Multiple Contexts in a Multi-ported VLIW Register File Implementation

Joost Hoozemans, Jens Johansen, Jeroen van Straten, Anthony Brandon, Stephan Wong

Computer Engineering Lab, Delft University of Technology, The Netherlands
Email:{j.j.hoozemans, a.a.c.brandon, j.s.s.m.wong}@tudelft.nl
{j.johansen, j.vanstraten}@student.tudelft.nl

***Abstract*—The register file is an expensive component in the design of any processor, especially, when considering the additional ports that are needed to support multiple datapaths within a wide-issue VLIW processor. In a recent work, these additional resources were used to dynamically reconfigure the register file to support a dynamically reconfigurable VLIW core. The design can be perceived as a single 8-issue, two 4-issue, or four 2-issue VLIW cores. Consequently, the multi-ported design can operate in different modes, namely as *one*, *two*, or *four* register files, respectively, corresponding to the active number of cores. The implementation of the register file design on FPGAs using Block RAMs still results in unused resources due to the coarseness of the Block RAMs.**

In this paper, we propose to re-purpose these unused BRAM resources to additionally support multiple contexts next to earlier-mentioned modes. In this manner, the 8-issue, 4-issue, and 2-issue cores have access to 4, 2, and 1 contexts, respectively. Consequently, we can avoid saving and restoring of the task states in a multi-task environment, turning context switching from a traditionally time-consuming event to an almost instantaneous event. The advantage of this is the reduction of interrupt latency and task switching latency, which are important in real-time and embedded systems.

Our results show that our technique can improve interrupt latency by a factor of 17.4× compared to using a software register spill routine, depending on the behavior of the memory system. Likewise, the task switching time can be improved by 6.7×.

I. Introduction

The ρ-VEX processor [1] is a dynamically reconfigurable VLIW processor that can adapt its organization to the requirements of different workloads. One of its most important run-time parameters is the issue-width that allows for adaptation towards the ILP of the task(s) at hand. The design can be configured as a single 8-way ($1 \times$ 8-way), two 4-ways ($2 \times$ 4-way), four 2-way ($4 \times$ 2-way) VLIW processor core(s), or combinations of those: e.g., two 2-ways and one 4-way. This capability requires the design of an extensive register file to support these different modes. In the worst case, the register file must provide:

- 8 write ports and 16 read ports when running in the 1×8-way mode
- 4 architecturally separate register files when running in the $4 \times$ 2-way mode

This work has been supported by the Almarvi European Artemis project nr. 621439.

To design a register file that satisfies these requirements we use techniques such as Block RAM (BRAM) duplication and a Live Value Table (LVT), which we will discuss in Section II.

A major drawback of the current design is the large resource utilization. The BRAMs used to implement the register file on the FPGA need to be duplicated multiple times to provide the necessary amount of read and write ports. Every BRAM has a capacity of 512 32-bit words (2KiB); however, the architecture only requires 64 32-bit registers. Because of this, the resulting design has an enormous storage capacity of which at most an eighth is used by the processor in any particular configuration.

The design presented in this paper aims to convert the drawback of the high BRAM usage of the register file for wide-issue VLIW softcore processors into an advantage by using the overcapacity to store different execution contexts. The actual utilization of the BRAM storage capacity will increase from $\frac{1}{8}$ to $\frac{1}{2}$. Support for multiple contexts in hardware relieves the core from having to spill and restore its entire register file contents to and from memory in the event of a task switch or interrupt. In a multi-tasking environment, this concept changes task switches, which are traditionally very time-consuming, into a virtually instantaneous event. Faster context switching has advantages in numerous computing scenarios, as it will increase responsiveness for interactive workloads and improve interrupt latency and task switching speeds in real-time systems. In the following, we illustrate several cases in which our work can improve performance:

