Using System Hyper Pipelining for a Multi-Threaded FORTH Compatible Stack Processor Mapped on an FPGA

Tobias Strauch

***Abstract*— FORTH compatible stack processors are still subject to research and development. They also face new multi-processing and multi-threading challenges. Most notoriously the multicore processor G144 from the inventor of FORTH Charles Moore. This paper outlines an alternative concept to generate a mutli-threaded FORTH compatible stack processor by using System Hyper Pipelining (SHP), which overcomes the limitations of classical multi-threading methods by adding thread stalling, bypassing and reordering techniques to better cope with the challenges of Symmetrical Multi-Processing and Simultaneous Multi-Threading. SHP is ideal for FPGAs with their high number of registers and their flexible memory usage. The paper gives results for a FORTH compatible stack processor mapped on an FPGA.**

Keywords—FORTH, C-Slow Retiming, Symmetrical Multi-processing, Simultaneous Multi-threading, Multi-processor Systems

I. Introduction

Charles Moore, the inventor of FORTH introduced a multicore array chip based on a FORTH compatible stack processor in [1]. This design uses the traditional approach of generating a multicore chip based on an array-like orientation. This paper now proposes an alternative approach of generating a multi-threaded stack processor by using System Hyper Pipelining, which is a successor of C-Slow Retiming.

C-Slow Retiming (CSR) provides C copies of a given design by inserting registers and reusing the combinatorial logic in a time sliced fashion. CSR therefore improves the performance per area factor. Leiserson et. al. introduced the concept of C-Slow Retiming (CSR) in [2]. In section II, the System Hyper Pipelining technology is shown and how it differs from CSR. Section III outlines a thread controller which copes with a high number of threads. Section IV outlines, how a FORTH based stack processor can benefit from SHP. A system is proposed in section V before the results are given in section VI.

II. CSR and SHP Technology

System Hyper Pipelining (SHP) has been introduced by Strauch in [3]. This paper gives a 2-page introduction for the readers' convenience again. SHP is based on C-Slow Retiming (CSR). It enhances CSR with thread stalling, bypassing and reordering techniques by replacing the original registers of the design with memories and by adding a thread controller (TC). In the remainder of this paper, the word "thread" (T) is used synonym for the execution of a program or algorithm.

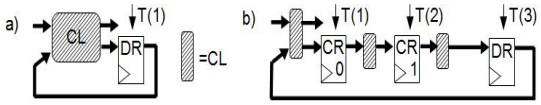

Figure 1: a) Simplified single clock design. b) Applying CSR technique.

Figure 1a shows the basic structure of a sequential circuit with its inputs, outputs, combinatorial logic (CL) and original design registers (DR). The sequential circuit handles one thread T(1). Figure 1b shows the CSR technique. The original logic is sliced into C (here C=3) sections. This results in C functionally independent design copies T(C=1..3) which use the logic in a time sliced fashion. Each thread has its own thread ID (TID). For each design copy it now takes C "micro-cycles" to achieve the same result as in one cycle (called "macro-cycle") of the original design. The implemented registers are called "CSR Registers", (CR) and are placed at different C-levels (CRn).

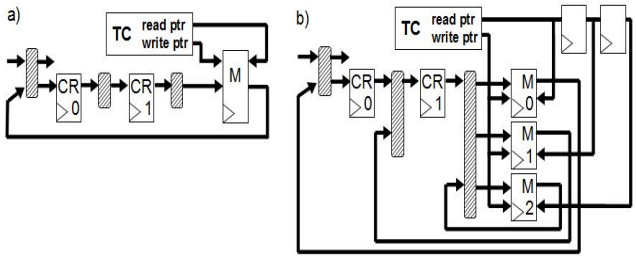

Figure 2: a) SHP-ed design with thread controller, memories and CRs. b) Advanced SHP.

