# VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors

Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose
Department of Electrical and Computer Engineering
University of Toronto
10 King's College Road
Toronto, Canada
{yiannac,steffan,jayar}@eecg.utoronto.ca

## ABSTRACT

While *soft processors* are increasingly common in FPGA-based embedded systems, it remains a challenge to scale their performance. We propose extending soft processor instruction sets to include support for vector processing. The resulting system of vectorized software and soft vector processor hardware is (i) **portable** to any FPGA architecture and vector processor configuration, (ii) **scalable** to larger yet higher-performance designs, and (iii) **flexible**, allowing the underlying vector processor to be customized to match the needs of each application. Using our robust and verified parameterized vector processor design and industry-standard EEMBC benchmarks, we evaluate the performance and area trade-offs for different soft vector processor configurations using an FPGA development platform with DDR SDRAM. We find that on average we can scale performance from 1.8x up to 6.3x for a vector processor design that saturates the capacity of our platform's Stratix 1S80 FPGA. We also automatically generate application-specific vector processors with reduced datapath width and instruction set support which combined reduce the area by up to 70% (61% on average) without affecting performance.

## Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures*

## General Terms

Measurement, Performance, Design

## Keywords

Soft processor, FPGA, vector, SIMD, microarchitecture, VESPA, VIRAM, ASIP, SPREE, custom, application specific

## 1. INTRODUCTION

As FPGAs continue to gain traction as complete embedded systems implementation platforms, so do the *soft processors* which are often the heart of those systems. Soft processors are implemented using the programmable fabric of an FPGA, and have become primary parts of the design environments provided by the major FPGA vendors: for example Altera's Nios II and Xilinx's Microblaze. Soft processors are compelling because they provide a very easy way for a software programmer to target the hardware of an FPGA without having to program in hardware description language (HDL). The architecture of a soft processor can also be customized to match the requirements of target applications. However, a challenge for soft processors is to scale-up performance by using increased FPGA resources.

There are several options for how to architect additional resources to improve the performance of a soft processor. While these options are similar to those for a hard processor, an FPGA substrate leads to vastly different trade-offs and conclusions. For example, one option is to consider a wider-issue superscalar or VLIW design—however, such architectures are challenging to scale due to the port limitations of FPGA on-chip block memories [8] (which are typically limited to only two accesses per cycle). A second option is to pursue thread-level parallelism via multiple uniprocessors—but this approach significantly complicates software design and debugging. A third option is to provide custom hardware for key functions through the use of custom instructions or accelerator components, often to exploit the data-level parallelism available in the application (eg., Altera's C2H or Xilinx's AccelDSP)—however, this approach specializes the resulting system, generally making future development and modification more difficult without robust and automated ASIP (Application-Specific Instruction-set Processors) design tools, and does not easily port to a new system. A fourth option that we pursue in this work is to exploit data-level parallelism through a programmable vector processor.

### 1.1 Scalable, Portable, and Flexible Soft Vector Processors

A vector processor supports vector instructions that each describe multiple units of independent work, such that an operation can be performed across many operands in parallel. We propose extending a soft processor instruction set to include vector processing—in particular, by augmenting a scalar soft processor with a vector processing unit, allowing a single thread of control to command large amounts of data parallelism. We call our vector architecture *VESPA* (Vector-Extended Soft Processor Architecture). While the goal of VESPA is to exploit similar data-level parallelism as that targeted by custom accelerators, soft vector processing offers the following advantages.

**Scalability** A vector processor design provides a relatively straightforward means to scale performance: by increasing the number of vector *lanes* (the per-operand datapaths of a vector processor), a vector processor designer can easily increase the amount of data-level parallelism exploited. This also allows designers to easily scale the vector processor design to exploit the increased logic capacity of new generation FPGA devices that continue to grow with Moore's law. VESPA is shown to scale performance on average between 1.8x to 6.3x for 2 to 16 lanes. We believe that improved vector processor designs will achieve more linear speedup compared to the first generation VESPA soft vector processor described in this work.