- Frequently used threads: Kernel threads, like schedulers, must be frequently executed. In a traditional core implementation, timers interrupt the core and trigger context switching in order to execute such threads. In our work, these threads can be maintained within the core and thereby remove the need for context switching. For example, an application is executing in the 8-issue mode using 1 out of 4 contexts. When the scheduler needs to execute, the current thread can be scheduled to run on a 4-issue core - this mode switch only takes several cycles when using generic binaries [2]. In the remaining 4-issue core, the execution of the scheduler can be resumed by using its own context that remained “dormant” within the core.
- Dynamic switching of execution by different cores: When threads require more resources, e.g., when their ILP

978-1-4673-9406-2/15/$31.00 ©2015 IEEE

increases, our processor design allows for it to claim additional datapaths to execute the code more efficiently. This does mean that another thread must be stalled for a while. However, in our case, the context of the second thread does not need to be saved into the memory and can remain within the core until it is resumed. In the latter, another context switching operation is saved.

- Context-cycling after cache misses: When our processor is running in the 8-issue (4-issue) mode, it can have up 4 (2) contexts stored within each core. This means that when one thread is encountering a cache miss, thus execution is stalled, the core can easily switch to another thread (context) and continue execution, i.e., Switch-on-Event Multi-Threading SoEMT.
- Embedded real-time systems with multiple tasks that require stringent real-time constraints (e.g., control loops with sensors and actuators). A single core can process more events using multiple contexts [3]. Therefore, a softcore can be used as microcontroller on an FPGA which would save the designer from having to design hardware circuits to handle some events or having to resort to a multi-core system where distinct events are handled by a dedicated core.

The register file of our ρ-VEX is a complex topic, as it is also instrumental in supporting the core's dynamic reconfigurability [4]. We limit the scope of this paper to evaluating the benefits from multiple hardware contexts. It must therefore be noted that the costs of this design (see Table I) are paid not only for multiple contexts, but also to support the dynamic reconfigurability. Our approach in this paper gives us a $17.4\times$ reduction in interrupt latency and $6.7\times$ reduction in context switching time.

II. Background

The multi-ported register file is a challenging component in the design of softcore VLIW processors. Wide-issue VLIW processors like the ρ-VEX need register files with a large number of read and write ports. The VEX instruction set architecture (ISA) supports operations that use two source registers and one destination register. Because of this, the number of write ports required is equal to the issue-width, and the number of read ports is equal to twice the issue-width. Creating such complex register files using FPGA LUT resources is very expensive and scales very poorly with the number of ports. The reconfigurable ρ-VEX design and the implementation of its multi-ported register file are introduced in [5]. Moreover, in [6], the idea of using a Live Value Table (LVT) is discussed that enables the use of banked memories with duplication to create multi-ported BRAM memories. The ideas presented in this paper are built upon a register file design that is implemented using this technique. We will discuss the concepts and challenges briefly in this section.

Creating RAM memories that have more read ports is straightforward and achieved by duplicating the BRAM and writing data into each block simultaneously. In this way, each BRAM contains the same data, and their read ports can be used independently of each other. Increasing the number of write ports, however, is more difficult. Several solutions exist in literature. The simplest solution is to divide the register file into banks, each connected to one of the write ports [7]. This solution restricts the range of registers each write port can write to and thus reduces the freedom the compiler has to schedule instructions. Another solution introduced in [8] increases the size of each bank to the original register file size and renames the registers in between the compiler and assembler. This solution enables a banked design with the same scheduling freedom as an actual multi-ported register file but utilizes a multiple of the number of registers. Note that this technique does not necessarily require more BRAMs since their size is a lot larger than the 64 registers specified in the VEX ISA. It does, however, increase the number of bits required to specify the source and destination registers in instructions.

