Parallel Processing Letters
© World Scientific Publishing Company

Rapid Prototyping of the Data-Driven Chip-Multiprocessor (D^2-CMP) using FPGAs

Konstantinos Tatas, Costas Kyriacou
Computer Engineering Department, Frederick University,
P.O Box 24727, 1303 Nicosia, CYPRUS

and

Paraskevas Evripidou, Pedro Trancoso
Department of Computer Science, University Cyprus,
P.O Box 20537, 1678 Nicosia, CYPRUS

and

Stephan Wong
Computer Engineering Laboratory, Delft University of Technology, The Netherlands

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

This paper presents the FPGA implementation of the prototype for the Data-Driven Chip-Multiprocessor (D^2-CMP). In particular, we study the implementation of a Thread Synchronization Unit (TSU) on FPGA, a hardware unit that enables thread execution using dataflow-like scheduling policy on a chip multiprocessor. Threads are scheduled for execution based on data availability, i.e., a thread is scheduled for execution only if its input data is available. This model of execution is called the Data-Driven Multithreading (DDM) model of execution. The DDM model has been evaluated using an execution driven simulator. To validate the simulation results, a 2-node DDM chip multiprocessor has been implemented on a Xilinx Virtex-II Pro FPGA with two PowerPC processors hardwired on the FPGA. Measurements on the hardware prototype show that the TSU can be implemented with a moderate hardware budget. The 2-node multiprocessor has been implemented with less than half of the reconfigurable hardware available on the Xilinx Virtex-II Pro FPGA (45% slices), which corresponds to an ASIC equivalent gate count of 1.9 million gates. Measurements on the prototype showed that the delays incurred by the operation of the TSU can be tolerated.

Keywords: Data-Driven Multithreading, FPGA, Multicore

1. Introduction

The trend, for the last two decades, of building high-performance processors with large caches, dynamically extracting ILP from sequential programs and employing out-of-order execution, has reached its limit. The effects of the memory wall and power consumption getting out of hand have forced the industry to turn to multiple cores per chip. However, the switch made by the industry did not address the fundamental issues that caused the problem of the memory wall and power consumption. Instead, it has resorted to an engineering workaround, lowering the clock frequency and placing multiple cores on a chip. The switch to multiple cores per chip makes concurrency be the major issue in achieving high-performance. Decades of research and development in parallel processing based on sequential processors has shown that handling concurrency efficiently is not a trivial task. The computer architecture community has resisted the calls for a move to naturally parallel models of execution because the sequential processors were dominating the marketplace. However, now that mainstream computer architecture has moved to the *Concurrency Era* it makes sense to adopt a naturally concurrent model instead of continuing to add ad-hoc constructs to the sequential model. One such model is the Data-Flow model of execution. The Data-Flow model does not rely on the program counter for scheduling but instead it schedules instruction based on the availability of required data [1, 2, 3].

In this paper we present the implementation of a hardware prototype of a Data-Driven Multithreading (DDM) chip multiprocessor architecture (D^2-CMP). In DDM, a program is composed of threads of instructions, which are related among themselves through producer-consumer relationships. The scheduling of these threads is done in a data-flow like manner, therefore exploiting the maximum available parallelism and providing tolerance to memory and communication latencies. The first implementation of the DDM model was the D^2NOW [4]. The D^2-CMP is a chip multiprocessor that combines the benefit of the DDM model with those of the multi-core technology [5]. The scheduling of the threads is done by a memory-mapped device, the Thread Synchronization Unit (TSU) [6]. The interaction of the processor with this unit does not require changes to the processor, therefore allowing the use of commodity processors. Consequently, the DDM implementation may easily evolve by upgrading and exploiting the latest technologies present in the state-of-the-art microprocessors. This fact contrasts with other nonblocking multithreaded systems that require either special design processors or modified existing processors [14, 15, 17].

The DDM model of execution has been implemented on a distributed shared memory network of workstations and evaluated using an execution driven simulator [7]. For a 32-node system, the execution of seven Splash-2 benchmarks, resulted in achieving an average speedup of 26.

Evaluation and benchmarking using software simulations tends to be time consuming. To further investigate the potential of the DDM model of execution on

a multi-core system, we have developed a 2-node DDM chip multiprocessor on a Xilinx Virtex-II Pro FPGA, that has two PowerPC microprocessors embedded on the chip. Executing benchmarks on the FPGA-based prototype is orders of magnitude faster than the software simulations, thus enabling us to explore a much larger application domain.