**Portability** In contrast with the approach of modifying software to exploit a specific custom hardware accelerator, a vectorized program remains portable: the vectorization of the software is mostly independent of the implementation of the vector processor. At one extreme, the implementation could simply be a sequential processor capable only of executing vector element operations one-at-a-time. The vector processor could instead be capable of many parallel vector operations, at a cost of greater area. This allows a system designer to easily adjust the computation power of the vector processor without having to adjust or re-factor the software. In addition, by not requiring any specific FPGA architecture features, a vector processor can migrate to new FPGA families. For example, VESPA was developed for a Stratix-III FPGA while in this paper we evaluate using Stratix-I.[1] Hence the same vectorized code can easily be ported between different vector processor implementations and different underlying FPGAs.

**Flexibility** In contrast with hard IC-designed vector processors, a soft vector processor can be customized to match the needs of target applications. In addition to adjusting architectural parameters, VESPA can also trade some portability by automatically reducing the width of each lane as well as the instruction set supported to achieve average area savings of 61% with 16 lanes.

## 1.2 Related Work

Yu *et. al.* [18] recently demonstrated the potential for vector processing as a simple-to-use and scalable accelerator for soft processors. In particular, through simulation the authors show that (i) a vector processor can potentially accelerate three benchmark kernels better than Altera's `C2H` behavioral synthesis tool (even after manual code restructuring to aid `C2H`), and (ii) how certain vector operations can exploit FPGA architectural features. In this paper we measure the performance benefits of VESPA through a detailed evaluation using a real FPGA-based system with DDR DRAM on the industry standard EEMBC benchmarks, and further demonstrate that tuning the vector processor architecture to match application requirements can dramatically reduce area while maintaining performance.

Other prior work has implemented vector processors on FPGAs, including a vector floating-point processor for matrix multiplication [14] and an integer SIMD processor for multimedia kernels [2]—but neither investigated architectural variation nor customization.

## 1.3 Contributions

This paper makes the following contributions: (i) we present VESPA, a fully-parameterized and robust vector processor core, implemented on an FPGA development platform and connected through first-level caches to an off-chip DDR DIMM—allowing us to account for the behavior of the memory hierarchy and modern DRAM; (ii) we quantitatively measure both the performance benefit and area cost of a range of vector architecture configurations across a hand-vectorized subset of the industry-standard EEMBC benchmarks—fully capturing the effects of pipeline stalls, memory system contention and latency, and communication and parallelism between the scalar and vector cores; (iii) we demonstrate the value of generating application-specific vector processors to dramatically reduce area without affecting performance by reducing the lane width and instruction set support to match the application.

## 2. TRADITIONAL VECTOR PROCESSING

There is a large body of research on vector processing [1, 6]—including decades of commercial development of vector supercomputers by Cray Inc.—that we summarize here. Vector processors provide direct ISA support for operations on whole vectors—i.e., on multiple data elements rather than on a single scalar. These instructions can be used to exploit data-level parallelism in an application to essentially execute multiple loop iterations simultaneously.

Conventional microprocessors increasingly exploit data-level parallelism via SIMD (single-instruction, multiple-data) support [5], including IBM's Altivec, AMD's 3DNow!, MIPS's MDMX, and Intel's MMX/SSE/AVX support. However, such SIMD support is typically limited to a fixed number of elements and exposes this limit to the application. In contrast, true vector processing abstracts from the software the actual number of hardware vector lanes. This abstraction is key for FPGA-based systems, allowing one to exploit flexibility by scaling the number of lanes while maintaining the portability of the application to different hardware implementations.

## 2.1 VIRAM

The VIRAM vector processor [9–11] was developed with the motivation of including parallel processing and DRAM on a single die. While this is hardly conventional, the VIRAM ISA is a relatively straightforward extension of the MIPS ISA—hence we designed VESPA to implement the VIRAM vector ISA with some minor modifications (described in Section 3). A VIRAM processor can operate on any vector *length* (the number of elements in a vector) up to a *maximum vector length* (MVL), with each element containing a maximum number of bits specified by the vector *width*. In the VIRAM processor, each vector element is executed in a 15-stage pipelined datapath referred to as a vector *lane*, and the processor can contain any number of lanes from one

[1]We will soon move to a Stratix-III platform.

to MVL. Both the vector length and width can be configured by software at runtime allowing the hardware to optionally adapt to the application's requirements. For example, reducing the vector width results in a corresponding increase to the MVL, since multiple smaller-width elements can be combined to fit in the maximum width of the register file and datapath—a feature that VESPA does not yet support. Note that changing the processor's MVL should alter the vector length since software should use the MVL to set the vector length.

