Multicore-based Vector Coprocessor Sharing for Performance and Energy Gains

Spiridon F. Beldianu and Sotirios G. Ziavras
Electrical and Computer Engineering Department
New Jersey Institute of Technology
Newark, NJ 07102, USA

ACM Transactions on Embedded Computing Systems
Special Issue on Application Specific Processors
Vol. 13, No. 2, September 2013

For most of the applications that make use of a dedicated vector coprocessor, its resources are not highly utilized due to the lack of sustained data parallelism which often occurs due to vector-length variations in dynamic environments. The motivation of our work stems from (a) the mandate for multicore designs to make efficient use of on-chip resources for low power and high performance; (b) the omnipresence of vector operations in high-performance scientific and emerging embedded applications; (c) the need to often handle a variety of vector sizes; and (d) vector kernels in application suites may have diverse computation needs. We present a robust design framework for vector coprocessor sharing in multicore environments that maximizes vector unit utilization and performance at substantially reduced energy costs. For our adaptive vector unit, which is attached to multiple cores, we propose three basic shared working policies that enforce coarse-grain, fine-grain and vector-lane sharing. We benchmark these vector coprocessor sharing policies for a dual-core system and evaluate them using the floating-point performance, resource utilization, and power/energy consumption metrics. Benchmarking for FIR filtering, FFT, matrix multiplication and LU factorization shows that these coprocessor sharing policies yield high utilization and performance with low energy costs. The proposed policies provide 1.2-2 speedups and reduce the energy needs by about 50% as compared to a system having a single core with an attached vector coprocessor. With the performance expressed in clock cycles, the sharing policies demonstrate 3.62-7.92 speedups compared to optimized Xeon runs. We also introduce performance and empirical power models that can be used by the runtime system to estimate the effectiveness of each policy in a hybrid system that can simultaneously implement this suite of shared coprocessor policies.

Categories and Subject Descriptors: C.1.2 [**Computer Systems Organization**]: Processor Architectures – Multiple Data Stream Architectures (Multiprocessors); *Array and Vector Processors*; *Single-instruction-stream, multiple-data-stream processors (SIMD)* C.1.4 [**Computer Systems Organization**]: Processor Architecture – Parallel Architectures;
General Terms: Performance, Power.
Additional Key Words and Phrases: Vector coprocessor, coprocessor sharing, multicore, FPGA prototyping, Xilinx MicroBlaze.

---

1. INTRODUCTION

SIMD architectures are very efficient for multimedia data processing and scientific applications because they can process simultaneously multiple data elements based on a single vector instruction. VIRAM [Kozyrakis and Patterson 2003b], SODA [Lin et al. 2006] and AnySP [Woh et al. 2010] are single-chip vector microprocessors, and their instruction sets support a comprehensive set of vector operations. Vector microprocessors have been shown to be more effective in embedded media applications than superscalar and VLIW processors [Kozyrakis and Patterson 2002]. Also, a 2-dimensional (matrixoriented) SIMD extension was developed in [Sanchez et al. 2005]. Due to recent advances in programmable devices that have substantially increased their logic cell densities, some FPGA-based soft vector processors have been proposed as well [Cho et al. 2006; Lin et al. 2006; Yiannacouras et al. 2008; Yu et al. 2009; Yang and Ziavras 2005]. An automated co-design tool chain in [Hagiescu and Wong 2011] produces SIMD hardware accelerators and appropriate software for performance and energy gains. The VEGAS soft vector architecture in [Chou et al. 2011] is attached to a single soft Nios II/f Altera processor. It comprises a parameterized number of vector lanes, a scratchpad memory and a crossbar network for shuffle vector operations.

Nevertheless, many high-performance and embedded applications dealing with streams of data cannot efficiently utilize dedicated vector processors for various reasons. Firstly, individual programs often display limited percentage of vector code due to substantial flow control or other operating system tasks. The utilization of an available Vector coProcessor (VP) is then proportional to the vectorized part of the code and the rest of the time the VP will be idle [Azevedo and Juurlink 2009]. Secondly, even with substantial vector code, the needed vector length may often vary across applications or within a single application, as in multimedia [Woh et al. 2010]. Thirdly, several applications have many data dependencies within sequences of instructions, a problem exacerbated without loop unrolling or other compiler optimization techniques [Gerneth 2010]. Such limitations deter efforts to sustain high VP utilization, especially for superpipelined floating-point units (FPUs) in VPs.

Since our sharing policies handle multiple threads, an overview of multithreading that increases the throughput of super-scalar microprocessors is pertinent. Coarse-grain multithreading switches to another thread when the currently active thread stalls. It allows an active thread to run to completion without interruption if it does not encounter stalls. Fine-grain multithreading switches between threads in consecutive clock cycles in order to tolerate short latency stalls, thus increasing the overall thread throughput. Finally, simultaneous multithreading (SMT) [Eggers et al. 1997] allows all threads to compete simultaneously for individual resources. SMT alleviates limited per thread instruction level parallelism to take advantage of superscalar processor resources.

Static power due to leakage current will become an even larger source of power consumption in future technologies. The shrinking of transistors yields increased static power contribution to the total energy consumption [Keating et al. 2007]. Our experimental results in a later section for our three coprocessor sharing policies clearly demonstrate that the total energy consumption of a given application increases as the VP usage decreases. Thus, increasing the utilization of the resources will reduce the total energy consumption for a given task.

In order to increase the overall utilization and throughput of a VP embedded into a multicore chip, a mechanism must be developed for its simultaneous sharing by multiple cores. The terms scalar processor and core processor will be used interchangeably from now on. Sharing could also support multithreading inside the VP with the threads coming from one or more applications. Unlike VP architectures for single cores which are designed with a fixed SIMD width (i.e., vector register size) aiming to service one application at a time, we propose adaptive VP sharing for multicores in order to support multiple-SIMD execution relying on thread-level parallelism (TLP). This design approach can maximize the VP utilization and throughput for two reasons: (a) different cores often handle different vector lengths, thus not being able to individually utilize dedicated VP resources fully; and (b) different vector kernels in the same or different applications often have diverse VP-based computation needs [Woh et al. 2010]. To simultaneously alleviate these drawbacks of rigid VPs while also releasing on-chip real estate for other important design choices, we had previously proposed adaptive VP sharing for multicores that integrates our three basic VP sharing architectures, namely *coarse-grain temporal* (CTS) *sharing*, *fine-grain temporal sharing* (FTS), and *vector lane sharing* (VLS) [Beldianu and Ziavras 2011]. However, that work did not investigate power/energy consumption, and did not present any performance and power estimation models that could be used by the runtime system to fine-tune VP sharing at runtime (based on the needs of individual applications or collections of them simultaneously competing for VP resources). Our system was implemented in the SystemVerilog high-level language and only performance benchmark results were recorded. Here we present an improved VP sharing integration that, besides several architectural improvements and new vector instructions, is implemented on an FPGA and is synthesized in VHDL. Our implementation of various benchmarks on the target Xilinx FPGA device yield accurate figures for performance and power, thus leading to important conclusions about such versatile VP sharing systems. Also, we introduce highly accurate performance and power estimation models.

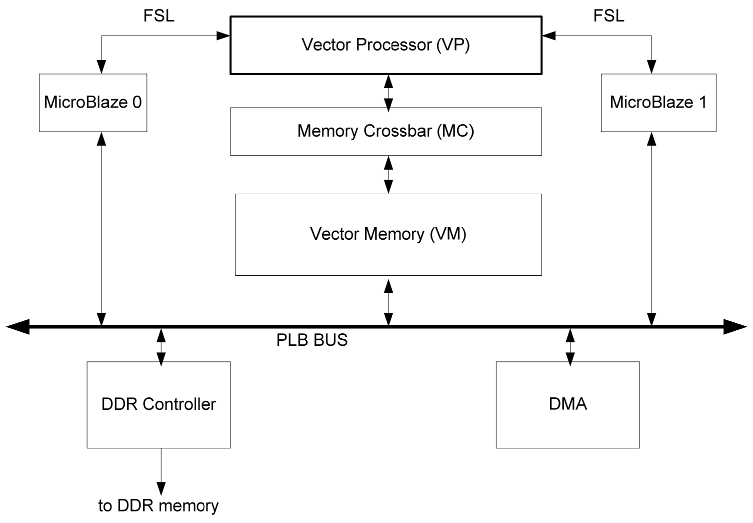

Figure 1. Architecture of the FPGA-based VP sharing prototype (PLB: Xilinx Processor Local Bus, used mostly for data transfers via DMA control; FSL: Xilinx Fast Simplex Link)

A brief introduction of the three VP sharing policies is pertinent. *CTS sharing* consists of temporally multiplexing the execution of sequences of vector instructions or threads containing them. A scalar processor takes exclusive control of the entire VP and then releases it by executing a lock and unlock instruction, respectively. It runs a thread to completion or until it stalls due to a resource conflict (e.g., DMA access conflict). Such a stall forces thread switching for the VP. *FTS sharing* involves spatial (i.e., resource-based) multiplexing of vector instructions coming from the same or different scalar processors. In the former case, the scalar runs in the multithreading mode. A scalar issues an SIMD instruction in a given VP clock cycle according to a chosen arbitration scheme, the simplest one being round robin. The benefit of this approach is that the VP utilization will be increased since data hazards do not exist between instructions issued by different threads or processors, and the VP resource idle times due to data transfers are eliminated or reduced. In FTS, vector instructions coming from different cores or threads can simultaneously execute in the same VP using the same pipelined resources (e.g., adder and multiplier). As shown in Section 3, this type of VP sharing provides the best performance and energy savings.

