Experimental Evaluation of Blocking and Non-Blocking Multithreaded Code Execution

Murali Annavaram
Dept. of EECS
University of Michigan
Ann Arbor, MI 48105
annavara@eecs.umich.edu

Walid A. Najjar
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523
najjar@cs.colostate.edu

Lucas Roh
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
roh@mcs.anl.gov

Technical Report CS-97-108

Experimental Evaluation of Blocking and Non-Blocking Multithreaded Code Execution

Murali Annavaram
Dept. of EECS
University of Michigan
Ann Arbor, MI 48105
annavara@eecs.umich.edu

Walid A. Najjar
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523
najjar@cs.colostate.edu

Lucas Roh
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
roh@mcs.anl.gov

Abstract

The objective of multithreaded execution models is masking the latency of inter processor communications and remote memory accesses in large-scale multiprocessors. Several such models combine aspects of dataflow-like execution with the von Neumann model in an attempt to provide both efficient synchronization (as in the dataflow model) and efficient exploitation of program locality (as in the von Neumann model). We refer to these models as *data-driven multithreading models*. One of the factors that distinguishes these models is the thread execution strategy: A thread can be either non-blocking or blocking.

Another factor is the architectural support for dynamic synchronization: The locality present within and among threads can potentially be exploited by a proper storage hierarchy for synchronization store (operand storage). Two storage models have been proposed for data-driven multithreaded execution. One is frame based, in which all the threads belonging to a code-block share one storage segment called frame; the other is framelet based, in which each thread has its own storage segment, called framelet.

This article experimentally compares two thread execution models and their related storage models. The first is a *blocking* execution model that relies on a scheduler for the allocation of threads to processors and exploits *inter thread* locality within a code-block. It relies on the frame storage model and assumes a certain amount of compile time data distribution to minimize network accesses. The second is a *non blocking* execution model in which threads are dynamically scheduled based on data availability. It relies on the framelet storage model and makes no assumptions about the static allocation of data to processors. The experimental evaluation takes into account the impact of the storage hierarchy design on the performance of the two models.

Keywords: Multithreaded execution, compilation, storage models.
Note: A version of this repoprt has been submitted to the *Int. Journal of Parallel Processing*.

1 Introduction

The increasing gap between processor and memory speeds has become a central problem in the design of high-performance computer systems. This gap is due to the increase in clock speeds of CPU chips and the higher demand put on the memory system by superscalar architectures capable of issuing multiple instructions per cycle. In large-scale multiprocessor systems the memory latency problem is compounded by the network latency when data is to be fetched across the network. Solutions to the latency problem attempt either to reduce latency, by using caching and distributed caching, or to tolerate it, by using decoupled architectures and multithreaded execution models.

Multithreaded execution has been proposed as a model particularly for parallel program execution. Multithreading views a program as a collection of concurrently executing threads that are asynchronously scheduled based on the availability of data. Its main advantages are (1) masking the latency of remote memory access and fine-grain synchronization by multiplexing the execution of multiple threads on the same processor, and (2) providing high processor utilization by allocating CPU resources to ready threads.

The multithreaded execution model can be viewed as a bridge between the sequential von Neumann execution model and the dataflow model. In many ways it combines features of both models: the concurrent and asynchronous execution of dataflow, which can provide implicit and efficient synchronization, and the sequential execution of the von Neumann model, which can efficiently exploit intrinsic program locality and, hence, efficient storage hierarchy. In fact, most multithreaded machines (experimental or production) today can be viewed as either derived from the von Neumann model or the dataflow model. In designs closer to the von Neumann model, threads tend to become larger, and data structure locality can be better exploited. Examples of these designs include HEP [37], Tera MTA [2], J-Machine [10], and M-Machine [14]. In designs closer to the dataflow model, latencies are better tolerated, and parallelism is more easily exploited. Examples are Monsoon [32, 33], P-RISC [30], *T [31], EM-4 [36] and the EARTH project [19]. The Hybrid Architecture (IHA), proposed by Iannucci [20] extends the von Neumann model with dataflow features to perform synchronizations. The Threaded Abstract Machine (TAM) [8] is a software implemented multithreaded model that has been ported to a number of platforms, such as the TMC CM-5 and the Cray T3D.

The performance of multithreaded model is determined by the complex interaction of a number of inter related architectural and compilation issues, such as code generation, thread firing rules, synchronization schemes, and thread scheduling. The relation between these issues and the tradeoffs between various alternatives for each of these issues requires extensive experimental evaluation.

A major design issue in multithreading is whether thread execution can be blocked. This issue affects the design of the storage hierarchy (storage model), thread synchronization and firing, and the code generation strategies. In a *blocking* strategy, when a thread initiates a long-latency operation such as a remote memory read, the thread is blocked and the execution is switched a ready thread. This strategy is used, for example, in Iannucci's Hybrid Architecture, the Tera MTA and the EARTH machine. In a *non blocking* strategy, threads are generated so that they cannot block through the use of split-phase operations. This means that a long-latency operation terminates a thread, and the result of a long latency operation, such as remote memory access, is sent to another thread. A context switch does occur when a thread is terminated, but the specific thread state need not be saved and later resumed. This strategy is used, for example, in Monsoon, *T, and the EM-4. In this article, we present a quantitative evaluation of the two design alternatives on the performance and storage design decisions, thereby providing insight into the design of multithreaded systems. The issues analyzed include code generation strategies, implications for data distribution and access, and storage hierarchy performance.

Our results indicate that the non blocking model is very efficient in overlapping execution and communication. The blocking model has smaller numbers of threads of larger granularity. Moreover, the total number of instructions and synchronizations is less in the blocking model

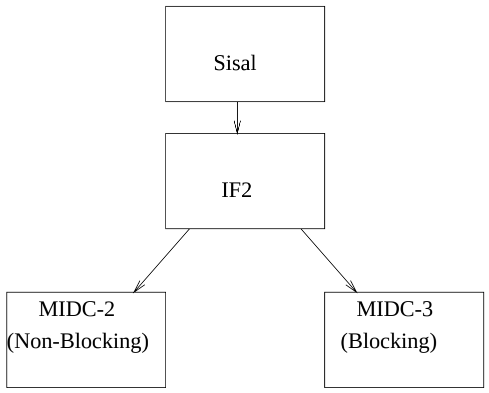


Figure 1: Code Generation Phases

than in the non blocking model. It does, however, exhibit less locality of access to its storage hierarchy and can potentially suffer from a poor distribution of data.

