Data-Driven Multithreading Using Conventional Microprocessors

Article *in* IEEE Transactions on Parallel and Distributed Systems · November 2006

DOI: 10.1109/TPDS.2006.136 · Source: IEEE Xplore

3 authors:

Costas Kyriacou
Frederick University
34 PUBLICATIONS **232** CITATIONS
SEE PROFILE

Paraskevas Evripidou
University of Cyprus
109 PUBLICATIONS **942** CITATIONS
SEE PROFILE

Pedro Trancoso
University of Cyprus
107 PUBLICATIONS **1,025** CITATIONS
SEE PROFILE

Data-Driven Multithreading Using Conventional Microprocessors

Costas Kyriacou, Paraskevas Evripidou, *Member*, *IEEE*, and Pedro Trancoso, *Member*, *IEEE*

Abstract—This paper describes the Data-Driven Multithreading (DDM) model and how it may be implemented using off-the-shelf microprocessors. Data-Driven Multithreading is a nonblocking multithreading execution model that tolerates internode latency by scheduling threads for execution based on data availability. Scheduling based on data availability can be used to exploit cache management policies that reduce significantly cache misses. Such policies include firing a thread for execution only if its data is already placed in the cache. We call this cache management policy the CacheFlow policy. The core of the DDM implementation presented is a memory mapped hardware module that is attached directly to the processor's bus. This module is responsible for thread scheduling and is known as the Thread Synchronization Unit (TSU). The evaluation of DDM was performed using simulation of the Data-Driven Network of Workstations ($\mathrm{D^2NOW}$). $\mathrm{D^2NOW}$ is a DDM implementation built out of regular workstations augmented with the TSU. The simulation was performed for nine scientific applications, seven of which belong to the SPLASH-2 suite. The results show that DDM can tolerate well both the communication and synchronization latency. Overall, for 16 and 32-node $\mathrm{D^2NOW}$ machines the speedup observed was 14.4 and 26.0, respectively.

Index Terms—Dataflow, multithreading, nonblocking threads, cache prefetching, multiprocessors, network of workstations, high performance computing.

1 Introduction

Data-driven Multithreading (DDM) is a nonblocking multithreading model based on the Decoupled Data Driven model of execution [1], [2]. This model decouples the synchronization from the computation portions of a program allowing them to execute asynchronously. In this model a thread is scheduled for execution in a data-driven manner, i.e., after all of its required data have been produced. As a consequence, no synchronization or communication latencies are experienced after a thread begins execution. The performance improvement is achieved at the expense of extra off-chip hardware. This hardware is responsible for scheduling the threads for execution based on data availability.

This paper presents the model, design, and performance evaluation for DDM. The main benefit of the proposed implementation is that it is built using conventional microprocessors. The DDM specific hardware is implemented as an off-chip memory-mapped module attached to the processor's bus. Therefore, the DDM implementation may easily evolve by upgrading and exploiting the latest technologies present in the state-of-the-art microprocessors. This contrasts with other nonblocking multithreaded systems that require either special design processors [3], [4] or modified existing processors [5], [6]. Alternative, software-only approaches such as TAM [7] suffer from performance degradation, in comparison to our proposed solution.

While Data-Driven scheduling leads to irregular memory access patterns that have a negative impact on cache effectiveness, it can be used for short-term cache management. This can be achieved by ensuring that the required data is preloaded in the cache, before a thread is fired for execution. Furthermore, DDM scheduling information can be used to ensure that data preloaded in the cache is not evicted before the corresponding thread is executed. We call this cache management policy the *CacheFlow* policy.

As a case study for DDM, an implementation of DDM is presented. This architecture is named *Data-Driven Network of Workstations* ($\mathrm{D^2NOW}$). In this architecture, the computation is carried out by off-the-shelf Intel Pentium microprocessors and the support for the data-driven synchronization of threads is provided by the *Thread Synchronization Unit* (TSU). The TSU is a memory-mapped hardware module attached directly on the processor's bus. The $\mathrm{D^2NOW}$ implementation includes also an efficient communication system composed of both a dedicated fine-grain interconnection network for fine and medium-grain communication as well as an Ethernet network to support coarse-grain block data transfers.

To evaluate $\mathrm{D^2NOW}$, we have conducted experiments for machine configurations from 1 to 32 nodes. The workload used on these experiments consists of nine scientific applications, seven of which belong to the SPLASH-2 suite [8]. The other two applications represent standard algorithms used in large scientific applications such as the block matrix multiply and the trapezoidal method of integration.

- *C. Kyriacou is with the Computer Science and Engineering Department, Frederick Institute of Technology, 7 Yianni Frederickou Str., Pallouriotissa, PO Box 24729, 1303 Nicosia, Cyprus. E-mail: cskyriac@cs.ucy.ac.cy.*
- *P. Evripidou and P. Trancoso are with the Department of Computer Science, University of Cyprus, 75 Kallipoleos Ave., PO Box 20537, 1678 Nicosia, Cyprus. E-mail: {skevos, pedro}@cs.ucy.ac.cy.*

Manuscript received 27 Dec. 2004; revised 22 July 2005; accepted 4 Oct. 2005; published online 24 Aug. 2006.
Recommended for acceptance by A. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0318-1204.

Results with fine-grain threads have shown an average speedup of 12.8 on a 32-node machine, compared to the baseline sequential execution on a single node. Increasing the thread granularity by a factor of 8 resulted in an average speedup of 16.8. When increasing the cycle time of the interconnection network by a factor of five, the average speedup reduction observed among all applications is only 13.4 percent. Hence, the DDM model can tolerate well the communication latency. Finally, the experimental results show that the use of the *CacheFlow* policies significantly reduce the cache miss ratio. As a consequence, the speedup is improved by 32 percent. The average speedup for all applications on a 32-node machine with all *CacheFlow* optimizations is 26.0.

Overall, DDM proves to be a promising architecture as it effectively hides synchronization and communication latency, independently of the network technology that is used. In addition, the CacheFlow policy reduces significantly the cache miss ratio, resulting in speedups for certain applications that are close to linear.

2 Data-Driven Multithreading

Data-Driven Multithreading is a nonblocking multithreading model of execution that provides effective latency tolerance by allowing the computation processor to produce useful work, while a long latency event is in progress. This model of execution has been evolved from the dataflow model of computation. In particular, it originates from the dynamic dataflow Decoupled Data Driven (D^3) graphs [1], [2], where the synchronization part of a program is separated from the computation part. The computation part represents the actual instructions of the program executed by the computation processor. The synchronization part contains information about data dependencies among threads and is used for thread scheduling.

In the DDM model, scheduling of threads is determined at runtime based on data availability, i.e., a thread is scheduled for execution only if all of its input data is available in the local memory. As with all dataflow models, its major benefit is the fact that it exploits implicit parallelism. Effectively, scheduling based on data availability can tolerate synchronization and communication latencies.

2.1 DDM Model

A program in DDM is a collection of re-entrant code blocks (see Fig. 2). A code block is equivalent to a function or a loop body in the high-level program text. Each code block comprises of several threads. A thread is a sequence of instructions equivalent to a basic block. Code blocks are essential for the management of threads by the TSU. Only some of the code blocks of a program are resident in the TSU at a time. When a code block becomes resident, the synchronization structures of its threads are loaded from the main memory into the TSU. This limits the size of memory required for the implementation of the TSU.

A producer/consumer relationship exists among the threads. In a typical program, threads called producers, create data for other threads, called consumers. Scheduling of code blocks, as well as scheduling of threads within a code block is done dynamically at runtime according to data availability. The instructions within a thread are fetched by the CPU sequentially in control-flow order. Nevertheless, the CPU can reorder the sequence of instructions internally to exploit the advantages of out-of-order execution.

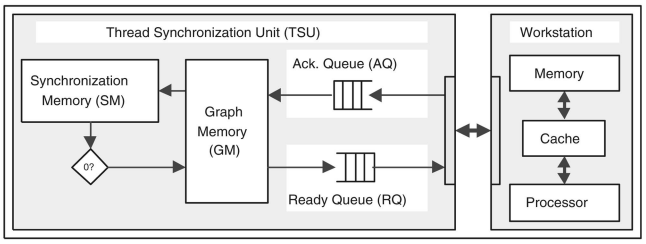

Fig. 1. DDM processing node.

