Journal of Universal Computer Science, vol. 6, no. 10 (2000), 1015-1033
submitted: 11/4/00, accepted: 3/7/00, appeared: 28/10/00

Data Driven Network Of Workstations (D^2NOW)

Paraskevas Evripidou
(Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, CYPRUS, skevos@ucy.ac.cy)

Costas Kyriacou
(Department of Computer Science, University of Cyprus P.O. Box 20537, 1678 Nicosia, CYPRUS cskyriac@ucy.ac.cy)

Abstract: This paper presents the Data Driven Network Of Workstations (D^2NOW), a multithreaded architecture that is based on the Decoupled Data Driven model of execution. This model decouples the synchronization from the computation portions of a program and allows them to execute asynchronously. At compile time a Multithreaded program is created with a Data-Driven thread synchronization graph superimposed on it.
D^2NOW is built using commodity control-flow microprocessors. The support for the data driven synchronization of threads, is provided by the Thread Synchronization Unit (TSU). The TSU is attached in the COAST (Cache On A STick) L2 Cache slot of Pentium workstations and thus it has an implicit interface, using snooping, to the Pentium microprocessor. Workstations are connected via a Telegraphos interconnection network, which is a high throughput ATM-like switch. Telegraphos uses short packets and guarantees no packet-drop, which is a must for fine grain data-driven computation. D^2NOW exhibits the tolerance to long memory and communication latencies, of the data-driven model, with very little overhead and also exploits short-term optimal cache placement and replacement policies. In our prototype implementation the TSU is implemented using FPGAs and it has very low hardware overhead.

Key Words: Multithreading, NOW, Distributed Shared Memory
Category: B.3.2, C.1.2, C.1.3

1 Introduction

Multithreading has emerged as one of the most promising and exciting approaches for the exploitation of parallelism. It utilizes techniques developed in several independent research directions such as Data-Flow, RISC, compilation for Instruction Level Parallelism (ILP) and dynamic resource management [Iannucci et al. (94)]. Multithreading combines the exploitation of program locality and latency tolerance via task switching. A thread of control is very similar to the notion of process from multiprogramming. The main difference is that a thread in a multithreaded machine is visible at the architecture level [Dennis and Gao (94)]. Multithreaded architectures have been traditionally implemented as tightly coupled multiprocessors.

In blocking multithreading a thread may begin execution before all of its operands are available. A thread suspends whenever a missing operand is needed or a synchronization is required, and the processor switches to another thread ready for execution. The processor should provide hardware support for more than one concurrent program counters and register files and have the ability to switch among threads efficiently [Iannucci panel (94)]. In non-blocking multithreading, a thread may begin execution only if all of its operands are available, and run to completion without suspension. Simultaneous multithreading exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle.

The Data-Driven Network of Workstations (D^2NOW) has evolved from the dataflow model of computation. D^2NOW is a multithreaded architecture that utilizes conventional control-flow workstations, augmented to support dataflow sequencing based on the Decoupled Data-Driven (D^3-model) [Evripidou (97), Evripidou and Gaudiot (90)] model of execution. However, D^2NOW differs from other dataflow machines in that instructions are not synchronized and scheduled individually, but are combined into larger blocks of instructions, called threads.

A node in D^2NOW consists of an of-the-shelf control-flow workstations with an add-on card called the Thread Synchronization Unit (TSU) [Evripidou (97)]. The TSU provides data consistency and thread synchronization for non-blocking multithreaded architectures built with conventional, single threaded, microprocessors. A thread is scheduled for execution only if all of its operands are available in the cache, thus no synchronization or communication or memory latencies will be experienced. Hence there is no need for thread suspension and switching to other threads, eliminating the need for multiple program counters and register files.

At compile time a program is partitioned into a number of threads of variable granularity. Each thread is made out of the computation part and the synchronization part. The TSU provides the thread synchronization of these threads and feeds them to the microprocessor for execution. The guiding principle in the generation of the threads is to fully exploit the ILP capabilities of the target processor. During program execution, the TSU schedules each thread based on data availability. Scheduling based on data availability exploits short term optimal cache placement and replacement policies: before scheduling a thread for execution, it is made sure that all required code and data is already stored in the cache, and that blocks related to threads scheduled for execution are not removed from the cache, before the thread is executed.

In D^2NOW, whenever a thread produces data for a remote workstation, the producer workstation is responsible for transferring the data to the remote workstation. Since a thread is scheduled for execution only if all required data is available, there is no need for remote read operations. This reduces the overall communication cost, since remote read operations are much more expensive than remote write operations.

Scheduling based on data availability provides tolerance to long memory and communications latencies inherent in large-scale multiprocessors, thus making the proposed architecture truly scalable and easily programmed.

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

The block diagram of the D^2NOW is shown in Figure 1. A number of TSU-augmented workstations (Pentium based PCs) are connected through an interconnection network (Telegraphos Switch). The TSU is attached on the COAST (Cache On A STick) slot. COAST is a slot found on many Pentium motherboards, that allows the replacement of the L2 cache, located on the motherboard,

