

# A Multithreaded Soft Processor for SoPC Area Reduction

Blair Fort, Davor Capalija, Zvonko G. Vranesic and Stephen D. Brown  
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering  
University of Toronto  
10 King's College Road  
Toronto, ON, Canada M5S 3G4

{fort,davor,zvonko,brown}@eecg.toronto.edu

## ABSTRACT

The growth in size and performance of Field Programmable Gate Arrays (FPGAs) has compelled System-on-a-Chip (SoPC) designers to use soft processors for controlling systems with large numbers of intellectual property (IP) blocks. Soft processors control IP blocks, which are accessed by the processor either as peripheral devices or/and by using custom instructions (CIs). In large systems, chip multiprocessors (CMPs) are used to execute many programs concurrently. When these programs require the use of the same IP blocks which are accessed as peripheral devices, they may have to stall waiting for their turn. In the case of CIs, the FPGA logic blocks that implement the CIs may have to be replicated for each processor. In both of these cases FPGA area is wasted, either by idle soft processors or the replication of CI logic blocks.

This paper presents a multithreaded (MT) soft processor for area reduction in SoPC implementations. An MT processor allows multiple programs to access the same IP without the need for the logic replication or the replication of whole processors. We first designed a single-threaded processor that is instruction-set compatible to Altera's Nios II soft processor. Our processor is approximately the same size as the Nios II Economy version, with equivalent performance. We augmented our processor to have 4-way interleaved multithreading capabilities. This paper compares the area usage and performance of the MT processor versus two CMP systems, using Altera's and our single-threaded processors, separately. Our results show that we can achieve an area savings of about 45% for the processor itself, in addition to the area savings due to not replicating CI logic blocks.

## 1. INTRODUCTION

The opportunity for larger and more complex System-on-a-Chip (SoPC) designs has been facilitated through the growth in size and performance of FPGAs. These large systems usually contain several IP blocks. To manage the complexity of these systems, SoPC designers utilize soft processors to control the IP blocks. This allows them to write simple software programs for the processor to communicate with and control the IP blocks. In the case of a system comprising a large number of IP blocks, the designer can either create a complex program or multiple simple programs to control the IP blocks. Multiple programs have the advantage over single complex programs due to the simplicity of their creation and debugging. In order to run

multiple programs on a uniprocessor system, software-based context switching is required. This is usually facilitated by the addition of an embedded operating system. On the other hand, one can utilize a CMP system which enables multiple programs to execute simultaneously without support for context switching.

A system can have IP connected to a processor as a peripheral or as a custom instruction (CI). A peripheral allows multiple processors to access it, at the cost of performance drop caused by the overhead of using memory instructions for communication, which have inherent latency. Moreover, if multiple processors concurrently access the same IP block, one of the processors is granted access to the IP block, while others wait idle. On the other hand, custom instructions are embedded into a processor's datapath and read and write data from the processor's register file. This can result in lower data access latency and consequently in higher throughput. However, in a multiprocessor system, each processor executing a program which requires the CI will contain its own copy of CI logic. This can be detrimental for the area usage if the employed custom instruction consumes a large amount of logic and is used by multiple programs.

This paper introduces a multithreaded soft processor and evaluates it versus other soft processing systems that provide multithreading support. We compare the area usage and performance of a uniprocessor, two CMP systems and an MT processor. The uniprocessor and one CMP system are based on the Altera Nios II processor. The processors employed in the second CMP system and the MT processor were designed by us. We investigate architectural enhancements for improving the performance of the multithreaded systems; more specifically, a mechanism for performance improvement in the presence of multicycle custom instructions and shared peripheral devices. There are three main contributions of this paper. First, it gives a comparison of an MT soft processor versus a single threaded processor and CMP systems. Second it proposes architectural enhancements specific to FPGA-based multithreaded systems. Lastly, it presents an approach for performance evaluation of soft multithreaded systems conducted in real time on an FPGA.

The remainder of this paper is structured as follows. First, related work is discussed in Section 2. Section 3 presents necessary background information. Section 4 describes the architecture of our single-threaded UT II and multithreaded UTMT II processors. Our experimental environment and methodology are explained in Section 5. Section 6 presents

the obtained experimental results. Lastly, in Section 7, we give conclusions and propose future work.

## 2. RELATED WORK

Soft processors for the FPGA platform have become a focus of research. Currently, the major FPGA companies, Altera and Xilinx, provide the in-order soft processors Nios/NiosII[1, 2] and Microblaze[3], respectively. In the design of SoPC systems, these processors have rapidly grown in popularity due to their flexibility and ease of integration into the system. On the other hand, the trend of supporting multithreaded execution has been taking place not only in high-performance and desktop systems but in embedded systems as well. In both worlds the motivation is to leverage increased presence of multithreaded workloads. Several embedded processors with multithreading support are commercially available. While workloads for embedded systems based on ASIC and FPGA platforms tend to be similar, hardware-based multithreading capabilities on FPGA platforms targeting multithreaded workloads have not yet been fully explored.

### 2.1 Soft processor design

In the past few years, significant research efforts have been made in the area of soft processors. Plavec et al. [4] present a methodology for efficient soft processor design. They produced a generic processor which is instruction set compatible with Altera Nios processor but synthesizable in most modern FPGAs. Recent research addressed the design of high-performance functional units for programmable logic. Metzgen [5, 6] describes the design of the high-performance 32-bit ALU which is a critical component of the Altera Nios II processor. The novel design of the ALU results in significant area savings and clock frequency increase for the Nios II over its predecessor, the Nios 1.1 processor. Our processors described in this paper follow the author's guidelines. Yianacouras et al. [7] present a framework for automatic generation of custom soft processors to explore the microarchitecture trade-off space. The authors discuss the performance and area impact of various functional unit implementations and pipeline depths. In particular, they investigate area versus performance trade-off for the shifter unit implementation and demonstrate the detrimental effects of pipeline stalls arising from multicycle operations on performance.