VLS lane sharing assumes a divisible VP consisting of independent vector lanes with their own execution units. A vector lane is an independent vector subunit containing its own bus interfaces, processing units and vector registers; during its operation it does not compete for resources with any other lane, except for external accesses going to the same memory modules. VLS facilitates the simultaneous allocation of distinct vector lanes, or collections of them, to distinct scalar processors for seamless processing. Based on the chosen set of vector-lane allocation and scheduling policies, a hardware scheduler external to the lanes determines at runtime how to group vector lanes to meet the requirements of applications running on the cores. Therefore, if multiple cores are simultaneously assigned the VP space, each core can use exclusive lanes forming a small-sized VP (as compared to the full-sized VP that comprises all of the lanes, say M). Assuming that each lane contains n elements per vector register, a small-sized VP with m lanes, where m<M, can operate on vector registers having $m \times n$ elements. Obviously, each vector register in the complete VP contains M$\times n$ elements. VLS proves useful when the degree of vectorization in an application running on a core is moderate, thus not requiring the full VP coprocessor space, or when multiple cores require simultaneously vector processing with different vector lengths. Also, VLS could be extended to cases where a VP subset simultaneously handles multiple threads issued by the same or different cores. This approach will be investigated in future work.

The main difference of our work from [Kozyrakis and Patterson 2003b; Woh et al. 2010; Yu et al. 2009] consists of introducing (a) an architecture for vector coprocessor design that can integrate mechanisms for the coarse-grain and fine-grain mixing of threads issued by one or multiple cores, (b) configurable vector lanes that can be grouped for assignment to distinct cores in a manner that eliminates internal resource conflicts, as well as (c) configurable vector register length. Our main objective as compared with all previous aforementioned works, where just one thread can use the entirety of the VP resources, is to provide a hybrid VP architecture framework for sharing the vector coprocessor among multiple scalar cores. This architecture is even more suitable for shared-bus multicores, the current focus of commercial microprocessor technology.

The rest of the paper is organized as follows. Section 2 presents the details of our three basic vector-sharing architectures. Section 3 presents relevant performance, power and energy results for popular vector-dominant floating-point applications and it is followed by a comparative analysis. Section 4 proposes performance and power models. Finally, conclusions are drawn in Section 5.

2. BASIC VP SHARING ARCHITECTURES

2.1 VP Architecture Framework

In order to validate the FTS, CTS and VLS vector-sharing contexts, we have prototyped a system on a Xilinx Virtex-5 XC5VLX110T FPGA. It consists of two scalar processors, an 8-way data-path partitioned VP with an 8-way vector memory load/store unit for parallel data memory accesses, a VP-memory interconnecting crossbar, and an 8-bank low-order interleaved on-chip vector memory. MicroBlaze, a 32-bit embedded RISC soft core provided by Xilinx, forms each scalar; it employs the Harvard architecture and uses the FSL interface to connect with up to eight coprocessors [Xilinx Inc. 2010b]. Instructions issued to our VP use a 32-bit FSL bus. Since the Xilinx EDK (Embedded Development Kit) tool kit limits the operating frequency of MicroBlaze to 125 MHz, without loss of generality we optimized the entire design for this target frequency.

Figure 1 presents the complete system prototype that was implemented on this Virtex-5 FPGA using the Xilinx ISE tools. The Vector Processor (VP), Memory Crossbar (MC), Vector Memory (VM) and Vector Memory Controller (VMC) are custom IPs that we designed in VHDL, and the rest of the system was generated using the Xilinx EDK tool, version 12.3. The VP basic structure conforms to the VIRAM lane-based architecture [Kozyrakis and Patterson 2002; 2003a and 2003b] that was proposed to connect to a single core. The vector lane space in our design can be partitioned among multiple cores, as needed. This adaptable structure can be used to assign varying numbers of vector lanes to the cores throughout execution based on individual application needs, as per the VLS design choice. Each vector lane contains a subset of the elements from a larger vector register, one FPU and a memory load/store (LDST) unit.

Figure 2 shows the overall structure for vector lane sharing. Our current FPGA-based prototype has M=8 lanes and L=8 memory banks. The LDST unit from each lane can operate with or without a vector stride, and can also carry out indexed memory accesses using the crossbar going to the memory. The crossbar allows for concurrent accesses from LDST units to distinct memory banks and also provides round-robin arbitration when many LDST units are accessing the same memory bank.

A distinct Vector Controller (VC) is attached to each scalar processor from which it receives instructions. Such instructions can be of two types: vector instructions to move and process data, which are forwarded to vector lanes, and control instructions which are forwarded to the Scheduler. Control instructions are used for communications between scalar processors and the Scheduler, for purposes such as acquiring VP resources and the current status of the VP. The scalar processor always receives an acknowledgement word in response to a control instruction. The VC forms a pipeline with two clock cycles latency, where the first stage is used for decoding, and the second stage is used for hazard detection and register renaming. All three types of data hazard (i.e., RAW, WAR and WAW) are resolved in the latter stage. Also, in this stage the VC requests from the Scheduler access to the instruction bus in order to broadcast the vector instruction to the vector lanes. It is the Scheduler’s responsibility to arbitrate between requests coming from both VCs and to acknowledge the one that will get access to the instruction bus. After decoding and hazard detection, the VC broadcasts the vector instruction to its assigned lanes by pushing it with the appropriate vector element ranges into small instruction FIFOs located in the respective lanes. The Scheduler handles the control instructions coming from the scalar processors. Based on requests from the cores, the Scheduler properly configures the vector lanes. Also, as mentioned previously the Scheduler is responsible for arbitrating on concurrent requests coming from both VCs; control signals for the instruction bus are then asserted based on the arbitration decision.

As Figure 3 shows, the lane has a LDST unit (left side) and an FPU (right side). Similar to VIRAM, the LDST unit works with the MC memory crossbar. As mentioned, it can operate with or without a vector stride, and can also carry out indexed memory accesses using the crossbar going to the memory. Additional features are added in our implementation: vector element load/store instructions, where just one element from the vector is loaded or stored, and shuffle instructions to transfer elements between different lanes using a communication pattern stored in any vector register. For shuffle instructions, the LDST unit computes the target lane and destination element, and the data is transferred via the MC crossbar using the data path for standard memory accesses. The MC Arbiter can distinguish between memory and shuffle transfers in order to forward properly the data to the appropriate destination. Table I presents the LDST instructions supported in our current implementation.

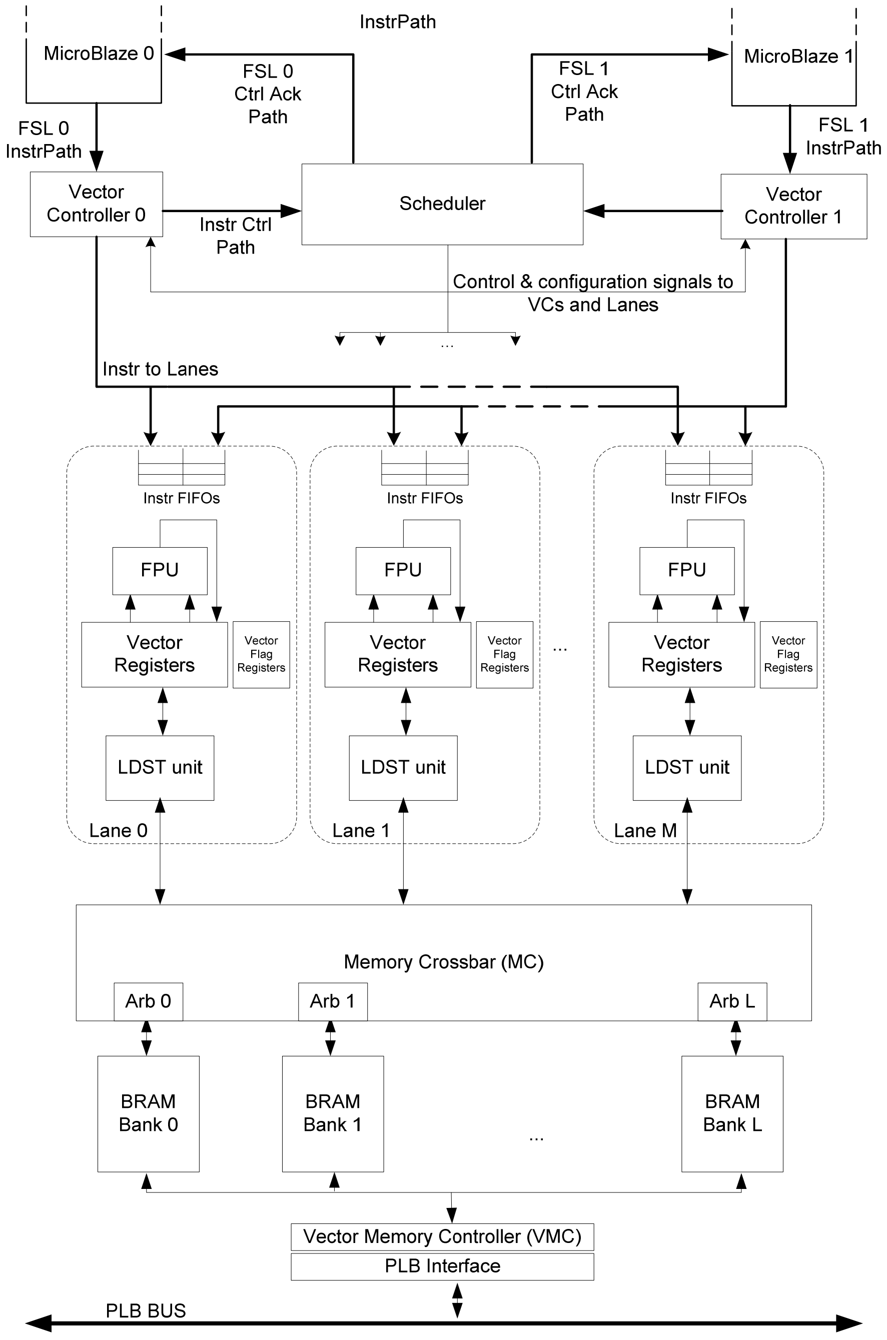

Figure 2. M vector lanes shared between two MicroBlaze processors (FSL serves as the instruction path between a MicroBlaze and its associated Vector Controller, through the Scheduler; BRAM: Xilinx Block RAM; each MUX in the figure is part of the respective lane).