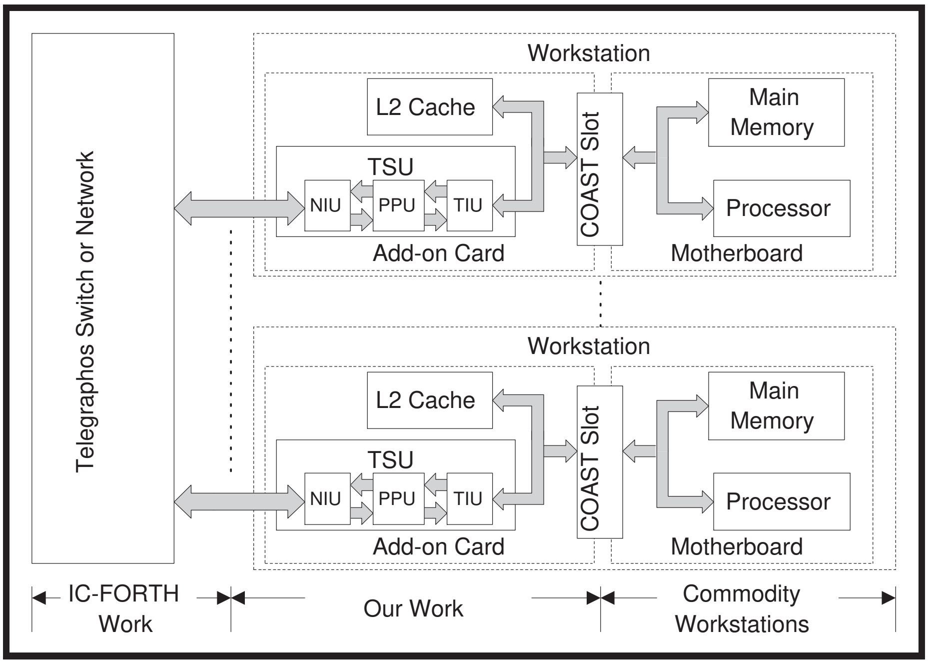

Figure 1: Data Driven Network of Workstations

by an add-on cache card. Several experimental architectures are reported in literature where the network interface unit (usually refered as the Communication Assist) is plugged into existing machines. In most cases the network interface is plugged into the PCI I/O slot. Other architectures integrate the network interface with the graphics bus [Minnich et al. (95)] ,or the SIMM attachment [Banks (93)]. In D^2NOW though, both the processor and the TSU need to access the L2 cache, thus, dual ported RAM is required for the L2 cache. Thus, the TSU should be attached on the motherboard in such a way to have direct access on a modified L2 cache. The use of the COAST slot allows the replacement of the motherboard L2 cache, and the connection of the TSU directly on the processors bus. The COAST slot is no longer provided on todays motherboards. The L2 cache is also integrated with the Pentium Pro chip. Our selection of the COAST slot is justified by the fact that our present goal is the proof of concept. Our future goal is to integrate the TSU with the Pentium processor in a multimodule chip.

The interconnection network is built around the Telegraphos switch [Markatos (96)], developed at ICS-FORTH of the University of Crete. It is a short packet, low overhead, low latency ATM-like switch. Telegraphos guarantees no packet-drop, which is a must for fine and medium grain data-driven computation. The TSU is interfaced with the Telegraphos interconnection network through the Network Interface Unit (NIU). The NIU communicates with the TSU via queues. A packet transfer is initiated whenever a packet is placed in the queue. Communication is carried out solely by the hardware, thus the communication

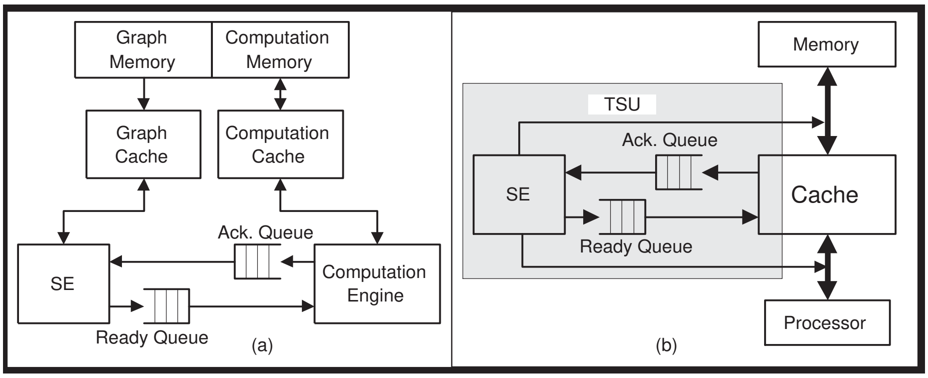

Figure 2: (a) Decoupled Processing Element (b) Processing Element with TSU

system is free from any system call overheads.

In the D^2NOW architecture each processor has its own local memory, which aggregates to a single address space distributed shared memory. Shared virtual memory is employed rather than shared physical address memory. The physical memory space of each workstation is mapped into a global virtual memory space. Remote memory references are directed to the NIU for further processing.

3 Execution Model

The abstract model of execution of the D^2NOW has as its starting point the dynamic Data-Flow D^3 graphs. The basic unit of computation, however, is the thread not the instruction. A key feature of the D^3-model of execution is that the synchronization part of a program is separated from the computation part. The computation part represents the actual instructions of the program. The synchronization part contains information about data dependencies among instructions and it is used for instruction scheduling.

The D^3-model is depicted in Figure 2a. The D^3-graph of the program is stored in the Graph Memory hierarchy. The actual code is stored in the Computation Memory hierarchy. Instruction synchronization and scheduling is carried out by the Synchronization Engine (SE). The actual code of the program is executed by the Computation Engine (CE). Instructions ready for execution, that is, all of their inputs have already been produced, are placed by the SE in the Ready Queue (RQ). The CE reads the instructions from the RQ, executes them, and notifies the SE about the executed instructions via the Acknowledgement Queue (AQ).

In D^2NOW a thread is a sequence of instructions that is executed sequentially and produces a single output. A thread contains no jumps or suspension points, with the exception of the Switch Actor, a thread that is used to implement conditional branching. A producer/consumer relation exists among threads. The data needed by a thread is produced by other threads, called the producers. The data produced by a thread might be needed by other threads, called the consumers.

A program is a collection of code blocks called the context blocks, representing functions or loops. Each context block comprises of several threads. Scheduling of context blocks is done at by the compiler. Scheduling of threads within a context block is done dynamically at run time by the TSU. Whenever the execution of a context block is completed, a unit within the TSU, called the garbage collector is triggered to release the memory reserved by the block and activate an interrupt to load the next context block.

Synchronization occurs only at the top of a thread. Each thread is associated with a synchronizing parameter, called the Ready Count, that indicates the number of inputs still needed to be produced, before the thread is ready for execution. This count is decremented whenever an input value of the thread is produced. A thread is enabled, that is, ready for execution, when its Ready Count reaches zero.

A D^2NOW processing node is shown in Figure 2b. Threads that are ready for execution, that is, all of their inputs have already been produced, are placed in the Ready Queue (RQ). Threads that have already been executed are placed in the Acknowledgement Queue (AQ) for post processing, that is, after the processor completes the execution of a thread, it stores in the AQ the status number of the completed thread and then reads the next thread to be executed from the RQ.

The SE fetches the completed threads from the AQ and updates the Ready Count of the consumer threads. If any of these consumers is ready for execution, it is placed in the RQ and waits for its turn to be executed. The RQ is divided into two stages the Waiting stage and the Firing stage. The SE places a ready thread in the Waiting stage. It then proceeds to determine if the required cache blocks reside in the cache. If they are not in the cache, the SE triggers their placement in the cache. A thread moves in the Firing stage only when all the cache blocks it needs are in the computation cache. Thus the computation processor does not encounter any cache misses or page faults. We call this cache policy: cache-flow [Evripidou (99)].

We have developed extensions to the basic model, such as hierarchical matching and variable resolution support, that makes it more suitable for Multithreaded execution [Evripidou (99)]. The entire dynamic data-flow graph is mapped on the logical space (virtual-memory) of the machine. Therefore, for each instantiation of a thread, we know at execution time where its data will reside by using standard virtual memory translation techniques. For each thread there is a synchronization point, in the logical space, that keeps track of the number of inputs that have been produced. Therefore, all the data-driven synchronization operation can be implemented by references/updates to the logical space. These references are mapped into the physical space using conventional virtual space mapping techniques.