## 3. VESPA

In this section we describe VESPA: a MIPS-based scalar processor with a highly configurable VIRAM-like vector coprocessor.

### 3.1 Scalar Instruction Set

The ISA used for our scalar processor core is a subset of MIPS-I which excludes floating-point, virtual memory, and exception-related instructions; floating point operations are supported through the use of software libraries. This subset of MIPS is the set of instructions supported by the SPREE system [16, 17] which is used to automatically generate our scalar processor.

### 3.2 Scalar Processor

Our scalar processor is a 3-stage MIPS-I pipeline with full forwarding and a 4Kx1-bit branch history table for branch prediction. The processor was automatically generated by SPREE from a datapath description and can be connected to the vector coprocessor without manual modification. Execution in the scalar processor continues even when the vector coprocessor is stalled, with the exception of (i) communication instructions between the two cores and (ii) scalar load/store instructions, which are serialized to maintain sequential memory consistency.

The SPREE framework was modified in a number of ways to better meet our needs. First, an integer divider unit was added to the SPREE component library along with instruction support for MIPS divide instructions. Second, the original SPREE supported only on-chip memories as a component in its library. We expanded the library to include components with external bus interfaces that will allow the user to attach the processor to a memory system of their choice. Finally, we needed to insert support for the coprocessor interface instructions to the vector coprocessor. These instructions allow the SPREE processor to send data to the vector coprocessor and vice versa. With these changes in place we can automatically generate new scalar processor cores and attach them directly to our memory system and vector coprocessor without modification.

### 3.3 Vector Instruction Set

We selected the VIRAM instruction set [11] to support vector operations for two primary reasons: (i) the VIRAM ISA was designed with embedded/multimedia applications in mind and directly supports predicated execution and fixed-point operations, which are common in many EEMBC benchmarks; and (ii) the ISA was designed as a coprocessor interface to MIPS, matching our existing infrastructure [15]. We currently support all integer and fixed-point vector instructions except for divide and multiply-accumulate, all load/store variants, six flag operations, and five vector manipulation instructions. Some exclusions were necessary to better accommodate an FPGA implementation: for example, the inclusion of the vector multiply-accumulate instructions (which require 3 reads and 1 write) would require further register file replication, banking, or a faster register file clock speed to overcome the 2-port limitations on FPGA block RAMs.

**Table 1: Configurable parameters for the vector coprocessor.**

| Parameter | Symbol | Values |
|---|---|---|
| Vector Lanes | L | 1,2,4,8,16,... |
| Vector Lane Width | W | 1,2,4 (bytes) |
| Maximum Vector Length | MVL | 2,4,8,16,... |
| Memory Crossbar Lanes | M | 1,2,4,8, ...L |
| Each Vector Instruction | - | on/off |

### 3.4 Vector Coprocessor

While the vector instruction set was borrowed from VIRAM, the vector coprocessor architecture was designed from scratch and hand-written in Verilog with built-in parameterization. It communicates with the scalar processor through the MIPS coprocessor interface instructions, while sharing the instruction stream and data cache with the scalar processor core—sharing the data cache avoids many data coherence issues. The presence of a vector data cache, the relatively short pipelining, and the accommodation of few register file ports differentiate the VESPA vector architecture from VIRAM and the recently proposed soft vector processor from Yu *et. al.* [18].