### 2.2 Multithreading

System-on-chip designs providing multithreaded processing are built with the aim to offer better area and power efficiency compared to multiprocessor systems. There are two approaches for supporting hardware-based on-chip multithreading. The first approach is CMP where multiple processors are placed on a single chip. The second approach is to augment a uniprocessor with additional hardware to allow multiple threads to share the processor resources. In this architecture the additional hardware maintains the thread state and performs context switching between threads. The two main classes of uniprocessor multithreading are fine-grained multithreading (FGMT) and simultaneous multithreading (SMT). FGMT processors allow only one thread to issue instructions in a given clock cycle, whereas SMT processors allow instructions from multiple threads to be issued in the same clock cycle. Tullsen et al. [8] have shown that supporting either class of mul-

tithreading can improve processor throughput at the expense of some minimal additional hardware. The performance improvement stems from higher pipeline utilization in the presence of cache misses, pipeline hazards and branch mispredictions. Some commercially produced FGMT processors include HEP [9] and Tera [10]. These processors allow up to 128 concurrent hardware threads. Systems utilizing these processors have had limited success, mainly due to their inability to achieve reasonable single thread performance. This inability holds true for our processor as well, but since our design is targeting FPGAs, it can be used in the presences of suitable multithreaded workloads, otherwise it maybe easily substituted with a uniprocessor.

Multithreaded architectures targeting application-specific workloads have recently become apparent. A study by Crowley et al. [11] characterizes processor architectures used in networking technology. The studied architectures range from a superscalar out-of-order processor through a chip multiprocessor, an FGMT processor to an SMT processor. The authors observe that workloads comprising a sufficient level of parallelism result in better performance on the SMT processor for the given processor resources.

The bulk of the past research was aimed at superscalar out-of-order processors for ASIC technology. However, commercially available FPGA-based soft processors are predominantly simple in-order single-issue processors. Consequently, it is not obvious whether the observed benefits and performance improvements will apply to these FPGA-based soft processors.

### 2.3 Multithreading on FPGAs

As previously investigated, multithreading allows better use of the chip area and power efficiency on an ASIC platform. Research efforts to exploit these benefits in FPGA-based systems have been limited. So far, two approaches have been explored to deliver soft processors with multithreading support[12, 13]. Dimond et al. [12] introduced the customizable threaded architecture (CUSTARD). CUSTARD is a parameterizable processor that combines support for multiple threads and automatic instruction set customization. Automatic instruction set customization is provided by a C compiler that targets the CUSTARD instruction set. The CUSTARD architecture can be customized to support two multithreading types: block multithreading (BMT) and interleaved multithreading (IMT). The authors investigate architectural improvements that can be applied to multithreaded processors. Most notably, they demonstrate the area savings by the elimination of data forwarding logic that is no longer required. The results show that a four-threaded IMT processor produces better results than a BMT processor in terms of clock cycles, area usage and maximum frequency. While the IMT processor provides higher throughput for a given chip area, it is not clear how the processor compares to available industrial or academic soft processors in terms of area and speed. The authors compare their 4-threaded IMT and BMT configurations against their single-threaded processor. They report a relatively small area increase (28% and 40%) for the multithreaded configuration as compared to their single-threaded configuration. However, the area of their single-threaded processor is approximately 1800 slices and maximum frequency obtained is around 30 MHz. Commercially available processors usually consume less area and achieve higher frequencies.

Weaver et al. [13] take an entirely different strategy to automatically generate a soft processor with multithreading capabilities. They applied C-slow circuit retiming to the Leon 1 SPARC compatible soft processor. Circuit retiming is a process of rearranging storage elements in a synchronous circuit without altering the combinational logic in order to optimize the circuit’s performance. C-slow retiming is a technique where each register is replaced with a sequence of C separate registers prior to retiming. Applying C-slowing to a processor enables interleaved execution of C distinct instruction streams. The resulting design provides higher throughput as long as C distinct instruction streams are fed into the pipeline. To fully achieve multithreading capabilities one must manually increase the instruction fetch bandwidth by modifying the memory controller and caches. Modifications include architectural changes to keep track of which data corresponds to which thread. Any particular thread runs slower due to the increased pipeline latency but the overall system runs at a significantly higher clock rate, thus increasing the throughput if programs can take advantage of the multithreading capabilities. For FPGA implementations, a 2-slow design is considerably more efficient than 2 distinct processor cores because most processors contain considerably more logic than registers. This additional efficiency is due to the FPGA design procedure called register packing. Their C-slow retiming method is applied to circuits in the post-placement phase which could limit the ability to manage the design of the whole multithreaded system. While the authors show that throughput can be increased, they do not give a detailed performance evaluation of the resulting processor on standard workloads.

### 3. BACKGROUND

#### 3.1 Altera’s Nios II

Nios II [2] is Altera’s second soft processor. It is a general-purpose RISC soft processor with 32-bit instruction words and datapath, integer only ALU, 32 general purpose registers and MIPS-style instruction format. The Nios II processor family comprises three different processor cores: Economy (Nios II/e), Standard (Nios II/s) and Fast (Nios II/f). Table 1 [6, 14] highlights the main characteristics of the cores.

All three cores are single issue in-order execution processors. The Nios II/f is highly optimized for performance, whereas on the other end of the scale, Nios II/e was designed to achieve the smallest possible core size. The Nios II/e saves logic by allowing only one instruction to be in flight at any given moment. This eliminates the need for data forwarding and branch prediction logic, among other things. The shifter unit implementation differs across all three cores. The Nios II/e has a serial shifter, while the other two processors use a barrel shifter if the FPGA contains digital signal processing (DSP) blocks with hardware multipliers.

#### 3.2 Custom Instructions

The CI feature allows designers to add their own functional units to a Nios II processor. The source operands of CIs can be registers if required by the design. In addition, CIs can connect to signals outside the processor. The Nios II processor can support up to 256 custom instructions.