Figure 2a shows the modifications of a CSR-ed design towards SHP. Assuming the DRs are now replaced by a memory (M). The incoming design states / threads are stored at the relevant address (write pointer) based on the TID. D is the number of threads which the memory can hold (memory depth). The outgoing thread can now be freely selected within D available threads (read pointer), except the threads already passing through the design logic. A CSR-ed design has usually many shift registers. DRs are followed by a series of CR

registers. In the SHP-ed version, many memory data outputs are connected to CRs. In this case, the shift registers at the outputs can be replaced by registers at the read address inputs of the memories (Figure 2b). The memory is sliced into individual sections (M0, M1, M2) and each section has a delayed read of the thread. The outputs can now be directly connected to the relevant combinatorial logic and the shift registers can be removed. The same trick can be applied on the shift register chains at the inputs of the memory.

$$Fcsr = Forig * C * r^{C} \text{ with } r \sim 0.93 \quad (1)$$

$$0\ Hz <= Ft <= Forig * r^{C} \quad (2)$$

$$Fshp = \Sigma\ Ft <= Fcsr \quad (3)$$

We define Forig as the maximal speed of the original design. The maximal speed of a CSR-ed design can be estimated by using Equation 1. Fcsr is C times the original speed Forig reduced by a correction factor r^C, which considers the delay inserted on the critical path by the CRs. r is technology dependent. Based on empirical data, r is roughly 0.93 for a Virtex-6 FPGA. Equation 2 says, that in an SHP-ed design, a single thread can now run at any speed (over a long period) between 0 Hz (stalled) and Forig * r^{C}. The maximal speed of an SHP-ed design Fshp is the sum of all active threads (Equation 3). Fshp cannot be greater than Fcsr.

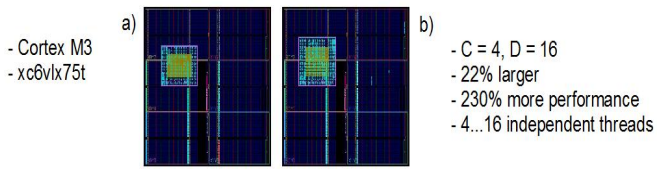

Figure 3: SHP based performance per area improvement based on a Cortex M3 example.

Figure 3a shows a Cortex M3 (as it can be found in [4]) implementation on a a Virtex-6. With C=4 and D=16, the SHP-ed version (Figure 3b) is just 33% larger (occupied slices) but can achieve 230% more performance (overall 330%) compared to the original implementation. In other words, SHP improves the performance per area factor if the application can utilize this performance gain by using at least 4 independent threads.

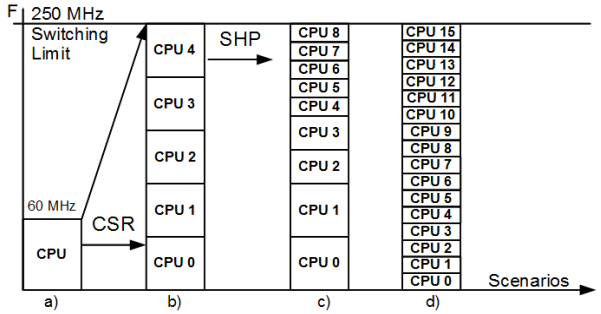

Figure 4: Histogram of different scenarios (a-d) of running CSR and SHP.

Figure 4 shows the advantages of CSR and SHP over the original design. The x-axis shows different scenarios. Assuming a single CPU runs at 60MHz on an FPGA (Figure 4a). It can be seen, how CSR improves the system performance of the original system implementation, (Figure 4b). When using CSR, the system performance is not necessarily limited by the critical path of the original design, but - for instance - by the switching limit of the FPGA (e.g. 250MHz) or the external memory access instead.

There are two key observations when SHP is used on a design. First, for executing multiple programs on multiple CPUs (symmetrical multi-processing (SMP)) or for executing multiple threads on a CPU (simultaneous multi-threading (SMT)), SHP allows a more efficient usage of the system resources. It adds the possibility to dynamically scale the system performance over a minimum (C, Figure 4b), and a maximum (D, Figure 4d) set of design copies, whereas any solution in-between can be realized (Figure 4c). This load balancing is handled by a thread controller (TC).