The developed prototype has been used to validate the simulation results achieved and to measure the hardware requirements for the implementation of the TSU. Measurements have shown that a DDM chip multiprocessor can be implemented with a moderate hardware budget, since two TSUs have been implemented on a single FPGA with the two PowerPC processors embedded in it. The hardware required for the two TSU D^2-CMP implementation amounts to 45% of the available Xilinx xcv2p30 Virtex-II Pro FPGA slices. This amount may be translated to an ASIC equivalent gate count of 1.9 million gates. In terms of performance, the delays incurred by the operation of the TSU can be tolerated due to the asynchronous operation of the processing nodes and the TSU.

This paper is organized as follows. Section 2 gives an abstract description of the DDM execution model. Section 3 describes architecture of the D^2-CMP and the TSU. The hardware prototype of D^2-CMP is given in Section 4, while Section 5 presents the results and evaluation of the proposed system. Section 6 and 7 present the related work and conclusion respectively.

2. The DDM Model

Data-Driven Multithreading [7, 8] is a non-blocking multithreading execution model that tolerates memory and synchronization latencies by scheduling threads for execution based on data availability. A thread is scheduled for execution only if its input data is available. The core of the DDM implementation is the Thread Synchronization Unit (TSU), a memory-mapped hardware module that is attached directly to the processor's bus. This module is responsible for thread scheduling.

A program in DDM is a collection of code blocks. 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. A producer/consumer relationship exists among threads. In a typical program, a set of threads, called the producers, create data used by other threads, called the consumers. Scheduling of threads within a code block is done dynamically at run time by the TSU, based on data availability. The instructions within a thread are executed sequentially in control-flow order.

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. 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. The size of this field is 32-bits.

- **Ready Count**: The number of thread dependencies not yet satisfied. This value is initialized with the total number of producers for this thread. A thread is ready for execution when its Ready Count is zero. The size of this field is 4-bits, limiting the maximum number of producer threads to 15.
- **Data Frame Pointer (DFP)**: This is a pointer to the data frame assigned for the thread/code block. The size of this field is 32-bits.
- **Consumer Threads**: These are pointers to the two consumers of the thread. If a thread has more than two consumers, then the first consumer field is set to zero and the second is a pointer to the consumer's list in a memory block within the TSU. The size of this field is 16-bits.

Figure 1 presents a DDM node consisting of the computation unit (conventional microprocessor with memory and cache) and the Thread Synchronization Unit (TSU). The TSU is attached to the processor's bus as a memory-mapped unit, and therefore there is no need for modifying the processor or adding extra instructions. The processor communicates with the TSU via a snooping unit that intercepts the memory references to the TSU addresses and directs them to the two queues: the *Ready Queue (RQ)* and the *Acknowledgement Queue (AQ)*. The RQ contains pointers 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 synchronization template of each thread. The SM contains the Ready Count values for each thread. Multiple instances of a thread such as loop iterations are assigned a separate entry in the SM.

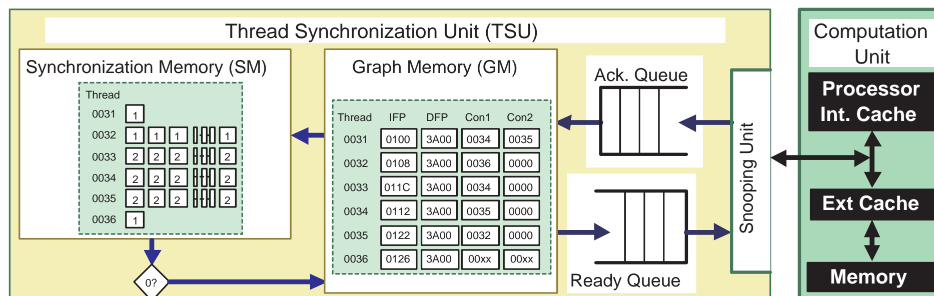

Fig. 1. A DDM processing node

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 the same function. The Block field identifies the code block, while the ThreadID identifies the thread within the code block.

The processor reads the address (IFP) 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.

2.1. *A DDM Example*

The DDM execution model can be described with a simple example such as the vector-matrix multiplication. Figure 2-a 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.

Figure 2-b 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. Non shaded boxes represent instances of values. Solid arrows represent data dependencies as well as the movement of data between threads, while dashed arrows represent only data dependencies among threads. Each processing node can execute threads with different instances from both code blocks.