Features	Nios II Cores		
	Economy	Standard	Fast
Area (LEs)	<700	<1400	<1800
FMax	≤ 150 MHz	≤ 135 MHz	≤ 135 MHz
Performance	1x	4.7x	7.5x
Pipeline	5-cycle	5-stage	6-stage
Instruction Cache	No	Yes	Yes
Data Cache	No	No	Optional
Branch Prediction	None	Static	Dynamic
Shifter Unit	1 cycle-per-bit	3-cycle barrel shifter	1-cycle barrel shifter
Multiply Unit	Software Emulation	3-cycle	1-cycle

Table 1: Nios II Processor Cores

### 3.3 Peripheral Devices

Peripheral devices are connected to Nios II cores via the *Avalon switch fabric*, which is a collection of point-to-point master to slave connections. A master can connect to multiple slaves and a slave can be connected to multiple masters. When a slave is driven by multiple masters, arbitration logic is automatically generated by Altera’s SOPC Builder [15]. Arbitration by default utilizes a round-robin master selection, but priorities can be associated to masters and used for selection.

### 4. PROCESSOR MICROARCHITECTURE

Previous work [8] has demonstrated that FGMT and SMT processors provide similar performance improvements for 4 or less threads. The SMTs exhibit better performance if more threads are available. Moreover, recent work [7] has shown that both 3 and 4 stage pipelined soft processors are optimal in terms of performance and area. These arguments guided our design choices which resulted in a multithreaded processor designed as a 4-way FGMT processor, more specifically as an IMT processor. In the IMT processor threads are issued in a round-robin manner one thread at one time. The choice of an IMT and a four-stage pipeline provided the opportunity for elimination of feedback paths and branch logic [12]. Applying IMT specific modifications to Nios II/f and adding thread context extensions would result in the same architecture obtained by adding thread context support to the Nios II/e. It was assumed that processors have equal depth pipelines. Forwarding and branch logic can be fully eliminated only when  $n$  or more threads are introduced to a  $n$ -stage pipeline. Therefore, we first designed a 4-stage pipeline processor similar to the Nios II/e, and subsequently augmented it to enable 4-way IMT multithreading.

#### 4.1 UT II Microarchitecture

##### 4.1.1 Pipeline

UT II soft processor is an implementation of the Altera Nios II/e architecture. UT II provides the same features as Altera Nios II/e, but its physical implementation may be

quite different due to our limited knowledge of Altera Nios II/e's microarchitecture. It is implemented with a simple 4-stage pipeline without data forwarding and hazard detection. For this reason, only one instruction is allowed to occupy the pipeline at a time. Figure 1 depicts the UTMT II processor. The stages of the pipeline are the same for both processors. They are fetch, decode, execute and write back stages.

#### *Fetch Stage*

- Completes instruction fetches
- Inserts a trap instruction if an interrupt has occurred.

#### *Decode Stage*

- Decodes the current instruction
- Reads source operands from the register file
- Chooses between immediate and register data for the operands
- Increments the program counter (by 4)

#### *Execute Stage*

- Performs ALU operations
- Shifts are 1-bit per cycle (stalls pipeline until operation completes)
- Branch and jump instructions are resolved
- Control registers are read and written (if needed)

#### *Memory/Writeback Stage*

- Initiates and completes loads (can stall the pipeline)
- Initiates stores
- Initiates instruction fetch
- Writes results to the register file

### **4.1.2 Load and Store Instructions**

Both the load and store instructions start in the memory/writeback stage. Loads stall the pipeline in order to retrieve data to be written to the register file just in case it is required by the next instruction. There is a possibility that the next instruction does not require the data being loaded, but since the processor does not include any mechanisms to check if this is the case, the pipeline is stalled. On the other hand, store instructions do not write result to the register file, therefore the next instruction can start executing without performing checks. The UT II allows the next instruction to enter the fetch and decode stages, but not the execute or writeback stages. To simplify the memory buffer implementation, the instruction is not allowed to enter the execute stage, as now the buffer is reset every time the execute stage is active, since it is guaranteed that there are no outstanding loads or stores.

### **4.1.3 Shift and Rotate Instructions**

In order to minimize logic, the shift and rotate instructions are implemented using a serial shifter. This can cause a significant performance drop if multiplication is required by the application. Multiplication is emulated in software by using additions and shifts. There are two commonly used methods for increasing the shifter performance. The first method is to use a barrel shifter built using LUTs. However, as shown previously [7], this improves the performance at a high cost of consumed logic elements. The barrel shifter takes approximately 200 logic elements, or a third of the entire processor. A second method to improve the shifter performance is to use an on-chip hard multiplier.

## **4.2 UTMT II Microarchitecture**

The UTMT II processor is derived from UT II. The architectural limitations of UT II still provided many opportunities for extensions to support the multithreading. The design of multithreading is based on the fact that instructions from multiple threads are independent. Hence, instructions from different threads can be allowed to occupy different stages of the pipeline at the same point in time. This simple extension results in higher utilization of the pipeline without the necessity of adding forwarding and hazard detection logic to the pipeline. UTMT II supports the execution of 4 threads to allow potentially one instruction to be issued in every cycle. In effect, the outcome of this extension is to increase the throughput by up to four times without changing the pipeline logic. In other words, this architecture implements IMT where context switching is performed every clock cycle.

### **4.2.1 Multithreading Extensions**

Figure 1 gives a high-level overview of the UTMT II's microarchitecture. All pipeline stages require some modifications to support multithreading, as described below. Also, the control logic must be augmented to keep track of which thread is in which stage of the pipeline. A different ID is assigned to each thread. The thread ID ensures that context information for a given thread is only modified by instructions from that thread.

The fetch stage was increased to contain program counters for each hardware thread. Additional instruction masters were included to maximize the instruction fetch bandwidth. Instructions are issued from threads in round-robin manner and one thread cannot have more than one instruction in the pipeline. For this reason, a pipeline bubble can occur, if the current thread has no instructions fetched or it is stalled. Other threads are not allowed to issue in this cycle because instructions from these threads can already occupy the pipeline.