Several simulation experiments have been conducted in order to test the ability of the D^3-machine to tolerate latency and exploit parallelism and locality and are presented in [Evripidou (97), Evripidou (99)]. In summary the results have shown that the D^3-machine can tolerate communication latency. Increasing the communication latency from 1 to 5 (500%) cycles results to an increase in execution of as little as 25%. Going from 5 to 15 cycles increases the execution by a factor of 37%. Furthermore, it does exploit locality: the experimental results have shown that increasing thread length does indeed reduce execution time. Finally the experiments have shown that the D^3-machine in the most part neu-

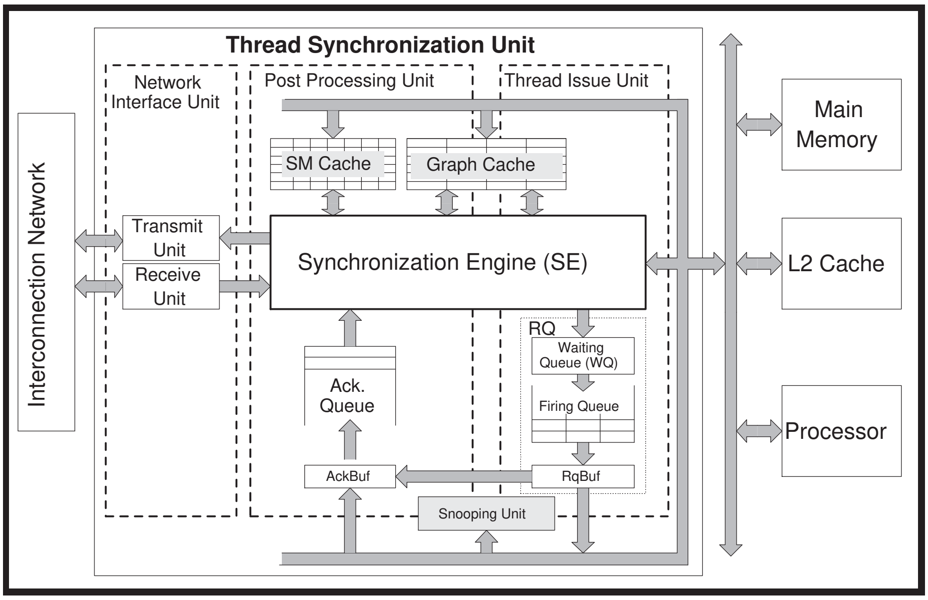

Figure 3: The Thread Synchronization Unit

tralizes the overhead associated with the data-driven synchronization. A five-fold increase in the processing time per thread in the SE resulted in an increase of the overall execution time of as little as 15%.

4 The Thread Synchronization Unit (TSU)

The purpose of the TSU is to provide hardware support for data-driven thread synchronization on conventional microprocessors. It integrates the functions of the SE, RQ and the AQ of the D^3-model. The internal structure of the TSU is shown in Figure 3.

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 SE acts as the control unit of the TSU. The function of the TIU is to schedule threads deemed executable. The PPU updates the Ready Count of the consumers of the completed threads, and determines which are ready for execution. The cache-flow policies for cache prefetching and replacement are implemented by both the TIU and PPU. The NIU is responsible for the communication between the TSU and the interconnection network. The Network Interface Unit (NIU) is made out of two units: the Transmit Unit and the Receive Unit. The design of the NIU depends on the interconnection network used.

At compile time a program is partitioned into a data-driven synchronization graph and the code threads. Each node of the graph represents one thread associated with its synchronization template. Each template contains the Instruction Frame Pointer (**IFP**), the number of input variables or Ready Count (**CountIn**), the Data Frame Pointers (**DFP1** and **DFP2**), and the consumer threads (**Consumer1** and **Consumer2**). If a thread has only one input, the value of DFP2 is set to 0000. If a thread has more than two inputs, then the value of DFP1 is set to 0000 and DFP2 is a pointer to the DFP list, a memory block that contains a list of DFPs in the TSU. A similar approach is also used for the consumers. If a thread has only one consumer then Consumer2 is set to 0000, while if a thread has more than two consumers, then Consumer1 is set to 0000 and Consumer2 is a pointer to the consumer's list, a memory block that contains a list of consumers in the TSU. The program loader loads the Graph Cache (GC) with the IFP, DFP1, DFP2, Consumer1 and Consumer2 of the threads of a context block. The **CountIn** is stored in the Synchronization Memory (SM) Cache. To reduce the size of the Synchronization Memory, only four bits are allocated for the Ready Count parameter of each thread. This limits the maximum number of producers for each thread to sixteen.

4.1 Interface between the TSU and the processor

The processor executes its instructions without any knowledge of the presence of the TSU. Five addresses are reserved for the communication of the processor and the TSU and are disabled from the cache. These locations are implemented as memory mapped registers in the TSU. When there is a need to exchange information between them, the microprocessor writes to one of these reserved memory locations. The TSU uses a snooping unit to intercept these addresses and process them accordingly. These locations are the following:

- **RqIptr** (Ready Queue Instruction Pointer): This location holds the address of the code of the next thread to be executed. The processor reads this location using the instruction (**mov eax,RqIptr**) and then branches to the next thread by executing the instruction (**jmp eax**)
- **RqCntx** (Ready Queue Context Register): This location holds the context or iteration number of the next thread to be executed. The processor copies the thread's context into the index register **esi** using the instruction (**mov esi,RqCntx**).
- **AqCntx** (Acknowledge Queue Context Register): This location holds the context or iteration number of the thread currently being executed. The processor initializes this register at the beginning of each context block using the instruction (**mov AqCntx,esi**).
- **AqStat** (Acknowledge Queue Status Register): The processor writes in this location a status word to inform the TSU about the status of the thread been executed.
- **AqData** (Acknowledge Queue Data): This location holds the data needed for remote write operations. It is accessed by the page fault handler using the instruction (**mov AqData,eax**)

4.2 Thread Issue Unit (TIU)

The function of the TIU is to schedule the threads deemed executable. Figure 4 shows the datapath of the TIU. After all inputs of a thread are evaluated by other threads, the Post Processing Unit (PPU) places the thread's ID number and context in the corresponding buffers of the Waiting Queue (WQ). The thread's