The register file used in the ρ-VEX uses the technique introduced by [6]. This scheme also duplicates the register file for each write port. However, instead of uniquely naming the registers in each bank, a Live Value Table (LVT) keeps track of which bank holds the most recent value of each register. It uses this information to multiplex the right bank to the read ports, as shown in Figure 1. The LVT needs to be implemented as a multi-ported LUT based RAM because it still needs one write port per register file write port. However, since it only needs to hold a bank address, it is much narrower than the original register file that the scheme seeks to replace. While this technique enables the register file to be implemented mostly with BRAMs instead of LUTs, it still scales poorly with the number of ports. The number of BRAMs required is equal to the product of the number of read and write ports. The depth of the LVT scales linearly with the number of registers in the register file while the width scales logarithmically with the number of write ports. The number of ports required for the LVT is equal to the number of ports on the register file.

III. Related work

In [9] the authors analyzed the high requirements that wide-issue VLIW processors pose on the register file. They discuss hypothetical FPGA primitives similar to existing BRAMs but featuring many more read and write ports. These primitives do not exist in current FPGAs, therefore, the use of large BRAM or LUT-based structures is required to emulate this behavior [6].

In [10], it is stated that "the context switch time is one of the most significant overhead factors in any operating system" and shows that high timer interrupt handling latency can impede schedulability of real-time tasks. In [3], it is measured that using a multi-threaded architecture with 4 register sets allows an autonomous guided vehicle to run at a 28% higher velocity. In [11], measurements were performed to quantify the interrupt latency of several embedded Linux distributions running on a Xilinx Microblaze.

There are numerous examples of processors which use the concept of multiple register files to enhance the context

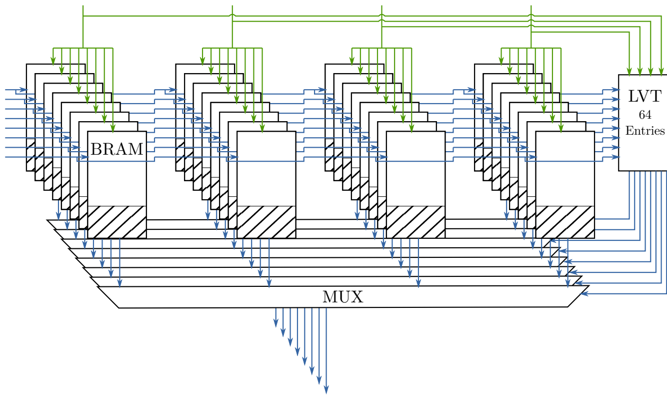

Figure 1. Block diagram of register file implementation using multiple banks of BRAMs. The green arrows indicate write ports, while the blue arrows indicate read ports. The shaded area represents the portion of the BRAM used for storing a single context.

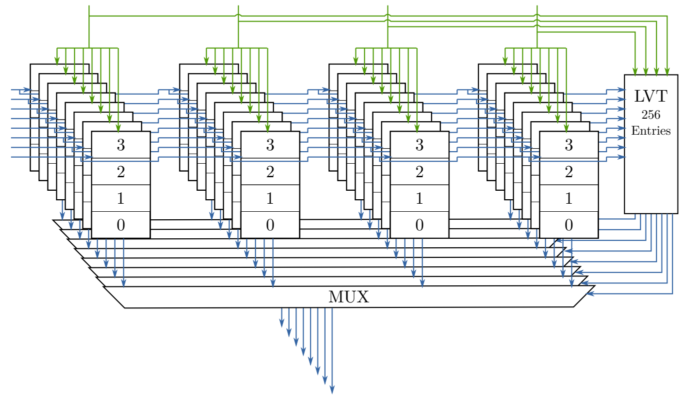

Figure 2. Block diagram of register file implementation supporting multiple contexts. Here the number of BRAMs is the same, but the LVT is larger.

switching time and interrupt latency in hardware. In [12], comparisons are made (by means of simulations) between increasing the number of cores and increasing the number of register sets in terms of increasing performance for a parallel workload. In [13], the MIPS architecture is extended by duplicating the register file multiple times and adding special instructions to switch between them when a context switch is required. In [14], the authors propose a novel architecture, which also supports holding multiple contexts in hardware simultaneously, and extend it with a dedicated cache to hold contexts to prevent spilling to main memory. Among other things the effects of the additional contexts on interrupt latency is investigated. Storing multiple contexts is also a requisite for (Simultaneous) Multi-Threading (SMT) [15]. An example of a VLIW processor with SMT support is the Itanium [16]. These technologies target high-end ASIC processors while this work targets the embedded (FPGA) domain.