The article is organized as follows: Section 2 presents the two models, their code generation and the architectural models. Section 3 describes the experimental evaluation including the framework used and the results obtained with a perfect cache. Section 4 presents performance results obtained by using a non ideal memory hierarchy, and Section 5 analyzes the impact of the non-ideal memory hierarchy. Section 6 presents related work in this area. Section 7 gives concluding remarks.

2 Execution Models: Architecture and Code Generation

The two multithreaded execution models considered in this article are based on data-driven dynamic execution of threads that are generated at compile time. This section describes the code generation strategy and the execution models.

2.1 Code Generation

The source language used for the multithreaded code generation is Sisal [26], a pure, first order, functional programming language. The functional nature of the language allows the compiler to easily extract parallelism at any granularity and generate code that takes the advantage of multiple threads of execution.

The compilation process converts the Sisal programs into two intermediate forms: MIDC-2 for the non blocking model and MIDC-3 for the blocking one. Both are derived from the Machine Independent Dataflow Code (MIDC) [34]. MIDC is a graph structured intermediate format: The nodes of the graph correspond to the von Neumann sequence of instructions and the edges represent the transfer of data between the nodes. It has been used to generate the executable code for other multithreaded machines (e.g., Monsoon and EM-4). Both MIDC-2 and MIDC-3 are highly optimized versions of MIDC, with optimization done both at the inter and intra thread level. Traditional optimizations such as dead code elimination and copy propagation are performed.

The compiler generating the code is guided by the following objectives: (1) minimize synchronization overhead, (2) maximize intra thread locality, (3) insure deadlock-free threads and, (4) preserve functional and loop parallelism in programs. The compilation phases of MIDC-2 and MIDC-3 code from the Sisal source code are shown in Figure 1.

1. Phase 1: The first phase of the code generation is the same for both the blocking and the non blocking models. This phase involves compiling the Sisal programs to an intermediate format, called IF2, using the OSC [5] compiler. IF2 is a block structured, acyclic data

dependence graph that allows operations that explicitly allocate and manipulate memory in a machine independent way. IF2 also makes certain assumptions about the allocation of data structures. For example, it requires that the elements of a one dimensional array reside in consecutively addressable memory locations.

2. Phase 2: Phase 2 differs for the two models principally in the handling of structure store accesses and the data storage models (called frames or framelets). The structure store access has a long latency if the required data is not in the cache. Long latency operations involving the structure store consist of remote memory reads, memory allocations, function calls, and remote synchronizations. The remote memory references can be handled either as split-phase access or single-phase access. In the split-phase access, the request is sent by one thread and the result is forwarded to another thread. In the single-phase access, the result is returned to the same requesting thread.

    - In the non blocking model the IF2 form is converted into the MIDC-2 form. In this format all the structure store accesses are turned into split phase accesses. A split phase access terminates a thread: the request is sent by a thread, but the result is returned to another thread always. In this model a thread never has to block on a remote memory access. This model makes no assumptions regarding data structure distribution.
    - In the blocking model (MIDC-3) the IF2 graph is statically analyzed at compile time to differentiate between local and remote structure store accesses. A local access does not terminate a thread whereas a remote one does. If the result of a structure store access is used within the same code-block where the access request is generated, the access is considered local. In this case, the thread will block until the request is satisfied. This model relies on a static data distribution to enhance the locality of access. Note that a data structure is often generated in one code block and used in several others, in which case only one of the consumer code-blocks has local access while the other consumers must have a remote memory access to this data structure.

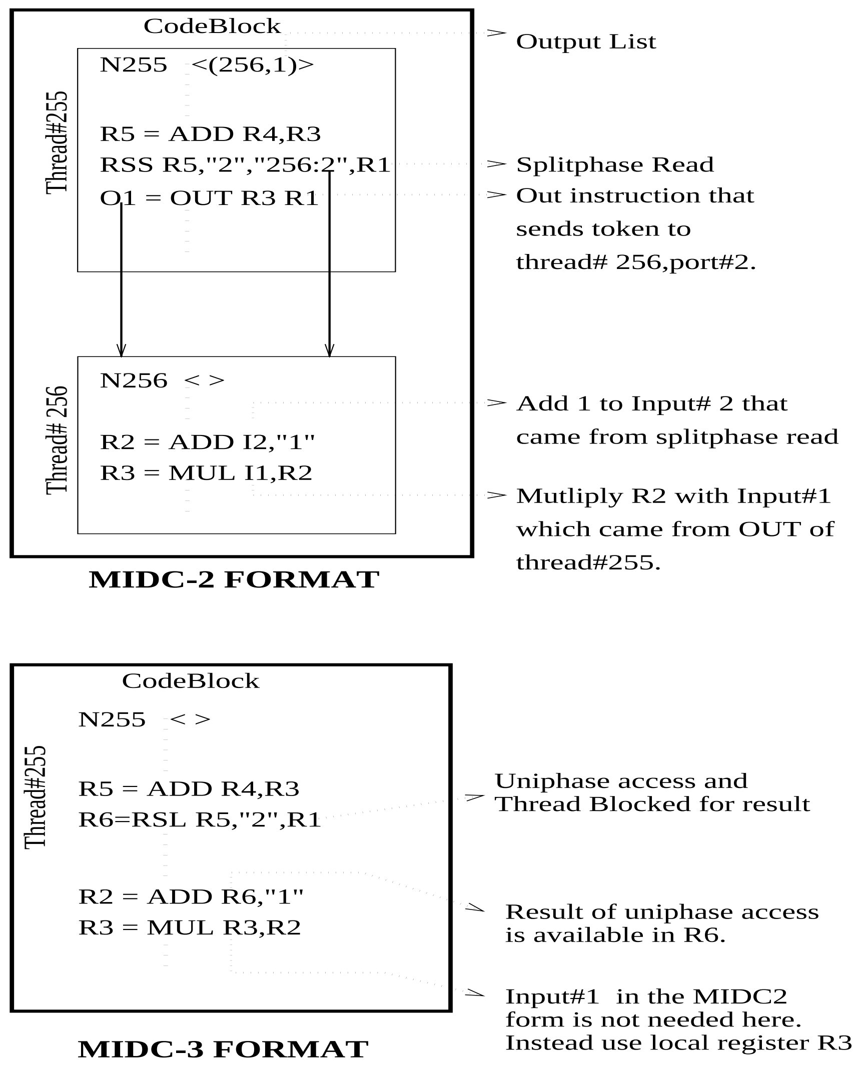
Example The examples in Figure 2 demonstrate the difference between MIDC-2 and MIDC-3. In MIDC-2, $Thread$ 255 performs a structure memory read operation. The read is performed as a split-phase access whereby the result is sent to $Thread$ 256. $Thread$ 255 does not block but continues execution until termination. Results of the split-phase read are forwarded to $Thread$ 256, which starts execution when all its input data is available. There is no restriction on which processor $Thread$ 255 and $Thread$ 256 are executed.