The decode stage contains the register file. The register file was replicated for all threads. Two approaches to replicating the register file were examined. First approach is to create four copies of the register file. This requires a simple modification to the pipeline, an additional multiplexer to choose between the register files. The second approach is to quadruple the number of registers in the register file and add a thread ID to the register address. A downside of this approach is that the dual-ported nature of the on-chip memory would allow only one thread to write to the registers during any given cycle, when considering the need for another thread to read for the register file during that same cycle. This is not a problem for the general case, but when the first performance optimization, discussed in section 4.2.2 is preformed, it is possible for multiple threads to simultaneously have data to be written back to the register file. Therefore, the first method of register file replication was employed.

The main concerns in the execute stage are the control registers, branch logic and custom instructions. In addition, problems can arise if custom instructions take multiple cycles to execute. The control registers were replicated, and logic was added such that threads access their corresponding control registers. Another concern is applying branches to the correct thread's program counter.

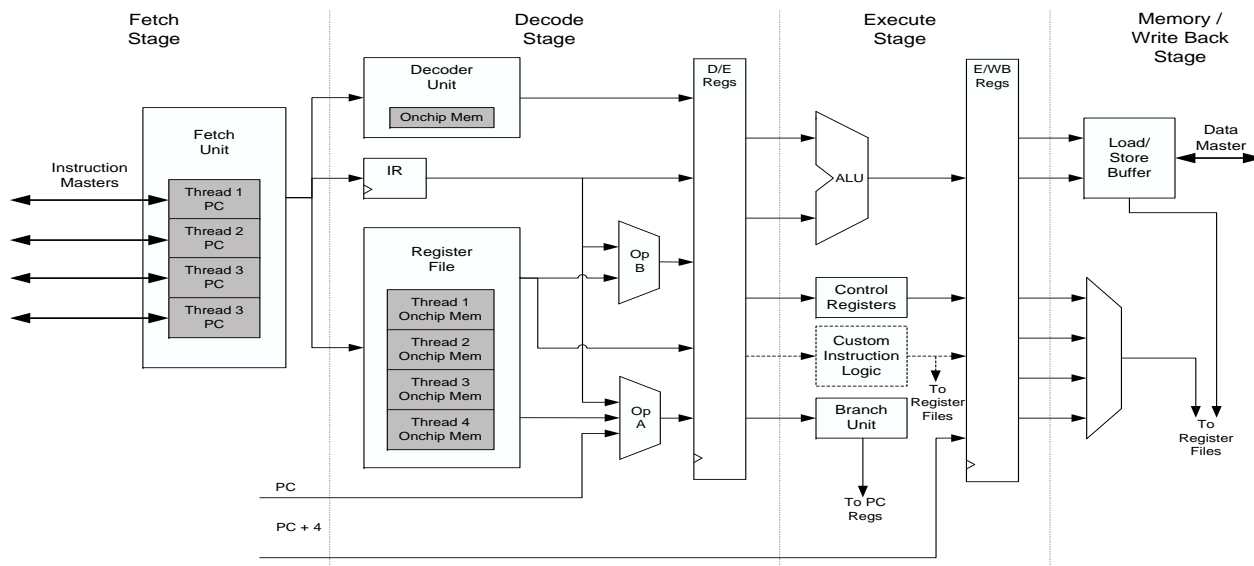


Figure 1: High-level UTMT II microarchitecture

In the memory/writeback stage logic was added to write results to the correct thread’s register file. Unlike the UT II processor, store instructions will stall the pipeline, same as for load instructions. An enhancement to minimize penalties due to load and store latencies is discussed in the following section.

### 4.2.2 Performance Enhancements

To achieve maximum performance, pipeline stalls must be eliminated. The stalls are more critical for the multi-threaded processor. This is due to the fact that stall in one thread will stall all the other threads and thus virtually disable the multithreading capabilities. We have eliminated data hazards stalls, but both accessing multi-cycle peripherals and multi-cycle custom instructions will cause pipeline stalls. We investigated two approaches for reducing these types of stalls. In both approaches pipeline stalls are limited to the thread causing the stall and other threads can continue executing. Also, stalls last until the stalled thread reaches the fetch stage for the first time after the outstanding CI or memory operation has been resolved.

The first method is generic and can be applied to all types of multicycle stalls. The method uses a queue for instructions accessing a multicycle IP block. The queue allows the processor to remove the instruction from the main pipeline allowing other threads to progress. While a thread is in the queue, it is not permitted to fetch instructions. The queue should be large enough to accommodate all threads without stalling the pipeline. Queued threads access the IP block one thread at the time. For simplicity, the register files for the different threads are in separate onchip memories to enable register write back for the instruction in the queue even when another thread is in the memory/write back stage. To evaluate this method, a FIFO buffer was added to the data memory controller. The FIFO keeps load and store instructions, thus removing them from the main pipeline, so that they do not stall the other threads, unless those threads are

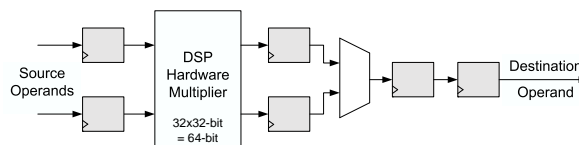


Figure 2: Multi-cycle shift unit

also in the buffer. Best performance of this enhancement requires the ability to write the result back to the register file immediately or to store it in a temporary register until the thread is in the memory/writeback stage. Our design writes directly to the register file, thus requiring replication of the register file to allow multiple threads simultaneous write access. The performance resulting from this approach can suffer when an IP block can allow access to more than one thread at a time.

A second method is applicable to IP blocks that can be pipelined and allow concurrent access to more than one thread. The main idea is to pipeline IP blocks to a multiple of  $N$ -cycles, where  $N$  is the main pipeline depth. As a result, the data becomes available when the thread is again in the same pipeline stage, thus the instruction may rejoin the main pipeline. Again, while an instruction is in the secondary pipeline, its thread may not issue any subsequent instructions. This method eliminates pipeline stalls for other threads, while improving upon the first method by allowing multiple instructions to be in flight within the IP block concurrently. To investigate the benefits of this method, we pipelined the shift unit as shown in Figure 2. As seen in the figure, a final register was added to achieve the necessary four-cycle latency for our pipeline.

The destination register’s ID is saved for each entry in the FIFO or pipelined IP. This information is used to write the result to the corresponding register after the IP has completed execution.