The function of *CBLOCK1* is to create the new column indexes *C_Index* and fire thread *new_context* that loads on the TSU a new instance of 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. The function of this switch thread 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 Figure 2-b 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. Figure 2-c 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

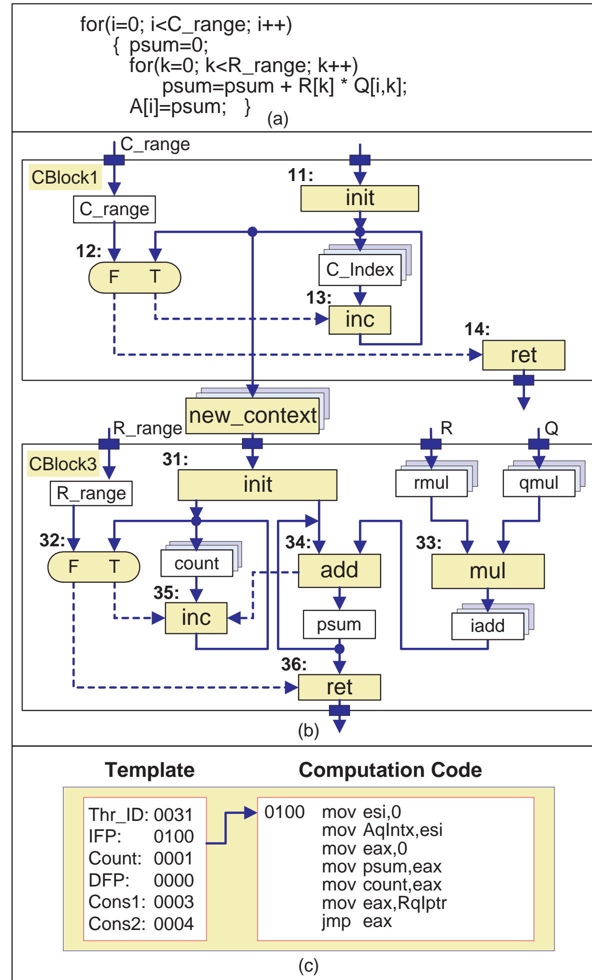

Fig. 2. DDM vector-matrix multiplication example

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 Figure 2, for simplicity, we assigned a simple arithmetic operation to each thread. In real applications a thread is assigned many more computation tasks.

3. The D^2-CMP

D^2-CMP is a chip multiprocessor implementation of the DDM execution model. A D^2-CMP chip consists of a number of execution cores, each associated with a level-1

cache and a TSU. A system interconnection network is also embedded in the chip. Figure 3 depicts the implementation of a 4CPU-4TSU D^2-CMP connected to an external shared memory.

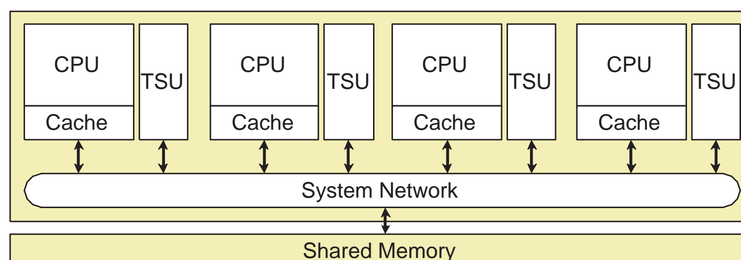

Fig. 3. 4CPU-4TSU D^2-CMP

Figure 4 depicts the internal structure of a TSU attached to the bus of a single processor. The TSU consists of two functional units: the Post Processing Unit (PPU) and the Thread Issue Unit (TIU). The PPU's function is to process the synchronization activities required by the threads that have just been executed by the processor, and determine which threads are ready for execution based on data availability. The TIU's function is to process the threads deemed executable by the PPU. The TIU provides the processor with the instruction frame pointer and the data frame pointer of the threads that are ready for execution. It should be noted that the TSU operates asynchronously with the processor, i.e. while the processor executes a given thread, the TSU operates on other threads. Furthermore, the operation of the two units of the TSU is also asynchronous, i.e. while the PPU operates on a given thread, the TIU operates on another.

4. D^2-CMP Prototyping using FPGAs

For validation and evaluation purposes, a 2CPU-2TSU D^2-CMP system has been prototyped on the Xilinx ML310 embedded development platform, equipped with a Xilinx xcv2p30 Virtex-II Pro FPGA [9]. The xcv2p30 device is a good platform for rapid prototyping of multi-core systems because it provides two embedded PowerPC microprocessors, hardwired on the FPGA chip, plus adequate FPGA fabric for implementing the TSU and the interconnection network. The prototype has been developed using the Xilinx Embedded Development Kit (EDK) [10]. The TSU was designed as an Intellectual Property (IP) block using reusable VHDL [11]. The bus and memory sizes were parameterized, facilitating easy modification and creation of various TSU versions in hardware, from rapid prototyping to ASIC implementation. A number of testbenches were developed in order to verify the functionality for various memory sizes and configurations. For further exploration, the TSU prototype was instantiated as a Processor Local Bus (PLB) slave.