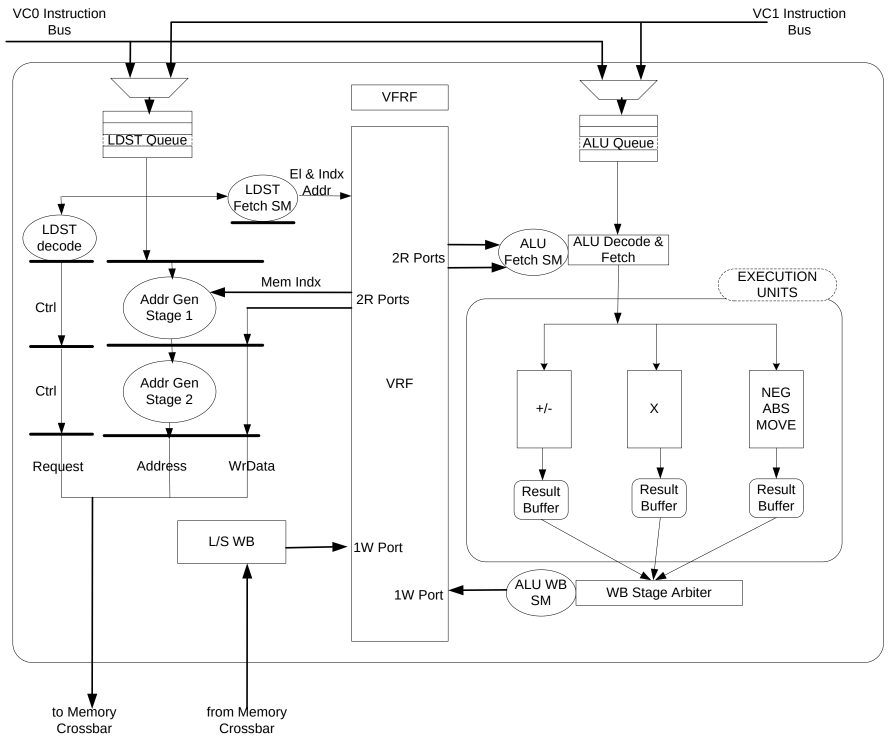

Figure 3. Vector Lane architecture

Table I. Load/Store (LDST) Instructions

	Details
VLD **VST**	Unit stride Vector Load and Store instructions
VLDS **VSTS**	Stride Vector Load and Store instructions. Stride could take values up to 1024.
VLDX **VSTX**	Indexed Vector Load and Store instructions.
VELLD **VELST**	Element Load and Store instructions.
VSHFL	Vector Shuffle instructions. The instruction takes 3 parameters: destination vector register, source vector register and vector register containing permutation information.

The initiation latency to fill-up the pipeline is 8, 13 and 8 clock cycles for a LDST, add-subtract/multiply and any other ALU instruction, respectively. The ALU of our current design contains 6-stage multiply and add single-precision FPUs. The latency parameters are provided by the Xilinx IP Core Generator and meet the requirements for a 125 MHz design frequency. The rest of the cycles, up to 13, are distributed as follows: the VC pipeline has two stages, one for hazard detection, and one for register renaming, scheduling and issue to lanes; the lane instruction FIFOs consume one cycle; lane decoding, operand fetching and issue to execution units takes two clock cycles mainly caused by the latency of BRAMs; the result buffers involve one cycle; finally, one clock cycle is taken for the lane to inform the hazard detection mechanism in the VC about instruction completion. Without loss of generality, the FPU can execute single-precision floating-point addition, subtraction and multiplication, and can also evaluate the absolute and negate operations. As shown in Figure 3 the LDST and ALU instructions involve separate paths. Therefore, it is possible to have concurrent execution of LDST and ALU instructions as long as there is no data dependence between them. LDST instructions are always executed and committed in order. ALU instructions are issued in order but might commit out of order due to different pipeline depths in the execution units. However, this does not violate data dependencies since instructions that execute at the same time in a lane have no data dependence. Table II summarizes the ALU instructions supported in our current implementation. Each lane assigned to a VC informs it upon instruction completion and the entire SIMD instruction is considered completed by the VC when all the lanes have completed this step. Each SIMD instruction is labeled by the VC with a unique tag and dedicated signaling from the lane to its assigned VC informs the latter about the completion of the instruction.

The elements corresponding to one vector register are distributed across multiple lanes in low-order interleaved fashion and the number of elements from a vector register corresponding to one lane is configurable. Each instruction consumes a start-up latency plus a number of cycles equal to the number of elements stored in the lane's vector register minus one. An instruction without dependencies consumes in the LDST or ALU unit a number of pipeline cycles equal to the size of the vector register used in the lane. Each lane contains a multi-ported Vector Register File (VRF) with 512 32-bit locations efficiently implemented with Xilinx FPGA 36Kbit BRAMs (Block RAMs). Each of the LDST and ALU units requires two reads and one write per clock cycle. Therefore, the memory has two write and four read ports (2W/4R), and is implemented by doing replication (2×) and multi-pumping with a double frequency [LaForest and Steffan 2010]. Each lane contains four configuration registers which are updated at runtime by the Scheduler. The first register contains the VC ID to which the lane is assigned while the second register contains the number of lanes assigned to the particular VC to which this lane is assigned. This register is updated when switching between the CTS/FTS and VLS operating contexts. The third register contains the fixed lane ID (or lane index). The fourth register contains the number of elements from a vector register which are located in the same lane. Since the VRF memory within each lane has fixed size, increasing the number of elements in a vector register will automatically decrease the number of available registers. Table III presents some valid combinations of the vector length and the number of available vector registers. It is a software decision to tune the vector length and the available number of registers in order to optimize the execution time and/or power consumption for a specific task. Besides the VRF memory in each lane, there is a Flag Vector Register File (FVRF) memory which contains 512 1-bit elements. Each bit is used as a mask for conditional execution of vector instructions on the corresponding element in the VRF.

The Vector Memory (VM) contains eight low-order interleaved Xilinx BRAM banks for a total capacity of 64 Kbytes (8 banks x 8 Kbytes per bank). Without crossbar conflicts in accessing the VM banks, eight 32-bit data transfers can be performed on each clock cycle using the eight LDST units, giving a peak bandwidth of 32Gbs with a design frequency of 125 MHz. Of course, this bandwidth will double with an expanded design for double-precision floating-point operations and respective data transfers. Each BRAM is a true dual-port memory; one port is used for data transfers between the VM and the VP's register file, and the second port is used for data transfers between I/O controllers and the VM through the PLB interface. Therefore, this architecture supports concurrency and yields high bandwidth for data transfers involving the VM.

2.2 Scheduling Procedure

The Scheduler controls the working context for the entire VP. Based on the chosen working state, the Scheduler provides configuration signals to all lanes and VCs. The signals for a particular lane provide information about: a) which VC the vector lane is assigned to, being VC 0, VC 1 or both; b) the total number of lanes assigned to the VC, including this particular lane; c) the offset/index of the lane in the lane array assigned to that VC; and d) the number of elements from a vector register which are located in this lane. The information from the first configuration signal (i.e., configuration a) is used by the lane to notify the appropriate VC of instruction completion, and the information derived from configurations b), c) and d) is used by the lane's LDST unit to properly translate addresses for memory accesses. The configuration signals provided to the VC by the Scheduler configure the former to work either in the exclusive context (i.e., one thread arriving from one scalar processor) or in the lane-sharing context (i.e., two distinct threads arriving from the two scalar processors).

Table II. ALU Instructions

Instr.	**Details**
VMUL	Vector-Vector Multiplication.
VADD	Vector-Vector Addition.
VSUB	Vector-Vector Subtraction.
VMULS	Vector-Scalar / Scalar-Vector Multiplication.
VADDS	Vector-Scalar / Scalar-Vector Addition.
VSUBS	Vector-Scalar / Scalar-Vector Subtraction.
VMOV	Vector move instruction.
VNEG	Vector negate instruction.
VABS	Vector absolute instruction.
VFLD	Load Vector Flag from Scalar instruction.
VFMOV	Vector Flag move instruction (from scalar to Vector).
VFNEG	Vector Flag complement instruction.

Table III. Examples of Vector Length and Number of Registers

	Elements per register (1 lane)	Vector Length	Number of available registers
8 lanes	4	32	32
8 lanes	8	64	32
8 lanes	16	128	16
8 lanes	32	256	8
4 lanes	4	16	32
4 lanes	8	32	32
4 lanes	16	64	16
4 lanes	32	128	8

Figure 4 shows some of the possible states for the Scheduler; each cell in the figure contains the state of the corresponding lane: which VC it is assigned to, the total number of lanes assigned to that VC, the lane index, and the number of elements from a vector register in that lane. STATE1 is similar to CTS in which all 8 lanes are assigned only to MB0 through VC0 and the Vector Length is 4×8=32. In STATE2, both scalar processors have access to all 8 lanes using FTS sharing; the value of VL is 8×8=64 with the application running on both MicroBlazes. In STATE3, each VC has 4 lanes assigned to it; the VL value for the application running on MB0 and MB1 is 16 and 32, respectively. Finally, STATE4 has 4 idle lanes with 4 lanes assigned to VC1 and the VL value is 64.

Each MicroBlaze can use a set of four indivisible instructions to communicate with the Scheduler. These are VP_REQ, VP_REL, VP_GETSTAT1 and VP_GETSTAT2; in response, the Scheduler always replies with a message. For a core to get access to the entire VP or to a subset of its lanes, the VP_REQ instruction is used. This instruction contains two parameters: i) *vl_size*, which indicates the required vector length (i.e., the number of vector elements in a vector register). *vl_size* can take the values: 4, 8, 16, 32, 64, 128 and 256; and ii) *perf_req*, a three-bit field which indicates the performance requirements, thus distinguishing among eight priority levels. However, without loss of generality, in our current implementation this field assumes two active values: *perf_req*=3'b000 corresponds to a low priority/performance application; and *perf_req*=3'b111 represents high priority. Based on the current VP state, any other pending VP requests, and the details of the current request, the Scheduler decides to grant a scalar processor request or not, and informs the requesting processor accordingly. In the extreme case where VP_REQ instructions arrive from both scalar processors in the same clock cycle, the Scheduler will reply to both of them but will positively acknowledge only one. Figure 5 shows the reply word in response to a VP_REQ instruction. For a successful request, the Scheduler will reply with the acquired VL value and the acquired performance fields. In the case of an unsuccessful request, the Scheduler will transmit the available VL value and the currently available highest priority.