The MIDC-3 code is generated from the same IF2 graph as the MIDC-2. The IF2 graph is statically analyzed. Since $Threads$ 255 and 256 belong to the same code-block[1], the read structure memory operation is a local, single-phase operation, and hence the two threads become a single thread. The thread blocks when the read operation is encountered and waits for the read request to be satisfied.

2.2 Execution Models

Figure 3 shows an abstract model of the processor organization used in our study. The *Execution Unit* executes ready threads, which generate new data values that are forwarded by the *Synchronization Unit* to the destination threads. The *Instruction Memory* stores thread instructions. Depending on the synchronization model used, either the *Framelet Store* or the *Frame 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 context 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.

[1]A code-block is a semantically distinguishable unit of code such as a loop or function body.


CodeBlock
Thread#255
Thread# 256
Output List
Splitphase Read
Out instruction that sends token to thread# 256,port#2.
Add 1 to Input# 2 that came from splitphase read
Mutliply R2 with Input#1 which came from OUT of thread#255.
MIDC-2 FORMAT
CodeBlock
Thread#255
Uniphase access and Thread Blocked for result
Result of uniphase access is available in R6.
Input#1 in the MIDC2 form is not needed here. Instead use local register R3
MIDC-3 FORMAT

```
N255   <(256,1)>

R5 = ADD R4,R3
RSS R5,"2","256:2",R1
O1 = OUT R3 R1

N256  < >

R2 = ADD I2,"1"
R3 = MUL I1,R2
```

```
N255   < >

R5 = ADD R4,R3
R6=RSL R5,"2",R1

R2 = ADD R6,"1"
R3 = MUL R3,R2
```

Figure 2: MIDC-2 and MIDC-3 Code Examples

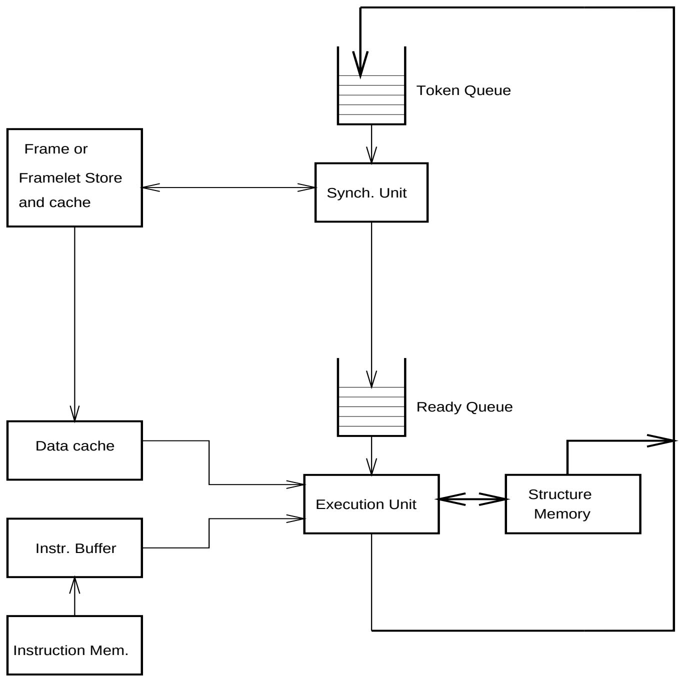


Figure 3: Abstract Model of a Processing Node

The *Structure Memory* represents the memory of the processing node and stores the program data structures. It is a distributed shared memory. The results after accessing the Structure Memory are returned as tokens to the Synchronization Unit. The tokens carry data values and information about the destinations to which these data values should be sent. From this base model, we introduce two variants of thread execution strategies.

Blocking Thread Model: In this model a thread may be suspended by a long latency operation such as a remote memory access or a synchronization. It assumes an architecture support for context switching: the saving of thread state and the selection of a new thread. The context switching overhead depends on the nature and extent of the available architectural support. If the machine has separate sets of hardware registers to support multiple threads, the switching overhead is simply the cost of hardware queuing of the thread (as in the Tera MTA). If there are no multiple register sets, the switching overhead involves the cost of saving and restoring the registers. The storage mechanism relies on the frame model. A frame represents a storage segment associated with each *invocation* of a code-block. All the threads within the code-block instance refer to its associated frame to store and load data values. 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. The frame model is used in several multithreaded machines (e.g., TAM [8], StarT-NG [3], and the EM-4 and EM-X [24]).

When an instance of a particular code-block is invoked, a frame is first allocated in a given processor's frame store. All the tokens generated for that code-block instance are stored in that frame. The virtual address carried by a token is of the form

$$<\textit{frame pointer, frame offset}>$$

A synchronization slot in the frame is associated with each thread. The synchronization slot is initialized with the count of the number of the inputs to the thread and is decremented with the arrival of each input. The thread is ready when the count reaches zero: A continuation corresponding to that thread is placed on the Ready Queue by the Synchronization Unit. A data value that is shared (i.e., read) by several threads in the same frame occupies only one location. The content of the frame is accessed by the Execution Unit via a read-only cache. Data values generated by the executing threads are sent to the Synchronization Unit, which writes them in the frame. The frame is deallocated when all the threads in the code-block have terminated. The cache organization is that of a conventional write-through cache. Because of their variable size, frames are not aligned with cache blocks.

Non Blocking Thread Model: In the non blocking model, once a thread starts its execution it runs until termination. Hence, the thread is activated only when all its inputs are available. All memory accesses are performed as split-phase accesses: the request is issued by the requesting thread, but the result is returned to the destination thread. A thread never has to block waiting for a remote memory access.

The storage mechanism for the non blocking threads is the framelet model. A framelet is a fixed-sized unit of storage that is associated with each thread instance; it includes a synchronization slot for that thread instance. This model has been described in detail in [35]. Simulation results have shown that over 99% of all thread instances can be accommodated with a framelet size of 128 bytes. A chain of framelets, with indirect references, is set up for those threads that have a larger input set.

In the framelet model a data value that is shared among several threads within a same code-block is replicated in the framelet of each thread instance. The framelet is deallocated when the thread instance completes its execution. Because their size is fixed, framelets are aligned with cache blocks. The virtual address of a data value in the framelet model is of the form

<context #, thread #, framelet offset>

The structure of memory organization of these two models is very similar to that of segmentation and paging: One relies on fixed size storage with easy addressing and allocation, while the other uses variable size storage with easier sharing of data.