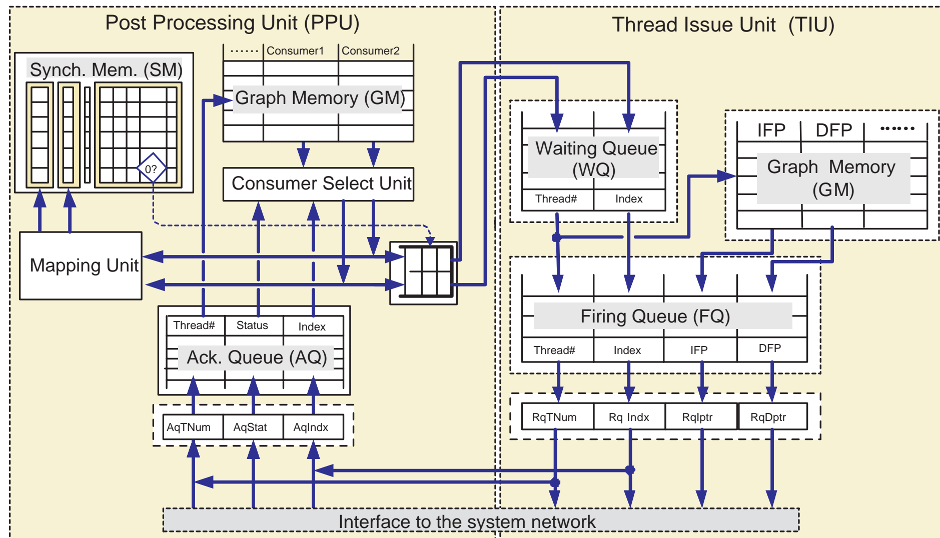

Fig. 4. Internal structure of the TSU

Figure 5 depicts the block diagram of the prototype that was generated automatically by the Xilinx Embedded Development Kit. The diagram shows the two PowerPC processors, each one with its own TSU attached to the Processor Local Bus (PLB). Each processor has its own on-chip distributed memory. The two processors have access to the external shared memory attached to the same PLB. The present D^2-CMP prototype is a bus-based multiprocessor. We are currently developing a crossbar-based network [12], to be used as the interconnection network of the system.

The size of the hardware structures of the TSU implemented on the prototype are presented in Table 1. These sizes are based on the requirements determined during the simulation of the D^{2}NOW [4]. Table 1 shows also the number of embedded RAM blocks (BRAMs) required for each structure.

Table 2 shows the FPGA utilization for the implementation of the TSU prototype, both in terms of the actual hardware needed, as well as in terms of the available hardware on the FPGA. The hardware device utilization shows that the FPGA can easily accommodate the 2CPU-2TSU implementation of the D^2-CMP. The total ASIC equivalent gate count for the implementation of the TSUs and the system interconnect is 1.9 million gates. This figure is provided directly by the Xilinx Embedded Development Kit (EDK).

5. D^2-CMP Evaluation

In order to evaluate the performance of the D^2-CMP prototype, a clock cycle counter is included in the TSU hardware prototype. All results are expressed with respect to the number of PowerPC clock cycles that run at 100MHz. To ensure that the

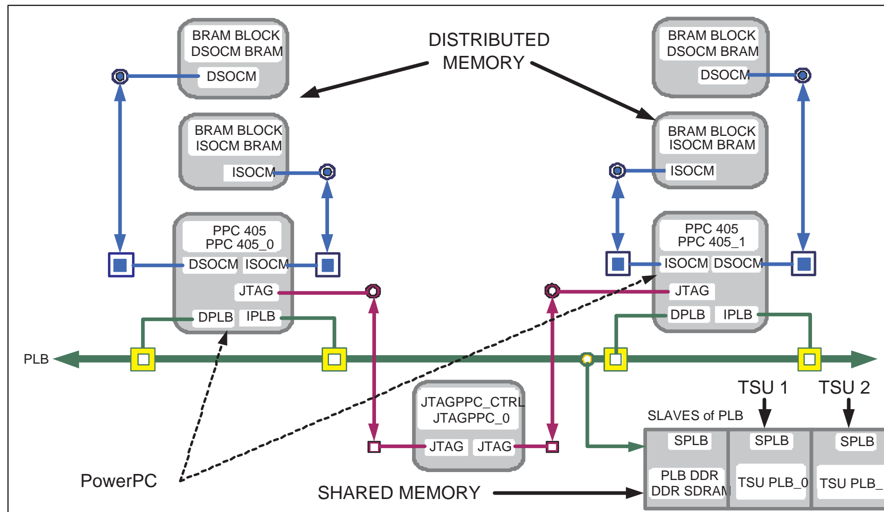

Fig. 5. Block diagram of the 2CPU-2TSU prototype

Table 1. TSU prototype structures size

Memory	Number of Entries (1 TSU)	Memory Size (bytes) (2 TSUs)	Number of BRAM (2 TSUs)
Graph Memory	256	4 KByte	8
Synchronization Memory	64	1 KByte (CAM)	8
Acknowledgement Queue	16	1 KByte	4
Waiting Queue	16	1 KByte	4
Firing Queue	16	1 KByte	4