At compile time a program is partitioned into a data driven synchronization graph and code threads. Each node of the graph represents one thread associated with its synchronization template. Each thread is identified by the thread number (*Thread#*) consisting of the triplet (*Context, Block, and ThreadID*). The *Context* field is set at runtime to distinguish between multiple invocations of the same code block or thread. This is useful for the implementation of multiple invocations of functions. The *Block* field identifies the code block, while the *ThreadID* identifies the thread within the code block. The synchronization template of each thread contains the following information:

- **Instruction Frame Pointer (IFP)**: This is a pointer to the address of the first instruction of the thread.
- **Ready Count**: This is a synchronization value set by the compiler, showing the number of input values or producers to the thread, still needed to be produced, before the thread is ready for execution. This count is decremented at runtime, whenever an input value is produced. A thread is ready for execution when its Ready Count reaches zero.
- **Data Frame Pointer (DFP)**: This is a pointer to the data frame assigned for the thread/code block. It is used to enable the reuse of code blocks with different data values. This field is set to zero during compilation, and is initialized by the runtime system, when the code block is loaded in the TSU.
- **Consumer threads (Consumer1 and Consumer2)**: These are the consumer threads. They are used to determine which Ready Count values to decrement after the thread completes its execution. If a thread has only one consumer then Consumer2 is set to zero. If a thread has more than two consumers, then Consumer1 is set to zero and Consumer2 is a pointer to the consumer's list, a memory block that contains the list of consumers. Using two consumers in the template allows the runtime system to process the consumers faster. The use of only two consumers is justified by the fact that code analysis has shown that the majority of DDM threads have either one or two consumers.

A DDM processing node is depicted in Fig. 1. It consists of an off-the-shelf workstation augmented with the TSU.

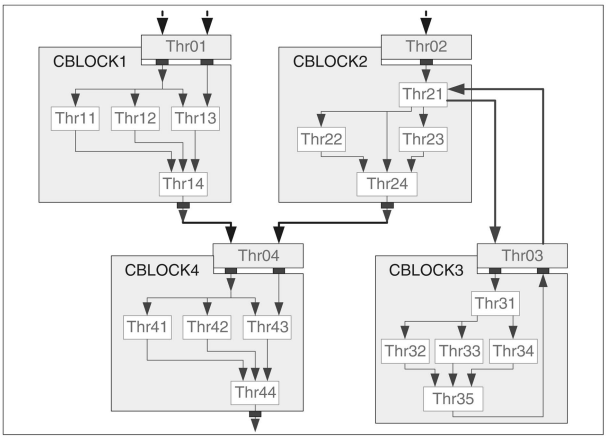

Fig. 2. Code block scheduling.

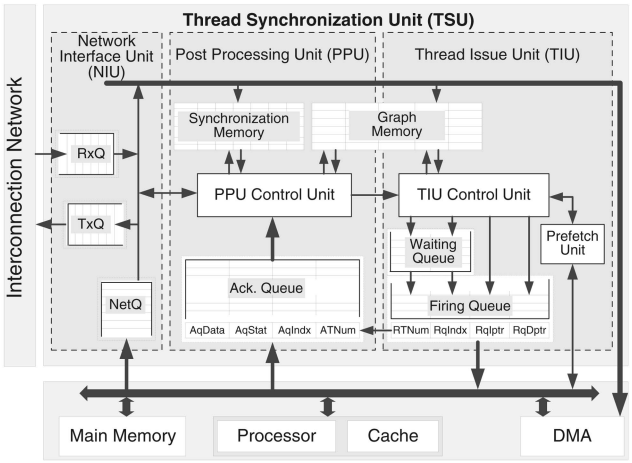

Fig. 3. Thread synchronization unit.

The computation processor communicates with the TSU via two queues: the *Ready Queue (RQ)* and the *Acknowledgement Queue (AQ)*. These queues are memory-mapped, i.e., there is no need for modifying the processor or adding extra instructions. The RQ contains pointers (IFP, DFP, and identification) to the threads that are ready for execution. The AQ contains identification information and status of executed threads. The main storage units in the TSU are the *Graph Memory (GM)* and the *Synchronization Memory (SM)*. The GM contains the IFP, DFP, and the consumers of each thread. The SM contains the Ready Count values for each thread.

The processor reads the address of the next thread to be executed from the RQ. After the processor completes the execution of a thread, it stores in the AQ the identification number and status of the completed thread and then reads the address of the next thread to be executed from the RQ. The control unit of the TSU fetches the completed threads from the AQ, reads their consumers from the GM and then updates the Ready Count of the corresponding consumer threads in the SM. If any of these consumers is ready for execution, i.e., its Ready Count reaches zero, it is placed in the RQ and waits for its turn to be executed.

2.2 Scheduling of Code Blocks

Scheduling of code blocks is done dynamically by the TSU at run time, according to data dependencies among code blocks. Each code block is associated with a synchronization value that shows how many data elements are still needed before the code block becomes ready. A consumer/producer relation exists among code blocks depending on their data dependencies. Each code block is associated with a special thread called the *Inlet Thread*. The inlet threads are used to provide synchronization among code blocks. The *Ready Count* of an inlet thread is equal to the number of inputs of the code block. An inlet thread is ready for execution when its *Ready Count* in the SM becomes zero. The inlet threads are responsible for loading the templates of the code block in the GM, initializing the SM according to the *Ready Count* of each thread, and updating the DFP field in the GM. Each code block is also associated with a special thread called the *Outlet Thread* that removes the code block from the TSU and release the memory reserved for the code block.

An example of scheduling of code blocks is depicted in Fig. 2. Thread *Thr01* is the inlet thread for code block *CBLOCK1*. The number of inputs to *Thr01* is the same as the number of inputs of code block *CBLOCK1*. When *Thr01* is fired, it loads on the TSU the code block of *CBLOCK1*. As shown in the example in Fig. 2, thread *Thr21* invokes code block *CBLOCK3* by enabling thread *Thr03* which is the inlet thread of *CBLOCK3*. In this case *CBLOCK3* returns a value back to the caller code block *CBLOCK2*. Thread *Thr03* must provide a data slot to send the return value back to thread *Thr21*.

2.3 TSU: Hardware Support for DDM

The purpose of the Thread Synchronization Unit (TSU) is to provide hardware support for data-driven thread synchronization on conventional microprocessors. The TSU is made out of three units: the Thread Issue Unit (TIU), the Post Processing Unit (PPU), and the Network Interface Unit (NIU). The PPU updates the *Ready Count* of the consumers of the completed threads, and determines whether they are ready for execution. The PPU forwards ready threads to the TIU. The function of the TIU is to schedule threads deemed executable by the PPU. The NIU is responsible for the communication between the TSU and the interconnection network. The TSU is memory mapped. Its internal structure and interface to the motherboard is shown in Fig. 3.

The TIU provides the computation processor with all information related to the thread scheduled to be executed next. The RQ is divided into two different queues: the Waiting Queue (*WQ*) and the Firing Queue (*FQ*). The thread identification number (Thread#) and iteration number of ready threads are first placed in the WQ. The *Thread#* is used to determine the thread's IFP and the DFP from the GM. The tuple (RTNum, RqIndx, RqIptr, RqDptr) is then shifted in the FQ, where *RTNum* is the *Thread#*, *RqIndx* is the iteration number of the thread, *RqIptr* is the address of the first instruction of the thread, and *RqDptr* is the address of the memory frame allocated for the data of the thread.

The computation processor passes information about the completed threads to the PPU through the AQ. This information is the thread number (*Thread#*), the iteration number (*AqIndx*), and the status of the completed thread

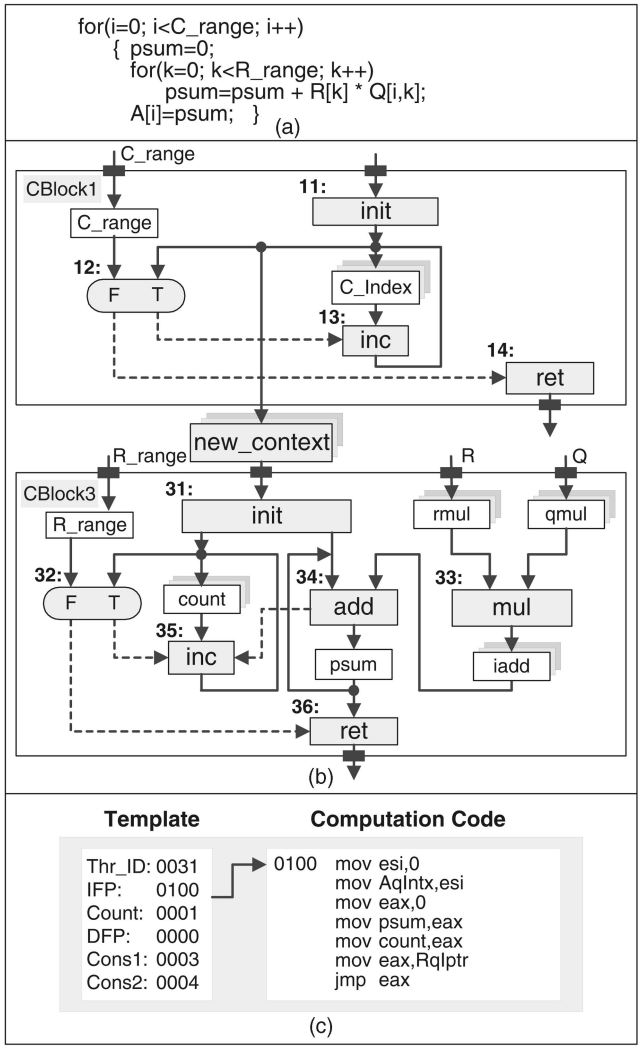

Fig. 4. DDM vector-matrix multiplication example.