## 5. EXPERIMENTAL FRAMEWORK

### 5.1 Environment

#### 5.1.1 Tools

For the implementation of the hardware and software part of the experimental system the following tools were used:

- Altera Quartus II and SOPC Builder for design of a soft system and its synthesis to FPGA configuration
- ModelSim for system simulation and initial verification
- Software development environment based on the GNU C/C++ tool chain and Eclipse IDE
- Design space explorer (DSE) to obtain the  $F_{max}$  and processor area

#### 5.1.2 FPGA development board

Nios Development Board (Stratix Professional Edition) was used as the FPGA-based platform for evaluation of designed multithreaded soft processing systems. The board includes a Stratix FPGA chip and several other components listed in Table 2.

Component	Characteristics
Stratix Chip	41250 logical elements 3.5M memory bits
SRAM (off-chip)	1 MB capacity
SDRAM (off-chip)	16 MB capacity
Flash device	8 MB capacity

Table 2: Nios Development Board Characteristics

### 5.2 Metrics

To evaluate the designed systems, we used the following metrics: area, maximum frequency ( $F_{max}$ ) and wall-clock execution time. Area metric is expressed as the usage of chip resources in terms of logic (LUTs and FFs) and memory bits. In order to conduct performance evaluation and comparison, all systems were synthesized with experimental frequency  $F_{exp} = 50$  MHz. Hence, the difference in the benchmark execution times on different systems account for the difference in instructions per cycle (IPC). The difference in the IPC performance stems from various characteristics that the studied systems employ. Since the instruction count for each benchmark is the same for all systems, the obtained execution times directly account for the difference in the number of clock cycles. This metric was used to investigate the differences in processor microarchitecture. To investigate the performance which stems from the differences in IPC and  $F_{max}$ , we determine the best achievable wall-clock execution times for each benchmark on every processor. These times are obtained from results achieved with  $F_{exp} = 50$  MHz and then prorated to  $F_{max}$ . For simplicity, in the following sections the measured performance is reported as best achievable execution time in seconds.

### 5.3 Experimental System

Figure 3 shows a high-level memory organization of the systems. Both systems incorporate a shared data memory and instruction memories dedicated to each processor. In

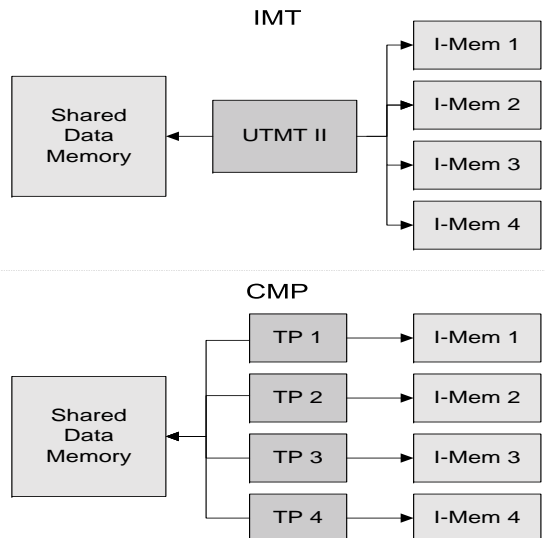


Figure 3: High-level system architecture

the MT system, the UTMT II processor connects to instruction memories via four instruction masters. In the CMP systems, each processor is connected to the shared data memory with its data master. On the other hand, the UTMT II processor accesses the shared memory using its single data master. Using only one data master eliminates some of the arbitration logic necessary in accessing the shared memory. This saves additional system area, although we focus on processor area only.

#### 5.3.1 Fetch bandwidth

In order to achieve highest pipeline utilization, the UTMT II processor must be able to simultaneously fetch instructions from all four threads. As mentioned before, the UTMT II processor incorporates four pre-fetch units that provide this functionality. To ensure this maximum bandwidth, all threads in the experimental system have separate instruction memories. Similarly, for the CMP systems, the maximum bandwidth is achieved if memory access conflicts among the processors are fully eliminated. Again, this is accomplished by employing four distinct instruction memories.

#### 5.3.2 Shared data memory model

To investigate the impact of resource sharing between multiple threads, our experimental system employs a shared data memory. The threads keep all their data in this memory. As depicted in Figure 3, the data memory used is single-ported, hence only one thread is granted access in each cycle. It is assumed that realistic embedded applications use large input data sets, thus a high capacity memory module is required (in terms of embedded systems). Since the available on-chip memory is insufficient in most cases, we use a large off-chip memory (SDRAM) as a shared resource. On the other hand, shared memory can be seen as an IP block implemented as a peripheral device shared by all threads. This allows performance evaluation of different processor's memory controller implementations. In particular, the performance evaluation of accessing a shared peripheral device with multicycle latency is possible.

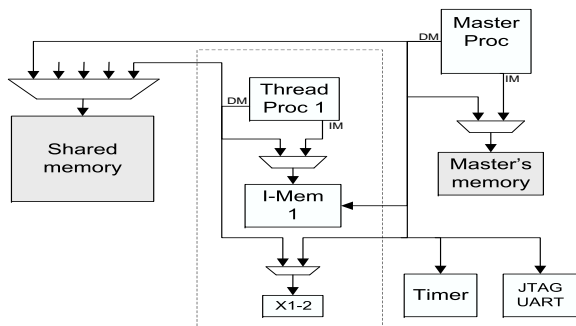


Figure 4: CMP implementation of the soft multi-threaded system (only one processor is shown)

### 5.3.3 Experimental multithreaded soft systems

Four multithreaded soft systems were implemented:

- CMP system built using four Altera Nios II/e processors
- CMP system built using four UT II processors
- MT system built using a UTMT II processor
- Uniprocessor system built using Altera Nios II fast (software-based multithreading)

These experimental systems were implemented to investigate the basic opportunities for multithreaded execution in FPGA-based systems. They serve as a platform to conduct the performance evaluation and comparison of UTMT II, UT II CMP and two Altera's processor systems. For the purposes of comparison with the UTMT II processor, the CMP systems employ four processors. In addition, we investigate the impact of several microarchitectural enhancements on each of our systems.