Secondly, threads don't interact with each other. There is no register dependency between the individual threads. The runtime of each thread is therefore deterministic. The variable latency that the execution per thread may experience due to different behavior in if-branches for instance is not an issue, because all threads work independent of each other.

III. The Thread Controller

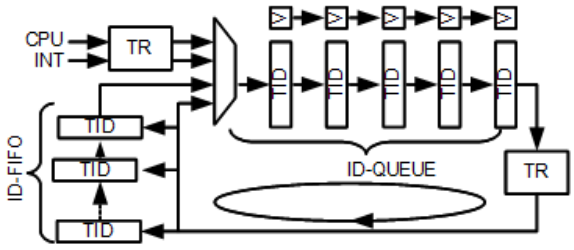

Figure 5: Thread Controller Mechanism.

A TC is used, which is controlled by a special function register set TCRS (Thread Controller Register Set). It is accessible by all active threads (T). Each thread has its own thread register (TR). Figure 5 shows how the Thread ID (TID) is provided for the SHP memories. When a thread is executed, its TID passes through the ID-Queue (IDQ). It is reinserted into the IDQ or into the ID-FIFO, if the relevant bit in the TR shows that the thread is still active and not on hold or killed. When less than C threads are valid, active threads need to be re-executed, but the valid bit V of the IDQ indicates, that its state copies should not be stored. When more than C threads are executed or an additional thread is inserted, then the TID is parked in the ID-FIFO.

Threads can be added to the active thread list by writing their program start address to the “Activate” register. This sets the thread-specific active A bit in the TR. Threads can be hold by setting the hold H bit or killed by clearing the active A bit in the TR. When the thread priority bit P is set in the TR, then

a thread execution has a higher priority than the threads stored in the ID-FIDO or threads resulting from an interrupt or the CPU. It is therefore directly re-inserted into the IDQ again and not stored into the ID-FIFO.

To cope with **fork-join** queueing, the following mechanism is implemented. A set of Ts an be started from a single main thread (MT) by successively writing the individual start addresses of the Ts to be started to the TCRS called "Activate and Count" (AC). By doing that, the number of Ts called (CT) by the MT is stored in the AC register. Optionally the MT stalls itself after that process. Each CT saves the MT's SID in the "forked thread register" (FT). When a CT is killed, it checks the FT and decrements the AC of the MT. If this number gets 0, the MT stalling bit is cleared by default and the MT continues. Alternatively the MT can read it's AC register to continue execution.

The TCRS can be programmed to set a group of consecutive threads into dynamic length instruction word (**DLIW**) mode. By doing that, a given number of threads are executed in parallel. The concept is similar to the very long instruction word (VLIW) concept, except the fact that the number of threads running in parallel can be dynamically defined (but must be lesser or equal to D).

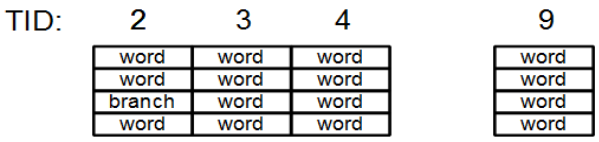

Figure 7: Grouping threads to run dynamic length instruction words.

Figure 7 shows an example. The threads with the TID 2,3 and 4 are running as a group using DLIW. Thread 9 is listed to show that alternative threads can still be actively running. The first thread in the DLIW (TID 2) starts the parallel execution when the number of subsequent threads (here 2) is written into its DLIW register. Branch instructions (words) are only consider by the initial thread (here TID = 2), the program counter of the subsequent threads are always derived from its trailing thread (by adding the value of 2 for instance).

This outlined TC has a low complexity (see result section). It can stall and bypass individual threads and it is capable of handling fork-join queues. By default, a thread runs completely independent of the other threads when its priority thread is set and when only less or equal number of C threads have the priority bit set. It can also group threads to run dynamic length instruction words. The DLIW method is accompanied by the message passing implementation, which is outlined in the sequel of this paper.