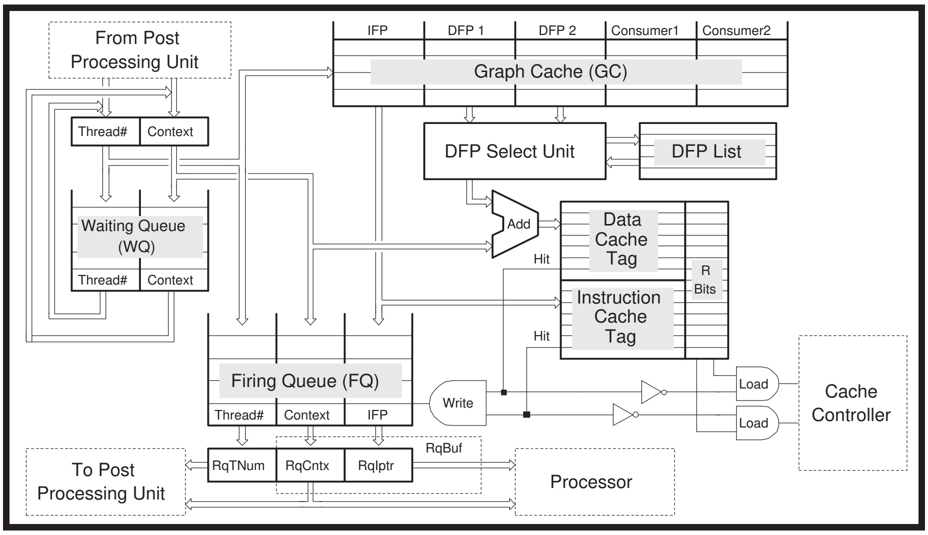

Figure 4: The Thread Issue Unit

ID number is used as the pointer to the Graph Cache that gives the Instruction Frame Pointer (IFP) and the Data Frame Pointers (DFP1 and DFP2) of the thread. The address of the data needed by the thread, is obtained by adding the DFP with the context (or iteration number). Both, the code address and data address of the thread, are compared with the contents of the Instruction Cache Tag and Data Cache Tag, respectively, to determine whether the code and data of the thread are already placed in the computation cache. If the result is a hit for both the code and data, then the triplet (Thread#, Context and IFP) are placed in the FQ, and the thread waits for its turn to be executed.

Four extra bits, the R-bits, are associated with the entry of each set or slot in the computation cache tag. These bits act as a counter indicating how many threads referencing that cache line are waiting in the FQ. If the value of the R-bits is not zero, some threads are scheduled for execution and the corresponding cache line can not be replaced. That line will not be replaced unless if the corresponding threads are executed. Whenever a thread is placed in the FQ, the values of the R-bits for both the code and instruction cache are incremented. These values are decremented by the PPU, when the corresponding thread is executed. If a cache miss results for either the Instruction Cache or the Data Cache, and the value of the R-bits for that entry is zero, the cache controller is signaled to load on the cache the required lines. If the value of the R-bits is not zero, the line can not be replaced, and the thread remains in the WQ.

Whenever the processor completes the execution of a thread, it reads the address of the next thread from the ready queue instruction frame pointer register (**RqIptr**) and its context from the ready queue context register (**RqCntx**). The thread ID number and context are also send to the **AckBuf** of the PPU. The top of the Firing Queue (FQ) is then shifted out in the **RqBuf**, to point to the next ready thread for execution.

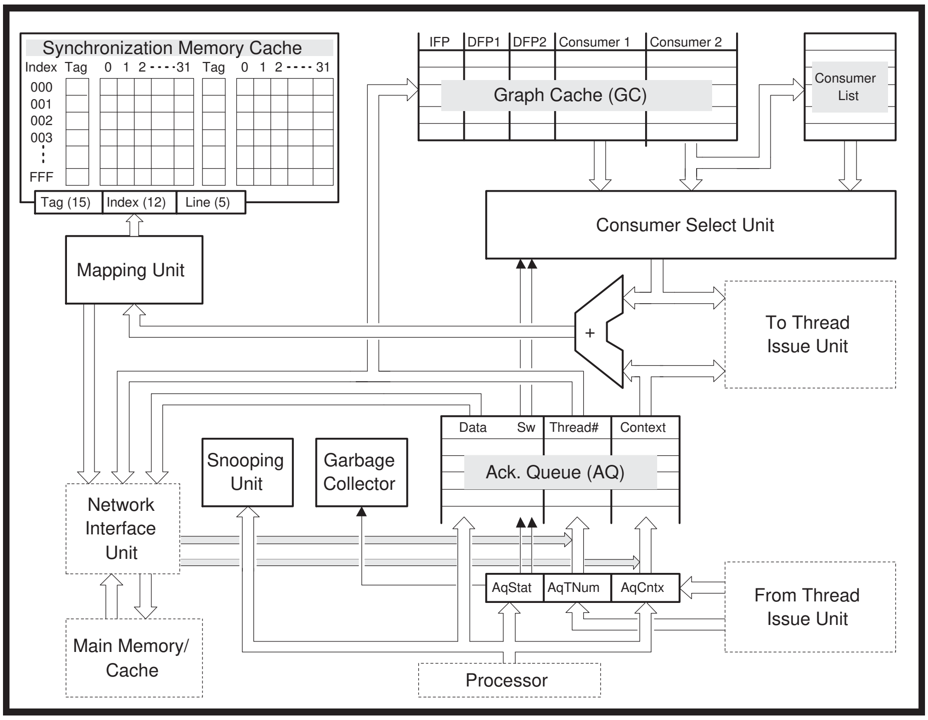

Figure 5: The Post Processing Unit

4.3 Post Processing Unit (PPU)

The Post Processing Unit (PPU) is responsible for the processing of threads that have already been executed by the processor. The datapath of the PPU is shown in Figure 5. The AqTNum and AqCntx registers at the input of the Ack Queue (AQ) contain the thread ID number and the context of the thread currently being executed by the processor. When the execution of the thread is completed, the processor writes a status number in the memory mapped register **AqStat**. The address of this register is snooped by the Snooping Circuit so that the SE is aware of the completion of the thread. The status number stored in the **AqStat** register is made out of three fields: the context field, the switch actor field and the garbage collector field.

The context field indicates whether the context register has been modified during the execution of the thread. If the value of the context field is 00 then the context register was not modified. If the value of the context field is either 01 for incremented or 10 for decremented, then the Context register is modified accordingly and the triplet **AqStat**, **AqTNum** and **AqCntx** are shifted into the AQ. The Thread ID number (Thread#) is a pointer to the Graph Cache (GC). The Consumer Select Unit reads the first consumer thread number from the GC. The consumer's number is then added with the context to determine its address in the memory. The Mapping Unit uses this address to determine whether it refers to a local or remote memory. If the address refers to a local memory, the corresponding entry of the Ready Count in the Synchronization Memory (SM) is decremented. If the content of the SM becomes zero, the thread is deemed executable and its ID number and context are send to the TIU for further processing. The same procedure is followed for the rest of the consumers.

Whenever the address produced by the Mapping Unit refers to a remote memory, the thread ID number, context and data are send to the NIU to be send to the remote workstation. Whenever the NIU receives a packet from a remote workstation, the thread ID number and its context are placed in the AQ. The received data is placed in the main memory.

The garbage collector field is used to indicate the completion of a context block. If the value of this field is 01, then the garbage collector is triggered to release the memory used by the context block, and load the next block.

4.4 Network Interface Unit

The NIU is made out of two units: the Transmit Unit and the Receive Unit. Both units have their own controller and their operation is independent from each other. The design of the NIU depends primarily on the selection of the interconnection network. Important issues in selecting the type of interconnection networks for a NOW are the overheads introduced, the communication latency, the quality of the interconnection, and the simplicity [Anderson et al. (95)]. The Telegraphos switch is chosen to be used as the interconnection network for the D^2NOW.