#### 3.4.1 Vector Architecture

The VESPA architecture is shown in Figure 1, which is composed of three pipelines: a 3-stage scalar processor pipeline, a 3-stage pipeline for vector control and vector scalar operations, and a 7-stage vector pipeline[2] which contains the `L` lanes of functional units beginning after the *hazard check* stage. The *replicate* pipeline stage divides the elements of work requested by the vector instruction into smaller groups that are mapped onto the `L` lanes. The *hazard check* stage observes hazards for the vector and flag register files and stalls if necessary. Execution occurs in the next two stages (or three stages for multiply instructions) where tt L operand pairs are read from the register file and sent to the functional units in the `L` lanes. The port limitations in FPGA block RAMs limit VESPA to executing instructions in a SIMD fashion with one operation in the execution stages at a time while traditional vector processors such as VIRAM rely on multi-ported, multi-banked register files and chaining to keep all functional units in the execution stages busy.

#### 3.4.2 VESPA Vector Parameters

Table 1 lists the ranges of configurable parameters for VESPA's vector processor. The number of vector lanes (`L`) determines the number of elements that can be processed in parallel; hence we use this parameter to scale the processing power of VESPA. The width of each vector lane (`W`) can be adjusted to match the maximum element size required by the application: by default all lanes are 32-bits wide, but

[2]Note that the VIRAM processor has 15-stages for functional units alone.

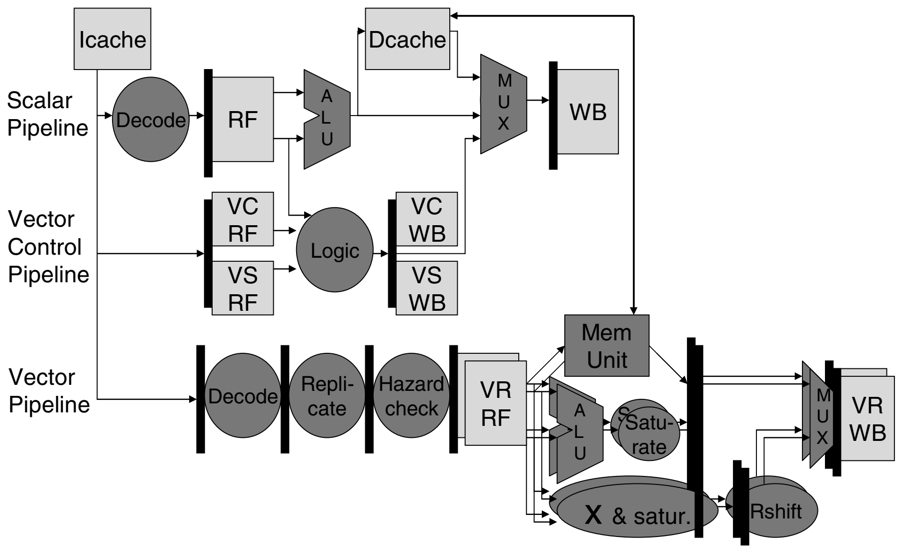


**Figure 1: The VESPA architecture. The black vertical bars indicate pipeline stages, the darker blocks indicate logic, and the light boxes indicate storage elements for the caches as well as the scalar, vector control (vc), vector scalar (vs), and vector (vr) register files.**

for some applications 16-bit or even 8-bit wide elements are sufficient. We can also use `W` to create application-specific vector processors (later in Section 6.1). The maximum vector length (`MVL`) determines the capacity of the vector register files: hence larger `MVL` values allow software to specify greater parallelism in fewer vector instructions, but increases the total area of the vector processor.

The number of memory crossbar lanes (`M`) determines the number of memory requests that can be satisfied concurrently, where $1 < \mathtt{M} < \mathtt{L}$. `M` is independent of `L` for two reasons: (i) a crossbar imposes heavy limitations on the scalability of the design, especially in FPGAs where the multiplexing used to build the crossbar is comparatively more expensive than for conventional IC design; and (ii) the cache-line size limits the number of memory requests one can satisfy from a single cache-line request. Thus a crossbar need not route to all `L` lanes, and the parameter `M` allows the designer to choose a subset of lanes to route to in a single cycle. Note that all integer parameters are generally limited to powers of two to reduce hardware complexity. Finally each vector instruction can be individually disabled thereby eliminating the control logic and datapath support for it.