IV. Benefits of a Multi-Threaded Stack Processor

A. ... compared to its single core implementation

This sections lists some of the benefits of an SHP based multi-threaded stack processor. First, it is compared to a single core implementation. It is assumed, that the application can be partitioned into individual threads to a certain degree. This certainly has its limit (Amdahl's Law), but applications in the field of automation and controlling for instance can usually be partitioned into individual tasks, whereas most of them can be executed in parallel.

The key benefit of SHP when applied on a single core is the increase of the performance-per-area (PpA) factor. This has been shown in [3] based on FPGAs and it is demonstrated in the result section of this paper again. The performance of a system is already increased when a second thread can be used. All threads of an SHP-ed processor can share the same data. The time sliced access to the program memory increases the memory bandwidth and reduces potential memory-wall problems.

B. ... compared to a multicore implementation

Charles Moore has introduced a multicore array stack processor GA144 [1]. A functional identical SHP-ed version can be realized on a smaller die size due to the increased ppa factor. Alternatively more performance could be realized on the same area when SHP is used.

The GA144 was used by Schneider et al. in a research project [5]. In their work, an application is partitioned into individual tasks, whereas each task is assigned to an individual core on the GA144. It is easy to understand, that in this case, tasks have to stall sometimes, because they need to wait on data from other tasks to be processed. It has been outlined in section II of this paper, that SHP allows the dynamic scaling of individual threads on an SHP-ed processor. If a task can be stalled (because it is waiting for data from other tasks for instance), then its associated thread can be stalled and therefore frees performance for other threads. A task on the SHP-ed processor array does not necessary consume performance nor logic area when stalled.

On an SHP-ed based processor array, multiple threads (D) on each element can share the same memory. On a traditional multicore array (GA144), data has to be transferred to the individual core/thread.

C. ... when used in a safety critical environment

FORTH compatible stack processors can be used in a safety critical environment. Most notoriously is the controlling of the Philae landing process, using FORTH and a radiation hardened processor [6]. A C-slow retimed processor can be used to generate a time redundant system, as outlined in [7]. It enables the detection of single event upsets (SEUs) and allows an on-the-fly recovery. The same technique can be applied on almost the same area on an SHP-ed stack processor.

As an alternative concept to detect malfunctions of an application running on a stack processor is the usage of different software tasks, which are aiming to deliver the same results. If the results differ, at least one task did not execute the code as expected. The "free" additional threads that come

with SHP and its increased PpA factor enable the execution of redundant tasks on almost the same die size without loosing a reasonable amount of system performance.

D. ... and what turned out to be not very beneficial

The idea of combining a FORTH compatible SHP-ed processor with OpenMP [8] was not very successful. Although some OpenMP concepts can be used in the FORTH language, the restriction comes from the write/read policies of private/shared variables used by individual tasks. Still, the programmer can use the TCRS to benefit from the implemented fork-join mechanism.

Another intriguing idea when working with an SHP-ed FORTH processor is that each thread can access the stack of alternative threads. It turned out that the resulting logic is too complex and therefore inefficient. Alternatively, threads can be synchronized using the TC's DLIW technique as well as the message passing method, which is outlined in the next section.

V. The Proposed System

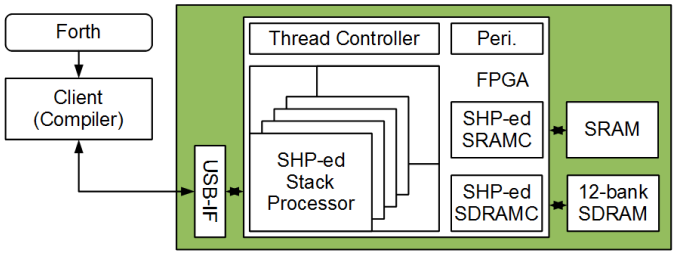

Figure 8: Overview of the Proposed System.

A. Overview

Figure 8 gives an overview of the proposed system. It is based on the diploma-thesis of G. Hohner, which is released on the OpenCores webpage [9]. A FORTH compatible program can be compiled by the original client (compiler). The program is then downloaded through an USB interface to a 12-bank wide SDRAM block. The original stack processor is enhanced by the SHP technology. A thread controller is added which can be programmed by the processor using special function registers (SFR). There are also some standard peripherals like GPIO, UART and Timer. The design is mapped on an FPGA. An external SRAM provides enough memory for data access.