(*AqStat*). An extra field is provided by the AQ, the *AqData* that is used to pass the data produced by threads that send single values to other nodes. The Thread# is used as an index to the GM to determine the Thread# of the consumers of the completed thread. The SM is indexed with the Thread# number of the consumer threads. If a consumer thread becomes ready, it is shifted to the WQ of the TIU. If a ready thread belongs to a remote node, it is forwarded to the NIU for further processing.

2.4 Example

The DDM execution model can be described with a simple example such as the vector-matrix multiplication. Fig. 4a depicts the code for the implementation of the vector-matrix multiplication example where the inner loop computes the vector inner product in DDM. Nested loops can be handled in two ways. In the first approach, nested loops are transformed by the compiler into a single level loop by changing loop indexes and counters accordingly. In the second approach, a nested loop is assigned to two code blocks. The first code block represents the outer loop and the second the inner.

Fig. 4b depicts the DDM-graph for the vector-matrix multiplication using the second approach. The outer loop is implemented with code block *CBLOCK1*, while the inner loop is implemented with code block *CBLOCK3*. Shaded boxes represent threads. Nonshaded boxes represent instances of values. Solid arrows represent data dependencies as well as the movement of data between threads, while dotted arrows represent only data dependencies among threads.

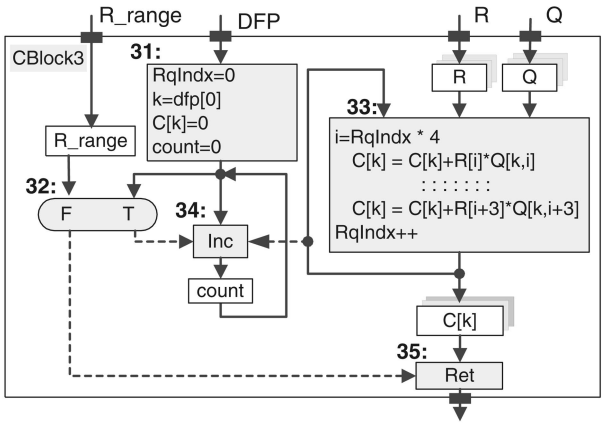

Fig. 5. Increasing thread granularity.

The function of *CBLOCK1* is to create the new column indexes *C_Index* and fire thread *new_context* that acts as the inlet thread for code block *CBLOCK3*. The inputs to *CBLOCK1* are the value for *C_range* that specifies the number of columns in the matrix, and a control input used to trigger the execution of the vector-matrix multiplication code. Thread *Thr11* initializes *C_Index* to zero. Thread *Thr13* increments the value of *C_Index*. Thread *Thr12* is a switch thread. Its function is to compare the value of *C_Index* with *C_Range* and trigger either thread *Thr13* or thread *Thr14* according to its predicate. Thread *Thr14* is a return thread. Its function is to release the memory allocated for the code block and if necessary inform the caller code block that it has completed its execution, or trigger the execution of other code blocks.

The graph at the bottom of Fig. 4b represents code block *CBLOCK3*. Its inputs are the number of rows *R_range* and the two vectors *R* and *Q*, where *Q* represents the current column of the matrix. Fig. 4c shows the code of thread *Thr31*. The first number in the template is the thread number (i.e., 0031). The second number is the starting address of the code of the thread (i.e., 0100). The third number indicates the number of inputs (arcs in the graph) to the thread. The next field is the DFP that is initially set to zero. The last two numbers are the consumer threads (i.e., 0003 and 0004). The last two instructions read from the AQ the address of the next thread scheduled for execution and then branch to that address, thus branching to the next thread.

In the example shown in Fig. 4, we assigned a simple arithmetic operation to each thread in order to show the implementation of DDM. In real applications a thread is assigned many more computation tasks. Fig. 5 shows how a computation thread can be assigned multiple instructions. In that example, *Thr33* consists of a loop that performs four multiplications for the inner product computation.

3 CacheFlow

One of the main goals of DDM is to tolerate latency by allowing the computation processor to produce useful work while a long latency event is in progress. Simulation results have shown that DDM can indeed tolerate communication and synchronization latency. In spite of these results, the scheduling based on data availability may have a negative effect on locality. Nevertheless, it can be exploited to implement efficient cache management policies that reduce cache misses. We call these policies the *CacheFlow* policies [9].

To implement the CacheFlow policies we split the RQ into two queues: the *Waiting Queue (WQ)* and the *Firing Queue (FQ)*. Two extra fields are also added in the GM that point to the offset addresses needed by each thread. Ready threads are first placed in the WQ. The TSU determines the offset addresses of the threads stored in the WQ and prefetches the corresponding data before shifting the thread into the FQ.

The *basic* CacheFlow implementation employs hardware prefetching to prefetch into the L2 cache the data needed by the threads scheduled for execution in the near future. All data needed by a thread resides in its data frame pointed by the DFP. Two extra fields are added in the thread's template pointing to the data needed by the thread. These fields are loaded into the Graph Memory at runtime as Data Offset Pointer 1 (DOP1) and Data Offset Pointer 2 (DOP2). If a thread has only one input, then DOP1 points to that value, while DOP2 is set to 0. If a thread has more than two inputs, then DOP1 is set to zero while DOP2 is a pointer to the DOP list, a memory block within the TSU that contains a list of DOPs.

One disadvantage of the *basic* CacheFlow policy is that it might increase the traffic on the processor's bus and snooping lines, since the prefetching unit must snoop the DOPs of all threads shifted from the WQ to the FQ. Another potential problem is that prefetching can cause cache conflicts, i.e., it is possible that before a thread executes, one of its prefetched cache block is replaced by another block, prefetched for a different thread. We call these conflicts *false cache conflicts* as they originate from the policy and not from the execution of the code. The possibility of false cache conflicts is reduced by keeping the size of the FQ as small as possible.

The first optimization, called the *false conflict avoidance*, prevents the prefetcher from replacing cache blocks required by the threads waiting in the FQ. This is achieved with the use of a table, called the Reserved Address Table (RAT), that contains the addresses of all cache blocks prefetched for the threads waiting in the FQ, as well as the thread currently running. The TIU with the *false conflict avoidance* CacheFlow support is depicted in Fig. 6. All addresses required by a ready thread, removed from the WQ, are determined using the information from the Graph Memory, and placed in the Tag Queue. These addresses are then compared with the contents of the RAT to determine if prefetching would cause a false cache conflict. A thread is shifted into the FQ if none of its addresses would result in a cache conflict. If it is detected that an address would result in a false cache conflict, then the tested thread is placed temporarily in a buffer, and the next thread from the WQ is tested. This is essential to avoid blocking. Threads waiting in the temporary buffer have precedence over the threads in the WQ. This is essential to avoid thread starvation, as a thread waiting in the temporary buffer is blocking its consumers from executing. The operations on the RAT table are done in parallel with the other operations on the TIU, PPU, and CPU, thus the latency due to the RAT operation can be hidden.