Table 2. TSU FPGA prototyping device utilization

Resource	Number/(Total Available)	% Utilization
Slices	6,025/13,696	45%
Slice FFs	4,968/27,392	18%
LUTs (4-input)	7,700/27,392	28%
BRAMs	28/136	20%
Equivalent Gates	1,939,188	- -

results are as accurate as possible we have repeated each experiment four times and computed the average number of cycles. Although the number of experiments is small, the deviation between measurements for the same experiment was always within 3%. Table 3 shows the measurements taken for the evaluation of the TSU prototype. These measurements take into consideration the clock counter overhead, i.e., the time needed to start and stop the cycle counter. The first column in Table 3 shows the number of CPU cycles required for the completion of the event measured. The second column shows the number of cycles after subtracting the counter overhead. The third column shows the number of cycles needed for the completion of the event, normalized with respect to the number of cycles needed to execute a fine-grained thread that performs a multiply-and-accumulate (MAC1).We use the MAC operation as a reference as it is commonly used in scientific applications. Its execution time can be found in row 4.1 of Table 4 as we also use the MAC operation in our own Matrix Multiplication benchmark.

Table 3. DDM and TSU overheads

Row#	Event	Cycles Measured	Cycles (without counter overhead	Time/MAC time
3.1	Counter Overhead	74	0	0
3.2	TSU Overhead	101	27	0.06
3.3	DDM Overhead	157	83	0.18
3.4	TSU Initialization	2,336	2,262	4.83

For the 2CPU-2TSU prototype, the PowerPC processor clock frequency is set to 100 MHz and the bus clock frequency to 50 MHz, hence the TSUs operate at half the speed of the processor. This is due to the limitations of the speed of the FPGA blocks. On a future ASIC implementation, it is expected that the processors and the TSUs will operate at the same frequency. This speed imbalance does not create any significant drawbacks on the performance of the system, since the TSU operates asynchronously with the CPU. The only drawback resulting from this difference in frequency is that the processor needs more cycles to access the memory mapped registers of the TSU. This increases the number of cycles needed for the execution of computation threads as well as the number of cycles needed to initialize the TSU structures such as the GM and the SM. The TSU critical path is identified in the control logic state machine. This critical path can be pipelined, therefore increasing the operating frequency, at the penalty of one additional clock cycle latency.

The number of CPU clock cycles needed by the TSU to process a thread with a single consumer is 27 (row 3.2 in Table 3). This refers to the number of cycles needed by the TSU to read the thread's information from the AQ, determine its consumer,

decrement its ready count, and load the RQ with the identification information of the consumer thread. This latency can easily be tolerated since the TSU operates in parallel and asynchronously with the CPU. Furthermore, as shown in Table 4, the number of cycles needed for the execution of a simple fine grained thread such as the multiply-and-accumulate thread is 468 cycles, which is much higher than the TSU overhead. The DDM overhead (row 3.3 in Table 3) refers to the overhead due to the execution of the instructions inserted in a thread to enable the implementation of the DDM model. These include the instructions that read the instruction frame pointer, data frame pointer, and index of the thread to be executed from the RQ, as well as the instructions that write the status of the completed thread to the AQ, and the branch to the next thread. This is equivalent to the number of cycles needed to execute an empty thread, i.e., a thread that does not perform any computation. The DDM overhead is 83 cycles, which is about one-fifth of the time needed by the processor to perform a MAC operation. This overhead is expected to be reduced to a half, in a real ASIC implementation where the processor and the TSU will operate at the same frequency. Furthermore, this overhead can be easily amortized by increasing the thread's granularity. For example this overhead for the execution of a thread with 16 iterations of the inner product of the matrix multiplication (see row 4.2 in Table 4) is only 2%.

The results in row 3.4 show the TSU initialization overhead. This is the number of CPU cycles needed to load the Graph Memory with the thread's template and the Synchronization Memory with the ready counts for a whole block. The time needed to initialize the TSU is high, compared to the time needed to execute a single fine grain thread such as the MAC1 thread. Nevertheless, this overhead is only 4.7% of the time needed to execute 16 thread iterations of the matrix multiplication benchmark (row 4.4 in table 4). Furthermore, it should be noted that loading the Synchronization and the Graph memories of the TSU is currently performed by the processor, by explicitly copying the necessary information from the memory to the TSU structures. The processor can be released from this task by using a DMA controller.