Example: A code-block consisting of three threads is shown in Figure 4. The corresponding frame memory model is shown in Figure 5. 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 then 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. 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 memory model corresponding to the same code block is shown in Figure 6. There is a separate framelet for each of the three threads(A, B, C). Each framelet contains the counter for the corresponding thread. In addition, 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 process is accomplished as two separate store operations.

Discussion of the Models: The main differences between the blocking and non-blocking models lie in their synchronization and thread switching strategies. The blocking model requires a complex architectural support to switch efficiently between ready threads. The frame space is deallocated only when all the thread instances associated with its code block have terminated execution, a state determined by extensive static program analysis. The model also relies on static analysis to distribute the shared data structures and therefore reduce the overhead of split-phase accesses by making some data structure accesses local. The non-blocking model relies on a simple scheduling mechanism: data-driven data availability. Once a thread executes, its framelet is deallocated, and the space is reclaimed.

The main difference between the frame model and framelet model of synchronization is the data duplication. In the frame model, variables that are *shared* by several threads *within* a code block are allocated only once in the frame; in the framelet model, on the other hand,

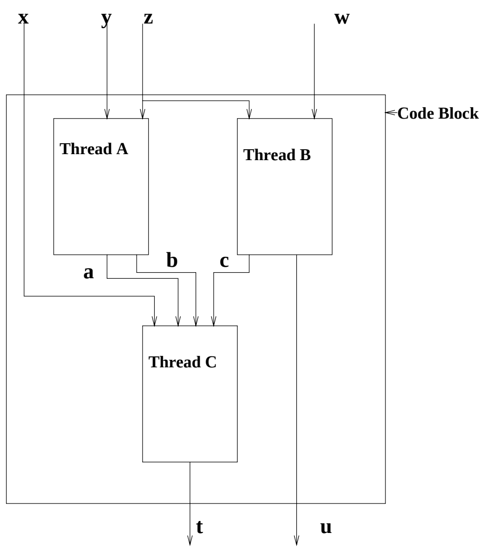


Figure 4: Code Block with Three Threads

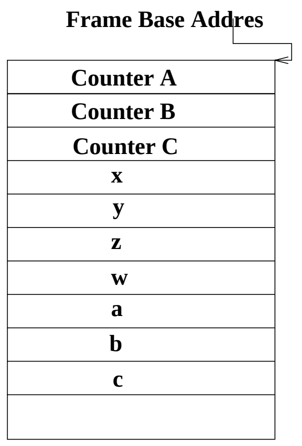


Figure 5: Frame Memory Representation

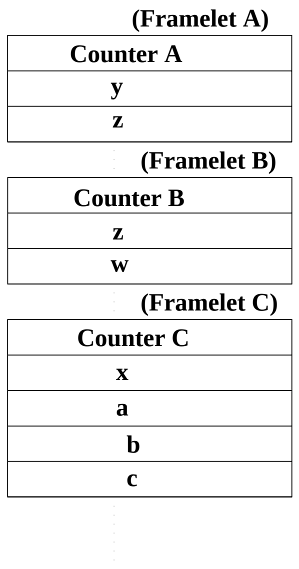


Figure 6: Framelet Memory Representation

these variables must be replicated to all the threads. The advantage of framelet model is that one can design special storage schemes that take advantage of the inter thread and intra thread locality and achieve a cache miss rate close to 1% [35].

3 Experimental Evaluation and Analysis

This section presents the results of executing multithreaded code in the blocking and non blocking models. In this section we assume an ideal storage hierarchy in which the requested data is always available in the cache. The effect of non ideal storage hierarchy under the two models is evaluated separately in Section 4.

3.1 Experimental Framework

Benchmarks: A set of seven Sisal benchmarks are used in these experiments: (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; and (7) ***WEATHER*** is a one level barotropic weather prediction code.

Architectural Parameters: The Execution Unit is a four-way issue super scalar CPU. The network interface of each processor has a bandwidth of two words per cycle. The network latency varies from 0 to 200 CPU cycles. The number of processors varies from one to ten.

The blocking execution model relies on a static data distribution to achieve local access on some data structures. The effects of this static data distribution are modeled using a *probability of success* that determines, for each data structure access, whether it is local or remote.

3.2 Dynamic Execution Statistics

The number of threads, instructions, and synchronizations executed by each benchmark in each model is architecture independent. The information is used to compare the efficiency of the two models. Table 1 shows the percentage change in these three parameters from the non blocking to the blocking model. The reduction in these parameters varies widely for the seven benchmarks. FFT shows a very minimal change. The reason for this behavior is that, although FFT has a large number of structure memory accesses, over 90% of these send their result to a thread outside the code-block boundary. Hence, only a very small fraction of the structure data accesses can be made local, and therefore the code generated for the blocking method is very similar to one for the non blocking model. On the other end of the spectrum, SDD shows the largest amount of change. SDD also has a large number of structure store accesses, almost half of which send the result to a thread within the same code-block.

Since blocking compilation potentially increases the thread size, the results of the average thread size for the blocking and non blocking strategies are shown in the Table 2. The thread size increase is near zero for FFT and as high as 60% for SDD. This behavior is again attributed to the number of memory accesses within a code-block boundary. Since the FFT has fewer accesses within the code-block boundary, the blocking strategy does not create larger threads. Hence, the thread size remains the same. SDD, on the other hand, has a large number of memory accesses within the code-block boundary. Hence, the code generation strategy leads to a larger thread sizes that block on a potential local access.

One of the primary difference between the two models lies in the duplication of data values that are shared among several threads within the same code-block in the non blocking model. This duplication results in increased synchronization. Table 2 shows the dynamic number

Table 1: Change (%) in number of threads, synchronizations and instructions from the non blocking to the blocking model

Benchmarks	Threads	Synch.	Inst.
AMR	−4.78%	−1.53%	−0.20%
FFT	−0.02	−0.11	0.0
PSA	−30.67	−27.39	−4.43
SDD	−41.70	−37.65	−6.38
SGA	−3.15	−5.38	−0.24
SIMPLE	−16.38	−16.27	−2.66
WEATHER	−21.19	−21.00	−5.53

Table 2: Comparison of thread size and synchronization per instruction (SPI) for blocking and non blocking models

Benchmarks	Thread Size			SPI		
	B	**N-B**	%	**B**	**N-B**	%
AMR	32.28	30.79	4.81	0.49	0.50	1.32
FFT	44.66	44.65	0.0	0.42	0.42	0.00
PSA	13.31	9.65	37.84	0.33	0.44	24.02
SDD	18.65	11.61	60.59	0.35	0.53	33.40
SGA	11.80	11.45	3.00	0.51	0.54	5.15
SIMPLE	14.46	12.42	16.41	0.48	0.56	13.98
WEATHER	19.60	16.35	19.87	0.47	0.56	16.38
Wtd Avg	18.20	15.02	26.08	0.43	0.52	17.31

of synchronizations per instruction. On average, the synchronizations per instructions are reduced by 17%.