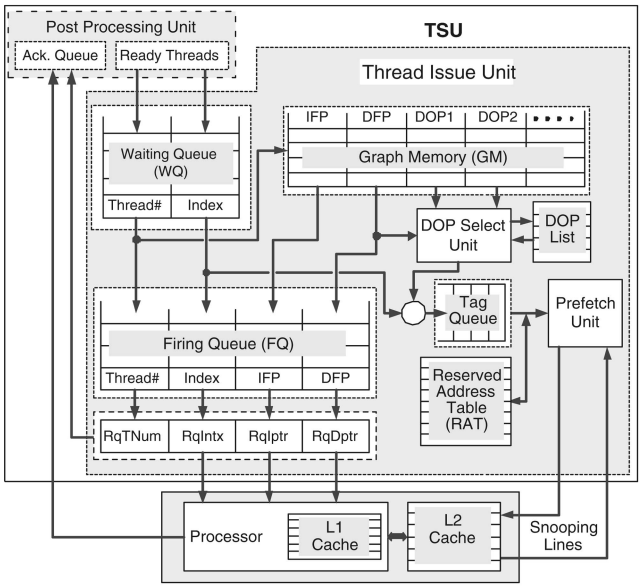

Fig. 6. TIU with *false conflict avoidance* CacheFlow support.

The implementation of the conflict avoidance optimization requires the knowledge of the cache size and associativity by the TSU. Modern processors, such as the Intel Pentium, provide information about the characteristics of the internal cache through processor specific registers. At boot time, the TSU can read these registers and configure the conflict avoidance circuitry accordingly.

The second optimization, called the *thread reordering*, attempts to exploit locality by reordering the threads in the WQ. To exploit temporal code locality the reordering mechanism reorders the threads in the WQ so that threads with the same identification number (Thread#) are placed near each other in the WQ. This reordering allows the code of a thread to be used many times before it is replaced from the cache. Furthermore, threads with the same Thread# are ordered according to their index (iteration number), thus exploiting spatial data locality. Reordering reduces further snooping overheads on the processor and the bus. The thread reordering mechanism operates in parallel and asynchronously with the rest of the TIU and, thus, it does not add any extra delays in the datapath of the TIU.

4 Data-Driven Network of Workstations (D^2NOW)

D^2NOW is a parallel machine based on the data-driven multithreading model of execution. D^2NOW is built using commodity workstations (Intel Pentium based PCs), connected through a fine-grain interconnection network. The workstations are augmented with a TSU card. An important design issue is where to connect the TSU on the existing

motherboard of each workstation. In most experimental architectures the extra hardware is plugged to the PCI slot, the graphics bus, or the DIMM/SIMM attachment. In D^{2}NOW the TSU should be connected to the motherboard in such a way to have direct access to the processor, as it requires to snoop the processor's memory requests. While the original implementation of the TSU was plugged to the COAST socket [10], it is possible to plug it to one of the two processor's slots of a dual processor motherboard.

The communication system affects the ability of D^{2}NOW in tolerating communication latency. Important factors in selecting the type of the communication system are the overheads introduced, the communication latency, the quality of the interconnection, and the simplicity of its implementation. Communication can be classified according to the amount of data that needs to be transferred per packet or communication session. In this sense, communication can be either fine-grain (one data value per message), medium-grain (up to few tens of values per message) or coarse-grain (hundreds to thousands of values per message). Data-driven multithreading employs data forwarding, i.e., a producer node forwards the data produced to its consumers as soon it is produced. Thus, the communication system must be able to process single-value workloads efficiently, with minimal CPU overheads due to system calls. To reduce as much as possible the involvement of the CPU in the communication, the interconnection network must be connected directly to the TSU.

D^{2}NOW employs three mechanisms for the implementation of the communication system. The first one relies on a fine-grain interconnection network, and communication mechanisms that handle only fine-grain communication (single-value messages). With this mechanism, the consumer's identification (node and thread number) as well as the data are passed to the Network Interface Unit (NIU) through the Post Processing Unit (PPU) (see Fig. 3). In the second mechanism, which is used to transfer medium size messages, the computation processor stores in a buffer, within the NIU, the consumer's identification as well as the starting address and size of the memory block. In the background, an embedded processor together with a DMA engine process the communication though the fine-grain interconnection network. The third mechanism utilizes an Ethernet network to support coarse-grain communication. In this case, large block data transfers are directed to the Ethernet network while fine-grain and medium-grain communications are handled by the fine-grain communication system through the fine-grain interconnection network. The use of the Ethernet network does not incur in any increase in cost as this network is already incorporated in D^{2}NOW in order to enable the system initialization and debugging.

5 Experimental Setup

5.1 Simulation Environment

In order to evaluate D^{2}NOW, we have built an execution driven simulator based on native execution. The simulator runs directly on the target processor. The execution of the application threads is interleaved with the simulation of the TSU and the interconnection network.

TABLE 1
Size and Latency of TSU Structures Used by the Simulator

TSU Structure	Capacity	Size	Latency
Acknowledgment Queue	64 Threads X 4 values	1 Kbyte	2 cycles
Synchronization Memory	64 Threads X 64 iterations	2 Kbytes	6 cycles
Graph Memory	64 Threads X 8 values	2 Kbytes	2 cycles
Waiting Queue	16 Threads X 2 values	128 bytes	2 cycles
Firing Queue	16 Threads X 5 values	320 bytes	2 cycles
Reserve Address Table	32 Threads X 1 value	128 bytes	4 cycles
NIU Queues	28 Threads X 18 bytes	504 bytes	2 cycles

All simulations were carried out on an 800MHz Intel Pentium III workstation with 256-MByte RAM. The processor has a 256-KByte L2 unified cache, a 16-KByte L1 data cache and a 16-KByte L1 instruction cache. All caches are 4-way set associative with a 32-byte line size.

The simulator identifies three types of entities: the processing units, the TSU units (PPU, TIU, and NIU), and the interconnection network. All timings produced by the simulator are in terms of CPU cycles. Each entity is assigned a cycle counter that reflects the present time of the entity. The simulator uses these cycle counters to determine which unit to simulate next. Priority is given to the unit with the smaller number of cycles. This ensures that the application's threads are executed in the correct sequence as on the real system. Each TSU unit is also assigned a task queue. This is a queue that contains pointers to the threads that are ready to be processed by the corresponding unit. Parallel processing systems are simulated by maintaining a set of cycle counters for each of the units (TSU and processor) of each node. The time needed to execute each thread is determined using the processor's time stamp performance counter. Measurements of the cache miss rates are obtained by reading the processor's performance monitoring counters [11].

The functional units of the TSU have been coded in VHDL and downloaded to an FPGA chip for functional verification and timing analysis. The simulator uses accurate latencies for the TSU operations that were counted from the FPGA implementation. The size and latency of the structures of the TSU, used by the simulator is shown in Table 1.

The fact that the host and target systems are the same does not allow us to reconfigure resources such as the processor microarchitecture and the cache. This is not a limitation since the D^{2}NOW simulator aims to simulate a parallel machine built with off-the-shelf workstations augmented to support data-driven multithreading. Native execution of the target system also implies that care should be taken to save and restore the state of the target processor/machine whenever the host switches between machine simulation and program execution. To ensure that the state of the cache of the target processor is not affected by the operation of the simulator, a buffer that contains the last 1K data memory references generated by the target processor is maintained. Before branching to the execution of a thread, the simulator fetches the data from the memory

locations stored in this buffer, thus the contents of the cache will be close to the contents of the cache on the real system. A similar approach is used for the code of the simulated program, i.e., the starting addresses of the last threads executed are read from the memory, so that the code of these threads is placed in the cache.

5.2 Application Suite

Nine scientific applications are used to evaluate the performance of D^{2}NOW. Seven of these applications, LU, FFT, Radix, Barnes, FFM, Cholesky, and Radiosity belong to the SPLASH-2 suite [8]. The other two applications, Mult and Trapez represent standard algorithms used in large scientific applications such as the block matrix multiplication and the trapezoidal method of integration, respectively.

5.3 Evaluation Methodology

In this work, all applications used are coded in C and compiled using the Microsoft Visual C++ V6 compiler. To port an application to the D^{2}NOW simulator, we first partition the code into DDM threads. Thread partitioning is currently done manually, by inserting extra code at the end of each thread that transfers control from the thread execution to the simulator. The application code is also annotated with extra instructions needed to produce the application's thread templates.

6 Experimental Results

6.1 Thread Granularity

Thread granularity refers to the amount of computation or number of instructions executed within a thread. Thread granularity affects the performance of a data-driven multithreaded architecture due to the overheads required for thread sequencing. At least two instructions are used to enable the data-driven thread sequencing: one to read the address of the next thread from the TIU and another to branch to that instruction. Furthermore, in many cases, it is necessary to include more instructions in a thread that are used to read the iteration number or data frame pointer of the thread from the TIU, or pass the status of the completed thread to the PPU. By increasing the thread granularity, the effect of the data-driven sequencing overheads is amortized, thus improving performance. Increasing thread granularity could also prove beneficial for cache performance since it improves spatial locality. Furthermore, by adding more loop iterations in a thread, less threads are executed, thus fewer branches and less pipeline flushes are encountered. Loop unrolling techniques can be employed to improve the pipeline performance of the processor. Nevertheless, increasing the thread granularity may have a negative effect on exploiting parallelism.