Table 4. Experimental results for the matrix multiplication benchmark

Row#	Event	Cycles	Cycles Normalized to MAC1	Cycles Normalized to MAC16
4.1	Single point per thread (MAC1)	468	1	- -
4.2	1 thread, 16 points per thread (MAC16)	4,170	8.91	1
4.3	8 threads, 16 points per thread	25,390	54.25	6.09
4.4	16 threads, 16 points per thread	47,368	101.21	11.36

The results shown in Table 4 correspond to the matrix multiplication benchmark.

Row 4.1 shows the number of cycles needed for the execution of a single thread that performs a multiply-and-accumulate operation (MAC1). The number of cycles needed for the execution of MAC1 includes the cycles due to the actual computation as well as the cycles due to the DDM overheads and the cycles due to cache misses. It should be noted that no data is placed in the cache prior the invocation of the MAC1 thread, hence all memory references result in cold cache misses. Measurements show that out of the 468 cycles needed for the execution of MAC1, 110 cycles correspond to the DDM and TSU overheads, and 328 cycles correspond to the cold cache misses. The remaining 30 cycles correspond to the execution of the actual multiply-and-accumulate code, which is basically the instruction *sum = sum + R[i]*Q[j]*. This code corresponds to 16 PowerPC assembly language instructions, out of which six are memory reference instructions (Table 5).

Table 5. Assembly listing of the MAC1 computation code

Address	PowerPC Assembly Code
0xffff0a8c	lwz r0,100(r31)
0xffff0a90	rlwinm r9,r0,2,0,29
0xffff0a94	addi r0,r31,8
0xffff0a98	add r9,r9,r0
0xffff0a9c	addi r9,r9,632
0xffff0aa0	lwz r11,0(r9)
0xffff0aa4	lwz r0,100(r31)
0xffff0aa8	rlwinm r9,r0,2,0,29
0xffff0aac	addi r0,r31,8
0xffff0ab0	add r9,r9,r0
0xffff0ab4	addi r9,r9,664
0xffff0ab8	lwz r0,0(r9)
0xffff0abc	mullw r9,r11,r0
0xffff0ac0	lwz r0,108(r31)
0xffff0ac4	add r0,r0,r9
0xffff0ac8	stw r0,108(r31)

Row 4.2 shows the number of cycles needed for the execution of a single thread that performs 16 multiply-and-accumulate operations (MAC16). Rows 4.3 and 4.4 show the number of cycles needed for the execution 8 and 16 MAC16 threads respectively by a single processor. From these results it can be seen that the number of cycles does not increase linearly with respect to the increase in the number of operations performed. This is due to the amortization of the DDM overheads and the TSU initialization overheads. Furthermore, increasing the thread granularity by performing more multiply-and-accumulate operations in a single thread, exploits lo-

cality and reduces cache misses. Locality is further exploited by executing multiple threads that reuse the same data values.

Figure 6 shows the effect of thread granularity on the number of cycles needed for the execution of (a) a 256-point matrix multiplication benchmark, and (b) a 256-point Splash-2 FFT benchmark. For both benchmarks we have increased thread granularity using loop unrolling. To obtain comparable results, the same loop unrolling factor has been applied to both the DDM as well as the sequential code. By increasing the Thread granularity the execution time (measured in CPU cycles) decreases because the DDM overhead is applied to larger threads. The DDM overhead is fully amortized when the thread granularity is 16 operations per thread.

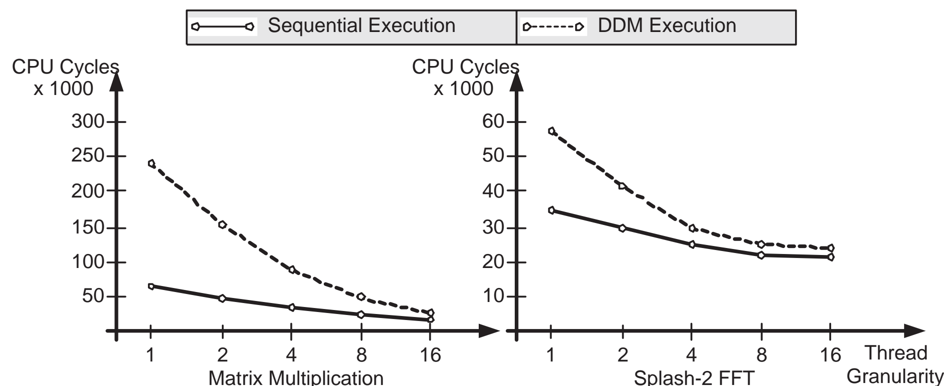

Fig. 6. Effect of thread granularity on overhead amortization

Large scale benchmarks are currently been ported on the FPGA-based D^2-CMP prototype. Furthermore, we are currently developing performance monitors on the FPGA that will enable us measure quantities such as the actual number of cache misses during the execution of a benchmark.