### 5.3.4 Experimental system implementation

Implementation of an experimental system encompasses hardware and software parts. The hardware part is a soft system synthesized into an FPGA configuration. The software part comprises thread programs and input data. Synthesis of the hardware part is a long and inflexible procedure. A small modification to the soft system requires recompilation of the system into the FPGA configuration. Thus, the goal was to develop a soft system that can be reused across all experiments. To achieve the required flexibility, the hardware experimental system was designed such that software modifications do not induce changes in the hardware part.

Figures 4 and 5 show the implementation of the experimental system based on four UT II processors and the UTMT II processor. The uniprocessor experimental system is similar to the CMP system, except that there is only one processor, and its associated peripherals. The interconnect between the components is implemented via *Avalon switch fabric* and it is generated by the SOPC Builder tool. Since the system in Figure 4 is symmetric, for simplicity only one of the four thread processors is shown. The data master of each processor is connected to the shared memory, the processor's instruction memory and associated synchronization mutual exclusion elements (mutexes). The system contains a master processor which is responsible for communication with the host computer and synchronization of threads. The

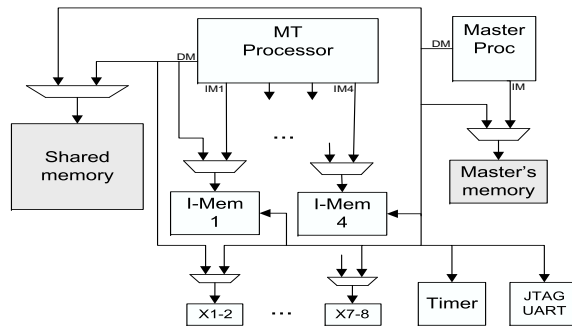


Figure 5: MT implementation of the soft multi-threaded system

master processor uses a separate memory containing both its instructions and data. This allows the master processor to access its off-chip memory without interfering with the thread processors. The master processor is an Altera Nios II/e processor with debug core functionality.

The shared data memory address space is statically partitioned among the threads at compile-time. Each thread has a private section of the memory designated for global, stack and heap data. In addition, input and output data for the thread is placed into the same private I/O section.

On startup of the experimental system, the master processor uses JTAG UART to download the instruction code of each thread from the host computer into the corresponding instruction memory. In addition, it loads each thread's input data from the flash memory into the corresponding I/O section in the shared memory.

The instruction memory for each thread processor is split into two physical instruction memories, totaling 64KB. First one, smaller in size, is initialized with a small *waiting program* which runs upon the startup of the system. The waiting program simply waits until the benchmark is loaded into the second part of the instruction memory. The waiting program communicates with the master processor using signals which are implemented via a mutex IP component provided by Altera. Upon receiving a signal the thread processor jumps to the beginning of the thread program. Waiting and benchmark programs are carefully linked to ensure that instructions are assigned to addresses of the on-chip instruction memory, and data references are assigned to the designated sections in the shared data memory.

Upon the completion of execution, thread programs signal the master processor. The execution time of each benchmark program is measured by the master processor using the timer and it is reported at the end of the experiment. Correctness verification is performed after all the benchmarks have completed execution. The benchmark output in the I/O section is compared against the correct output stored in the flash memory. The verification was necessary to ensure the correctness of our processors.

## 5.4 Benchmarks

We base our benchmarks on the embedded benchmark suite MiBench [16]. We selected seven benchmarks and ported them to our experimental system. The benchmarks were compiled with Nios II GCC compiler using the highest optimization level (O3). In addition, since none of the

Benchmark	Characteristics
Dijkstra	Matrix size 100x100 Calculates 20 shortest paths
SHA	Computes message digest for 304 KB text file
CRC32	Calculates CRC-32 on a 1336KB binary file
QSort	Sorts 50000 double precision floating point numbers
Rijndael	AES data encryption using 128-bit keys on a 304KB text file
Adpcm	Encodes pulse code modulation samples
Bitcount	Performs bit manipulations on 75000 numbers

Table 3: Benchmark characteristics

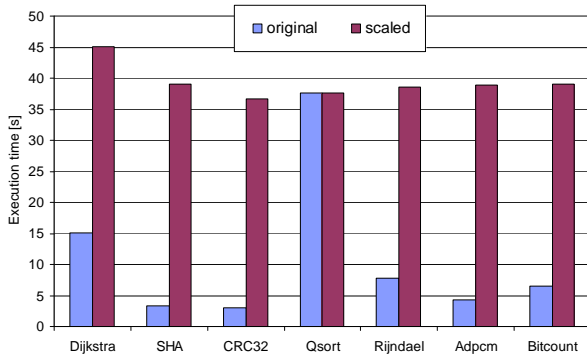


Figure 6: Execution times of original and scaled versions of benchmarks

processors include hard multiply or divide units, these functionalities are emulated in software. The characteristics of the benchmarks are given in Table 3. To represent a real-world multithreaded workload, we ran a combination of four benchmarks simultaneously, where each benchmark is run on a separate processor. Clearly, without competition among threads for shared data memory access, the total execution time would be determined by the slowest running benchmark. Additionally, a single benchmark would have the same execution time as it was the only thread executing in the system. By employing memory access contention, each benchmark is slowed down depending on its memory access intensity.

To make a valid assessment of concurrent execution of multiple benchmarks, the benchmarks were scaled to run for approximately the same amount of time. Table 4 lists the scaling ratios for all benchmarks. For the scaling ratio of  $N$ , a particular benchmark is executed  $N$  times. Figure 6 shows how we balanced the benchmarks. The left bar represents the execution time for the original benchmark. The right bar shows the execution time for the scaled benchmark. These bars correspond to the execution times obtained by running the benchmarks on Nios II/e processor at  $F_{exp}$ .