### 3.4.3 VESPA Portability

To maintain device portability in VESPA, the architected design makes very few device-specific assumptions. First, it assumes the presence of a full-width multiply operation which is supported in virtually all modern day FPGA devices—VESPA does not assume any built-in multiply-accumulate, rounding, saturation, or fracturability support since the presence of these features is more rare and can vary from device to device. In fact, rounding and saturation is required for VIRAM's fixed point support but in VESPA it is performed in the FPGA programmable logic instead to preserve portability. Second, with respect to block RAMs VESPA assumes no specific sizes or aspect ratios, nor any particular behavior for read-during-write operations on same/mixed ports. VESPA only uses one read port and one write port for any RAM hence limiting the need for bi-directional dual-port RAMs. These few assumptions allow the VESPA architecture to port to a broad range of FPGA architectures without re-design. However, although VESPA was not aggressively designed for high clock frequency, any timing decisions made *are* specific to the Stratix III it was designed for, hence some device-specific retiming may be needed to achieve high clock rates on other devices.

## 4. MEASUREMENT METHODOLOGY

In this section we describe the components of our infrastructure necessary to execute, verify, and evaluate VESPA. Specifically, we describe our hardware platform, verification process, CAD tool measurement methodology, benchmarks, and compiler.

**Hardware Platform** We use the multi-FPGA Transmogrifier 4 (TM4) [4] to host the complete vector processor systems. The platform has four Altera Stratix EP1S80F1508C6 devices each connected to two 1GB PC3200 CL3 DDR SDRAM DIMMs clocked at 133 MHz (266 MHz DDR). We use Altera's Quartus II 7.2 CAD software to synthesize our processor systems onto one of the four Stratix I FPGAs and clock the system at 50 MHz. Note our design was intended for a faster Stratix III FPGA where it has a clock speed of 135 MHz, but we used the TM4 board since it was readily available. Each design consists of the MIPS scalar processor generated by SPREE, the VIRAM coprocessor extension, separate 4KB, first-level, direct-mapped instruction and data caches with 16 byte cache lines, and a custom-made DDR controller employing a closed-page policy that connects to one of the 1GB DDR DIMMs.

**Testing** All instances of VESPA are fully tested in hardware using the built-in checksum values encoded into each EEMBC benchmark. Debugging is performed using Modelsim and is guided by comparing traces of all writes to

**Table 2: EEMBC Benchmarks Used.**

| Benchmark | Suite | EEMBC Dataset | Largest Data Type Size | Percent of Supported VIRAM ISA used |
|---|---|---|---|---|
| AUTCOR | Telecom | 2 | 4 bytes | 9.6% |
| CONVEN | Telecom | 1 | 1 byte | 5.9% |
| FBITAL | Telecom | 2 | 2 bytes | 14.1% |
| VITERB | Telecom | 2 | 2 bytes | 13.3% |
| RGBCMYK | Digital Ent. | 5 | 1 byte | 5.9% |
| RGBYIQ | Digital Ent. | 6 | 2 bytes | 8.1% |

the scalar and vector register files. This trace is extracted from RTL simulation using Modelsim and compared against an analogous trace obtained from instruction-set simulation using the MINT [13] MIPS simulator which we have augmented with the VIRAM extensions.

**FPGA CAD Tools** A key value of performing FPGA-based processor research directly on an FPGA is that we can attain high quality measurements of the area consumed and the clock frequency achieved—these are provided by the FPGA CAD tools. We use aggressive timing constraints to maximize the CAD tool's effort for default optimization settings but with register retiming and register duplication set to *on*. Through experimentation we found that these settings provided the best area, delay, and runtime trade-off. We also performed 8 such runs for every vector configuration to average out the non-determinism in modern CAD algorithms. The relative silicon area of each FPGA resource relative to a single logic element (LE) was supplied to us by Altera [3], and we used these equivalent areas to calculate the total silicon area consumed on the Stratix 1S80 measured in units of *equivalent LEs*—the silicon area of a single LE including its routing.

**Benchmarks** As listed in Table 2, for this study we use six benchmarks from the Telecom and Digital Entertainment suites of EEMBC, the industry-standard benchmark collection. We execute the largest dataset for each benchmark with the test harness and benchmarks uncompromised, potentially allowing us to calculate and report official EEMBC scores. However, in this paper we report only number of cycles in lieu of wall-clock-time, as clock frequency is essentially constant at 135 MHz (on Stratix III) across our designs as seen in Section 5.1. Note that cycle counts are collected from a complete execution on our hardware platform as described above.

