At the associativity of 8 the conflict misses are almost zero. Moreover the capacity misses are small for cache sizes of 16 KB. Table 3 does indicates a noticeable reduction in miss rates with the increase in block size. When the block size is small a few tokens are sufficient to fill the block and the next incoming token will cause a compulsory miss. Due to the high locality the token causing the miss will have the virtual address close to the block that is just filled.

Benchmarks	Miss Rates % Associativity				
	1	2	4	8	Full
AMR	6.36	6.03	6.00	6.00	6.00
FFT	6.33	6.03	6.02	6.02	6.02
PSA	2.58	2.62	2.62	2.62	2.62
SDD	14.55	14.01	13.97	13.97	13.97
SGA	9.84	8.50	8.07	7.98	7.98
SIMPLE	7.73	7.27	7.21	7.23	7.23
WEATHER	15.60	15.03	14.86	14.79	14.79
Weighted Avg	9.82	9.27	9.15	9.12	9.12

Table 2: Effect of Associativity: Frame-based Cache, 16 Kbytes, 64 byte blocks

Benchmarks	Miss Rates % Block Size			
	32	64	128	256
AMR	10.59	6.01	3.60	2.26
FFT	11.13	6.02	3.32	1.78
PSA	5.01	2.62	1.42	0.85
SDD	19.36	13.97	10.41	7.93
SGA	11.12	7.98	6.35	5.78
SIMPLE	11.43	7.23	4.72	3.03
WEATHER	20.38	14.79	11.86	10.57
Weighted Avg	13.46	9.13	6.66	5.25

Table 3: Effect of Block size: Frame Based 8-way associative Cache, 16K bytes

By increasing the block size these misses are reduced.

4.2 Framelet-based Simulation

Cache Size. Figure 6 shows the effect of varying the cache size from 4KB to 256 KB for a 4-way set-associative cache with 64, 128, 256 byte blocks. The incremental improvement in the miss rates starts to decrease after a cache size of just 8 KB. For the traces used here the ratio of the cache size to the trace size is in the range of 0.5% to 1% when the cache size is 8 KB. Hence the cold start misses are not dominant for a cache size of 8 KB. This behavior is explained by the high locality in the trace. The cache size of 8 KB can effectively reduce the capacity misses. An associativity of 4 can reduce the conflict misses thereby making the misses just compulsory misses.

Set-Associativity. Table 4 shows the effect of varying the associativity. The results are derived using a 16 KB cache per processor with a 64-byte block and direct-mapped, 2-way, 4-way, 8-way set-associative and fully associative. Here, the associativity has negligible effect on the cache miss rates and the reason

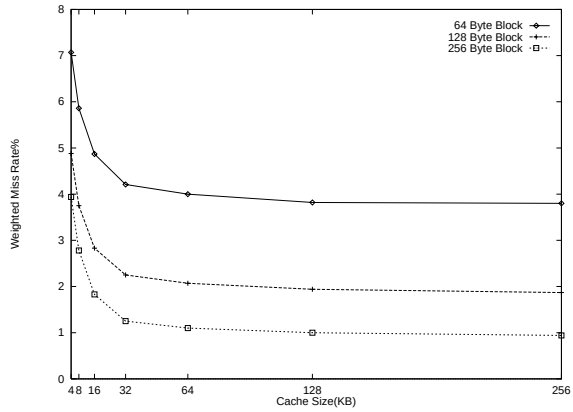

Figure 6: Effect of Cache Size : Framelet Based 4-way associative cache.

for this is similar to the reason given for the Frame model.

Block Size. When there exists sufficient locality in the address trace the compulsory misses can be reduced by increasing the block size. Table 5 shows the effect of block size on the miss rates. An 8-way associative 16K bytes cache is used with block sizes set to 32, 64, 128 and 256 bytes. At the associativity of 8 the conflict misses are almost zero. Moreover the capacity misses are small for cache sizes of 16 KB. Table 3 do indicate a noticeable reduction in miss rates with the increase in block size.