To examine the effect of thread granularity on performance, we tested different thread granularities. For this work, a thread granularity factor of 1 corresponds to a thread that performs the computation for a single point or a single iteration of a loop, while a thread with granularity factor N performs N iterations of a loop. To isolate the effect of thread granularity, we do not employ the CacheFlow or communication assist optimizations for these experiments, i.e., only fine-grain communication is used.

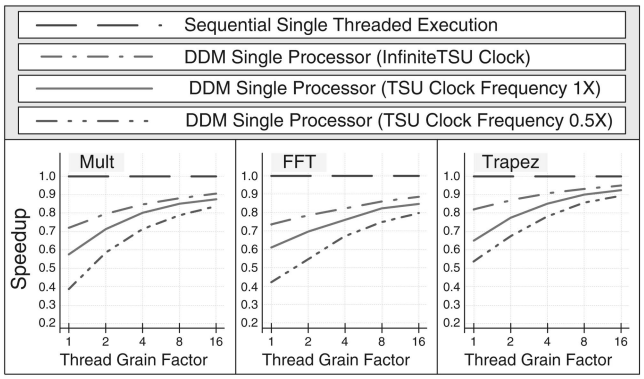

Fig. 7. Single processor comparison.

We have conducted experiments to compare the execution time of DDM threads on single processor with the execution time of the sequential code on a single processor. Furthermore, to investigate the effect of the speed of the TSU, we have conducted experiments with different TSU clock frequencies ranging from infinite speed to half of the normal TSU clock frequency. The normal TSU frequency is the same as the CPU frequency. Fig. 7 depicts the speedup for single node configurations using single threaded sequential, and DDM execution for varying thread granularity and TSU clock frequency. The distance between the single threaded execution line and the DDM execution with normal TSU clock frequency line (*TSU Clock Frequency 1x*) represents the performance losses due to the DDM overheads, TSU latency, and loss of locality. With thread granularity factor of 8 and above, the DDM speedup is close to the sequential, especially for Trapez. The distance between the normal TSU clock frequency and the *"Infinite TSU Clock"* lines indicates the loss in performance due to the TSU latency. This distance is reduced when the granularity increases, since by increasing thread granularity increases the computation time of threads, allowing more time to the TSU to carry out synchronization operations. The *"TSU Clock Frequency 0.5x"* indicates the loss in performance due to the increase in the TSU latency when the speed of the TSU is reduced to half of the normal. This reflects the possible mismatch in the TSU speed and the speed of future processors. The distance between the line for DDM execution with normal TSU clock frequency (*TSU Clock Frequency 1x*) and the line for DDM execution with half the TSU clock frequency (*TSU Clock Frequency 0.5x*) is reduced when the granularity increases, and approaches zero when thread granularity factor is 16.

Fig. 8 shows the speedup obtained on 8, 16, and 32 node machine configurations, for threads with grain factor ranging from 1 to 16. All applications show a significant improvement when increasing thread granularity. Out of all applications, Trapez has the best speedup characteristics. This is due to the fact that for this application the communication-to-computation ratio is very low. By increasing the thread granularity factor from 1 to 8, for Trapez on a 32-node configuration, the speedup increases from 19.3 to 25.5 (32 percent increase). The lowest speedup increase is observed for Radix. For this application, a significant improvement is observed when increasing the thread granularity factor from 1 to 2, while it

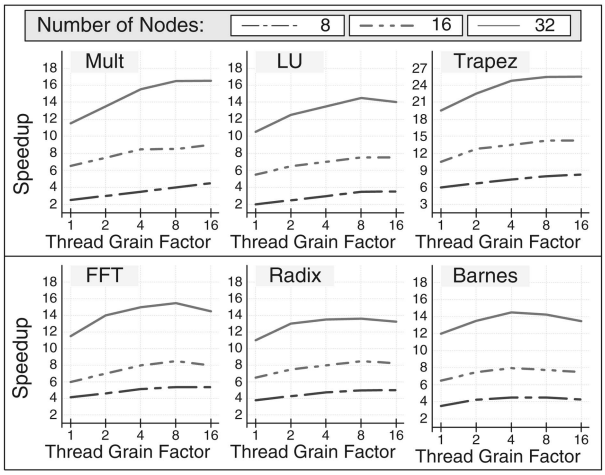

Fig. 8. Effect of thread granularity on performance.

does not show any improvement when further increasing the thread granularity factor. For most applications, the peak value is obtained when the thread granularity factor is eight. Increasing the thread granularity factor beyond eight results in a decrease in performance. The largest decrease is observed for the FFT application. Overall, the average speedup on a 32-node machine configuration, for all applications increases from 12.8 (granularity factor 1) to 16.8 (granularity factor 8), which corresponds to a 27 percent increase. Therefore, throughout the rest of this work, the thread granularity factor is set to 8. Notice that the overall speedup obtained is relatively low, because these results were obtained employing only fine-grain communication mechanisms.

6.2 Communication Latency

To examine the ability of DDM in tolerating communication latency, we have conducted experiments with varying interconnection network latency cycles (period of the clock of the interconnection network), ranging from ideal (0x) to five times the baseline (5x). The speedup, compared to single processor execution, with varying communication latency for three of the applications is depicted in Fig. 9. The rest of the applications show a similar behavior. These results were obtained without using the Ethernet network. Consequently, large block data transfers were transferred through the TSU with the medium-grain communication mechanism, and the fine-grain interconnection network. The effect of the interconnection network latency on the speedup is represented by the distance between the curves for each communication latency. The closer these curves are to the ideal communication latency (0x), the smaller the impact on the speedup.

The speedup decrease is lower for Trapez (3.3 percent average among all machine configurations), since the communication needed by this application is low. The speedup variation is higher for FFT (average 20.7 percent with peak 22.8 percent), due to its high volume, bursty communication characteristics, and its strict data dependencies. Moderate results are obtained for Radix (average 18.3 percent). The remaining applications show a variation that ranges from 10 percent to 20 percent. Overall, the decrease observed for each application is low, average 13.4 percent among all applications, compared to the

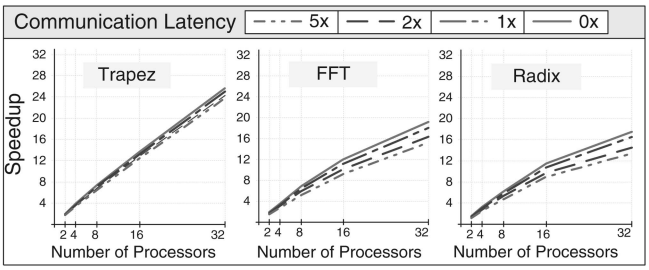

Fig. 9. Speedup for varying communication latency cycles: ideal (0x), baseline (1x), double (2x), and five times (5x).

increase in the communication latency by a factor of 5. Consequently, it is possible to conclude that the communication latency can indeed be tolerated by DDM and the block data transfer mechanism. In addition, the results show that DDM is able to tolerate communication latency independently of the network technology as the results are not sensitive to the changes in the communication latency.

Regarding the communication facility, we have performed experiments with three different communication configurations. In the first, all communication was handled by the fine-grain interconnection network, with only fine-grain communication support by the TSU. In the second, block data transfer is handled by the fine-grain network and the TSU that was modified to support medium and coarse-grain communication. In the third, fine and medium-grain communication was handled by the interconnection network and the TSU, while coarse-grain communication was handled by an Ethernet network. In average, for all applications, results have shown a 17 percent improvement in the speedup when using the second communication configuration compared to using the first one, (20.4 average speedup on a 32-node machine). If Ethernet is added for handling coarse-grain communication, the speedup is further increased to 21.6. For optimal communication, single value packets are transferred through the TSU fine-grain communication mechanism, multivalue packets (with less than 384 bytes) are transferred through the TSU medium-grain communication mechanism, while the Ethernet is used to transfer messages with more than 384 bytes [12].

6.3 CacheFlow

In order to evaluate the ability of the proposed cache management policy in reducing cache misses, we have conducted simulations for systems with 2, 4, 8, 16, and 32 processors, using the basic prefetch as well as the two proposed optimizations. To examine the effect of the problem size on the effectiveness of the CacheFlow management we have used, for most applications, two problem sizes: Data Size 1 (corresponding to 64K integer data matrices) and 2 (corresponding to 1M integer data matrices). Next, we present the analysis of different parameters that affect the performance.