**Compilation Framework** Benchmarks are built using a MIPS port of GNU `gcc` 4.2.0 with `-O3` optimization level. Initial experiments with this version of `gcc`'s auto-vectorization capability showed that it is in its infancy, preventing us from automatically generating vectorized code from key EEMBC program loops. Instead we ported the GNU assembler to support VIRAM vector instructions. Hand-vectorized assembly EEMBC routines were provided to us by Kozyrakis who used them during his work on the VIRAM processor [11].

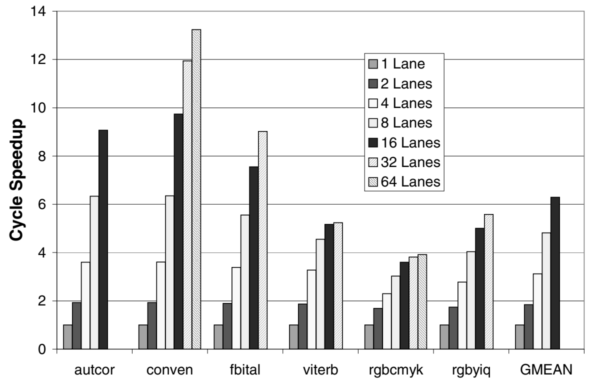


**Figure 2: Cycle speedup relative to one lane for an increasing number of vector lanes. Only 16 full vector lanes fit on the Stratix I FPGA, but using the area reduction customization in Section 6.1 we can generate customized VESPA processors with up to 64 lanes for some benchmarks**

## 5. EVALUATING SCALABILITY AND FLEXIBILITY

In this section we evaluate the scalability and trade-offs within VESPA by varying the following parameters: the number of lanes, the MVL and the memory crossbar size.

### 5.1 Scaling the Number of Vector Lanes (L)

Increasing the number of vector lanes in a vector processor is a simple and scalable way to invest area to improve performance: with more vector lanes, more element operations can be performed in parallel. In this section we vary the number of vector lanes in VESPA and measure the area of the implementation as well as the performance across our vectorized benchmark set. For this experiment we use a 32KB data cache size—the effect of cache size is examined more closely in Section 7.

Figure 2 shows the speedup attained by increasing the number of vector lanes. Speedup is measured relative to a vector processor with a single lane executing the identical benchmark binary—we do not compare against the nonvectorized benchmark here since the vectorized code has the advantage that it is hand-written in assembly. The figure shows speedups ranging inclusively across all benchmarks from 1.7x to 13.2x. On average the benchmarks experience a 1.8x speedup for 2 lanes, with a steady increase to 6.3x for 16 lanes—at which point the FPGA capacity saturates. Using the area reduction customization described later in Section 6.1 we can fit up to 64 lanes for benchmarks which do