3.3 Execution Time

The execution parameters described above have a direct effect on the execution time. The graphs in Figures 7 and 8 show the execution time versus the average network latency with five and ten processors, respectively.

The execution time in these plots is the weighted average of the seven benchmarks used. An ideal memory hierarchy for the structure store is used. Hence, local data is always available in the cache. The network latency varies from 0 to 200 cycles. Each graph, has four plots corresponding to the blocking model and one plot corresponding to the non blocking model. The four plots for the blocking model correspond to four probabilities of success of compile time data distribution and scheduling. A 100% success indicates an optimal allocation of data structures and threads in which no remote access is performed. Obviously, this case is unrealistic but is used here for comparison. The 90%, 50% and 10% success probabilities are more realistic situations, corresponding to the average percentage of local structure store accesses. The overhead of thread switching is neglected in computing the execution time for the blocking model.

The performances of the blocking model with 90% or 100% success rate are very close. These two plots outperform the non blocking model with both five and ten processor networks. But the non blocking model performs better than the blocking model when the success rates of data distribution are less and when the network latency is smaller. The effect of added network latency on the non blocking model is more severe (higher slope) than on the blocking one. In the non blocking model all accesses are split-phase, and hence the thread is always switched whenever there is a memory reference. But when the network latencies are large,

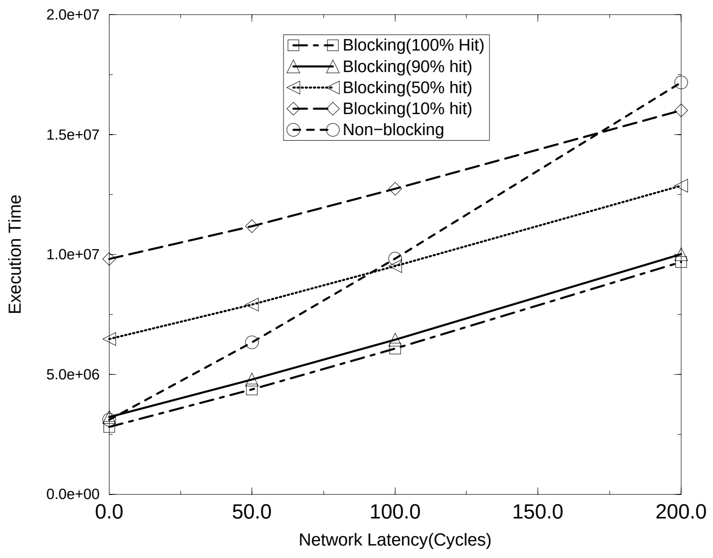


Figure 7: Execution Time versus Average Network Latency (5 processors and ideal memory hierarchy)

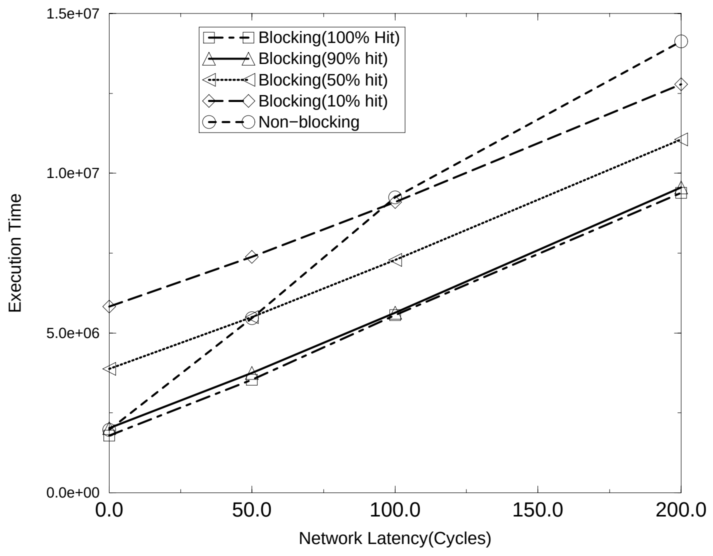


Figure 8: Execution Time versus Average Network Latency (10 processors with ideal memory hierarchy)

there are not enough ready threads to hide the latency. As expected, this effect is made worse when the processor utilization is low. Hence, even a small percentage success of compile time data distribution in blocking model will increase the performance at high network latencies. There, the blocking model does better than the non blocking.

The down side of the blocking model is that its performance is highly dependent on the success rate of the data distribution and the effectiveness of the scheduler to schedule the threads whose data is always local. When the network latencies are small, it is even more important to get a high success rate for the data distribution. Otherwise the thread will be blocked on a remote reference in spite of having some other ready threads.

4 Evaluation and Analysis of Storage Models

The preceding section presented the performance of the two models of execution assuming an ideal storage hierarchy (no cache misses). The performance evaluation results presented in this section are based on realistic cache and memory models. The blocking model of execution uses a frame-based storage, whereas the non blocking model uses a framelet-based one. Both storage models are analyzed by a trace driven cache simulation. The effects of the cache size, set-associativity, and block size are examined for each of the two models.

The traces are generated from the simulation of a ten processor machine[2]. 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% for all cache configurations used in the simulation. Hence, the mean value closely represents the actual miss rate of each processor.

The simulator output is a set of token traces, one per processor. A trace consists of a collection of references to the storage hierarchy. A reference can be either the arrival of a data token (i.e., a data value generated by this or another processor) or an access to a synchronization counter. 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 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 models. The cache sizes range from 4 KB to 256 KB per processor. The framelet-based and the frame-based storage models use 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 the frame and framelet models.

4.1 Trace Size

One of the presumed drawbacks of the framelet model is duplication of the tokens. The framelet stores tokens corresponding to one thread. Hence, multiple threads of a code-block sharing a data value will have multiple copies of this value in the frame model, on the other hand, only one copy of this value is stored. Table 3 shows the percentage of replication in the framelet model: for all benchmarks it is under 3.5%. The explanation is that even though the frame model has no replicated data tokens within a same frame, it still requires one or more accesses to synchronization counters: one for each thread that will read the shared data token. Hence, the trace sizes are nearly equal.

[2]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 the simulator.