6.3.1 Effect of Firing Queue Size on Performance

Prefetching must be completed early enough to ensure that data is prefetched before the thread using that data is fired for execution. Nevertheless, prefetching must not be initiated too early, to avoid replacing cache blocks already prefetched by threads waiting in the Firing Queue. The

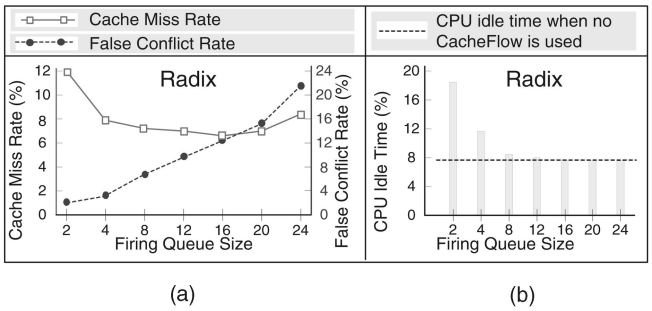

Fig. 10. FQ size effect on cache miss rate, false cache conflicts, and CPU idle time.

effect of the size of the Firing Queue (FQ) for *Radix*, on the cache miss rate and the false conflicts when the *basic prefetch CacheFlow* is employed, is depicted in Fig. 10a. For these results, a thread is shifted into the FQ as soon as the prefetching operation is initiated. As expected, the cache miss rate is higher when the FQ size is small. This is due to the fact that a thread might be fired before the prefetching is completed, resulting in cache misses. As the size of the FQ increases, there is more time for the prefetch unit to complete the prefetching operations, since the processor will execute other threads. Increasing the size of the FQ, increases also the number of false cache conflicts, resulting in more cache misses. The rest of the applications behave in a similar way. For all applications, the minimum cache miss ratio is obtained when the FQ size is 16. Increasing the size of the FQ to more than 16, results in a gradual increase of the cache miss ratio.

To reduce the cache miss rate when the FQ is small, we have changed the FQ shifting policy. A thread is shifted into the FQ after its data prefetching is completed. This change affects also the CPU idle time. The effect of the size of the Firing Queue (FQ) for *Radix*, on the CPU idle time when the *basic prefetch CacheFlow* is employed, is shown in Fig. 10b. Measurements of the CPU idle time show that there is a significant increase in the CPU idle time when the FQ is too small. For all applications, the CPU idle time when the FQ size is over 16 is the same as the idle time obtained without the CacheFlow.

6.3.2 Effect of DDM and CacheFlow on Cache Miss Rate

Fig. 11 shows the L2 cache miss rate for the sequential single threaded execution on a single processor and the different DDM configurations with CacheFlow on a 32-node system. Note that in order to avoid misleading results, for the measurement of the cache miss rate, we have scaled down the data size of the sequential single threaded execution to match the data size for each of the nodes in the 32-node DDM system, i.e., the data sizes used for the sequential single thread execution are the same as those used by each node in the data-driven multithreaded execution. As expected, the baseline DDM configuration shows a higher miss rate than the sequential (increase from 6.9 percent to 9.8 percent), which corresponds to a 42 percent increase for the average of all applications. This reflects the loss of locality for both the code and data. The Basic Prefetch CacheFlow implementation reduces the miss rate from 9.8 percent to 3.1 percent (68 percent decrease compared to

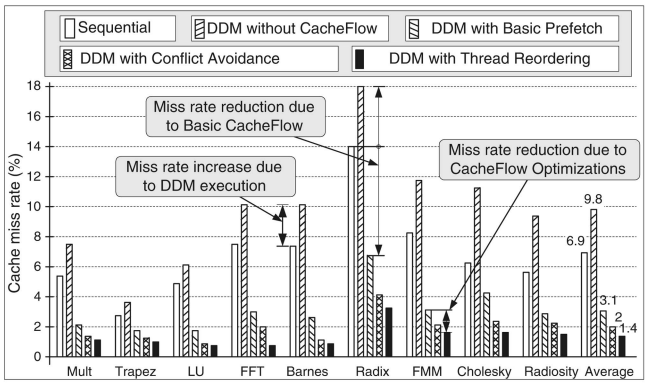

Fig. 11. Cache miss rate for small data sizes.

the baseline DDM). It is important to notice that the reduction achieved by the Basic Prefetch CacheFlow results in miss rate values lower than the original sequential execution. The use of the two CacheFlow optimizations results in further reductions on the miss rate, which becomes 2.0 percent and 1.4 percent, respectively.

6.3.3 Effect of Problem Size on Cache Miss Rate

The effect of the problem size on the cache miss rate is depicted in Fig. 12. By increasing the problem size by a factor of 16, the average cache miss rate for the sequential execution is increased from 7.9 percent to 9.7 percent (23 percent increase). This increase is justified by the fact that the working set for the large problem size does not fit in the cache, resulting in more cache misses. The cache miss rate increase for the DDM execution without the CacheFlow management, is increased from 9.7 percent to 12.0 percent (24 percent increase). The increase of the miss rate is significantly reduced when the different CacheFlow optimizations are used, (7 percent, 5 percent, and 8 percent miss rate increase, respectively). This shows that CacheFlow is efficient in keeping the miss rate low independently of the problem size.

6.3.4 Effect of CacheFlow on Speedup

Fig. 13 shows the speedup for the three CacheFlow implementations, compared to sequential execution, for machine sizes ranging from 2 to 32 processors. The effect of

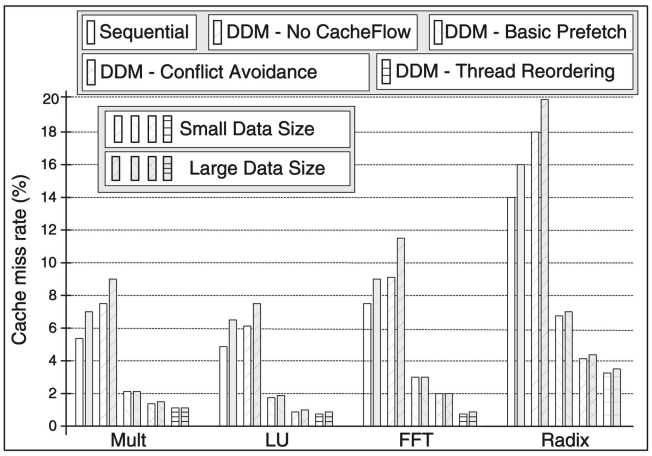

Fig. 12. Effect of data size on cache miss rate.

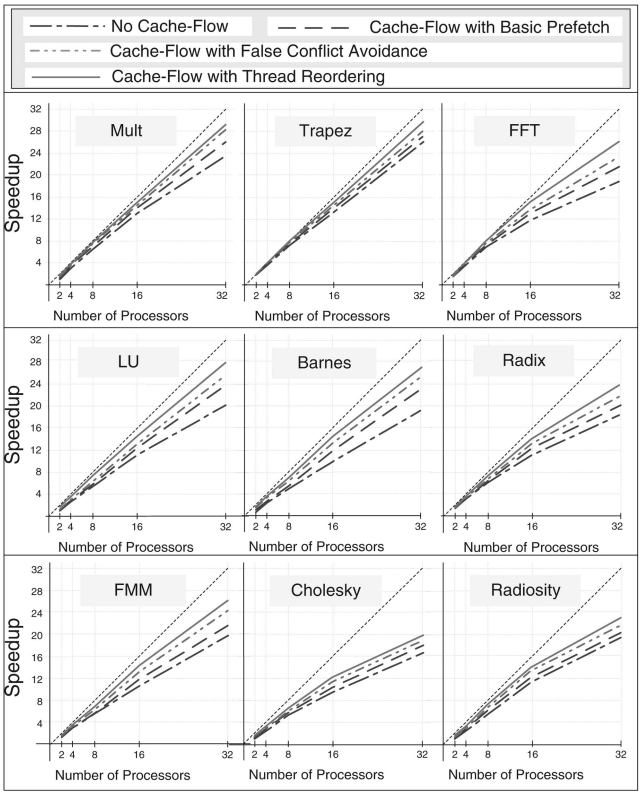

Fig. 13. Effect of CacheFlow on speedup.

the CacheFlow policy on the speedup is shown by the distance of the speedup lines for each application. The smallest speedup improvement is obtained for Trapez, Cholesky, and Radiosity applications. The biggest speedup improvement is obtained for LU and Barnes applications. The speedup improvement is consistent with the reduction in the cache miss rate. For example, Trapez has a 72 percent reduction in the miss rate and an 18 percent increase in speedup, while LU has a 92 percent reduction in the miss rate and a 38 percent increase in speedup.