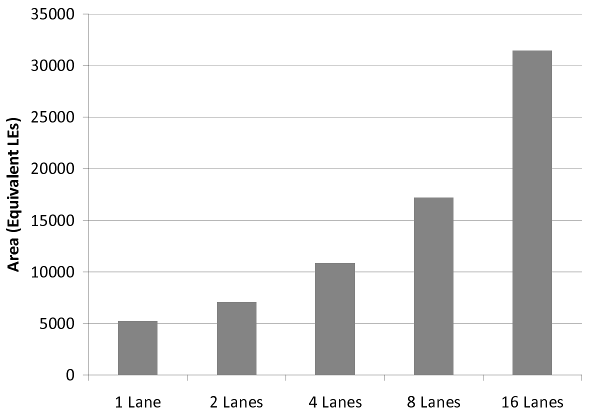


**Figure 3: Area of the vector coprocessor for an increasing number of vector lanes on the Stratix 1S80.**

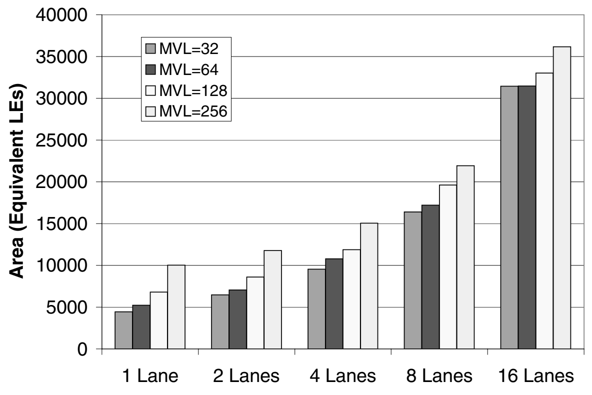


**Figure 4: Silicon area of the vector coprocessor across different MVL and lane configurations.**

not require the processing of full vector lanes. The continued scaling of these benchmarks is also shown in the figure but their performance scaling is significantly limited by the 16-byte cache line size which necessitates several cache line accesses to satisfy the memory requests across so many lanes.

Ideally, speedup would increase linearly with the number of vector lanes, but this is prevented by a number of factors: (i) only the vectorizable portion of the code can benefit from extra lanes, hence benchmarks such as CONVEN that have a blend of scalar and vector instructions are limited by the fraction of actual vector instructions in the instruction stream; (ii) some applications do not contain the long vectors necessary to scale performance, for example VITERB executes predominantly with a vector length of only 16; (iii) the movement of data is a limiting factor as each load/store instruction contains start and finish delays and each cache line request requires at least 2 cycles—limiting streaming-type benchmarks such as AUTCOR, RGBCMYK, and RGBYIQ.[3]

Figure 3 shows the total area cost of the vector coprocessor in VESPA as we increase the number of vector lanes. The growth seen in the figure indicates that the vector lane logic quickly dominates the area. At 16 lanes the vector coprocessor consumes 31500 LEs worth of silicon area, which is less than a quarter of the 130160 LEs worth of silicon on the Stratix 1S80 FPGA. However, we are unable to double the number of lanes further since each vector lane requires a 32-bit multiplier while the 1S80 supports only 22 32-bit multiply operations. Newer generations of FPGAs have both significantly more logic resources and also an increased proportion of supplied multipliers [7]. Moreover, the configurability of FPGAs can be leveraged to customize and significantly trim the area needed by the vector processor as we will show in Section 6.1.

All results in this paper are with respect to the Stratix 1S80 FPGA on our hardware platform, but VESPA was originally designed to target a Stratix III 3SL200C2. On such a device we can scale the number of vector lanes up to 64 as shown in Table 3 (synthesis results only). The table shows that the clock frequency remains steady until 64 lanes when the broadcast of the base address to each lane for address generation becomes critical. The amount of logic used grows at 570 ALMs per lane. There is one 32-bit multiplier (built out of four 18-bit multipliers) for the scalar processor and one for each lane resulting in a growth of $4*(1+\mathtt{L})$ until 16 lanes where some multipliers are used for address generation as well. Finally the block RAM usage stays constant until 32 lanes where the more wide and shallow register file begins wasting the M9Ks storage capacity.

The linear area cost may become expensive in the stringent embedded design space, however two factors help to counter this cost. First, along with the faster speeds and lower power consumption of each new FPGA generation comes an enormous boost of logic resources. Soft vector processors allow a designer to write code once and automatically extract more parallelism at the rate of Moore's law by increasing the vector lanes in the vector processor implementation without any redesign. Second, a soft vector processor can be customized to reduce the area cost, examples of this are shown in Sections 6.1 and 6.2.

## 5.2 Altering Maximum Vector Length (MVL)

Increasing the MVL allows a single vector instruction to encapsulate more element operations, but also increases the vector register file size and hence the total number of FPGA block RAMs required. This growth is potentially exacerbated by the fact that the entire vector register file is replicated to achieve the three ports (2-read and 1-write) necessary, since current FPGAs have only dual-ported block RAMs. The performance impact of varying the MVL results in an interesting tradeoff: higher MVL values result in fewer loop iterations, in turn saving on loop overheads—but this savings comes with more time-consuming vector reduction operations. For example, as `MVL` grows, the $Log(\mathtt{MVL})$ loop iterations required to perform a tree reduction that adds all the elements in a vector grows with it. We examine the resulting impact on both area and performance below.