The synthesizable ARPA-MT [17] and RTBlaze [18] processors also use SMT to improve schedulability and performance for embedded real-time systems. However, all the resource investments in this core are only used for SMT. The ARPA-MT core has a single execution pipeline. The fetch and decode circuits as well as the register file need to be duplicated for each thread slot.

In contrast, the ρ-VEX uses the additional resources to support: 1) a very wide VLIW to exploit ILP, 2) multiple hardware contexts and 3) a multi-core configuration (in other words, all contexts can be active and executing at the same time). Therefore, it uses the additional resources in a more efficient way compared to the previous work.

IV. Implementation

Figure 1 shows the implementation of a register file with four write ports and eight read ports ($4W \times 8R$), using BRAMs and an LVT. The $8W \times 16R$ version would be 4 times as large. The hatched area represents the part of the BRAM that is actually used to store the 64 registers used by the ρ-VEX. The figure shows that a large part of the BRAMs is unused. Because the ρ-VEX can be configured as four independent processors, it also needs four separate register files. However, the total number of read and write ports is the same for one large 8-issue processor or four separate 2-issue processors. Because of this characteristic, the same multi-ported register file can be used in each configuration. The number of registers, however, needs to be quadrupled, for a total of 256 registers, since each core needs a separate register file of 64 registers. The BRAM resources on contemporary FPGA boards provide more than sufficient storage capacity to accommodate this, so there is no added cost in BRAM resources. However, the LVT does need to increase in size, to keep track of the most recent location of all 256 registers.

Figure 2 shows how the multiple contexts can be stored in the previously unused space of the BRAMs. Creating four separate register spaces is a necessary cost to enable the ρ-VEX to be split into four separate processors. However, not all of the register spaces are used when the core is configured as a single 8-issue processor or two 4-issue processors. This creates the opportunity to re-purpose these unused register spaces as alternative register windows, which can be used to store the register context of inactive processes. Since the four register windows are implemented as a larger continuous address space, the uppermost bits can be used to select one of the four register windows.

The ρ-VEX utilizes more registers than just the 64 general purpose registers. It also has the following registers, that must be stored for a context switch:

1) A special 32-bit register used to store the return address for a function call (the link register).
2) Eight 1-bit registers used for conditional branching.
3) The program counter.
4) Various control registers, used for example for interrupt handling.

These registers cannot easily be stored in BRAMs, as the control logic needs to be able to access all these registers at once. Therefore, these registers are implemented in LUTs. To support running as 4×2-issue processors, all these registers need to be duplicated as well, and can thus be used as part of

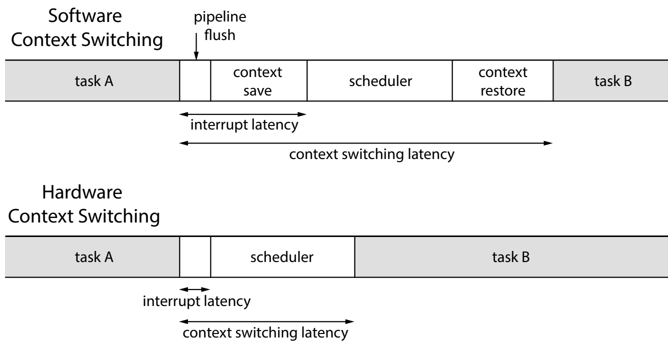

Figure 3. Context switching and interrupt latency definition.

the hardware contexts. Some additional hardware is required to use these registers for context switching, as not every lane would necessarily need access to all duplicates of the registers for reconfiguration only, while this is necessary for context switching. However, when this is done, the only registers which need to be spilled and restored are those registers which are used by the context switching routine, or scheduler itself. Because the additional hardware cost is small, our context switching design incorporates this feature.