4.4.1 The Telegraphos Switch

The Telegraphos [Katevenis et al. (95), Markatos (96)] switch is a high performance switch that can be used in high speed networks, NOWs, and multiprocessor networks. It is a low latency, fixed size packet switch based on virtual circuit, and it utilizes hop by hop credit-based flow control. Currently there are two implementations of the Telegraphos switch: Telegraphos I, a board level prototype implemented with FPGAs and RAM chips, and Telegraphos II, a standard cell ASIC implementation. Telegraphos I is a 4 channel switch, with each channel carrying in parallel 8 data bits and a flag bit. Each packet consists of 9 bytes: one byte for the header and 8 bytes for the payload. The flag bit is used to identify the header of each packet, which identifies the virtual circuit number. Each link can support up to 256 virtual circuits. Due to the slow FPGAs used for the implementation of Telegraphos I, the cycle time is 75 ns, giving a 107Mbps throughput in each direction. The packet size in Telegraphos II is twice that of Telegraphos I. The clock cycle time is reduced to 40 ns (using a 16 bit internal datapath), giving a 400Mbps throughput. Preventive flow control is employed to ensure that packets are never dropped, due to buffer overflow in the Telegraphos switch. This is obtained by using a hop-by-hop credit-based flow control mechanism. With this mechanism, a packet is transmitted to the next switch or node only if there is available buffer space. To implement this mechanism, Telegraphos has an extra credit link associated with each data link. Each credit link carries 1 flag bit and 8 data bits per clock cycle. The flag bit is used to indicate that the data bits contain a valid credit. The 8 data links contain the number of the virtual circuit for which the credit refers to. In Telegraphos II the credit links

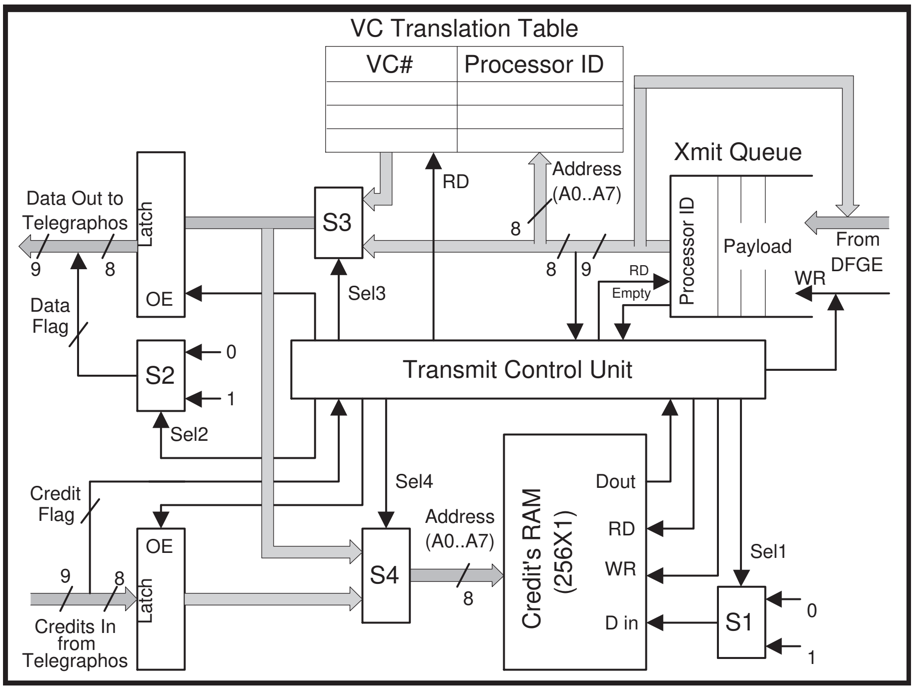

Figure 6: Transmitter Unit

are multiplexed with the data links. We chose to use the Telegraphos switch because: (a) It has a short packet size. Each message of the D^2NOW fits exactly in one Telegraphos II packet. (b) It uses static routing, thus less overheads and in order delivery of packets is quarantined. (c) Packet delivery is quarantined due to the preventive credit based flow control used. (d) It has only a 3 clock cycles cut through latency, thus low communication latency. (e) It is fast. It has 40 ns cycle time, thus 400 Mbps throughput.

4.4.2 Transmitter Unit

The block diagram of the transmitter unit is shown in Figure 6. The transmitter unit performs two functions. The first one is to monitor the state of the interconnection network, by keeping a record of the credits for each virtual circuit. The second is to assemble and transmit packets of data to the interconnection network.

Information needed to be send to other processors is first stored in the Xmit Queue by the PPU, in the form of a packet. Each packet contains the receiving processors ID number, the base and context values of the receiving thread, as well as the calculated value. The first entry of each packet is always the receiving processor ID number. The Transmit Control Unit monitors the 'Empty' signal of the Send Queue to check if a packet waits to be transmitted. In such a case the first entry in the queue is used as the address for the Virtual Circuit Translation

Table. The function of this look-up table is to determine the virtual circuit number of the destination processor. This table is static because the Telegraphos switch uses static (hardwired) virtual circuits. After finding the virtual circuit number (VC#), the Transmit Control Unit reads the corresponding entry in the Credits RAM. Each entry in this RAM indicates whether there is an empty slot in the Telegraphos switch for a specific virtual circuit. This is a 256X1 RAM, since the Telegraphos switch can support up to 256 virtual circuits per link. If there is an empty slot, the Transmit Control Unit sends first, the virtual circuit number to the Telegraphos switch through the Data Out lines. The rest of the packet is then transmitted, one byte per cycle. After sending a packet to the switch, the Transmit Control Unit resets the content of the credit RAM for that virtual circuit. This is required because only one slot is allocated for each virtual circuit in the Telegraphos switch. If a packet is to be send through a virtual circuit that has no empty slot, that is, its credit is zero, then that packet has to wait until the corresponding slot is empty. In this case, the Transmit Control Unit has to wait until it receives a credit for that virtual circuit, and then transmit the packet. If the blocked packet remains in the queue, then all of the following packets in the queue will be blocked as well. In order to avoid blocking the rest of the packets, the packets that can not be send to the interconnection network, are recycled in the Xmit Queue. The Transmit Control Unit also monitors the state of the Flag bit of the Credits In lines. If the flag bit is at logic 1, then a credit is received for the virtual circuit specified by the data lines of the Credits In lines. In this case, the Transmit Control Unit reads the data from the Credit In line and updates the content of the Credit RAM for the specified virtual circuit.

4.4.3 Receiver Unit

The block diagram of the Receiver Unit is shown in Figure 7. The Receiver Control Unit monitors the Flag bit of the Data In lines send by the Telegraphos switch. If the flag bit is at logic 1, then the packet contains valid data, and the first byte represents the virtual circuit number. The positive edge of the Flag bit of the Data In lines is used to trigger the clock synchronization circuit. The clock synchronization circuit is required to ensure that data is read correctly, since there might be a phase difference between the clock of the TSU and the clock of the switch. The clock frequency of the receiver unit is three times the frequency of transmission of the Telegraphos switch. The receiver reads each byte from the switch on the positive edge of the second clock pulse of each cycle. The Receiver Control Unit reads the next 18 bytes from the Data In lines and stores them the Rcve Queue for further processing by the PPU.