Benchmark	Scaling ratio
Dijkstra	3
SHA	12
CRC32	12
QSort	1
Rijndael	5
Adpcm	9
Bitcount	6

Table 4: Scaling ratios for benchmarks

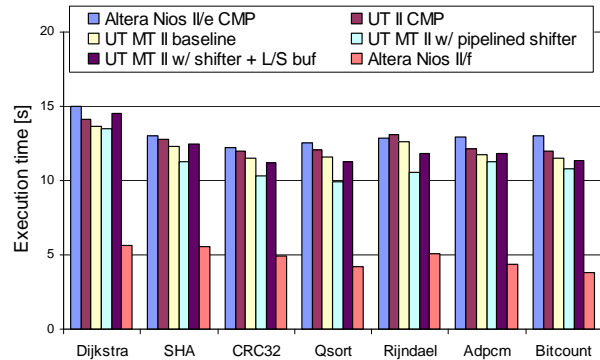


Figure 7: Single thread execution times

## 6. RESULTS

Figure 7 depicts the execution times (after prorating to the  $F_{max}$  of each processor) when a given benchmark is run as a single thread on each system implementation. The leftmost bar corresponds to a Nios II/e processor in the CMP system. The next bar is for the simplest implementation of a UT II in the CMP system. The third bar corresponds to UTMT II baseline system. The fourth bar represents UTMT II with a DSP shifter, while the fifth bar is for UTMT II which is further augmented with a load/store buffer. The rightmost bar is for the Nios II/f. As shown, the performance of our processors is comparable to commercial processors. The results show that single thread performance is worse with the addition of the load/store buffer. This occurs because the pipeline no longer stalls the round robin fetch mechanism when an instruction enters the load/store buffer, which may require extra cycle(s) before the thread is allowed to fetch new instructions, once the load/store instruction has been resolved.

Since our multithreaded implementation supports four threads for a workload, we use several mixes of four benchmarks. Figure 8 gives an example of a four benchmark mix. The leftmost bar corresponds to a multiprocessor system with four Nios II/e processors (Nios II/e CMP). The next bar corresponds to a multiprocessor with four UT II processors (UT II CMP). The third bar represents the UTMT II baseline system. The fourth bar corresponds to the UTMT II with the DSP shifter and the rightmost represents UTMT II with the shifter and load/store buffer. In multiprocessor configurations each benchmark is run on a separate processor. We can observe from the figure that Adpcm benchmark runs considerably faster on the UTMT II employing a load/store buffer. This is due to the fact that the Adpcm

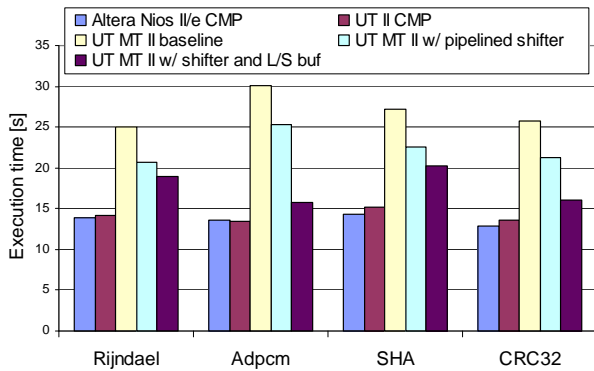


Figure 8: Execution times of each benchmark in a four-threaded benchmark mix

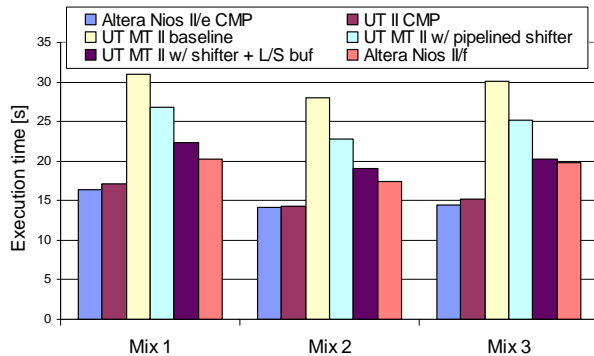


Figure 9: Completion times for three four-threaded benchmark mixes

benchmark is a memory non-intensive application as characterized by the study on MiBench suite [16]. Without the inclusion of the load/store buffer the memory non-intensive benchmarks are slowed down by other memory intensive benchmarks in the same benchmark mix. With the addition of load/store buffer memory non-intensive benchmarks can progress through the pipeline without being stalled by other threads. On the other hand, in the case of Rijndael benchmark, the improvement is not that significant. Rijndael is a memory intensive benchmark, thus most of the time the thread is in the load/store buffer.

Figure 9 shows the results for three different benchmark mixes. The third mix in the figure is the mix from Figure 8. The first mix comprises Dijkstra, SHA, QSort and CRC32 benchmarks, while the second mix consists of Rijndael, Adpcm, QSort and Bitcount benchmarks. The time shown reflects the completion time of all four benchmarks in a particular benchmark mix, determined by the longest running benchmark in the mix. The sixth bar reflects the total execution time when each benchmark in the mix is run sequentially on the Nios II/f processor. While this bar denotes the execution time for the benchmarks, it does not include the operating system overhead required to perform context switching. In addition, this is the only processor which uses an instruction cache. Moreover, this advantage is accentuated by the fact that the entire benchmark fits into the instruction cache. We chose this arrangement to allow Nios II/f processor to fully utilize its pipeline.

Figure 10 demonstrates the performance improvement gained by implementing a pipelined shifter over a multicycle shifter. The difference between the 4-cycle shifter and the

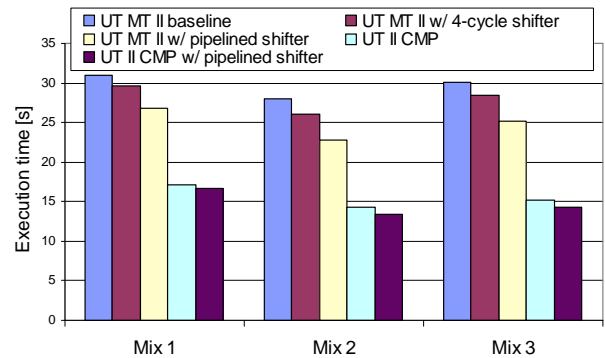


Figure 10: Comparison of UTMT II and UT II CMP with pipelined and non-pipelined 4-cycle shifter

pipelined shifter is that the 4-cycle shifter stalls all threads, whereas the pipelined shifter allows other threads to continue through the pipeline. Therefore, the pipelined shifter is the full implementation of the second performance enhancement described in section 4.2.2. On the other hand, the performance improvement for UT II CMP is limited. Again this shows that pipeline stalls due to multicycle IP significantly impact the performance of other threads in UTMT II. The three mixes used are the same as in the previous experiment.