B. The FORTH Stack Processor and its SHP-ed Version

The CPU is a FORTH compatible stack processor with 6 stages and two 32-bit wide stacks. It supports all common FORTH commands (see [9] for more details).

The CPU is slightly optimized so that it can be used for an automatic transformation process towards an SHP-ed version, which is done by a tool called CoreMultiplier [10]. Based on empirical data of other CPUs of comparable complexity, the parameter C was set to 4, which results in a good performance-versus-area (occupied slices) trade-off. In other words, 3 (C-1) registers are automatically inserted into each path in the CPU by a timing driven algorithm (, whereas some of the registers are merged into their adjacent memory blocks again, see section II). The parameter D was initially set to 16. Less than 16 threads do not reduce the number of occupied memory resources. Due to the high registers count of the CPU, D was then reduced to 8 so that the stacks can share FPGA memory resources.

The CPU accesses an external SRAM using an SHP-ed SRAM controller for data transfer and a external SDRAMs using an SHP-ed SDRAM controller for the program code. The external SDRAM is based on 16-bit wide devices so that a pair generates a 32-bit wide interface. Three individual SDRAM pairs build an SDRAM list with 12 banks. This allows individual threads to access individual banks. Each thread can access the complete SRAM range and the complete SDRAM range as program memory.

C. The Message Passing Extension

Before SHP was applied on the original design, one additional coprocessor register COR is added. A new FORTH instruction CTC (copy to coprocessor) writes the stack value into COR. An additional instruction CFC (copy from coprocessor) writes the COR value back onto the stack. The SHP-ed version was then modified so that each thread writes the stack's value into the COR of the thread with the next TID. This adjacent thread can then read this value one cycle later by using the CFC word. This mechanism is very helpful when the DLIW method is used and a message needs to be passed to another thread.

D. The FPGA Board

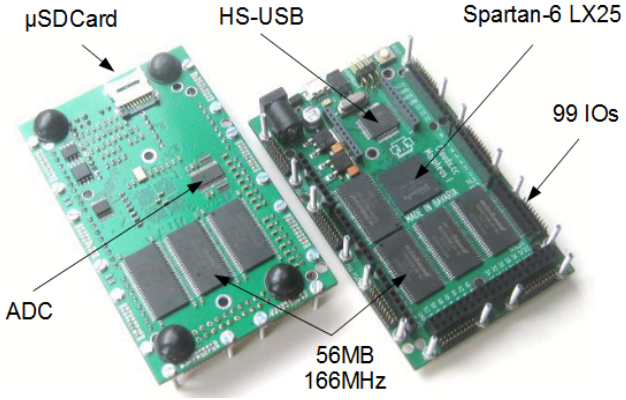

Figure 9: FPGA Board with unique SDRAM access structure.

The proposed system is mapped on a Spartan 6 LX25 FPGA board with a unique SDRAM access structure (see Figure 9). It allows the individual threads on the CPU to access up to 12 individual SDRAM banks. Without that infrastructure, the memory bandwidth would be a major bottleneck (memory wall).

E. The Software Compiler and Client

The FORTH compiler and the client are taken over from the original source [9], which is written in Java. The two

additional instructions (CTC and CFC) are added to the compiler code. The communication with the board was enhanced for HS-USB using an FTDI [11] DLL and an FTDI USB chip.

VI. Results

In this section, the original design variations are compared to the system hyper pipelined version of its single core implementation. The original work includes a feature to automatically generate a multicore solution by modifying a parameter called "core". By increasing the core parameter, the number of CPUs and CPU stacks increases accordingly.