Table 3: Trace Sizes

Benchmarks	Trace Size Frame	Trace Size Framelet	Replication(%)
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

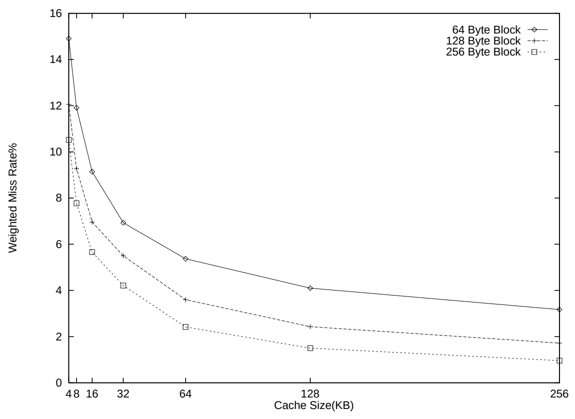


Figure 9: Effect of Cache Size : Frame Based 4-way Associative Cache.

4.2 Frame-based Simulation

Cache Size: Figure 9 shows the effect of varying the cache size from 4 KB to 256 KB per processor for a 4-way set-associative cache with 64, 128, and 256 byte blocks[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 overall miss rates. Since compulsory misses are unaffected by the cache size, the improvement in the miss rates diminishes as the cache size increases. Given the trace size, we chose to restrict our further analysis to a cache size of less than 16 KB. The ratio of cache size to the trace size ranges from 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.

Set-Associativity: Table 4 shows the miss rates for direct-mapped, 2-way, 4-way, 8-way set-associative, and fully associative 16 KB cache per processor with 64-byte blocks. The associativity has no appreciable effect on the miss rates. This result indicates that the percentage of conflict misses is small, which is an indication of the high reference locality in the traces. The high locality means that all the references to a given frame are clustered in a 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 sufficient locality exists 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.

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

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

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 5: Effect of Block size: Frame Based 8-way Associative Cache, 16 KB

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

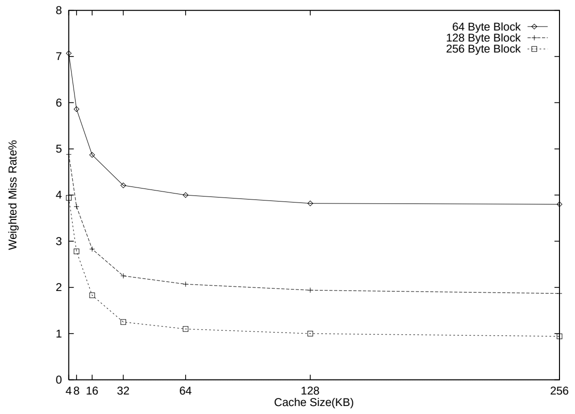


Figure 10: Effect of Cache Size : Framelet Based 4-way Associative Cache.

An 8-way associative 16 KB cache is used with block sizes set to 32, 64, 128, and 256 bytes. At an associativity of 8, the conflict misses are almost zero. Table 5 indicates a noticeable reduction in miss rates with the increase in block size. The reason is as follows. 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. Because of the high locality, the token causing the miss will likely have a virtual address close to the block that is just filled. By increasing the block size, these misses are reduced.

4.3 Framelet-based Simulation

Cache Size: Figure 10 shows the effect of varying the cache size from 4 KB to 256 KB for a 4-way set-associative cache with blocks of 64, 128, and 256 bytes. The incremental improvement in the miss rates starts to decrease after a cache size of just 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. For the traces used here, the ratio of the cache size to the trace size ranges from 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.

Set-Associativity: Table 6 shows the effect of varying the associativity. The results are derived for a 64-byte block and direct-mapped, 2-way, 4-way, 8-way set-associative, and fully associative 16 KB cache. Here, the associativity has negligible effect on the cache miss rates, the same reason given for the frame model.

Block Size: When sufficient locality exists in the address trace, the compulsory misses can be reduced by increasing the block size. Table 7 shows the effect of block size on the miss rates. An 8-way associative 16 KB 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.

4.4 Discussion of the Two Storage Models

Figure 11 compares the best case performance of the frame and framelet models for a cache size of 16 KB. The best performance is achieved when the block size is 256 byte blocks for both models. Since associativity 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 does significantly better

Table 6: Effect of Associativity: Framelet Based Cache, 16 KB, 64 byte blocks

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 7: Effect of Block size: Framelet Based 8-way Associative Cache, 16K bytes

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

than the frame model. Further analysis of these benchmarks showed that they have a high percentage of large frames (larger than 256 bytes) in the trace. In these cases the counter frequently belongs to a different cache block from that in which the data is stored. Hence, each data storage results 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 need less than 64 bytes of framelet, making the counter access and the input access to the same block.

The cumulative miss rate in Figure 11 shows 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 benchmarks used.

5 Effect of Non Ideal Memory Hierarchy

The results presented in Section 3.3 give the weighted execution time of the seven benchmarks with an ideal memory system. The data needed for execution is assumed to be in the cache always. In reality, however, the memory system performance depends on the cache hit ratio and cache miss penalty. This section studies the effect of cache misses on the overall performance. The execution performance with a non ideal memory hierarchy is considered here. The results presented here are obtained by combining the execution time results of Section 3.3 with the miss rates of Section 4. Two parameters are studied: network latency and the effect of the miss penalty on the performance.

Network Latency The plots in Figures 12 and 13 show the execution time versus average network latency with five and ten processors. These plots take into consideration the effect of

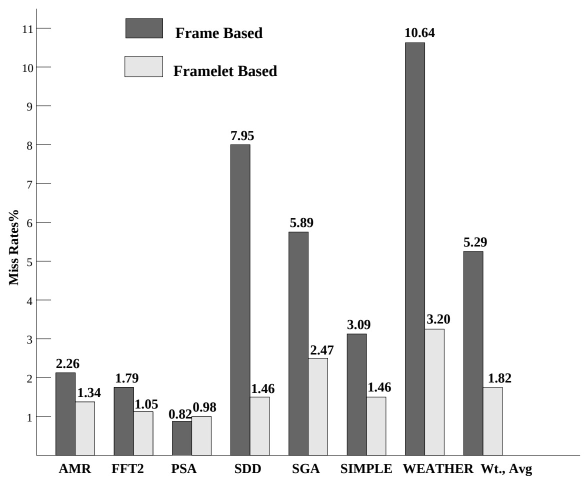


Figure 11: Comparison of Models: Best Performance Parameters, 16 KB, 4-way Associative, 256 byte block Cache

cache misses on the execution time. Comparing these plots with the plots in Figures 7 and 8 we can make the following inferences.