4.3 Discussion of the Two Models

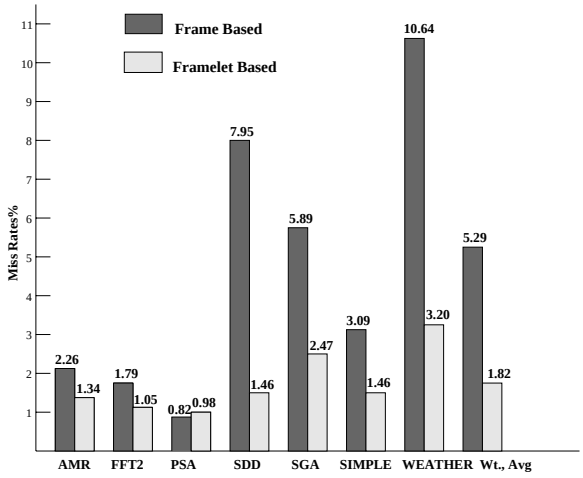

Figure 7: Comparison of Models: Best Performance Parameters,16 KB, 4-way associative, 256 byte Block cache

Benchmarks	Miss Rates % Associativity				
	1	2	4	8	Full
AMR	4.61	4.61	4.63	4.64	4.67
FFT	3.89	3.87	3.87	3.87	3.88
PSA	3.88	3.88	3.88	3.88	3.88
SDD	4.65	4.62	4.63	4.63	4.64
SGA	5.24	5.10	5.08	5.08	5.08
SIMPLE	4.71	4.63	4.61	4.60	4.60
WEATHER	6.57	6.43	6.40	6.38	6.37
Weighted Avg	4.95	4.88	4.87	4.86	4.87

Table 4: Effect of Associativity: Framelet Based Cache, 16KB, 64 byte blocks

Benchmarks	Miss Rates % Block Size			
	32	64	128	256
AMR	8.87	4.64	2.48	1.34
FFT	7.35	3.87	2.03	1.05
PSA	7.71	3.88	1.95	0.98
SDD	8.74	4.63	2.54	1.47
SGA	8.82	5.08	3.25	2.44
SIMPLE	8.75	4.60	2.50	1.45
WEATHER	10.07	6.38	4.24	3.19
Weighted Avg	8.79	4.86	2.83	1.82

Table 5: Effect of Block size: Framelet Based 8-way associative Cache, 16K bytes

The size of the traces both in frame and framlet model are very close as can be seen from the Table 1. This indicates that there are not many replicated tokens sent to the threads belonging to the the same frame. Figure 7 compares the best case performance of the Frame and Framelet model for cache size of 16 KB. The best performance is achieved when the block size is 256 byte blocks for both models. Since associativity, as stated earlier, does not have a large impact on miss rates a 4-way associativity is used for both. Except for PSA, the performance of the Framelet model is better than the Frame model. For SDD, SGA and WEATHER the Framelet model did significantly better than the Frame model. Further analysis of these benchmarks showed that they have a high percentage of large size frames (larger than 256 bytes) in the trace. In these cases it is frequent that the counter would belong to a different cache block than the data that is stored. Hence, each data storage would result in accesses to two different cache blocks. In the Framelet model, since each framelet holds just one thread, the counters and the inputs for the thread belong to the

same cache block more frequently. In fact 99% of the threads needs less than 64 bytes of framelet making the counter access and the input access to the same block.

The cumulative miss rate in Figure 7 show that overall the Frame model has about three times the miss rate of the Framelet model for the benchmarks used. Obviously the scope of these results is limited by the banchmarks used.

5 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 the present work and other related ones. A fair comparison must address several other issues related to the execution model (e.g. blocking or non-blocking threads), the semantics of the source language (strict or non-strict) and the code generation strategy. The discussion will therefore be limited to a simple description of related work.

Culler *et al.* [CSvE93] demonstrates that the performance of the storage hierarchy to an extent 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. An important question in parallel architecture is the problem of organizing the storage hierarchy to operate in concert with the scheduling of computations in parallel programs. One simple approach is TAM's scheduling policy which favors threads within the currently executing quantum.