Lane	**L0**	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**
STATE1	VC0 8 0 4	VC0 8 1 4	VC0 8 2 4	VC0 8 3 4	VC0 8 4 4	VC0 8 5 4	VC0 8 6 4	VC0 8 7 4
STATE2	VC0/1 8 0 8	VC0/1 8 1 8	VC0/1 8 2 8	VC0/1 8 3 8	VC0/1 8 4 8	VC0/1 8 5 8	VC0/1 8 6 8	VC0/1 8 7 8
STATE3	VC0 4 0 4	VC0 4 1 4	VC0 4 2 4	VC0 4 3 4	VC1 4 0 8	VC1 4 1 8	VC1 4 2 8	VC1 4 3 8
STATE4	Idle	Idle	Idle	Idle	VC1 4 0 16	VC1 4 1 16	VC1 4 2 16	VC1 4 3 16

Figure 4. State examples for the Scheduler

In response to the VP_GETSTAT1 instruction, the Scheduler will reply with the following information: status of VC1 and VC0 (idle or busy), number of lanes assigned to VC1 and VC0, and performance status of VC1 and VC0. In response to the VP_GETSTAT2 instruction, the Scheduler will reply with: status of VC1 and VC0 (idle or busy) and Vector Length assigned to VC1 and VC0. The VP_REL instruction is used to free all the VP resources previously acquired by a scalar processor.

In our current VP prototype, three types of software-based adaptation are facilitated to take advantage at runtime of any available VP resources: (a) at the core-run software level, where the core changes at runtime the routine that implements a needed vector kernel based on the available VP resources (the routines may be parameterized by vector length or performance level); (b) closer to the VP level, the Scheduler is able to appropriately configure the working context of the VP based on its current state and the current set of requests coming from the scalar cores; and (c) at the lane level, where the Scheduler can configure some of the vector lane parameters (e.g., the number of elements per vector register contained in a vector lane).

Based on its current state and the request parameters, the Scheduler decides if any resources are available and replies with a successful or unsuccessful acknowledge message. Based on this information and the application routines that it has to run, the scalar processor makes the final decision on the number of lanes to acquire. To avoid the duplication of stored code, generic parameterized routines may be developed (e.g., routines with such parameters as the vector length, number of registers to be used, etc.). Figure 6 shows the current algorithm run by the Scheduler and Table IV presents some examples of Scheduler state transition based on a request coming from one of the scalar processors. Under CTS each vector kernel in a thread runs to completion before releasing all the VP resources. In VLS our current scheduler gives equal priority to competing threads by assigning the same number of exclusive VP lanes to each thread. In FTS our current scheduler can accommodate simultaneously multiple threads in any given cycle as long as they need different VP resources; when competing for the same ALU or LDST unit, the scheduler applies round-robin arbitration per unit. In future work, we will develop scheduling algorithms that may assign different numbers of lanes to the cores based on known thread priorities; this will be beneficial primarily to VLS. In FTS, an instruction-based weighted round robin policy can be used to control the lanes. This solution will introduce flexibility to control individual thread performance (i.e., to satisfy thread quality of service at the expense of adding more complex arbitration logic). Since our main objective here is to demonstrate the viability of VP sharing among cores and threads, for the sake of brevity the development of very sophisticated scheduling schemes will become a future research objective. Also, as our experimental results dictate in the next section, the specific configuration to be chosen could be driven by power/energy and performance tradeoffs. Thus, future work will focus on designing schedulers to dynamically change the working context or adjust the number of active lanes in order to optimize a given objective function involving performance, power and/or energy.

31 28	27 17	16 14	13 1	0 0
OP_CODE	Acquired VL/ Maximum Avail VL	Acq PERF / Avail Perf	RSVD	SUCC

Figure 5. Scheduler to MicroBlaze reply word in response to a VP_REQ

```
if 8 lanes IDLE {
        if req_perf=low {
                assign 4 lanes to VC;
                VL=requested_VL;
                REPLY=SUCC;
        }
        if req_perf=high {
                assign 8 lanes to VC;
                VL=requested_VL;
                REPLY=SUCC;
        }
}
if 4 lanes IDLE {
                assign 4 lanes to VC;
                VL=requested_VL;
                REPLY=SUCC;
}
if all 8 lanes BUSY {
        if requested_VL = current_VL {
                assign 8 lanes to VC;
                 VL=requested_VL;
                REPLY=SUCC;
        } else {
                REPLY=UNSUCC;
        }
}
```

Figure 6. Scheduler algorithm

Table IV. Examples of Transition for Scheduler states

Scheduler state	**Request parameters**	**Reply**	**Scheduler next state**
all 8 lanes IDLE	MB0 req req_vl=64 req_perf=high	SUCC VL=64 perf=high	8 lanes assigned to VC0 VL=64 els per lane=8
all 8 lanes IDLE	MB0 req req_VL=128 req_perf=low	SUCC VL=128 perf=low	8 lanes assigned to VC0 VL=128 els per lane=32
all 8 lanes assigned to VC0 VL=64	MB1 req req_VL=64 req_perf=high	SUCC VL=64 perf=high	8 lanes assigned to VC0/1 VL=64
	MB1 req req_VL=128 req_perf=high	UNSUCC	8 lanes assigned to VC0 VL=64
4 lanes assigned to VC0 VL=64	MB1 req req_VL=128 req_perf=high	SUCC VL=128 perf=low	4 lanes assigned to VC0 4 lanes assigned to VC1

```
STEP 1.1 Lock DMA resource
STEP 1.2 Transfer data from DDR to
         Vector Memory (VM)
STEP 1.3 Unlock DMA resource

STEP 2.1 Acquire VP resources
STEP 2.2 Call VP routine to process data from VM
STEP 2.3 Release VP resources

STEP 3.1 Lock DMA resource
STEP 3.2 Transfer processed data from VM to DDR
STEP 3.3 Unlock DMA resource
```

Figure 7. Main MicroBlaze routine for CTS, FTS and VLS sharing

```
STEP 2.1
while (ack != IDLE) { //wait until the VP is idle
   VP_REQ ack;        // Scheduler returns a positive
                      //or negative reply;
}
STEP 2.2
  VLD VR0, A;   // Processor starts using the VP; loads
   //the vector register (A is address in Vector Memory)
  …
  VST VR4, B;       // Processor finishes the routine;
                    // saves the vector result
                    // (B is address in Vector Memory)
STEP 2.3
VP_REL ack;    // Unlock the VP resources and receives
               // a reply if successful or not;
```

Figure 8. CTS vector sharing MicroBlaze routine

In our current implementation, under CTS only one VC can issue an instruction to vector lanes at any time. In the FTS context, both VCs can issue simultaneously instructions to the lanes. The lane execution pipeline is capable of processing simultaneously instructions issued by both scalar processors since multiple vector instructions can simultaneously reside in the pipeline. Under these circumstances, FTS requires vector register renaming because the scalar processors must be assigned distinct vector registers. Usually small- and medium-scale SIMD machines are currently used as stream processors. Data can be streamed into the VM of our VP-based structure using the DMA capability; the program then operates on this data using the VM as a data workspace, and the results are streamed back to the main memory using again DMA control. This data streaming can occur simultaneously with arithmetic computations. Figure 7 shows how the main routine of a MicroBlaze is developed for CTS/FTS and VLS sharing, and Figure 8 shows steps 2.1 to 2.3 for CTS sharing. Just before a thread becomes active, the software may clear all vector registers using a VP clear instruction. Another possibility is to implement additional hardware to support a local reset controlled by the Scheduler and triggered when the VP space is exclusively acquired by one of the scalar processors. When the scalar processor finishes the vector routine, it releases the coprocessor by issuing a VP_REL instruction. Prior to this instruction the MicroBlaze code makes sure that no vector register is dirty; also, the state of the vector processor for the respective MicroBlaze program is saved back into the memory. Therefore, the state of the VP must be saved before the VP is released in a shared environment.

Under FTS, vector instructions received from both scalar processors share the VP resources. This context resembles fine-grain multithreading in superscalar processors, and increased throughput is expected because there are no data dependencies between instructions coming from different processors.

Under VLS, if the *req_perf* value is low (*req_perf=3'b000*), the Scheduler splits the VP into two distinct lower-sized VPs with each one having its own vector length. For example, if MB0 requests a VL=32 with *req_perf=low* and MB1 requests a VL=64 with *req_perf=low*, the final state of the Scheduler will be: 4 lanes assigned to VC0 with 8 elements per lane from the same vector register and 4 lanes assigned to VC1 with 16 elements per lane from the same vector register. Then, the VP will serve simultaneously two threads of different vector lengths.

2.3 Resource Consumption

Table V shows resource consumption figures for the VP and VM in the Virtex XC5VLX100T FPGA device. Virtex-5 FPGAs contain a column-based architecture comprised of logic slices, 36-Kbit block RAMs called BRAMs (RAMB36_EXP), DSP slices (DSP48E) and many I/O hardwired IPs. Each logic slice can implement functions using four 6-input look up tables (LUTs) and four flip-flops; the LUTs can also be configured to realize dual-output 5-input LUTs. A LUT is a 64-bit memory capable of then realizing any of 32 or 64 functions. The DSP48E slice is based on a 25x18 bit multiplier and a 48-bit adder/subtractor/accumulator. Note that a vector lane contains an LDST unit, an ALU unit, a VRF and a FVRF; the VP contains 8 lanes, 2 VCs and one Scheduler. Except for the last row in the table, the percentage values are shown relative to the total design resource consumption. As expected, most of the design is occupied by ALU units. Each lane consumes 1719 slice LUTs and 2600 slice registers (i.e., 10.3% and 11%, respectively, of the entire design), and the device consumption collectively by the VC and Scheduler is less than 4%. The overall device consumption by the VP and VM is shown in the last row of Table V it is 16,765 slice LUTs and 23,628 slice registers, which represent 24% and 34%, respectively, of the VLX110T resources. The rest of the FPGA resources can be used for the realization of scalar processors, buses, DMAs, I/O controllers and other IPs.