Table 5 depicts the complexity of each system in terms of maximum frequency ( $F_{max}$ ), the number of logic elements and the total number of memory bits used. A key advantage of UTMT II is its simplicity compared to the multiprocessor configurations based both on Nios II/e and UT II processors. Note that while the numbers for the Nios II/f appear better, its real performance is likely to be significantly worse if one takes into account the OS overhead and cache efficiency which would be affected by the conflict misses due to switching between threads. We observe that our baseline MT processor achieves about 45% and 25% area savings in terms of LEs over our UT II CMP and Altera’s Nios II/e CMP, respectively. After applying the performance enhancements the area savings are 40% and 20%. Also, we observe that the pipelined shifter adds 90 LEs to the UTMT II, but for the UT II CMP it results in an increase of 428 LEs. This shows the opportunity for area reduction due to sharing CI logic. The area usage, in terms of LEs and memory bits, of the processors is displayed in Figure 11.

## 7. CONCLUSIONS

Our investigation has shown that it is attractive to use a multithreaded processor in the FPGA environment because of the significant area savings. To achieve comparable performance to a CMP system, it is essential to prevent a thread from blocking the progress of other threads in the pipeline. This can be achieved either by queuing instructions before accessing an IP block or by pipelining an IP block to a multiple of  $N$  cycles, where  $N$  is the main pipeline depth, as explained in section 4.2.2. The load/store buffers demonstrate the first approach, while the shifter unit demonstrates the second approach. The implementation results for UT II CMP and UTMT II, given in Table 5, were obtained by a straightforward application of the Quartus II compiler. We believe that superior implementations of our processors could be obtained by optimizing the design for a specific target FPGA.

Metric	Nios II/f	Nios II/e CMP	UT II CMP	UT II CMP DSP shifter	UTMT II DSP shifter	UTMT II DSP shifter/ L/S Buffer
$F_{max}$	154	150	112	112	116	111
LEs	1,392	2,200	2,892	3,320	1,645	1,735
Memory bits	7,088	4,096	8,192	8,192	5,120	5,404

Table 5: Maximum frequency and area results

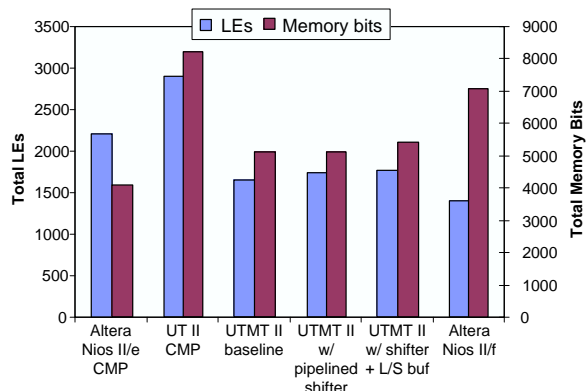


Figure 11: Area Usage

## 7.1 Future work

As part of the future work, we will investigate the reduction of the pipeline bubbles in the presence of multicycle IP blocks. For instance, an improvement can be achieved by allowing a thread to fetch a new instruction even if the current instruction is executing a multicycle IP block, providing that the current instruction does not write a value back to the register file.

Another avenue for investigation is to extend the instruction issue logic to improve performance when the workload is less than four threads. In addition, bubbles can possibly be eliminated by supporting more than four threads.

## Acknowledgments

This work was supported in part by NSERC, and the Edward S. Rogers Sr. Scholarship. We would like to thank Franjo Plavec for his advice and direction. Also, we would like to thank the anonymous reviewers for their many valuable comments.

## 8. REFERENCES

- [1] Altera Nios. <http://www.altera.com/products/ip/processors/nios/>. 2005.
- [2] Altera Nios II. <http://www.altera.com/products/ip/processors/nios2/>.
- [3] Xilinx. Microblaze processor reference guide embedded development kit edk 7.1i. 2005.
- [4] Franjo Plavec, Blair Fort, Zvonko G. Vranesic, and Stephen D. Brown. Experiences with soft-core processor design. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*, page 167.2, Washington, DC, USA, 2005. IEEE Computer Society.
- [5] Paul Metzgen. A high performance 32-bit alu for programmable logic. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International symposium on Field Programmable Gate Arrays*, pages 61–70, New York, NY, USA, 2004. ACM Press.
- [6] Paul Metzgen. Optimizing a high-performance 32-bit processor for programmable logic. In *International Symposium on System-on-Chip*, 2004.
- [7] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. The microarchitecture of FPGA-based soft processors. In *CASES '05: Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 202–212, New York, NY, USA, 2005. ACM Press.
- [8] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, New York, NY, USA, 1995. ACM Press.
- [9] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE - Real-Time Signal Processing IV*, pages 241–248, 1981.
- [10] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *ICS '90: Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, 1990.
- [11] Patrick Crowley, Marc E. Fluczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *ICS '00: Proceedings of the 14th International Conference on Supercomputing*, pages 54–65, New York, NY, USA, 2000. ACM Press.
- [12] R.G. Dimond, O. Mencer, and W. Luk. CUSTARD - a customisable threaded FPGA soft processor and tools. In *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications*, 2005.
- [13] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, pages 185–194, New York, NY, USA, 2003. ACM Press.
- [14] Altera Nios II Cores. [http://www.altera.com/products/ip/processors/nios2/cores/ni2-processor\\_cores.html](http://www.altera.com/products/ip/processors/nios2/cores/ni2-processor_cores.html).
- [15] Altera's SOPC Builder. <http://www.altera.com/products/software/products/sopc/>.
- [16] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.