6. Related Work

The traditional von Neumann model of execution used in most current systems leads to severe limitations, such as large needs for communication and synchronization as well as poor memory performance. Currently, several research projects have adopted the dataflow principles. Explicit Data Graph Execution (EDGE) [13] proposes a new instruction set architecture that enables the scaling to window sizes of thousands of instructions to hold large code blocks. EDGE uses direct instruction communication, i.e. a producer instruction delivers its produced data directly to its consumer instructions, thus enabling instructions to execute in dataflow order, with instruction firing as soon as all of their operands are available. The TRIPS processor [14] is an instance of the EDGE architecture which uses large cores consisting of a matrix of execution units.

Another threaded dataflow project is the Scheduled Dataflow (SDF) [15, 16] architecture. SDF is a multithreaded architecture that decouples the synchronization from the computation of non-blocking threads. SDF has its origins in the PL/PS-Machine (Pre-load/Post-store), a multithreading machine that decouples memory accesses from thread execution. An SDF processor consists of two pipelines: the execution pipeline and the synchronization pipeline. The synchronization pipeline is responsible for scheduling the non-blocking 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 results from an execution are stored by the execution pipeline in registers and then post-stored in the memory by the synchronization pipeline. This decoupling eliminates stalls due to memory accesses and cache misses.

WaveScalar [17] is a dataflow instruction set architecture and execution model designed for scalable, low-complexity/high-performance processors. WaveScalar investigates the area/performance tradeoffs of a tiled dataflow architecture. In the WaveScalar ISA each instruction executes in place in the memory system and explicitly communicates with its dependents in dataflow fashion. The architecture caches instructions and their values in a simple grid of "alu-in-cache" nodes called the WaveCache.

Microthreading is a static block interleaving multithreading model with explicit switching introduced in [24]. Thread switching is specified by the compiler (i.e. static or compiler assisted) by explicitly tagging the instructions where a context switch should take place. Such instructions are the conditional branch instructions and the instructions that use data that the compiler cannot guarantee that will be available in the cache. Instructions tagged to cause thread switching always do so, irrespective if the data is available or not, unless if there are no other threads able to run, i.e. all other threads are suspended. A microthreaded CMP [25] consists of a number of microthreaded pipelines connected via two shared communication systems.

The main difference of the above architectures with the DDM model and the D^2-CMP implementation is that they consist of a large number of simple execution units dedicated to support a specific ISA, while the DDM model can be implemented using any off-the-shelf high performance processor core. Another difference is that the above architectures support the execution of short sequences of instructions, i.e. fine-grain threads, while the DDM model supports the execution of threads with any granularity.

Another processor that has its origins in the dataflow model is the Fuce chip-multiprocessor [18], where the execution of threads is triggered using continuations, an event-driven coarse-grain execution model. The Fuce CMP consists of a small number of very simple RISC processor, acting as the main computation units, associated with a small RISC core called the preloading unit. The main difference between

the Fuce processor and the D^2-CMP is that in the Fuce model, thread synchronization is achieved by inserting the synchronization in the thread's code, executed by the preloading unit, while in the D^2-CMP the synchronization is achieved with the use of the synchronization memory.

FPGAs have been traditionally used for system rapid prototyping for years and a number of architectures for multiprocessor system hardware emulation have been developed, mostly utilizing external processor cores, while FPGAs are used to implement the interconnect, glue logic and hardware controllers [20, 21, 22]. RPM [19] is an FPGA-based rapid prototyping system for multiprocessor emulation. RPM is built using off-the-shelf processors, while the memory, cache, and communication controllers are implemented with FPGAs. The RPM system consists of eight computation node boards and an I/O board connected to a standard Futurebus+ backplane. FAST [20] is a hybrid hardware emulation environment that FPGAs, microprocessors and memory. It can be used for prototyping CMPs, and other novel computer architectures, as well as chip-level memory systems. FAST is implemented on a single printed circuit board consisting of four MIPS microprocessors, FPGA-based reconfigurable hardware, a SRAM-based memory hierarchy and an interconnection network. The RAMP [23] project aims to accelerate research on multiprocessor systems through the emulation of parallel processor prototypes. It presents a general framework for FPGA-based system that emulates components from disks to CMPs.