Table V. Resource consumption in the Virtex XC5VLX110T FPGA device

	Slice REGISTERs	Slice LUTs	RAMB36_EXP	DSP48E
LDST unit	713 (3.0%)	351 (2.1%)	-	2
ALU unit	1818 (7.7%)	1500 (8.9%)	-	2
VRF	258 (1%)	99 (<1%)	1	-
FVRF	44 (<1%)	44 (<1%)	-	-
LANE	2600 (11%)	1719 (10.3%)	1	4
VC	448 (1.9%)	296 (1.8%)	-	-
Scheduler	282 (1.2%)	357 (2.1%)	-	-
VP (8LANES)	21955 (92.9%)	14071 (84%)	8	32
VM	1820 (7.7%)	2874 (17.1 %)	16	-
VP+VM	**23628 (34%)**	**16765(24%)**	**16(16.2%)**	**32(50%)**

3. EXPERIMENTAL RESULTS AND ANALYSIS

3.1 Benchmarks

Four vector intensive programs, namely 32-tap FIR filtering, 32-point decimation-in-time radix-2 butterfly FFT, 1024x1024 dense matrix multiplication (MM), and LU decomposition were tested on our VP architecture. The performance evaluation numbers in our experiments include the I/O DMA transfer times (i.e., DDR to/from the VM). The routines for the VP were hand-coded, trying to improve the instruction throughput by using data prefetch via load instructions. Figure 7 shows how the main routine of each MicroBlaze processor is built for CTS sharing. With FTS and VLS sharing, there is no exclusive access to the VP so STEPs 2.1 and 2.3 are removed; that is, a request for VP resources can be granted without waiting for the VP to be idle. Except for LU decomposition, each MicroBlaze uses its own partition in the VM and there are no data dependencies between threads running on the two processors. In order to have exclusive access to the single DMA module, the Mutex IP core provided by Xilinx is used. The lock and unlock procedures for the DMA module require locking and unlocking the Mutex, respectively. For an in-depth evaluation of our architecture, for each benchmark we created several performance-power scenarios that involve loop unrolling, different vector lengths and instruction rearrangement optimization.

32-tap FIR filtering is implemented using the outer product [Sung and Mitra 1987] that avoids the reduction operation. Using a loop of 32 iterations and a given vector length for the VL, we compute VL results at the end of the loop. 45 FIR scenarios were produced for various combinations of: (i) CTS, FTS and VLS VP sharing contexts; (ii) vector lengths of 32, 64, 128 and 256; (iii) no loop unrolling, or unrolling once or three times; and (iv) instruction rearrangement optimization. FFT is implemented using a five-stage butterfly; each stage involves complex multiply and add vector operations, and a shuffle operation. 12 scenarios were produced for various combinations of: (i) CTS, FTS and VLS contexts; (ii) vector lengths of 32 and 64; (iii) no unrolling or unrolling once; and (iv) instruction rearrangement optimization. Since the number of vector registers for FFT is more than 16, we cannot use a vector length greater than 64 (see Table III).

MM is based on the same procedure as FIR filtering using Single-precision real Alpha X Plus Y (SAXPY) in a loop to obtain one row result at the end of the loop; 21 scenarios were produced for combinations of: (i) CTS, FTS and VLS contexts; (ii) vector lengths of 32, 64, 128 and 256; (iii) no unrolling or unrolling once; and (iv) instruction rearrangement optimization. LU decomposition consists of generating the L and U matrices from a dense 128×128 matrix using the Doolittle algorithm [Golub and Van Loan 1996]. Since the VP implementation assumes vector lengths having powers of two, the value of VL is successively halved during Gaussian elimination, starting with 128 and then becoming 64, 32 and 16. More specifically, VL is halved when the row/column size of the remaining submatrix is half of the current VL value. Therefore, the time for LU decomposition depends on the execution times for VL=128, 64, 32 and 16. Three scenarios were produced corresponding to the CTS, FTS and VLS contexts. Under FTS and VLS, the workload is split evenly between the two MicroBlaze processors.

For FIR, FFT and MM in the VLS and FTS contexts, both scalar processors run the same routine. For all benchmarking scenarios under CTS that keep the VP active throughout execution, the performance is independent of the number of involved cores and threads. Compared to the classic implementation where a VP is always tied to the same scalar processor, the advantage of CTS in a multicore environment is that VP ownership can change dynamically for more robust application realization.

3.2 Evaluation Procedure

Execution times and the utilization of lane units were obtained with ModelSim simulations using the RTL system model. The Xilinx XPower tool [Xilinx Inc. 2010a] was then used to calculate the dynamic and static power dissipation based on data stored in the simulation record files (.vcd files recording the switching activities of all the logic and wires in the FPGA, and which are generated by ModelSim during the timing simulations with the place-and-route netlist). To obtain realistic power figures, the timing simulations employed real floating-point input data. In all power calculations, all the design nets were matched; i.e., toggle information was extracted from all the nets in the netlist. Besides the execution times under various scenarios, we also produced figures for the average utilization of the ALU and LDST units (per vector lane). The ALU average utilization is defined as the number of results produced by a lane's arithmetic and logic execution unit in 100 clock cycles, and the LDST utilization is the number of data words sent or received to or from the MC crossbar in 100 clock cycles.

3.3 Performance and Power Analysis

Tables VI-IX show the ALU and LDST utilization and performance results in reference to the execution time for various configurations of the system: a) one scalar processor working without the VP and the DMA unit, and all data and instructions are pre-stored in the on-chip local memory; b) two scalar processors working without the VP and the DMA unit, and all data and instructions are pre-stored in the on-chip local memory; c) a scalar processor using exclusively the VP and the DMA unit (this represents CTS); d) two scalar processors working with the VP in the FTS context and the shared DMA unit; e) two scalar processors working with the VP in the VLS context and the shared DMA unit (each MicroBlaze acquires four lanes); and, for fair comparisons across platforms, f) 3.2 GHz Intel Xeon SL7DX (Nocona) processor in a commercial PC running same algorithmic implementation as VP scenarios except that the vectorized code was replaced with sequential C code (standard); the compilation was done using O3 option with no vectorization (no SSE extensions); and g) the same Xeon processor running optimized routines with FFTW library for FFT [Frigo and Johnson 2005], Intel Integrated Performance Primitives (IPP) [Intel IPP 2010] for FIR and LU factorization and Math Kernel (MKL) [Intel MKL 2011] libraries for matrix multiplication; the compilation was done using O3 option with SSE3 extensions. For each one of the c), d) and e) configurations, we present the results for three distinct scenarios that combine different vector lengths with loop unrolling. For FIR filtering, the results are shown in ns per dot product. For FFT, the results are in µs per 32-point complex FFT operation, and for MM the results are in µs for the calculation of a single element in the product matrix. Besides the total execution time for the LU decomposition of a 128×128 dense matrix, Table IX shows the time to process one single row for various vector lengths. Since recording a .vcd file for an entire LU decomposition task is impractical due to its size, Table XIV shows the power and energy dissipation for one processed row in Gaussian elimination.

Table VI. Performance comparison for 32-tap FIR

		Average utilization (%)		Execution Time (ns)	Speedup
		ALU	LDST		
One MB w/o VP		N/A	N/A	4060	1
Two MB w/o VP		N/A	N/A	2030	2
CTS	VL=32; no loop unrolled	17.51	8.86	371.25	10.93
	VL=128; no loop unrolled	39.24	19.94	165.56	24.52
	VL=128; unrolled three times	83.31	42.51	78.31	51.85
FTS	VL=32; no loop unrolled	34.97	17.70	186.01	21.83
	VL=128; no loop unrolled	75.66	38.24	85.98	47.22
	VL=128; unrolled three times	99.71	50.67	65.19	62.27
VLS	VL=32; no loop unrolled	27.68	14.09	234.12	17.34
	VL=128; no loop unrolled	49.51	25.29	131.28	30.92
	VL=128; unrolled three times	89.89	45.71	72.21	56.22
FTS	VL=4; unrolled three times	9.47	4.74	685.11	5.92
VLS		10.94	5.83	593.24	6.84
GPP Xeon - standard		N/A	N/A	340.08	11.94
GPP Xeon – IPP library		N/A	N/A	9.23	439.87

Table VII. Performance comparison for 32-point Complex FFT

		Average utilization (%)		Execution Time (µs)	Speedup
		ALU	LDST		
One MB w/o VP		N/A	N/A	160.01	1
Two MB w/o VP		N/A	N/A	80.01	2
CTS	VL=32; no loop unrolled	43.29	23.38	3.264	49.02
	VL=32; unrolled once	65.10	34.78	2.172	73.66
	VL=64; unrolled once	78.92	43.09	1.782	89.78
FTS	VL=32; no loop unrolled	76.28	42.39	1.844	86.76
	VL=32; unrolled once	87.20	46.44	1.618	98.89
	VL=64; unrolled once	89.45	48.60	1.573	101.72
VLS	VL=32; no loop unrolled	62.74	35.11	2.192	72.99
	VL=32; unrolled once	74.23	41.60	1.848	86.58
	VL=64; unrolled once	79.18	44.56	1.701	94.06
GPP Xeon- standard		N/A	N/A	100.01	1.60
GPP Xeon – FFTW		N/A	N/A	0.312	512.85

Table VIII. Performance comparison for Matrix Multiplication

		Average utilization (%)		Execution Time (µs)	Speedup
		ALU	LDST		
One MB w/o VP		N/A	N/A	130.90	1
Two MB w/o VP		N/A	N/A	65.45	2
CTS	VL=32; no loop unrolled	20.37	20.70	10.09	12.97
	VL=32; unrolled once	33.94	34.50	6.03	21.71
	VL=128; unrolled once	68.30	69.51	3.01	43.49
FTS	VL=32; no loop unrolled	40.59	41.29	5.055	25.89
	VL=32; unrolled once	67.09	68.20	3.048	42.95
	VL=128; unrolled once	97.32	98.91	2.114	61.92
VLS	VL=32; no loop unrolled	33.83	34.34	6.086	21.51
	VL=32; unrolled once	53.51	54.45	3.791	34.53
	VL=128; unrolled once	81.88	83.40	2.494	52.48
GPP Xeon - standard		N/A	N/A	20.56	6.36
GPP Xeon – MKL library		N/A	N/A	0.651	201.38