Whenever the Telegraphos switch sends a packet, it automatically decrements the credit counter for that specific virtual circuit. The switch can not transmit any more packets through this virtual circuit, unless a credit for that circuit is received. The Credit Unit monitors the Flag bit and the RD signal of the Rcve queue. If both of these signals are activated, then the PPU is reading the virtual circuit number from the Rcve Queue. In this case the virtual circuit number is latched to the Credits Out lines, to inform the switch that it can send more data.

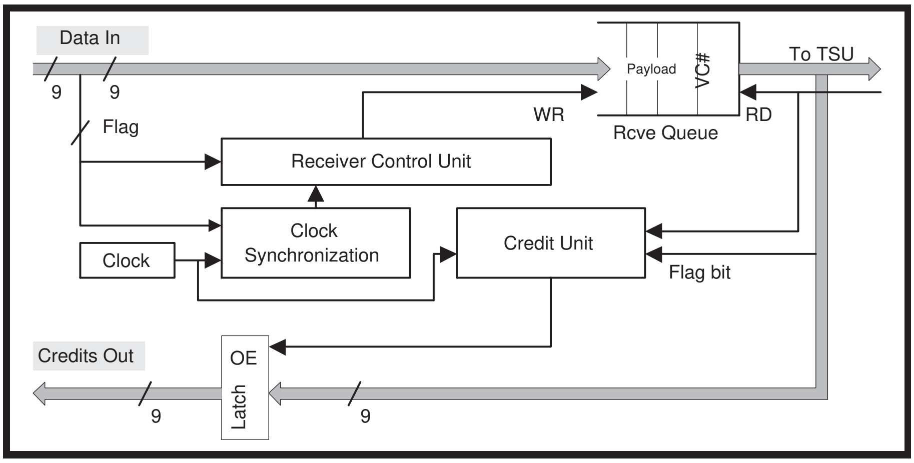

Figure 7: Receiver Unit

4.5 Mode of operation

The D^3-graph for the implementation of the inner product example is shown in Figure 8a. The graph represents one context block. Its inputs are the range or number of iterations (Rnge), the starting index (Stindex) and the two vectors (R and Q). Shaded boxes represent threads. Non shaded boxes represent instances of values. For simplicity this example consists of very fine threads (one or two operations). This is a hand coded example used to illustrate the operation of the system.

Figure 8b shows the dynamic data frames for the computation code. Each frame consists of the data location for each value needed by one iteration of the loop. This represents the data area of the context block. This memory block is created when the block is loaded in the TSU, and released by the garbage collector after the block has been executed. The content of the synchronization memory cache is shown in Figure 8c, in this example we have 6 threads so we need 6 synchronization points per iteration. The entries in this memory represent the number of producers of each thread, or the number of input arcs in the graph (**CountIn** field in the template). This memory block is also initialized when the context block is loaded.

The templates and computation code are shown in Figure 8d. The box on the left shows the graph template, and the box on the right shows the actual code. The template of each thread has six entries: the pointer to the code of the thread, the number of producers, two DFPs and two consumers.

Initiating a context block: The first thread initiates and triggers the execution of the loop. The value of the index register **esi** is initialized to the current context which is stored in the memory location **stindex**. The value of the sum **psum[esi]** is initialized to zero.

Switch to the next thread: The last two instructions of each thread (**mov eax,RqIfpr** and **jmp eax**) branch to the instruction specified by the RQ instruction frame pointer **RqIptr** register, that is, the address of the next thread to be executed. Recall that the **RqIptr** is a memory mapped register that is

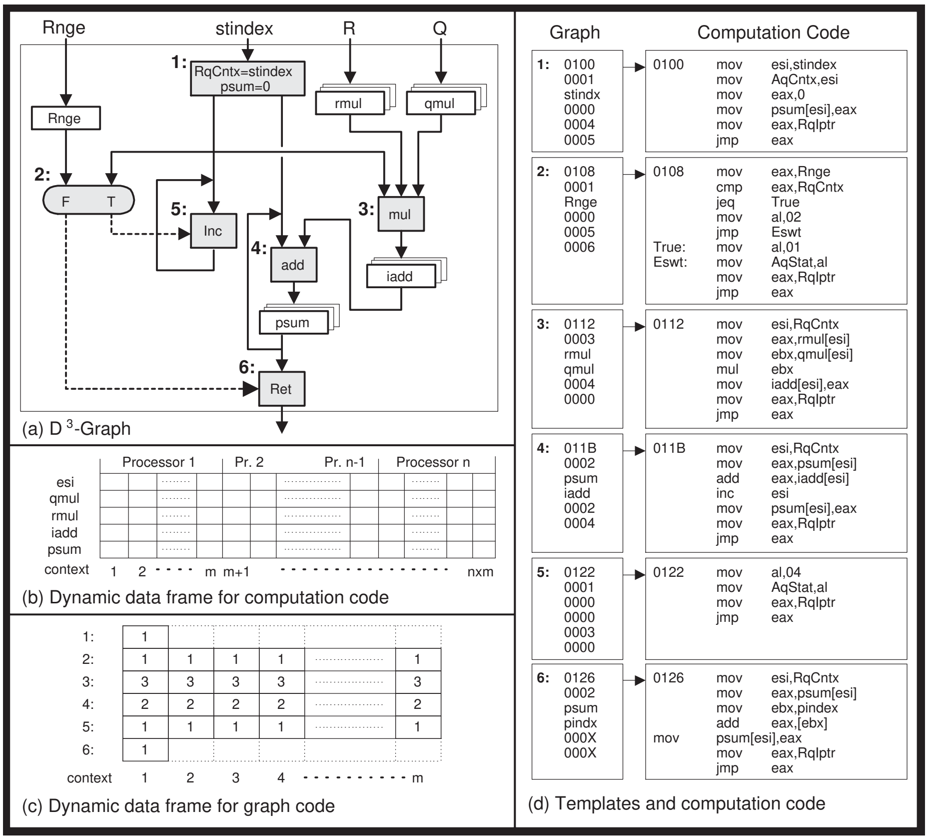

Figure 8: The inner product example

disabled from the cache. When instruction the **mov eax,RqIfpr** is executed, the Snooping Unit intercepts that address and sends the address of the next thread to the processor.

Changing the context: Thread#5 is used to increment the context. This is implemented by storing in the acknowledge queue status register **AqStat** the number 04, which instructs the PPU to increment the content of the acknowledge queue context register **AqCntx**. Incrementing the context register is done by the hardware, before the **AqCntx** register is shifted in the AQ.