- **The execution time with a non ideal memory system is higher for both the blocking and non blocking models.**
- **The performance of the blocking model degrades more than that of the non blocking model with a non ideal memory system. This is due to the higher miss rates of the frame memory used for the blocking execution model. Since the number of tokens generated both in the blocking and non blocking models is almost the same, and since the miss rate is higher for the blocking model, the memory access time is also higher, and hence the performance degrades.**
- **With a non ideal memory at a network latency of 100 cycles, the performance of the non blocking model is comparable to that of the blocking model with 50% success rate. Note that, with an ideal memory, the blocking model performed better than the non blocking one at 100 cycles network latency. The reason is, again, the higher miss rate of the blocking model.**

Miss Penalty **Figure 14 shows the effect of varying miss penalties on the execution time. Three miss penalties (8, 25, and 50 cycles) are used in this figure. A network latency of 50 cycles is assumed here. The non blocking model masks higher miss penalties effectively compared with the blocking model. The slopes of all four plots corresponding to the blocking model are the same. The slope is higher than that of the non blocking model. Since the miss rates are higher for the blocking model, the miss penalties affect the execution time more severely than the non blocking model.**

6 Related Work

Culler et al. [9] 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

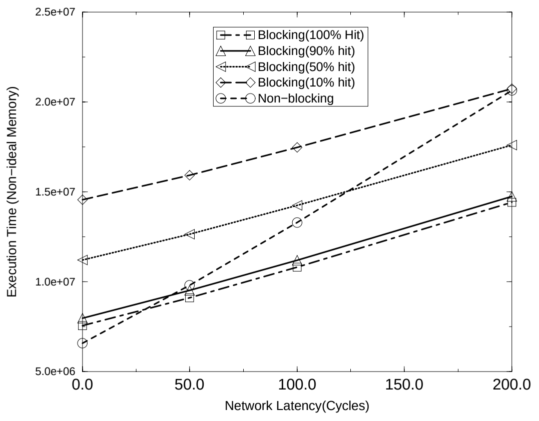


Figure 12: Execution Time versus Average Network Latency(5 processors ad 8 cycle miss penalty)

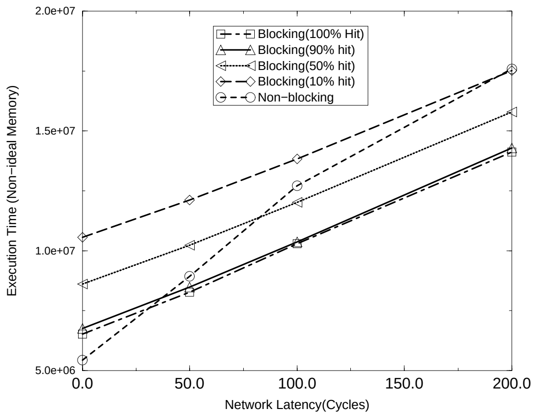


Figure 13: Execution Time versus Average Network Latency(10 processors and 8 cycle miss penalty)

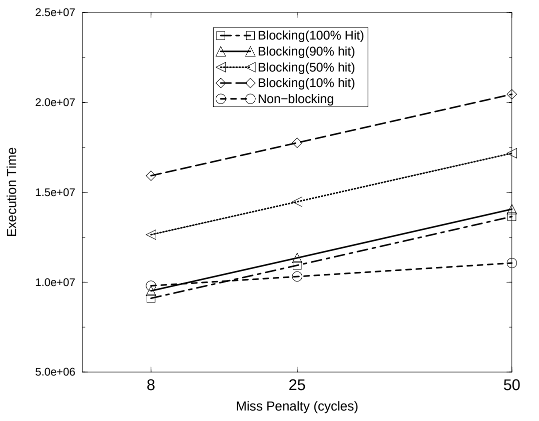


Figure 14: Execution Time versus Miss Penalty (5 processors, 50 cycle network latency)

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. The blocking model of execution presented in this article schedules the threads belonging to a code-block together. The concept of quantum is thus similar to the code-block based scheduling.

In Monsoon, thread scheduling cannot be controlled by the programmer. The token queues are completely hardware managed; software is not permitted to read or write the token queues. Hardware uses a pure FIFO scheme to schedule threads. A bad sequence of tokens could significantly increase run-time or resource usage of the entire program.

*T provides the *sched* instruction for a hardware supported scheduling of threads. It is also possible to explicitly read the continuation stack, opening the possibility to explicitly control thread scheduling. This flexibility of scheduling can be used by the compiler for achieving better performance.

The M-machine uses scoreboarding for scheduling the instructions from H-threads. An instruction is not issued till the register used by this instruction has a valid value. This type of scheduling is efficient but needs a large amount of hardware support. Scoreboarding is a sequential bottleneck that can reduce the performance.

Research on storage hierarchy design for multithreaded architectures is ongoing. Most of the research efforts are on incorporating caches into multithreaded executions and measuring their effectiveness. Cache designs in a dataflow model is discussed in [39] for DFM-II [40]. 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 [7]. The model is evaluated by using a relatively small set of hand-coded benchmarks. The caching mechanism attempts to preserve the working set of the program in the cache. Compulsory misses form most of the misses in the caches for multithreaded architectures. Hence, the techniques to reduce the compulsory miss penalty give significant performance benefits. Kavi et al. [22] 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 and avoid bringing in the cache block. In [35] it is shown that by using certain simple hardware support, the cache performance can be increased by an order of magnitude. Some blocks of the cache are reserved for satisfying the compulsory misses. This scheme, called reserve block scheme, reduces the miss penalty on compulsory misses. In [41] Tekkath and Eggers show that a minimal amount of contention misses occur due to inter thread communication. Therefore thread co-placement strategies designed to enhance inter-thread locality and reduce cache misses have minimal effects.

Among the proposed multithreaded architectures that are being developed, the Tera MTA [2] does not have a cache memory. The M-Machine [14] does not have 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 word granularity). The *T-NG [3] uses coherent caches for global shared memory. It also retains the message passing ability of the *T machine. The coherency is implemented by using a directory based coherency scheme.

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 [18]. Several designs have tried to combine the advantages of direct-mapped caches with those of set-associative caches, these designs include the victim-cache [21], MRU cache [38], hash-rehash cache [1], and half-half cache [42]. 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. With sufficient locality, this method can result in a lower average access time than either a pure direct-mapped or a pure set-associative cache of the same size.

While the thread execution models discussed in this paper rely on hardware support, several projects have developed software run-time systems that support lightweight thread execution on either a symmetric multiprocessors (SMPs) or distributed memory multiprocessors. An extensive discussion of these approaches is beyond the scope of this section. In the absence of architecture support, lightweight threads that support efficient inter-thread communication provide an efficient and attractive platform for parallel processing. Further information can be found in [15, 17, 12, 16, 11, 23, 27, 29, 13, 6, 25, 28, 4].