Table IX. Performance comparison for LU decomposition

		Average utilization (%)		Execution Time (µs) per row of size VL	Execution Time (µs) for entire LU dec.	Speedup
		ALU	LDST			
One MB w/o VP		N/A	N/A	N/A	1,034,340	1
Two MB w/o VP		N/A	N/A	N/A	517,170	2
CTS	VL=16	4.73	5.34	0.632	5,137	201.35
	VL=32	9.88	10.36	0.632		
	VL=64	20.11	20.42	0.632		
	VL=128	40.44	40.54	0.632		
FTS	VL=16	8.32	8.54	0.312	2,568	402.78
	VL=32	18.74	21.08	0.316		
	VL=64	39.93	41.36	0.316		
	VL=128	81.05	82.30	0.316		

VLS	VL=16	8.70	11.11	0.316	3,522	293.68
	VL=32	19.05	21.03	0.316		
	VL=64	39.62	41.05	0.316		
	VL=128	53.86	54.95	0.472		
GPP Xeon- standard		N/A	N/A	N/A	89,060	11.62
GPP Xeon – IPP library		N/A	N/A	N/A	587	1762.08

Table X. Average execution time in μs for the 32-tap FIR routine with various statistical average stall ratios obtained from multiple runs (VL=128; unrolled three times)

		Average stall ratio (%)				
		0	**25**	**50**	**75**	**100**
One CPU with VP	VL=128; unrolled three times.	78.31	98.25	117.78	137.55	157.42
CTS		78.31	78.54	79.19	86.95	92.07
FTS		65.19	69.84	76.11	83.15	91.61
VLS		72.21	73.86	78.44	85.01	92.81

From these performance results the following conclusions can be made: i) the best performance is provided by FTS followed by VLS and CTS; ii) a higher VL value increases the data-level parallelism, and therefore the performance; iii) loop unrolling increases the utilization of the units and also the overall performance; iv) with a low utilization of the units the speedup doubles from CTS to FTS (see VL=32 without loop unrolling for FIR, FFT, MM and LU); moreover, if the utilization from each thread is less than 50%, the speedup of FTS almost doubles as compared to CTS; v) for kernels with a high utilization of the lane units in the CTS mode, FTS can provide a speedup of 1.2 to 1.5 as compared to CTS. This is caused by the fact that FTS achieves close to 100% utilization (peak performance) and the VP can no longer accommodate more instructions in its pipeline; vi) thread-level parallelism can provide higher speedup than data-level parallelism and loop unrolling (for FFT, FTS with VL=32 and without loop unrolling yields almost the same performance as CTS with VL=64 and the loop unrolled once). Therefore, we can overcome the lack of data-level parallelism and inadequate compiler optimization (loop unrolling) for an application by simultaneously processing an additional thread. LU decomposition exhibits low utilization for low vector lengths. This is caused by the scalar code run by MicroBlaze that involves one floating-point division and two memory accesses per processed row; it can fully overlap VP code runs. As a consequence, two scalar processors in the FTS context provide a speedup of two as compared to the CTS context. This is a good example of applications where the fraction of sequential code is substantial and the utilization of the VP accelerator is low. Thus, adding threads from two or more processors will increase the speedup almost linearly for the same VP resources.

There are cases where VLS can provide better results than FTS. Table VI presents a scenario where each core issues instructions for FIR kernels requesting vector length smaller than the number of VP lanes. Since in VLS four exclusive lanes are assigned to each core, all eight lanes will be used. In FTS, four lanes will be idle in each execution cycle since all eight lanes simultaneously receive the same vector instruction. Therefore, for small vector sizes FTS forces several lanes to be idle, thus yielding performance inferior to VLS. CTS will perform worse than both since only one thread that utilizes half of the lanes is active in each cycle. Contrary to FTS, however, a thread that enters the VP under CTS completes execution without any interruption as long as all dependencies can be resolved internally. As compared to Xeon standard routines, FTS provides a speed-up between 5 (for FIR) and 63 (for FFT) despite the much lower operating frequency of our FPGA-based prototype. On the other hand, highly optimized routines running on Xeon outperformed all VP sharing schemes. However, if the execution times are translated into clock cycles for fairness since FPGA implementations run at much lower clock frequencies, then our best FTS-based scenario for FIR32 consumes just 8.15 clock cycles as compared to 29.54 cycles for Xeon; these numbers are averages for a single FIR32 run obtained after running a large number of consecutive FIR32 routines. In this case, the FTS-based cycle speedup is 3.62. The best FTS-based scenario for a complex FFT32 routine consumes 196 clock cycles while Xeon takes 998 cycles, for a speedup of 5.09. The 1024x1024 matrix-multiplication FTS scenario takes 262.75 cycles as compared to 2080 cycles for the respective optimized MKL matrix function running on Xeon, for a resulting 7.92 speedup. Finally, 321,078 clock cycles are taken by FTS to compute LU decomposition as compared to 1,878,411 cycles on Xeon, for a 5.85 speedup. Therefore, with the performance is expressed in clock cycles, our VP sharing techniques demonstrate 3.62-7.92 speedups compared to optimized Xeon runs.

In many cases a thread may stall at various times. Stalls may occur during the execution of a single or multiple threads running on a single core with a dedicated VP, or during the execution of threads running on multiple cores sharing a VP. Table X shows the average execution time for scenarios where each core runs FIR routines of random size interleaved with stalls of random duration. The stall ratio is defined as the ratio between the average duration of a stall and the average time that the routine utilizes the VP. Without stalls (i.e., the ratio is zero), CTS provides the same performance as a single core attached to a VP with the same total number of lanes (eight in our prototype); FTS gives the best performance. As the stall ratio increases, the performance between CTS and a single core with a VP increases. Also, the performance numbers for CTS and VLS approach that of FTS and become almost identical for a stall ratio of 100%.

Table XI. Power comparison for 32-tap FIR

		Dynamic Power (mW)		**Energy (nJ)**		**nJ/FLOP**
		VP	**VP, Crossbar and Memory**	**Dynamic**	**Total**	
One MB w/o VP		N/A		225.37	380.78	5.951
CTS	VL=32; no loop unrolled	92.02	114.19	42.39	190.89	2.982
	VL=128; no loop unrolled	185.43	225.66	37.36	120.14	1.877
	VL=128; unrolled three times	398.40	479.28	37.53	68.85	1.075
FTS	VL=32; no loop unrolled	182.37	220.98	41.10	115.50	1.804
	VL=128; no loop unrolled	359.56	432.74	37.21	71.61	1.118
	VL=128; unrolled three times	474.41	567.82	37.01	63.09	0.985
VLS	VL=32; no loop unrolled	140.84	187.76	43.96	137.61	2.150
	VL=128; no loop unrolled	238.09	319.13	41.89	94.41	1.475
	VL=128; unrolled three times	429.01	554.97	40.07	68.96	1.077
CTS VL=32; no loop unrolled 4 lanes used; other 4 lanes are power gated.		69.50	93.51	43.76	148.59	2.325

Table XII. Power comparison for 32-point Complex FFT

		Dynamic Power (mW)		**Energy (nJ)**		**nJ/FLOP**
		VP	**VP, Crossbar and Memory**	**Dynamic**	**Total**	
One MB w/o VP		N/A		8562.13	14687.38	22.949
CTS	VL=32; no loop unrolled	195.66	233.59	762.40	2068.01	3.231
	VL=32; unrolled once	279.46	330.07	716.91	1585.71	2.477
	VL=64; unrolled once	337.21	398.79	710.64	1423.44	2.224
FTS	VL=32; no loop unrolled	344.96	405.14	747.07	1484.46	2.319
	VL=32; unrolled once	390.97	456.32	738.32	1385.52	2.164
	VL=64; unrolled once	395.12	460.97	725.11	1352.72	2.113
VLS	VL=32; no loop unrolled	302.43	356.43	781.29	1658.09	2.590
	VL=32; unrolled once	347.54	406.09	750.45	1489.65	2.327
	VL=64; unrolled once	352.23	429.24	730.14	1410.53	2.203
CTS VL=32; no loop unrolled 4 lanes used; other 4 lanes are power gated.		147.87	178.30	781.66	1763.68	2.755

Table XIII. Power comparison for MM

		Dynamic Power (mW)		Energy (nJ)		nJ/FLOP
		VP	VP, Crossbar and Memory	Dynamic	Total	
One MB w/o VP		N/A		7806.88	12817.73	6.258
CTS	VL=32; no loop unrolled	131.68	166.22	1677.16	5713.16	2.789
	VL=32; unrolled once	234.75	296.69	1787.85	4198.25	2.049
	VL=128; unrolled once	433.28	555.02	1671.16	2875.68	1.404
FTS	VL=32; no loop unrolled	263.99	332.86	1682.61	3704.60	1.808
	VL=32; unrolled once	482.95	610.20	1859.89	3079.08	1.503
	VL=128; unrolled once	621.84	793.65	1668.25	2509.05	1.225
VLS	VL=32; no loop unrolled	222.48	311.86	1897.98	4332.38	2.115
	VL=32; unrolled once	386.59	513.59	1947.02	3463.42	1.691
	VL=128; unrolled once	508.44	668.76	1667.89	2665.49	1.301

Table XIV. Power comparison for LU decomposition

		Dynamic Power (mW)		Energy (nJ) per row processed		nJ/FLOP
		VP	VP, Crossbar and Memory	Dynamic	Total	
One MB w/o VP (row length 128)		N/A		2559.51	4473.54	17.473
CTS	VL=16	250.37	317.69	29.09	281.89	8.809
	VL=32	130.33	164.71	54.01	306.81	4.794
	VL=64	68.54	85.46	104.09	356.89	2.788
	VL=128	37.10	46.03	200.78	453.58	1.771
FTS	VL=16	371.26	471.15	27.21	152.01	4.750
	VL=32	198.56	252.94	42.01	168.41	2.631
	VL=64	105.12	132.95	79.92	206.32	1.611
	VL=128	68.59	87.24	148.88	275.28	1.075
VLS	VL=16	311.84	422.28	28.74	156.74	4.898
	VL=32	214.53	290.38	49.79	176.19	8.809
	VL=64	114.05	157.59	91.76	218.16	4.794
	VL=128	64.24	89.82	199.31	388.11	2.788