7. Conclusions

In this paper, we presented the hardware prototype of a 2-node chip multiprocessor implemented on a Virtex-II Pro FPGA. This prototype showed that a DDM chip multiprocessor can be implemented with a moderate hardware budget, a 2-node chip multiprocessor has been implemented on a single FPGA using less than 50% of the FPGA fabric, an equivalent of 1.9 million gates. Measurements on the prototype have shown that the DDM overheads can be tolerated due the asynchronous operation of the processor and the TSU. We are currently developing an 8-core D^2-CMP prototype on a Virtex-4 FPGA platform, based on the MicroBlaze soft processor core from Xilinx.

References

[1] Arvind, and R. S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture", In IEEE Transactions on Computers, 39(3), pp.300-318, (1990).
[2] J. Dennis and G. Gao, "An efficient pipelined dataflow Processor Architectures", In proceedings of Supercomputing '88, Orlando, Florida, pp.368-373, (1988).
[3] J. Gurd, C. Kirkham, and I. Watson, "The Manchester Prototype Dataflow Computer", Communications of the Association of Computing Machinery, 28(1), pp.34-52, (1985).
[4] P. Evripidou, and C. Kyriacou, "Data Driven Network of Workstations (D^2NOW)", Journal of Universal Computer Science (J.UCS), Vol. 6, Issue 10, pp.1015-1033, (2000).

[5] K. Stavrou, C. Kyriacou, P. Evripidou, and P. Trancoso, "Chip multiprocessor based on data-driven multithreading model", International Journal of High Performance Systems Architecture (IJHPSA),Vol. 1.Issue 1, pp.34-43, (2007).
[6] P. Evripidou, "Thread Synchronization Unit (TSU): A building block for High Performance Computers", In Proceedings of the ISHPC, pp.405-414, Japan. (1997).
[7] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-Driven Multithreading Using Conventional Microprocessors", IEEE Trans. on Parallel and Distributed Systems, Vol.17, No. 10, pp.1176-1188, (2006).
[8] C. Kyriacou, "Data Driven Multithreading using Conventional Control Flow Microprocessors", PhD dissertation, University of Cyprus (2005).
[9] http://www.xilinx.com/products/silicon_solutions/virtex_ii_pro_fpgas/index.htm
[10] http://www.xilinx.com/ise/embedded/edk91i_docs/edk_ctt.pdf
[11] M. Keating, and P. Bricaud, "Reuse Methodology Manual for System-On-A-Chip Designs", Springer (2006).
[12] R. Kumar, and et. al. "Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling", In: Proceedings of ISCA '05, pp.408-419, (2005)
[13] D. Burger, and et. al. "Scaling to the End of Silicon with EDGE Architectures", Computer 37(7), pp.44-55 (2004).
[14] K. Sankaralingam, and et. al. "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture", SIGARCH Computer Architecture News 31(2), pp.422-433, (2003).
[15] K.Kavi, R.Giorgi, and J.Arul, "Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation", IEEE Transactions on Computers, 50(8), pp.834-846, (2001).
[16] J.Arul, and K. Kavi, "Scalability of Scheduled Dataflow Architecture (SDF) with Register Contexts", ICA3, pp.02-14, (2002).
[17] Swanson, S., and et. al. "Area-Performance Trade-offs in Tiled Dataflow Architectures", In Proceedings of ISCA '06, pp.314-326, (2006).
[18] S. Amamiya, and et. al. "Fuce: the continuation-based multithreading processor", In Proc. of the 4th International Conference on Computing Frontiers, pp.213-224, Italy (2007).
[19] L.A. Barroso, and et. al. "RPM: A Rapid Prototyping Engine for Multiprocessor Systems". IEEE Computer Magazine. 28(2), pp.26-34, (1995).
[20] J.D. Davis, et. al. "A Chip Prototyping Substrate: the Flexible Architecture for Simulation and Testing (FAST)", SIGARCH Computer Architecture News 33(4), pp.34-43, (2005).
[21] R. C. Cofer, and B. F. Harding, "Rapid System Prototyping with FPGAs: Accelerating the Design Process", Newnes (2005).
[22] S. Hauck, G. Borriello, and C. Ebeling, "Mesh Routing Topologies for Multi-FPGA Systems", IEEE Transuctions on VLSI Systems, Vol. 6, No. 3, pp.170-177, (1998).
[23] S. Wee, and et. al. "A Practical FPGA-based Framework for Novel CMP Research", FPGA'07, pp.116-125, California, (2007).
[24] A. Bolychevsky, C. R. Jesshope, and V. Muchnick, "Dynamic Scheduling in RISC Architectures", IEE Proc. Comput. Digit. Tech. 143, pp.309-317, (1996).
[25] K. Bousias, N. Hasasneh, and C. Jesshope, "Instruction Level Parallelism through Microthreading - A Scalable Approach to Chip Multiprocessors", The Computer Journal, Vol. 49 No. 2, pp.211-233, (2006).