7 Conclusions

This article compares the performance of two data-driven multithreaded execution models and their associated synchronization schemes. The blocking model relies on frame-based data storage. A frame contains all the data related to a code block. All the thread instances that belong to that code block share the same frame. In the blocking model it is assumed that a static data distribution has been done at compile time, and therefore some of the structure accesses are assumed to be local. Local accesses are done with a single phase access. All other data structure accesses are done in split phase. Threads are split only on split-phase accesses but not on uni-phase access. This approach should lead to fewer of large granularity threads. In the non blocking model the data belonging to each thread instance is stored in a separate framelet. Access to data structures in the non blocking model is always done with a split-phase operation: the request is sent by one thread, and the results are received by another.

The evaluation of the program execution characteristics of these two models shows that the blocking model has a significant reduction in threads, instructions, and synchronization operations executed with respect to the non blocking model. It also has a larger average thread size and a lower synchronization per instruction. But the lower synchronization per instruction does not reduce the load on the synchronization processor. The traffic in terms of the tokens sent to the synchronization processor remains the same in both the blocking and the non blocking models, as is clear from the fact that the trace sizes obtained for both models are nearly equal. Although the number of data tokens is less in the frame model, there are a significant number of synchronization tokens (decrementing counters) that negate the advantage of fewer data tokens. Hence, synchronization overhead is the same for the frame and framelet models of synchronization.

The execution time of the blocking model is highly dependent on the success rate of the static data distribution. At a 100% or 90% success rate, the execution times are comparable and outperform those of the non blocking model. For a success rate of 50%, however, the execution time may be higher than that of the non blocking model: the performance depends largely on the processor utilization and the average network latency. When the network latency is low and the processor utilization high, the non blocking model performs as well as the blocking model with a 100% or 90% success rate.

A non blocking, dynamically scheduled, multithreaded execution model provides an efficient mechanism to overlap communication with computation, thus improving the processor utilization. An important issue in this model is the *efficiency* of the synchronization. Since the number of synchronizations required for the blocking and non blocking models is similar, the performance depends on the efficiency of accessing the structure store in the two models. Hence, an evaluation of two possible storage hierarchy models is carried out: one, associated with the blocking model is called the frame model; the other, associated with the non blocking model, is called the framelet model. The cache miss rates indicate that the framelet model of synchronization outperforms the frame model of synchronization. Hence, the execution time of the non blocking model is better than the blocking model under a non ideal memory hierarchy.

The results have provided several insights into multithreading operations. With the lack of a proper compile-time, data-distribution strategy, the blocking model of thread execution is at a disadvantage. A simple non blocking model requiring minimal hardware support can outperform the complex blocking model. The storage model performance indicates that a significant amount of inter thread locality can be exploited in this dynamically scheduled multithreaded model. A traditional cache organization is sufficient to exploit this locality. The previous work indicates that by tailoring the cache organization to take advantage of the thread execution model, the performance can be greatly improved.

References

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

[2] 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.

[3] B. S. Ang, Arvind, and D. Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. Technical Report 354, LCS, MIT, August 1994.

[4] G.D. Benson and R.A. Olson. A portable run-time for the SR concurrent programming language. In *Proc. Workshop on Run-Time Systems for Parallel Programming*, 1997.

[5] D. C. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proc. of 13th ACM Symp. on Operating Systems*, pages 152–164, October 1991.

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

[8] 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.

[9] 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.

[10] W. J. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[11] D.L. Eager and J. Zahorjan. Chores: Enhanced run-time support for shared memory parallel computing. *ACM Trans. on Computer Systems*, 11(1):1–32, February 1993.

[12] D.R. Engler, G.R. Andrews, and D.K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical Report TR 93-13, Dept. of Computer Science, University of Arizona, April 1993.

[13] E. Felton and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Scalable High-Performance Computing Conf.*, pages 84–89, April 1992.

[14] 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.

[15] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *J. of Parallel and Distributed Computing*, (37):70–82, 1996.

[16] V.W. Freeh, D.K. Lowenthal, and G.R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. Technical Report TR 94-11, Dept. of Computer Science, University of Arizona, 1993.

[17] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Supercomputing*, pages 350–359, Washington, D.C., November 1994.

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

[19] H. Hum, O. Macquelin, K. Theobald, X. Tian, G. Gao, P. Cupryk, N. Elmassri, L. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu. A design study of the EARTH multiprocessor. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, 1995.

[20] R. A. Iannucci. Toward A Dataflow/Von Neumann Hybrid Architecture. In *Proc. 15thInt. Symp. on Computer Architecture*, pages 131–140, 1988.

[21] 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.

[22] 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.

[23] D. Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[24] 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*, pages 14–23, June 1995.

[25] J. Kramer, J. Magee, M. Sloman, N. Duley, S.C. Cheung, S. Crane, and K. Twindle. An introduction to distributed programming in REX. In *Proc. of ESPRIT-91*, pages 207–222, Brussels, November 1991.

[26] 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.

[27] F. Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, CA, January 1993.

[28] Frank Mueller. Distributed shared memory threads: DSM-Threads. In *Proc. Workshop on Run-Time Systems for Parallel Programming*, 1997.

[29] B. Mukherjee, G. Eisenhauer, and K. Ghosh. A machine independent interface for lightweight threads. Technical Report CIT-CC-93/53, College of Computing, Georgia Institutre of Technology, 1993.

[30] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proc. 16thInt. Symp. on Computer Architecture*, pages 262–272, 1989. Also: CSG Memo 292, MIT Laboratory for Computer Science 545 Technology Square, Cambridge, MA 02139, USA.

[31] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proc. 19thInt. Symp. on Computer Architecture*, pages 156–167, May 1992.

[32] G. M. Papadopoulos. Implementation of a General-Purpose Dataflow Multiprocessor. Technical Report TR-432, MIT Laboratory for Computer Science, August 1988.

[33] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proc. 17thInt. Symp. on Computer Architecture*, pages 82–91, June 1990.

[34] 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*, Montreal, Canada, 1994.

[35] Lucas Roh and Walid Najjar. Design of storage hierarchy in multithreaded architectures. In *Proc. Int. Symp. on Microarchitecture*, pages 271–278, November 1995.

[36] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data-flow single chip processor. In *Proc. 16thInt. Symp. on Computer Architecture*, pages 46–53, May 1989.

[37] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE (Real Time Signal Processing)*, 298:241–248, 1981.

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

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

[40] 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.

[41] R. Thekkath and S. J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proc.* $21th$*Int. Symp. on Computer Architecture*, pages 176–186, Chicago, Illinois, 1994.

[42] 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.