In FPGAs the static power consumption is dominated by the leakage current in the transistors. Leakage current occurs across gates of transistors (through thin oxides) and sub-threshold leakage from drain to source. The dominant cause of dynamic power consumption is the charging and discharging of capacitance within the device as it manipulates or moves data during computation. The power Tables XI, XII, XIII and XIV show that: i) the lowest dynamic energy is provided by FTS followed by CTS and VLS, with the values having a small dispersion; ii) however, if static power is included, the advantage of FTS and VLS is substantial, especially for low average utilization (see the FIR benchmark for CTS, FTS, and VLS with VL=32 and no unrolling); iii) adding a new core that runs a thread has almost the same performance gain and total energy consumption as doubling the data-level parallelism and unrolling the loop once (see FFT under CTS with VL=64 and loop unrolled once as compared to FFT under FTS with VL=32 and without loop unrolling). Under similar LDST utilization, the MC crossbar and VM dynamic power consumption is higher in VLS than in any other VP sharing context. This is due to the high contention in the crossbar due to the presence of two LDST threads corresponding to two distinct VPs, with no synchronization for accessing the VM. Tables XI-XV also contain energy figures for a MicroBlaze without the VP. We can conclude that the best VP sharing scheme consumes 5 to 16 times less energy per operation than MicroBlaze. We did not include in the power analysis the Xeon general-purpose processor since it has very high power consumption (103 Watts) and is not suitable for high-performance embedded applications.

VP sharing in FTS with an increased number of cores requires either more vector controllers, one per core, or the capability of a controller to handle multiple threads coming from many attached cores. We have carried out simulations for the latter approach where each core in our prototype emulates a dual-threaded microprocessor. This approach suffices for current systems that normally contain less a dozen cores. The

FTS results show high throughput for threads with low VL and no loop unrolling because in this case FTS can accommodate the simultaneous execution of multiple threads, thus increasing the VP throughput. However, if individual threads have high utilization of VP resources, it will be difficult to accommodate simultaneously more threads under FTS. For example, the overall throughput of FIR with VL=128 and no loop unrolling is increased by about 20% with four threads compared to two threads. On the other hand, with loop unrolling FTS cannot easily facilitate additional threads for FIR since the utilization per thread is already 83%. VLS can facilitate better scalability if the lanes are assigned to the threads in a manner similar to the allocation of pages in virtual memory implementations. However, a study must be made of lane fragmentation and interconnection problems. To further improve scalability for increased numbers of cores, the design of suitable networks to interconnect cores to vector controllers is needed. As such a task is outside of the scope for the current paper, this is a future research objective.

Table XV. Advantages and disadvantages of the VP sharing schemes.

	CTS	**VLS**	**FTS**
Advantages	Simple to implement. No per instruction scheduling. Can take advantage of stalls in VP routines to increase the average utilization.	No per instruction scheduling. Increases utilization (due to increased number of elements per lane corresponding to one vector register).	Increases the overall throughput. Increases instantaneous utilization by mixing VP instructions from two or more cores in any lane. Low energy per operation.
Disadvantages	Low throughput since the instantaneous utilization does not increase (still one thread runs at any time). High energy per operation, especially for kernels with low VP utilization.	A single thread uses a lane. Crossbar dynamic power higher due to potential contentions. Complex task to assign lanes, especially if more than two cores share the VP. It can result to lane fragmentation problems for VPs with large numbers of lanes.	Needs arbitration (the complexity increases if more than two cores share the VP). Requires register renaming. May give worse results than VLS when the vector length is less than the number of lanes.

Table XI also shows the power and energy figures when a scalar processor issues VP instructions to four lanes. If the static power for the other four lanes is ignored, the total energy consumption is lower as compared to using all 8 lanes (CTS with VL=32 and without an unrolled loop). Thus, under low utilization the energy consumption due to the static power is substantial; it then becomes imperative to decrease the number of active lanes and power gate the idle ones. Even if the actual FPGA technology does not facilitate power gating, our future work will focus on finding the optimum number of lanes for given LDST and ALU utilizations that minimizes the total energy. Finally, Table XV summarizes the advantages and disadvantages of the VP sharing schemes.

4. PERFORMANCE AND POWER MODELING

Our ultimate objective is to develop a robust runtime framework that can make highly accurate predictions at runtime about performance and energy figures for various VP assignments to applications. Vector lanes could then be assigned effectively to resource-competing threads in ways that could minimize thread execution times, maximize thread throughput, minimize energy consumption for guaranteed performance or independent of performance (e.g., for battery-operated devices), etc. To this extent, we need to develop highly accurate models for performance and power prediction, in the latter case aided by empirical results.

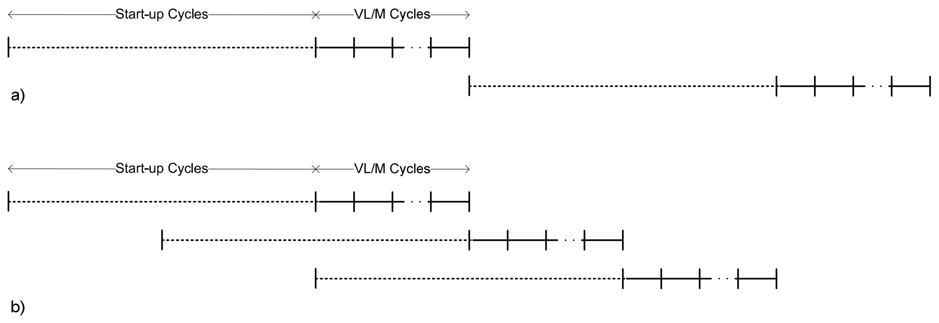

Figure 9. Execution of a) two data dependent instructions; b) three instructions without data dependencies.

Each ALU or LDST instruction finishes in $SU_{ALU/LDST} + VL/M$ clock cycles after it leaves the hazard detection stage in the VC. VL is the vector length, $SU_{ALU/LDST}$ is the start-up latency of ALU/LDST units and M is the number of lanes that receive this instruction. The instruction start-up time directly depends on the pipeline depth of the control stages and the functional unit implementing that instruction. In our implementation, for a LDST instruction with no contention in the crossbar the start-up time is eight clock cycles. For floating-point operations the start-up time is 13 clock cycles for multiply and add, and eight clock cycles for the rest of the instructions. Figure 9 shows how SIMD instructions are executed in each lane in two distinct cases: a) consecutive instructions with data dependence such that in $SU_{ALU/LDST} + VL/M$ clock cycles only VL/M results are produced; b) all instructions issued to lanes have no data dependence such that results are produced in each clock cycle. The average utilization of the ALU or LDST unit can be conveniently defined as the average number of ALU results produced or the average number of data transfers via the memory crossbar, respectively, in $SU_{ALU/LDST} + VL/M$ clock cycles. The number of results is the product of the average number of instructions $IP_{ALU/LDST}$ ready for execution (i.e., the average number of ALU or LDST instructions issued to VP lanes in $SU_{ALU/LDST} + VL/M$ cycles) and VL/M (i.e., the number of elements in each lane to be processed with an SIMD instruction). Equation 1 computes the ALU and LDST utilization.

$$U_{ALU/LDST} = \frac{IP_{ALU/LDST} \dfrac{VL}{M}}{SU_{ALU/LDST} + \dfrac{VL}{M}} = \frac{IP_{ALU/LDST}}{SU_{ALU/LDST} \dfrac{M}{VL} + 1} \qquad (1)$$

Peak performance is achieved when $IP_{ALU/LDST} = SU_{ALU/LDST} \cdot M / VL + 1$ which represents the maximum instruction parallelism needed to fully utilize one of the units. We can increase the utilization of the ALU and LDST units by: i) increasing the vector length VL; ii) reducing the number of lanes assigned to a VC; iii) increasing the average instruction-level parallelism $IP_{ALU/LDST}$; or iv) reducing the start-up time. The first option could be used whenever possible. However, there are applications with low or difficult to identify data parallelism. The second option increases the utilization of the units but degrades the overall performance since each VP instruction takes more time to execute. Instruction-level parallelism can be increased via loop unrolling and multithreading that involves two or more scalar processors. Improving the start-up time may not be an option, especially for FPGAs, since it involves reducing the pipeline depth of the VP, and therefore the design frequency.

The utilization of a unit in a lane can be estimated at runtime as a function of the average instruction throughput $IT_{ALU/LDST}$ and the number of vector elements used per lane (i.e., VL/M), as per Equation 2. This could be implemented easily by embedding appropriate hardware counters (profilers) in the design. Utilization figures were obtained by using Equation 2 for observation periods representing 1000 runs of the same kernel.

$$U_{ALU/LDST} = IT_{ALU/LDST} \cdot VL / M \qquad (2)$$

Our power results show that 90-95% of a lane's dynamic power consumption is due to the ALU unit and the VRF. Also, more than 99% of the total dynamic power consumption is consumed by the VP lanes, MC crossbar and Vector Memory (VM). Figure 10 shows a linear dependence between the ALU dynamic power consumption and the ALU utilization (that actually represents the ALU activity rate). Similarly, we observed that the dynamic power of the VRF is proportional to the number of accesses; since each ALU or LDST unit has two read ports and one write port, the contribution to the number of VRF accesses is evenly divided between these units. Therefore, we modeled the VRF dynamic power consumption as having a linear dependence on the average VRF utilization expressed as $(U_{ALU} + U_{LDST})/2$. The LDST unit requires a complex power model; however, since its impact on the total power budget of the lane is less than 10%, we can freely express it as a linear function of the LDST utilization. Moreover, the MC and VM dynamic power consumptions also show an almost linear dependence on the LDST utilization. The small errors are caused by fine grain effects like different memory access patterns, especially in the VLS context, and different toggling rates in netlist signals due to the randomness of the data used in simulations. Therefore, we can estimate the total dynamic power of the VP by Equation 5, where $P^d_{ALU} \cong K_{ALU} \cdot U_{ALU}$, $P^d_{LDST} \cong K_{LDST} \cdot U_{LDST}$, $P^d_{VRF} \cong K_{VRF} \cdot (U_{ALU} + U_{LDST})/2$, $P^d_{LANE} = P^d_{ALU} + P^d_{LDST} + P^d_{VRF}$ and $P^d_{MC\_VM} \cong K_{MC\_VM} \cdot U_{LDST}$ are the dynamic power figures for the ALU and LDST units, vector register file, lane, and memory crossbar and vector memory, respectively. K_{ALU}, K_{LDST} K_{VRF} and K_{MC_VM} are constant coefficients and are measured in mW per percent of utilization (mW/%).

$$P^d_{TOTAL} \cong M \cdot P^d_{LANE} + P^d_{MC\_VM} = M\left(P^d_{ALU} + P^d_{LDST} + P^d_{VRF}\right) + P^d_{MC\_VM} =$$
$$= M\left[\left(K_{ALU} + K_{VRF}/2\right)U_{ALU} + \left(K_{LDST} + K_{VRF}/2\right)U_{LDST}\right] + K_{MC\_VM} \cdot U_{LDST}$$
(5)

This equation applies if the utilization is the same for all the lanes. Otherwise, we have to compute the power consumption of each lane separately according to its own ALU and LDST utilization figures.