A hardware context switch is not entirely free in terms of cycles in the current ρ-VEX design. To avoid complicating the forwarding logic, context switches are only possible when the pipeline is empty. Because the ρ-VEX has a five stage pipeline, five cycles are needed to flush the pipeline before a context switch can occur. In addition, the context switches are currently controlled by the dynamic reconfiguration controller, which takes three additional cycles to decode and commit a new configuration. Two of these are spent still executing instructions in the old context.

V. Experimental Setup

Our measurements are carried out using the ρ-VEX VLIW softcore processor clocked at 37.5 MHz running on a Xilinx ML605 development board, which incorporates an XC6VLX240T Virtex 6 FPGA. We use a timer connected to the interrupt request input of the processor to generate interrupts at different rates to measure the impact of our approach on the performance of the system.

We quantify the impact on performance by measuring two different values, namely:

1) *Interrupt Latency*: The number of cycles elapsed between the moment an interrupt request is received by the core, and the first instruction of the interrupt handler being executed.
2) *Context switching latency*: The number of cycles elapsed between the moment a context switch is requested (due to an interrupt), and the first instruction being executed in the new context.

Figure 3 shows what these latencies are made up of, namely: pipeline flushing, saving context registers, running the interrupt service routine (in our case the task scheduler), and finally restoring the context registers. By using hardware contexts the latency of saving and restoring registers can be eliminated. We measured these quantities by creating a workload of four programs. At every timer interrupt a scheduler selects a different program to execute, and performs the context switch to that program. The programs themselves have no impact on the measurements, since they are purely dependent on the time it takes to save and restore all context registers.

In order to measure the difference between hardware and software context switching, we wrote a software and a hardware context switching routine. The software version saves the complete context to the stack of the currently running task, stores the stack pointer to a predefined memory location, and starts executing the interrupt handler. The interrupt handler then calls the scheduler in order to schedule the next task. The current stack pointer is then replaced with the stack pointer of the new task. Next, the application context of the newly selected task is restored from the stack, after which control is handed back to the application. The hardware switch routine does not need to save or restore all registers. Instead it only has to do so for the registers used by the interrupt routine, in this case the scheduler.

The scheduler utilizes a linked list in memory to determine which task to switch to; each entry representing a task, with a mapping to another task. When a task completes, the linked list is rebuilt such that the context switching code does not switch back to the completed task, and a context switch is requested immediately using a software trap instruction. When the last task completes, it signals completion to the platform.

Because cache behavior will impact the latencies for saving and restoring the contexts we perform the measurements for different memory access latencies. We measure using latencies from 0 (single cycle memory access) to 30 cycle memory access on cache miss. The cache itself consists of a separate instruction and data cache, respectively 32KiB and 8KiB in size. The size has intentionally been kept small, because the programs under test had to be small as well for the entire memory to fit on the FPGA; it is assumed that, under normal circumstances, larger caches will be used, but the running programs will also use wider regions of more memory. Both caches have single-cycle hit latency for reads. The data cache has a two-cycle latency for writes for both hits and misses, as long as one of the four write buffers is vacant.

To evaluate the context switching overhead in multi-process time-sharing systems, overall performance of the multi-task system is tested on hardware using the cached system. The timer is used to generate an interrupt at a fixed frequency, often referred to as the system "tick," in which a context switch is performed. Clearly, the context switching overhead is directly related to the frequency of the system tick [10]. The frequency of the tick is usually in the order of 50 to 1000 Hz. A lower frequency will lead to lower switching overhead, but higher frequencies will result in a more responsive system. Systems that require more responsiveness will therefore have a higher tick frequency. For example, the Linux kernel uses a system tick of 1000 Hz for desktop systems, but this can be reduced to 100 Hz for server systems to reduce overhead. On the other

Table I
RESOURCE USAGE OF REGISTER FILE WITH AND WITHOUT SUPPORT FOR MULTIPLE CONTEXTS.