Overall, the average speedup obtained for all applications on a 32-node machine with both of the CacheFlow optimizations is 26.0. The overall speedup does not reach the ideal linear speedup due to data migration, parallelization overheads, as well as strict data dependencies.

6.4 Comparison with Other Systems

Table 2 compares the speedup achieved by DDM and CacheFlow on a 16-node machine with the results reported by other researchers using the similar configurations and data sizes for the applications compared. The first system is a PRAM model [8], the second is an SGI-Origin 2000 system [13], and the third is a shared virtual memory system on four 4-node SMP clusters connected via a Myrinet network [13]. Results show that when using DDM without CacheFlow the PRAM and SGI systems achieve better performance for all applications, while SVM-HLRC achieves better results in one out of four applications. When using CacheFlow, DDM performs better than all other systems for all applications, with the exception of the *Barnes* application in the PRAM system. Overall, the proposed DDM system with CacheFlow performs as-well-as or better than the other systems. DDM has the advantage that it is built using commodity components.

TABLE 2
Comparison of DDM and CacheFlow Performance with Other Systems

Application	DDM	CacheFlow	PRAM	SGI-O	SVM-HLRC
LU	10.8	14.7	13.2	14.1	12.1
FFT	11.7	15.1	14.8	14.3	6.2
Radix	11.2	14.5	13.5	12.4	1.9
Barnes	10.3	14.1	15.0	13.8	9.5
Cholesky	9.7	12.3	10.4	--	--
Radiosity	11.5	14.2	13.6	--	--

7 Related Work

DDM is a nonblocking multithreading execution model that has its origins in the dynamic dataflow model. There have been several multithreading projects that use dataflow principles for thread sequencing. *Monsoon* [14] is one of the first multithreaded architectures that employ dataflow principles. Monsoon is build using a pipelined processor specifically designed to support dataflow sequencing and execution of very fine-grain threads. The successor of the Monsoon is the StarT (or *T) project [5] that utilizes the Motorola 88110 superscalar processor modified to support fine-grain communication and scheduling of user threads, while maintaining the latency hiding features of the Monsoon. Another nonblocking multithreading project that has its origins in the dynamic dataflow sequencing is the EARTH multiprocessor [6]. Each node consists of the execution unit, which is an off-the-shelf RISC processor, and the synchronization unit that provides dataflow-like synchronization. The instruction set of the processor is modified to support multithreading. The execution model of all three architectures mentioned above is similar to the DDM execution model. The main difference is that DDM can be implemented using any commodity microprocessor, without any modifications, while Monsoon is built using specifically designed hardware, and the *T and Earth are built using existing processors but modified to support dataflow-like execution (additional instructions in ISA).

The Threaded Abstract Machine (TAM) [7] is a software approach to tolerate latency through the execution of nonblocking threads. In this model, all synchronization, scheduling, and storage management are placed under the compiler control. When the execution of a thread is completed, the thread forks its consumer threads. This is obtained by decrementing the synchronization counters of the consumer threads, and if they become zero they are placed in the execution waiting queue. TAM has been implemented and tested on a variety of existing processors/machines. The difference between TAM and DDM is that in DDM thread scheduling is done by the hardware, according to data availability, while in the TAM is done by the software, incurring extra CPU overheads. Another difference is that in DDM the communication is handled by the

TSU with minimal CPU overhead, while in TAM, the communication is handled in software by executing special inlet threads. This adds extra CPU cycles that degrade the performance.

The Pebbles Multithreaded Model [15] is also an execution model evolved from the dataflow model of execution. Each node consists of an execution unit, a synchronization unit and the local memory. The three units communicate via queues. The execution unit executes nonblocking threads to completion. A thread is enabled for execution by the synchronization unit, when all its inputs become available. The main difference between DDM and the Pebbles model is that in DDM the synchronization counters are stored in the TSU while in the Pebbles they are store in the main memory in a frame allocated for each thread. To overcome the problem of the loss of locality due to the multithreaded execution, Pebbles employs a special storage hierarchy [16], while DDM achieves its performance by employing the CacheFlow policies.

In DDM, the computation of nonblocking threads is decoupled from synchronization. Scheduled Dataflow (SDF) [4] is a multithreaded architecture that also decouples the synchronization from the computation of nonblocking threads. A SDF processor consists of two pipelines: the execution pipeline and the synchronization pipeline. The synchronization pipeline is responsible for scheduling the nonblocking threads. The execution pipeline is responsible for the execution of threads. Each thread is assigned a register context. The synchronization pipeline preloads the data needed by a thread in its register context in the execution pipeline, before firing the thread for execution. The main difference between DDM and SDF is that in DDM the computation is carried out by an off-the-shelf processor while in SDF the computation is carried out by a custom designed processor. Another difference is that in SDF data is preloaded in registers while in DDM data is prefetched in the cache. The fact that in SDF all data must be preloaded into registers before the thread is fired, limits the size of threads to fine-grain, since there is a limited number of registers in each register context.

Out-of-order processors employ dataflow principles to rearrange the order of execution of instructions in order to optimize the utilization of the processor's resources. The processor relies on hardware mechanisms that determine dynamically data dependencies among the instructions stored in the instruction window. Explicit Data Graph Execution (EDGE) [17] proposes the scaling to window sizes of thousands of instructions to hold a large code block. The TRIPS processor is an instance of the EDGE architecture which uses large cores consisting of a matrix of ALUs. A program consists of large code blocks that are scheduled using control flow order. The instructions of code blocks are mapped to the ALUs and executed based on dataflow order set by the compiler. The difference with DDM is that code blocks (threads) are scheduled in dataflow order determined at runtime. The instructions within a code block are executed in a control flow set by the compiler, while in EDGE threads are scheduled statically in control flow order with the instructions within threads scheduled using dataflow principles.

Another recently proposed multithreaded architecture that employs dataflow firing rules is the WaveScalar [18]. This is a dataflow instruction set that combines fine-grain multithreading and dataflow synchronization mechanisms. An implementation of the WaveScalar instruction set is the WaveCache. It consists of small, simple processing nodes replicated across the die. Similarly to DDM, in a WaveScalar architecture, a node forwards the data produced to its dependents (consumers). The main difference between DDM and WaveScalar is that DDM is developed for distributed memory multiprocessor systems while in WaveScalar the processing nodes are incorporated in a single chip. One of our future goals is to implement DDM on a chip multiprocessor.

Another difference between DDM and the above nonblocking multithreaded projects is the exploitation by DDM of cache management policies (CacheFlow), that use prefetching to load in the cache data to be used in the near future. Data prefetching [19], [20] reduces cache misses by loading data into the cache before it is accessed by the processor. Data prefetching can be classified as hardware prefetching [21], [22], software prefetching [23], or thread-based prefetching [24], [25]. With software prefetching, prefetch instructions or directives are inserted in the code that load the required data into the cache before it is accessed by the processor. Hardware prefetching attempts to predict future cache misses using past access and miss patterns to preload the required data accordingly. CacheFlow uses a compiler-assisted hardware prefetching mechanism. The difference between CacheFlow and other hardware prefetchers is that most of the other prefetchers attempt to predict possible cache misses based on earlier misses, while in CacheFlow the addresses of the data needed by a thread scheduled for execution is either specified at compile time or its is determined at runtime when the thread becomes ready for execution. In addition, CacheFlow avoids unnecessary prefetching that would lead to extra bus traffic and cache pollution.

The work presented in this paper is based on the D^3 model of execution. An abstract description of this model is presented in [1]. A detailed hardware description, hardware requirements and timings of D^2NOW is presented in [26]. A detailed description of the communication system of D^2NOW is presented in [10]. The work on the CacheFlow policies has also be presented in [9]. The main contribution of this paper is that it demonstrates through a real implementation and using widely used well-known benchmarks that Data-Driven Multithreading can be efficiently implemented on commodity workstations using off-the-shelf processors such as the Intel Pentium.

8 Conclusions

This paper described the design and performance evaluation of the Data-Driven Multithreading model based on commodity microprocessors. Data-driven multithreading is

a nonblocking multithreading execution model that aims to tolerating internode latency by scheduling threads for execution based on data availability.

To evaluate DDM, we simultated the execution of scientific applications from the SPLASH-2 suite on a DDM implementation based on commodity workstations, the Data-Driven Network of Workstations (D^2NOW). Simulations were carried out for machine configurations with up to 32 nodes.