Table 1. Comparing Performance per Area of the Original Design and the SHP-ed Version.

	unit	original, core =				SHP, C = 4, D = 8
		1	2	3	4	
occS		1377	1865	2380	3143	1927
Perf.	MHz	45,44	45,44	45,44	45,44	159,12
PpA	kHz/occS	32,99	24,36	19,09	14,45	82,57
ΔPpA	%	100	73,83	57,86	43,81	250,23

Table 1 compares the results for the different implementations mapped on a Spartan 6 LX25 FPGA. The occupied slices (occS) of the original design with increasing core factor (1, …, 4) is shown. The performance remains stable for all 4 core variations at 45,44 MHz. This decreases the performance-per-area factor (PpA) due to the increased number of occS. The SHP-ed achieves a performance of 159,12 MHz on 1927 occS, which results in a PpA of 82,57 kHz/occS. This is a PpA increase of 250% compared to a single core implementation of the original core. The occS number of the SHP-ed version includes the design of the TC, which consumes just 226 slices of the FPGA.

Further performance tests are not conducted, because they heavily depend on how the algorithm can be partitioned into multiple independent threads. The runtime of the original program and the runtime enhancements when using the multithreaded SHP-ed version can easily be derived from the numbers given in Table 1.

VII. Conclusion

This paper showed how C-Slow Retiming (CSR) and parallel programming can be combined to a new method called System Hyper Pipelining (SHP). SHP benefits from the higher performance per area (PpA) factor, which can be achieved when using CSR. Additionally, SHP offers also flexible thread stalling, bypassing and reordering features which are used by multi-threading methods to improve the system performance.

SHP is applied on a FORTH compatible stack processor. This stringent transformation process can be automatically accomplished within seconds and results in a multi-threaded version of the stack processor. Fig. 4 shows, how the increased system performance can be distributed among multiple design copies by using a thread controller. Individual threads can run at different speeds and can even be completely stalled without consuming relevant power anymore. The paper shows how a thread controller enables fork-join operations by accessing its special function registers. Also very large instruction words (VLIW) can be executed by running consecutive threads. In the proposed system the VLIW can also have a dynamic length.

The system is mapped on an FPGA. The increased system performance though requires an enhanced memory access method to reduce the potential memory bottleneck. A hardware solution with 12 SDRAM banks is proposed. The time shared memory access works in-line with the time-shared mechanism used to duplicate the functionality of the FORTH compatible stack processor.

References

[1] GreenArrays. B001 - F18A Technology Reference. Available online: www.greenarraychips.com/home/documents/greg/DB001-110412-F18A.pdf, as of April 21, 2013.

[2] C. Leiserson and J. Saxe, "Retiming Synchronous Circuitry", Algorithmica, vol. 6, no. 1, pp. 5-35, 1991.

[3] T. Strauch, "The Effects of System Hyper Pipelining on Three Computational Benchmarks Using FPGAs", 11th Intern. Symposium in Applied Reconfigurable Computing, ARC 2015, 13-17 April 2015, Bochum, Germany, pp. 280 – 290

[4] Atmel, "AT91SAM ARM based Flashed MCU", Available online: http://www.atmel.com/Images/doc11057.pdf

[5] T. Schneider, I. Von Maurich, and T. Guneysu, "Efficient implementation of cryptographic primitives on the GA144 multi-core architecture", 24th Intern. Conf. on Application-Specific Systems, Architectures and Processors (ASAO), IEEE 2013, 5-7 June 2013, Washington, pp 67-74.

[6] MPE Microprocessor Engineering, "Comet Landing – a triumph for Forth in Hardware and Forth in Software", Press Release, 13th November 2014, Southampton, UK

[7] T. Strauch, "Using C-Slow Retiming in Safety Critical and Low Power Applications", First Intern. Workshop on FPGAs in Aerospace Applications, FASA 2014, 5th September 2014, Munich, Germany, pp. Tpd.

[8] OpenMP, "The OpenMP API Specification for Parallel Programming", Available online: www.openmp.org

[9] Opencores, Stockholm, Sweden, 2007, Available online: www.opencores.org/projects

[10] Edaptix, CoreMultiplier, Munich, Germany, Available online: www.edaptix.com/coremultiplier.htm

[11] FTDI, Avalable online: www.ftdichip.org