	Register File		Core	Increase over Core
	1 Context	4 Contexts		
Slice Registers	806	1392	8529	6.9%
Slice LUTs	10764	15591	35148	13.7%
RAMB18E1	128	128	147	0%
RAMB36E1	0	0	128	0%

hand, the Windows kernel uses 66 Hz. The frequency is varied between tests to evaluate its effect. In addition, the system is evaluated with varying bus latencies. The latencies used are estimates of what the average latency would be for a real off-chip memory system.

A cycle counter available within the ρ-VEX processor is used to measure the time from system reset to the program completion signal, which is given by the task switching implementation when all tasks have completed. For each timer and memory system configuration, both context switching implementations are evaluated. Because all other factors are kept constant, the difference in total execution time is only dependent on the context switching overhead. The speedup between the baseline and hardware context switching implementations is then determined to quantify this overhead.

VI. RESULTS

In Table I we show the increase in resource utilization of the register file when adding support for four contexts. As expected the number of BRAMs used does not increase. Only the number of registers and LUTs increases, since these are used to implement the LVT. While these increases seems large, when compared to the total usage of the core they are less significant. Additionally, note that this increase in resources in the register file is required to support the dynamic reconfigurability of the processor.

As we can observe in Table II, the interrupt latency is 87 cycles for software context switching. The interrupt latency when using hardware contexts is only 5 cycles, solely due to the pipeline flush performed by the trap handling logic. A full context switch, i.e., the time between a tick interrupt request and the execution of the first instruction in the new context, takes 174 cycles using the software implementation, compared to 26 cycles using the hardware contexts.

Table II
INTERRUPT AND CONTEXT SWITCHING LATENCY WITH SINGLE-CYCLE MEMORIES IN CYCLES.

	Software	Hardware	Reduction
Interrupt Latency	87	5	17.4×
Context Switch Latency	174	26	6.7×

In Table III, we can observe the results of the same experiments run using a cached memory system, with a bus latency of 20 cycles. We observe that the improvement due to the hardware context switching is greater in this system, with the improvement in interrupt latency increasing from 17.4 to 23.5×, and the improvement of context switching time increasing from 6.7 to 14.8×.

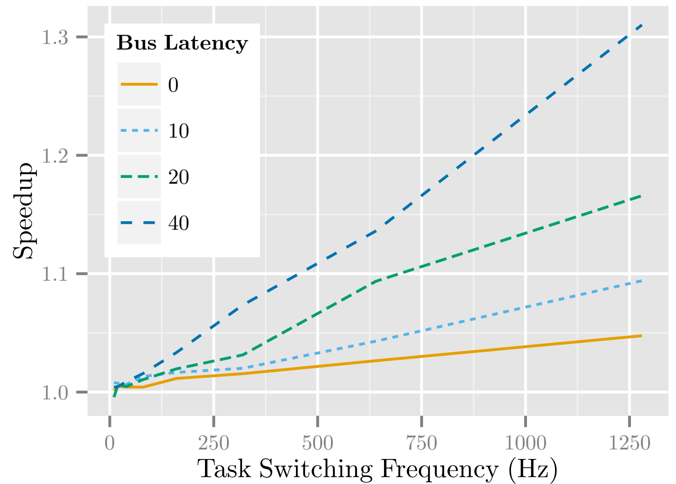

Figure 4. Speedup of the multi-task system due to the hardware context switching implementation.

Table III
INTERRUPT AND CONTEXT SWITCHING LATENCY WITH CACHE AND 20 CYCLES BUS LATENCY IN CYCLES.

	Software	Hardware	Reduction
Interrupt Latency	16798	713	23.5×
Context Switch Latency	31861	2148	14.8×

Figure 4 shows the speedup for different frequencies of the timer tick parameterized for different memory latencies, as measured on hardware using the cached system. It can be seen that in the region of higher task switching frequencies the difference between hardware and software context switching can be quite substantial depending on the memory system. A speedup of over 1.3× can be achieved for a bus latency of 40 cycles at a switching frequency of 1280 Hz.