Results showed that fine-grain threads achieve a poor performance with only 12.8 average speedup on a 32-node machine configuration. Increasing thread granularity by a factor of eight increases the average speedup to 16.8. The results also showed that DDM is able to efficiently hide the communication latency. When increasing the cycle time of the interconnection network by a factor of five, the maximum speedup reduction observed among all applications was only 23 percent. Finally, the *CacheFlow* policies proved to effectively reduce the cache miss ratio. This reduction resulted in a speedup increase of 32 percent.

Overall, the DDM model used on the D^2NOW architecture proved to tolerate well both the communication and synchronization latency. Furthermore, the use of *CacheFlow* reduces the local memory latency resulting in very high speedup values. Altogether, for 16 and 32-node machines, the observed speedup, compared to the sequential single node execution, was 14.4 and 26.0, respectively.

REFERENCES

[1] P. Evripidou, "D3-Machine: A Decoupled Data-Driven Multithreaded Architecture with Variable Resolution Support," *Parallel Computing,* vol. 27, no. 9, pp. 1197-1225, 2001.

[2] P. Evripidou and J.-L. Gaudiot, "A Decoupled Graph/Computation Data-Driven Architecture with Variable-Resolution Actors," *Proc. 1990 Int'l Conf. Parallel Processing (ICPP),* pp. 405-414, Aug. 1990.

[3] A. Agarwal et al., "Sparcle: An Evolutionary Processor Design for Multiprocessors," *IEEE Micro,* vol. 13, pp. 48-61, June 1993.

[4] K. Kavi, R. Giorgi, and J. Arul, "Scheduled Dataflow: Execution Paradigm, Architecure, and Performance Evaluation," *IEEE Trans. Computers,* vol. 50, no. 8, pp. 834-846, Aug. 2001.

[5] R.S. Nikhil, G.M. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," *Proc. Int'l Symp. Computer Architecture (ISCA),* pp. 156-167, 1992.

[6] H. Hum et al., "A Design Study of the EARTH Multiprocessor," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '95),* pp. 59-68, June 1995.

[7] D. Culler et al., "TAM: A Compiler Controlled Threaded Abstract Machine," *J. Parallel and Distributed Computing,* vol. 18, no. 3, pp. 347-370, 1993.

[8] S. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture (ISCA),* pp. 24-36, June 1995.

[9] C. Kyriacou, P. Evripidou, and P. Trancoso, "Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading," *Proc. EuroPar-04,* pp. 561-570, Aug. 2004.

[10] C. Kyriacou and P. Evripidou, "Communication Assist for Data Driven Multithreading," *Advances in Informatics,* (LNCS2563), Springer-Verlang, pp. 351-367, 2002.

[11] Intel, *IA-32 Intel Architecture: Software Developers Manual,* Series System Programming Guide, Intel, vol. 3, 2003.

[12] C. Kyriacou, "Data Driven Multithreading Using Conventional Control Flow Microprocessors," PhD Thesis, Dept. of Computer Science, Univ. of Cyprus, 2005.

[13] A. Bilas, C. Liao, and J.P. Singh, "Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems," *Proc. Int'l Symp. Computer Architecture (ISCA),* pp. 282-293, 1999.

[14] G. Papadopoulos and D. Culler, "Monsoon: An Explicid Token Store Architecture," *Proc. 17th Ann. Int'l Symp. Computer Architecture (ISCA),* pp. 82-91, May 1990.

[15] B. Shankar, L. Roh, W. Bohm, and W. Najjar, "Control of Loop Parallelism in Multithreaded Code," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '95),* pp. 131-139, June 1995.

[16] L. Roh and W.A. Najjar, "Design of Storage Hierarchy in Multithreaded Architectures," *Proc. Int'l Symp. Microarchitecture (Micro-28),* pp. 271-278, Nov. 1995.

[17] D. Burger et al., "Scaling to the End of Silicon with EDGE Architectures," *Computer,* vol. 37, no. 7, pp. 44-55, July 2004.

[18] S. Swanson and M. Oskin, "WaveScalar," *Proc. Int'l Symp. Microarchitecture (Micro-36),* pp. 291-302, Nov. 2003.

[19] D.K. Poulsen and P.-C. Yew, "Data Prefetching and Data Forwarding in Shared Memory Multiprocessors," *Proc. Int'l Conf. Parallel Processing (ICPP),* pp. 276-280, Aug. 1994.

[20] P. Trancoso and J. Torrellas, "The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding," *Proc. Int'l Conf. Parallel Processing (ICPP),* vol. 3, pp. 79-86, 1996.

[21] J.D. Collins, S. Sair, B. Calder, and D.M. Tullsen, "Pointer Cache Assisted Prefetching," *Proc. 35th Ann. Int'l Symp. Microarchitecture (MICRO-35),* pp. 62-73, Nov. 2002.

[22] T. Sherwood, S. Sair, and B. Calder, "Predictor-Directed Stream Buffers," *Proc. 33rd Int'l Symp. Microarchitecture (MICRO-33),* pp. 42-53, Dec. 2000.

[23] C.-K. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* pp. 222-233, Oct. 1996.

[24] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, "Dynamic Speculative Precomputation," *Proc. 34th Ann. Int'l Symp. Microarchitecture (MICRO-34),* pp. 306-317, Dec. 2001.

[25] A. Roth and G.S. Sohi, "Speculative Data-Driven Multithreading," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 37-48, Jan. 2001.

[26] P. Evripidou and C. Kyriacou, "Data Driven Network of Workstations (D^2NOW)," *J. Universal Computer Science,* vol. 6, no. 10, pp. 1015-1033, 2000.

Costas Kyriacou received the HND in electrical engineering from the Higher Technical Institute in Nicosia Cyprus in 1981, the BSc degree in electrical engineering from the Polytechnic Institute of New York in 1985, the MSc degree in electrical engineering from the Polytechnic University of New York in 1987, and the PhD degree in computer science from the University of Cyprus in 2005. He is currently an assistant professor in the Computer Science and Engineering Department at the Frederick Institute of Technology, Cyprus. His research interests include parallel processing, high-performance computer architectures, and reconfigurable computing techniques.

Paraskevas Evripidou received the HND degree in electrical engineering from the Higher Technical Institute in Nicosia, Cyprus, in 1981. In 1983, he joined the University of Southern California with a scholarship from the US Agency for International Development. He received the BSEE, MS, and PhD degrees in computer engineering in 1985, 1986, and 1990, respectively. Currently, he is a professor and the chairperson in the Department of Computer Science, University of Cyprus. He was an associate professor at the same University from 1994 to 2004. From 1990 to 1994, he was with the faculty of the Department of Computer Science and Engineering, Southern Methodist University, as an assistant professor (tenure track). His current research interests are in parallel processing, computer architecture, mobile and pervasive computing, and grid computing. Dr. Evripidou is a member of the IFIP Working Group 10.3, the IEEE, the IEEE Computer Society, and ACM SIGARCH. He is also a member of the Phi Kappa Phi and Tau Beta Pi honor societies. He was the program cochair and general cochair of the International Conference on Parallel Architecture and Compilation Techniques in 1999 and 1995, respectively. He was the program cochair of the Eighth Pan-Hellenic Conference on Informatics and program chair of the Third SEE Conference on e-Commerce.

Pedro Trancoso received the undergraduate degree in electrical and computer engineering from Instituto Superior Técnico (IST), Technical University of Lisbon, Lisbon, Portugal, in 1993, and the MSc and PhD degrees in computer science from the University of Illinois at Urbana-Champaign, Illinois, in 1995 and 1998. He is currently an assistant professor in the Department of Computer Science at the University of Cyprus, Nicosia, Cyprus. He has worked at the IBM T.J. Watson Research Center as a researcher (1997), at the University of Illinois at Urbana-Champaign as a visiting scholar (2000), and at Intercollege Limassol, Cyprus, as an assistant professor (1998-2001). He has published several papers in the area of computer architecture, with a focus on the memory hierarchy, intelligent memory technologies, architecture-aware optimizations for database workloads, power-performance efficient architectures, multicore architectures, and graphics processors. He was a recipient of a Fulbright scholarship to pursue his PhD studies, an EU-Mobility grant and a HPC-Europa grant to visit, as a researcher, the Supercomputing Center CESCA-CEPBA at the Universitat Politecnica de Catalunya, Barcelona, Spain, in 2002 and 2005. He is a member of the CoreGrid Network of Excellence, the IEEE, the IEEE Computer Society, and the ACM. More information can be found at http://www.cs.ucy.ac.cy/~pedro.