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 caching mechanism attempts to preserve the working set of the program in the cache. Kavi et al. [KHP+95] use a cache with a frame based storage (called SuperBlocks) in a dataflow execution model. Their mechanism uses a Cold Store bit to identify compulsory misses.

In [TE94] Tekkath and Eggers show that there is very minimal amount of contention misses due to inter-thread communication. Therefore thread coplacement strategies designed enhance inter-thread locality and reduce such misses have minimal effects. From a cache design perspective these results are quite similar to the ones reported here even though the thread execution model is very different.

Among the proposed multithreaded architecture that are being currently developed, the Tera MTA [ACC+90] does not have a cache memory. The M-Machine [FKD+95] does not have a a proper data cache but uses the local memory to cache remote data. This caching mechanism is also supported in the local TLB providing hardware support for the coherence mechanism at the block level (8 words granularity).

Because the processor speeds have been improving at a much faster rate than the memory speeds in the past decade, the design of on-chip caches has become more crucial. The details of various issues in cache designs are treated in [HP90]. A direct-mapped cache has the advantage of a shorter critical path length than a set-associative cache, at the expense of higher conflict misses. There have been several designs which try to combine the advantages of direct-mapped caches with those of set-associative caches such as the victim-cache [Jou90], MRU cache [SR86], hash-rehash cache [AHH88], and half-half cache [THG95]. In general, these schemes split the cache into two parts: one is direct-mapped while the other is set-associative. The idea is to simultaneously send the desired address to both parts of the cache and to *assume* that the direct-mapped portion contains the data. If the assumption turns out to be false, the set-associative portion of the cache provides the data (if available) at a slightly greater cost. If there is sufficient locality, this method could result in a lower average access time than either a pure direct-mapped or a pure set-associative cache of the same size.

6 Conclusion

A non-blocking dynamically scheduled multithreaded execution model provides an efficient mechanism to overlap communication with computation and thus improving the processor utilization. An important issue in this model is the efficiency of the synchronization among threads and the scheduling of threads. This paper reports on the experimental trace-driven

evaluation of two possible storage hierarchy models designed for such an execution.

The results yielded several insights into multithreading operations: First a significant amount of inter-thread locality can be exploited in this dynamically scheduled multithreaded model. Second, even a traditional cache organization can achieve a good performance. Third, by tailoring the cache organization to take advantage of the thread execution model, the performance can be greatly improved.

These results demonstrate that a significant amount of locality can be exploited in a multithreaded architecture.

References

[AAC94] B. S. Ang, Arvind, and D. Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. Technical Report 354, LCS, Massachusetts Institute of Technology, August 1994.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith. The Tera computer system. In *Int. Conf. on Supercomputing*, pages 1–6. ACM Press, 1990.

[AHH88] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Trans. on Computer Systems*, 6:393–431, November 1988.

[CP90] D. E. Culler and G. M. Papadopoulos. The explicit token store. *J. of Parallel and Distributed Computing*, 10(4), 1990.

[CSS+91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, 1991.

[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.

[FKD+95] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S Lee. The m-machine multicomputer. In *Proc. Int. Symp. on Microarchitecture*, November 1995.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[Ian90] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages.* Kluwer, 1990.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Int. Symp. on Computer Architecture*, pages pp. 364–373, May 1990.

[KHP+95] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam. Design of cache memories for multi-threaded dataflow architecture. In *Int. Symp. on Computer Architecture*, pages 253–264, June 1995.

[KSS+95] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Int. Symp. on Computer Architecture*, June 1995.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[RN95] Lucas Roh and Walid Najjar. Design of storage hierarchy in multithreaded architectures. In *Proc. Int. Symp. on Microarchitecture*, November 1995.

[SR86] K. So and R. N. Rechtschaffen. Cache operations by MRU-Change. *Intl. Conf. on Computer Design*, pages pp. 584–586, October 1986.

[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.

[TE94] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proc.* 21^{th} *Int. Symp. on Computer Architecture*, Chicago, Illinois, 1994.

[THG95] K. B. Theobald, H. H. Hum, and G. R. Gao. A design framework for hybrid-access caches. In *Int. Symposium on High-Performance Computer Architecture*, pages pp. 144–153, January 1995.