Using a linear approximation method we can find the values for the K coefficients. They are shown in Table XVI along with the mean absolute estimation error for the VP, and collectively for the VP, MC and VM. Therefore, the utilization of the lane units can be used to estimate the dynamic power consumption within a 10% confidence interval.

These performance and power models will enable us to design effective hardware IPs for monitoring the utilization and power consumption of VP units. This information will be collected by the runtime system to estimate power and performance figures for dynamically assigning lanes to cores and individual threads. Appropriate objective functions will be developed.

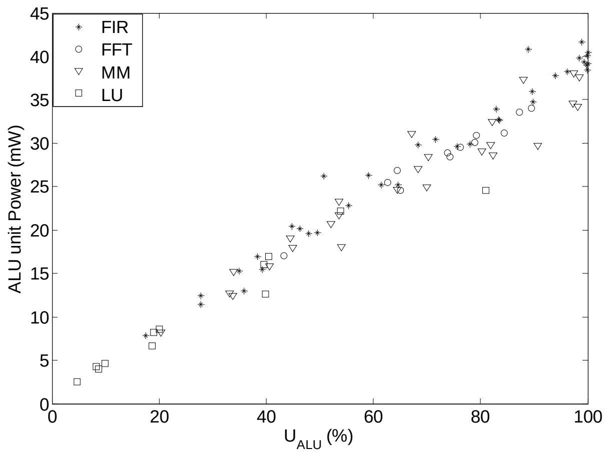

Figure 10. ALU power consumption vs. ALU utilization;

Table XVI. Mean Absolute Error for Dynamic Power Estimation

	Mean Absolute Error (%)	
	VP	**VP, MC and VM**
FIR	5.83	7.71
FFT	7.12	9.93
MM	5.23	7.24
LU	5.01	5.57
OVERALL	5.87	8.14
$K_{ALU} = 0.3723$; $K_{LDST} = 0.0967$; $K_{VRF} = 0.2814$; $K_{MC\_VM} = 1.5197$ (mW/%)		

5. CONCLUSIONS

We presented an architecture to realize three architectural contexts for the implementation of shared vector coprocessors in multicores, in order to efficiently utilize silicon resources. Coarse-grain temporal sharing (CTS) consists of temporally multiplexing sequences of vector instructions ideally arriving from different threads. However, providing a per-core exclusive access to the vector resources does not maximize their utilization. Fine-grain temporal sharing (FTS) consists of spatially multiplexing individual instructions issued by different scalar processors, in order to increase the utilization of the functional units. Finally, vector-lane sharing (VLS) consists of simultaneously allocating distinct vector lanes or collections of them to distinct cores. We evaluated the performance and energy consumption for these coprocessor sharing contexts by implementing several floating-point applications on an FPGA-based prototype. FTS exhibits the biggest speedup and smallest energy consumption, and is followed by VLS. Moreover, under low resource utilization FTS doubles the speed-up and reduces the energy consumption by 50% as compared to the case where a core has exclusive access to the vector coprocessor.

We introduced also a performance model for these coprocessor sharing contexts as well as a power estimation model based on observations deduced from the experimental results. These models suggest several techniques to increase the performance or reduce the energy consumption: i) increase the data-level parallelism by increasing the vector length; ii) increase the instruction-level parallelism at compile time by loop unrolling or other techniques; iii) use multiple threads in a multiprocessor environment to increase the vector coprocessor utilization. Our analysis showed that the last technique can be superior to the former two combined. Therefore, the lack of adequate data-level parallelism in an application can be overcome by sharing the coprocessor resources among many cores.

Future work will focus on providing Quality-of-Service (i.e., the VC and Scheduler will assign coprocessor resources based on the priorities of the active threads). Also, we will refine the performance and power models to help us design a Scheduler capable of modifying dynamically the coprocessor working context or adjusting the number of active lanes in order to minimize a given objective function (involving performance and power/energy). Finally, preemptive coprocessor scheduling will be investigated and this sharing approach will be extended to coprocessors of other type.

ACKNOWLEDGMENTS

The authors would like to acknowledge Mr. Christopher Dahlberg for implementing some of the benchmarks.

REFERENCES

AZEVEDO, A. AND JUURLINK, B. 2009. Scalar Processing Overhead on SIMD-Only Architectures. In *Proceedings of 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 183-190.

BELDIANU, S. F. AND ZIAVRAS S. G. 2011. On-Chip Vector Coprocessor Sharing for Multicores. In *Proceedings of 19th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. IEEE Computer Society proceedings.

CHO, J., CHANG, H., AND SUNG, W. 2006. An FPGA based SIMD processor with a vector memory unit. In *Proceedings of IEEE International Symposium on Circuits and Systems*. IEEE 525-528.

CHOU C.H., SEVERANCE A., BRANT A.D., LIU Z., SANT S., AND LEMIEUX G. 2011. VEGAS: soft vector processor with scratchpad memory. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (FPGA '11). ACM, New York, NY, USA, 15-24.

EGGERS S., EMER J., LEVY H., LO J., STAMM R., AND TULLSEN D.. 1997. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17, 5 (September 1997), 12-19.

FRIGO, M., AND JOHNSON, S. G. 2005. The Design and Implementation of FFTW3. In *Proceedings of the IEEE*. IEEE, 93, 2, 216-231.

GERNETH, F. 2010. FIR Filter Algorithm Implementation using Intel SSE instructions: Optimizing for Intel Atom architecture. *Software White Paper on Intel Embedded Design Center*. (http://download.intel.com/design/intarch/papers/323411.pdf).

GOLUB, G. H. AND VAN LOAN, C. F. 1996. Matrix Computations 3rd Ed. Johns Hopkins, Baltimore, USA.

HAGIESCU A. AND WONG. W.F. 2011. Co-synthesis of FPGA-based application-specific floating point SIMD accelerators. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays* (FPGA '11). ACM, New York, NY, USA, 247-256.

INTEL IPP 2010. Integrated Performance Primitives for Intel Architecture Reference Manual, Intel Corp. 2010. http://software.intel.com/en-us/articles/intel-ipp.

INTEL MKL 2011. Intel Math Kernel Library Reference Manual, Intel Corp, 2011. http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation.

KEATING , M., FLYNN , D., AITKEN, R., GIBSONS, A. AND SHI, K. 2007. *Low Power Methodology Manual for System on Chip Design*. Springer Publications, NewYork, USA.

KOZYRAKIS, C. AND PATTERSON, D. 2002. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Proceedings of 35th Annual IEEE/ACM International Symposium on Microarchitecture*. 283–293.

KOZYRAKIS, C. AND PATTERSON, D. 2003a. Overcoming the limitations of conventional vector processors. *SIGARCH Comput. Archit. News*, 31(2):399–409.

KOZYRAKIS, C. AND PATTERSON, D. 2003b. Scalable, vector processors for embedded systems. *IEEE Micro*. 23, 6, 36–45.

LAFOREST, C.E. AND STEFFAN, J. G. 2010. Efficient Multi-Ported Memories for FPGAs. In *Proceedings of 18th annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, Monterey, CA, 41-50.

LIN, Y., LEE, H., WOH, M., HAREL, Y., MAHLKE, S., MUDGE, T., CHAKRABARTI, C., FLAUTNER, K. 2006. SODA: A low-power architecture for software radio. In *Proceedings33rd Annual International Symposium on Computer Architecture*. IEEE, Boston, MA, 89-101.

SANCHEZ, F., ALVAREZ, M., SALAMI, E., RAMIREZ, A., VALERO, M. 2005. On the Scalability of 1- and 2-Dimensional SIMD Extensions for Multimedia Applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 167-176.

SUNG, W. AND MITRA, S. K. 1987. Implementation of digital filtering algorithms using pipelined vector processors. *Proceedings of the IEEE*. IEEE, 75, 9, 1293- 1303.

WOH, M., SEO, S., MAHLKE, S., MUDGE, T., CHAKRABARTI, C., AND FLAUTNER, K. 2010. AnySP: Anytime Anywhere Anyway Signal Processing. *IEEE Micro*. 30, 1, 81-91.

XILINX INC. 2010a. XPower Estimator User Guide. Xilinx, www.xilinx.com/support/documentation /user_guides

XILINX INC. 2010b. MicroBlaze Processor Reference Guide, 2008. http://www.xilinx.com/support/ documentation/sw_manuals/mb_ref_guide.pdf

YANG, H. AND ZIAVRAS S. 2005. FPGA-Based Vector Processor for Algebraic Equation Solvers. In *Proceedings of IEEE International Systems-On-Chip Conference*. IEEE, Herndon, VA, 115-116.

YIANNACOURAS, P., STEFFAN, J. G. AND ROSE J. 2008. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, Atlanta, GA.

YU, J., EAGLESTON, C., CHOU, C. H.-Y., PERREAULT, M., AND LEMIEUX, G. 2009. Vector Processing as a Soft Processor Accelerator. *ACM Transactions on Reconfigurable Technology and Systems*. ACM, 2, 2, 1-34.