Switch Actors: Thread#2 is a switch actor. It compares the value of **'Rnge'** with the present value of the context register, to determine whether the end of the loop is reached. If the end of the loop is not reached, then the switch actor triggers thread#5, otherwise thread#6 is triggered. This is shown in the graph with dotted lines, indicating data dependencies rather than data movement. In the computation code, this is implemented by storing in the register **AqStat** the number 01 if the condition is true, or 02 if the condition is false. The Consumer Select Unit of the PPU uses the content of the **AqStat** to select one of the two consumers.

Returning from a context block: The last thread in the block is Thread#6. This thread sends the partial sum to the next processor and triggers the garbage collector. This is implemented by writing in the **AqStat** register the number 10h (garbage collector field = 01).

4.6 Shared Memory Implementation

Shared virtual memory is employed in the D^2NOW. Each processor views the global virtual memory as its own virtual memory. Remote memory references are carried out implicitly through short messages.

Memory mapping is done according to the thread ID number and the context, as shown in figure Figure 8b. A remote memory write operation is issued when Thread#4, in figure Figure 8d, attempts to write the calculated result in (psum[esi]) with context (m+1), where (m) is the last iteration assigned to each processor for that specific context. The virtual address for (psum[m+1]) belongs to a remote processor, thus the instruction (**mov psum[esi]**) will cause a page fault. The page fault handler will store the produced data in the AqData field of the Acknowledge Queue in the PPU. This is done by executing a single instruction (**mov AqData,eax**). Thus the cost of the page fault is very low. A mapping unit within the TSU is responsible for determining the destination workstation of the remote write, and initiate the write operation by shifting the virtual address (Thread ID number, the context) and the data from the AQ to the NIU for further processing. Please note that no page faults occur for local memory references since we preallocate the pages and cache blocks before we allow a thread to enter the firing stage of the TSU.

The TSU is presented only with virtual addresses produced either at compile time, contents of the Graph Cache, or at run time, contents of the Synchronization Memory Cache. These addresses depend on the Threads ID number and the context. There are two cases where the TSU needs to use physical addresses. The first case refers to remote write operations where the receiving TSU needs to know where to store the received data. In this case the TSU stores the data into the appropriate physical address.

The second case is due to the need to check if the code and data required by a thread are placed in the cache, before it is transferred in the Firing Queue. Since the cache is addressed using physical addresses, it is required that the entries in the Graph Cache that specify the Instruction Frame Pointer and the Data Frame Pointers contain physical addresses. To facilitate the address translation we incorporate a TBL within the TSU. In case of a TBL miss the TSU access the virtual memory map that resides in the main memory.

5 Implementation Issues

The TSU is implemented using the Xilinx XL4005E Field Programmable Gate Arrays (FPGAs) as well as standard components like SRAM and FIFO chips. The components needed for the implementation of the TSU are listed in Table 1. Nine FPGAs are used for the implementation of the TSU. These FPGAs are not fully utilized. This is done in order to reduce the routing delays in the FPGAs and thus obtain higher speeds. The hardware needed will be reduced significantly on future ASIC designs.

Function	Device	Comment
Thread Issue Unit		
Waiting Queue + Thread# and Context Registers	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 180 (95% of resources), I/O pins = 102
Firing Queue + RqBuf	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 176 (95% of resources), I/O pins = 104
Graph Memory (IFP, DFP1 and DFP2) + DFP List	Four MT58LC64K16	64K X 16 Sync. Burst SRAM (10 ns access time)
TIU controller + DFP Select Unit + Adder	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 120 (60% of resources), I/O pins = 96
Post Processing Unit		
Ack. Queue	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 160 (80% of resources), I/O pins = 74
Ack. Registers + Snooping Circuit + Garbage Collector	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 60 (30% of resources), I/O pins = 84
PPU controller + Consumer Select Unit + Adder	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 80 (40% of resources), I/O pins = 70
Synchronization Memory Cache	Four MT58LC64K16 One XL4005E FPGA	64K X 16 Sync. Burst SRAM (10 ns access time)
Graph Cache (Consumer1 and Consumer 2) + Consumer List	Three MT58LC64K16	64K X 16 Sync. Burst SRAM (10 ns access time)
Network Interface Unit		
VC Translation table	KM68257CJ15	32KX8 SRAM with 15 ns access time. Only a small fraction of the memory area is used. Fast and cost effective
Xmit Queue	Am7202A-15	1KX9 FIFO with 15 ns access time. Can hold up to 58 packets. (18 bytes per packet)
Credits RAM + Transmit Controller + Multiplexers	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 108 (55% of resources), I/O pins = 39
Rcve Queue	Am7205A-15	8KX9 FIFO with 15 ns access time. Can hold up to 256 packets. (One packet for each virtual circuit)
Receiver Control + Credits Circuit + Clock Synchronization	XL4005E FPGA	Frequency: 50 MHz (cycle time = 20 ns) CLB used = 94 (48% of resources), I/O pins = 24

Table 1: Component List

The cycle time for all units of the TSU is 20 ns. The timing analysis for the operation of the TSU is given in Table 2. The minimum time needed by the TIU to process one thread is 120 ns. This is the case of threads that have only one DFP and their code and data is already placed in the cache. The minimum time needed by the PPU to process one thread is 100 ns. This is the case of threads that have only one consumer that refers to local memory. At least 20 ns must be added for any extra consumer or DFP. The TIU and the PPU operate asynchronously, and concurrently. Thus the synchronization latency is reduced to the maximum latency introduced by either the TIU or the PPU. Furthermore the synchronization latency in most cases (RQ not empty) does not affect the critical path of the computation.

We coupled the TSU with a 100 Mhz Pentium microprocessors. Based on the simulations results we got for the D^3-machine we expect that even for moderate number of instructions per thread the scheduling overhead of the TSU is negligible. The current trend for designing high performance microprocessors is to place them with the L2 cache in a multichip module (MCM). We envision that the TSU could also be placed in the same MCM. Thus, our work on the TSU could be easily adapted by such microprocessors.

Function	Number of Cycles	Time
Thread Issue Unit (Cycle Period = 20ns)		
Read Waiting Queue	1	20 ns
Read IFP and DFPs	2	40 ns
Read extra DFPs	1	20 ns (each DFP)
Check Cache Tag (1 IFP and 2 DFP)	3	60 ns
Write into Waiting Queue of Firing Queue	1	20 ns
Post Processing Unit (Cycle Period = 20ns)		
Write into Ack. Queue	1	20 ns
Read Consumer1 and Consumer2	2	40 ns
Read extra Consumers	1	20 ns
Get Address from Mapping Unit	1	20 ns
Update Synchronization Cache	2	40 ns (for each Consumer)
Network Interface Unit (Cycle Period = 20ns)		
Form a Header (read VC and check credits RAM)	6	120 ns
Send a byte	2	40 ns
Send a packet (18 bytes)	6 + 18 X 2 = 42	840 ns
Read a byte	2	40 ns
Read a packet (18 bytes)	36	720 ns

Table 2: Timing Analysis

6 Related Work

D^{2}NOW is a non blocking Multithreading machine that has its origins in the dynamic data-flow sequencing. There have been several multithreading projects that use data-flow sequencing. The *Monsoon* departed form the idea of associative memory for token matching and introduced the concept of Explicit Token Store (ETS). A memory location, within the activation frame of each function allocation is established where each synchronization takes place. In general, the allocation of activation frame has to be done dynamically at run time. The successor of the Monsoon is the Start (or *T) project [Boon et al. (94), Boon et al. (98a), Boon et al. (98b)] that utilizes off the shelf microprocessors while maintaining the latency hiding features of the Monsoon.