VII. CONCLUSIONS

The concept of using additional register files to speed up multi-threading performance has been applied in numerous designs in the past. In this paper, we apply the concept to an existing design, exploiting the overcapacity of the BRAMs in the existing implementation of the multi-ported register file and the additional logic required by the parameterized reconfigurability of the ρ-VEX softcore. We have demonstrated that the proposed design can decrease the interrupt latency by a factor of over 20 times in a realistic environment. Likewise, the total context switching time can be decreased by a factor of over 10 times. In a simple multi-task system the effect of this is apparent as the decrease in overhead results in a speedup of 1.3× in the most extreme case evaluated. For applications with few real-time requirements, where the

system tick frequency would be relatively low, the speedup is negligible, as the task switching code would not be executed as often. However, embedded real-time systems that need to process large numbers of events will benefit most from the improvements.

REFERENCES

[1] S. Wong and F. Anjam, “The Delft Reconfigurable VLIW Processor,” in *17th International Conference on Advanced Computing and Communications*, 12 2009, pp. 244–250.

[2] A. Brandon and S. Wong, “Support for dynamic issue width in VLIW processors using generic binaries,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 827–832.

[3] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, “Interrupt service threads-a new approach to handle multiple hard real-time events on a multithreaded microcontroller,” *RTss WIP sessions, Phoenix*, pp. 11–15, 1999.

[4] F. Anjam, M. Nadeem, and S. Wong, “Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–6.

[5] S. Wong, F. Anjam, and F. Nadeem, “Dynamically Reconfigurable Register File for a Softcore VLIW Processor,” in *Design, Automation Test in Europe Conference Exhibition*, March 2010, pp. 969–972.

[6] C. LaForest and J. Steffan, “Efficient Multi-ported Memories for FPGAs,” in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’10. ACM, 2010, pp. 41–50.

[7] M. Saghir and R. Naous, “A Configurable Multi-ported Register File Architecture for Soft Processor Cores,” in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, vol. 4419, 2007, pp. 14–25.

[8] F. Anjam, S. Wong, and F. Nadeem, “A Multiported Register File with Register Renaming for Configurable Softcore VLIW Processors,” in *International Conference on Field-Programmable Technology (FPT), 2010*, Dec 2010, pp. 403–408.

[9] M. Purnaprajna and P. Ienne, “Making Wide-issue VLIW Processors Viable on FPGAs,” *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 33:1–33:16, Jan. 2012.

[10] G. Buttazzo, *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011, vol. 24.

[11] A. Ronnholm, “Evaluation of Real-Time Operating Systems for Xilinx MicroBlaze CPU,” Master’s thesis, Malardalens University, 6 2006.

[12] R. Thekkath and S. Eggers, “The Effectiveness of Multiple Hardware Contexts,” *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 328–337, Nov. 1994.

[13] N. Rafla and D. Gauba, “Hardware implementation of context switching for hard real-time operating systems,” in *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS),* , Aug 2011, pp. 1–4.

[14] K. Tanaka, “PRESTOR-1: a Processor Extending Multithreaded Architecture,” in *Innovative Architecture for Future Generation High-Performance Processors and Systems, 2005*, Jan 2005.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[16] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski, “A 32nm 3.1 billion transistor 12-wide-issue itanium processor for mission-critical servers,” in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, Feb 2011, pp. 84–86.

[17] A. Oliveira, L. Almeida, and A. de Brito Ferrari, “The arpa-mt embedded smt processor and its rtos hardware accelerator,” *Industrial Electronics, IEEE Transactions on*, vol. 58, no. 3, pp. 890–904, March 2011.

[18] T. P. Wijesinghe, “Design and implementation of a multithreaded softcore processor with tightly coupled hardware real-time operating system,” Master’s thesis, 2008. [Online]. Available: http://search.proquest.com/docview/250936948?accountid=27026