Commercial multithreaded machines such as the Tera [Alverson et al. (94)] have been around for some years now. Tera is a pipelined multithreaded multiprocessor that enables multithreaded execution at al levels of parallelism. The compiler and run time system exploit parallelism at fine and medium grain levels, while the operating system exploits parallelism at coarse grain level. The main difference between Tera and the D^{2}NOW machine is that Tera is using blocking multithreaded processors with multiple register files, while the D^{2}NOW machine is implemented with Pentium processors running in a non blocking multithreading mode. In the Tera machine, a thread suspends if it encounters a long latency. In the D^{2}NOW machine a thread is scheduled for execution only if all required data is available in the cache. Threads run to completion in the D^{2}NOW machine. Another difference is that in the Tera machine, medium and coarse grain thread scheduling is done by the software, while in the D^{2}NOW machine thread scheduling is done by the hardware.

Simultaneous Multithreading (SMT) [Eggers et al. (97), Lo et al. (97)] is used in superscalar processors to allow multiple threads to issue instructions at each cycle. SMT uses multiple threads to compensate for low single-thread ILP, by keeping busy all units of a superscalar processor all times. The main difference between SMT and D^2NOW is that D^2NOW is based on non-blocking multithreading, while SMT is based on blocking multithreading. Another difference is that in D^2NOW there is no need for hardware modifications on the internal structure of the processor.

The run time system employed by the D^2NOW machine is similar to that of the Threaded Abstract Machine (TAM) [Culler et al. (93)]. TAM is a software approach to tolerate latency. In this model all synchronization, scheduling and storage management are placed under the compiler control. TAM is implemented for a variety of existing processors.The difference between TAM and D^2NOW is that in the D^2NOW machine thread scheduling is done by the hardware, according to data availability, while in the TAM is done by the compiler.

7 Concluding Remarks

The TSU is a hardware mechanism that supports the design of Multithreading platforms with conventional microprocessors. Its mode of operation is based on the Decoupled Data-Driven (D^3-model) of execution. The decoupling of the synchronization and computation of a multithreaded program and their asynchronous execution allows us to concentrate in designing the synchronization module and readily adopt the state-of-the-art microprocessor technology for the computation.

The TSU provides a minimal hardware approach in designing Multiprocessors based on the D^3 model of execution. We are developing a prototype implementation of a TSU-based multithreaded platform: the Data-Driven Network of Workstations (D^2NOW). The TSU is designed using FPGAs. It has cycle time of 20 ns and it takes about 5-6 cycles to performed the preprocessing and as many for the post processing of each thread. We coupled the TSU with a 100 Mhz microprocessors. Thus, for even moderate number of instructions per thread the scheduling overhead of the TSU is expected to be negligible. The current project serves as proof of concept. The next generation of the TSU will be designed with ASIC methodology and incorporated with the processor and the L2 cache in a multichip module MCM.

References

[Alverson et al. (94)] Gail Alverson, Bob Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. *Integrated Support for Heterogeneous Parallelism.* Kluwer Academic Publishers, 1994. Edited by Robert Iannucci *et al.*

[Anderson et al. (95)] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations). *IEEE Micro*, 1995.

[Banks (93)] D. Banks and M. Prudence. A high performance network architecture for a PA-RISC workstation. *IEEE Journal of Selected Areas in Communication*, 1993.

[Boon et al. (94)] Boon Seong Ang, Arvind, and Derek Chiou. Start the next generation: Integrating global caches and dataflow architecture. In *Proceedings of the International Conference on Computer Systems and Education, IISc, Bangalore, India*, 1994.

[Boon et al. (98a)] Boon S. Ang, Derek Chiou, Larry Rudolph, and Arvind. The startT voyager parallel system. In *Proceedings of the International Conference* *on Parallel Architectures and Compilation Techniques (PACT98)*, Paris, France, October 1998.

[Boon et al. (98b)] Boon S. Ang, Derek Chiou, Daniel Rosenband, Mike Ehrlich, Larry Rudolph, and Arvind. The startT voyager: A flexible platform for exploring scalable smp issues. In *Proceedings of SuperComputing 98*, Orlando, Florida, November 1998.

[Culler et al. (93)] D. Culler *et al.* Tam: A compiler controlled threaded abstract machine. *JPDC*, June 1993.

[Dennis and Gao (94)] Jack Dennis and Guang Gao. Multithreaded Architectures: Principles, Projects and Issues. In R. Iannucci et al., editor, *Multithreaded Computer Architecture a Summary of the State of the Art.* Kluwer Academic Publishers, 1994.

[Eggers et al. (97)] Eggers *at al.* Simultaneous multithreading: A platform for next generation processors. *IEEE Micro*, pages 12–18, September/October 1997.

[Evripidou (97)] P. Evripidou. Thread Synchronization Unit (TSU): A building block for High Performance Computers. In *Proceedings of the International Symposium on High Perfomance Computing, Fukuoka, Japan*, Nov. 1997.

[Evripidou (99)] P. Evripidou. D^3-machine: A Decoupled Data-Driven Multithreaded Architecture with Variable Resolution Support. *Parallel Computing.* In Press.

[Evripidou and Gaudiot (90)] P. Evripidou and J-L. Gaudiot. A Decoupled Graph/Computation Data-Driven Architecture with Variable Resolution Actors. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[Iannucci et al. (94)] Robert A. Iannucci *et al. Multithreaded Computer Architecture a Summary of the State of the Art.* Kluwer Academic Publishers, 1994.

[Iannucci panel (94)] Panel Discussion. Architectural implementation issues for multithreading. In R. Iannucci et al., editor, *Mutlithreaded Computer Architecture a Summary of the State of the Art.* Kluwer Academic Publishers, 1994.

[Katevenis et al. (95)] M. Katevenis, P. Votsalaki, A. Efthymiou, and M. Stratakis. Vc-level flow control and shared buffering in the telegraphos switch. In *IEEE Hot Interconnects III Symposium*, 1995.

[Lo et al. (97)] J. Lo *at al.* Converting thread level parallelism into instruction level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, pages 322–354, August 1997.

[Markatos (96)] E. Markatos and M. Katevenis. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *Symposium on High Performance Computer Architectures*, 1996.

[Minnich et al. (95)] R. Minnich, D. Burns, and F. Hady. The memory integrated network interface. *IEEE Micro*, 1, 1995.