The area impact of increasing MVL is isolated entirely within the vector register file and hence affects solely the FPGA block RAM usage. But because of the discrete sizes and aspect ratios of those block RAMs, the area savings from decreasing MVL will plateau. For example, the Stratix M4K block RAM has 4096 bits and can output a maximum of 32-bits in a single cycle, hence a 16-lane vector processor requires 16 of these M4Ks to output the 32-bit elements

[3]RGBCMYK and RGBYIQ are more adversely affected because they employ strided memory access instructions to extract and write every 3rd or 4th byte of the stream.

**Table 3: VESPA design characteristics on Stratix III 3SL200C2.**

| Design Characteristic | 1 lane | 2 lanes | 4 lanes | 8 lanes | 16 lanes | 32 lanes | 64 lanes |
|---|---|---|---|---|---|---|---|
| Maximum Clock Frequency (MHz) | 135 | 137 | 140 | 137 | 137 | 136 | 122 |
| Logic Used (ALMs) | 2763 | 3606 | 4741 | 7031 | 11495 | 20758 | 38983 |
| Multipliers Used (18-bit DSPs) | 8 | 12 | 20 | 36 | 76 | 172 | 388 |
| Block RAMs Used (M9Ks) | 45 | 46 | 46 | 46 | 46 | 78 | 144 |

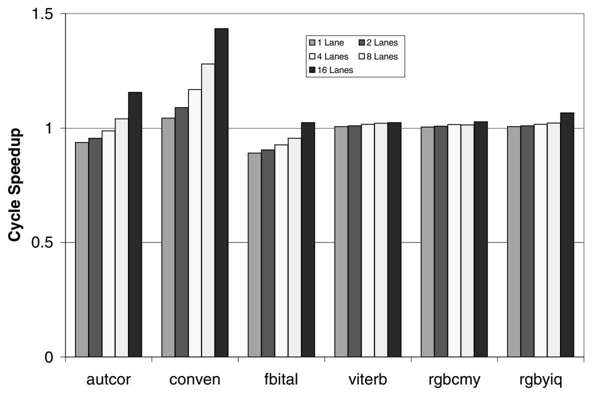


**Figure 5: Cycle speedup measured when MVL is increased from 32 to 256.**

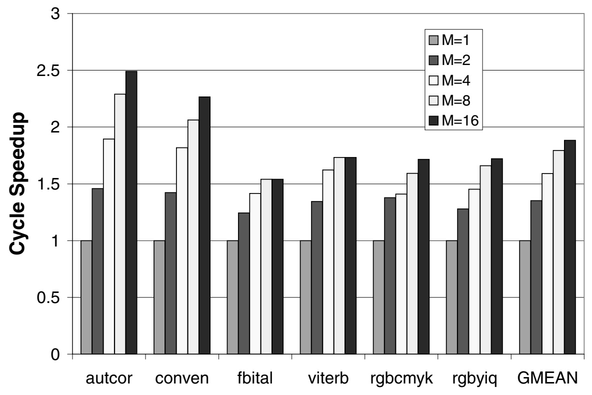


**Figure 6: Speedup achieved on a 16-lane vector processor for different values of M, the memory crossbar size, compared to the same vector processor with $M = 1$.**

for each lane. This results in 64Kbits being available which exactly matches the demand of the MVL=64 case, but when MVL=32 the block RAMs are only half-utilized resulting only in wasted area instead of area savings.

Figure 4 shows the area of the vector processor for different values of MVL and for a varying number of lanes. The graph shows that increasing the MVL causes significant growth when the number of lanes are few, but as the lanes grow and the functional units dominate the vector coprocessor area, the growth in the register file becomes less significant as seen in the 16 lane vector processor. Of particular interest is the identical area between the 16 lane processors with MVL equal to 32 and 64. This is an artifact of the discrete sizes and aspect ratios of FPGA block RAMs. At 16 lanes the vector processor demands such a wide register file read that multiple FPGA block RAMs are required to match that width. As a result, the storage space for vector elements is distributed among these block RAMs causing, in this specific case, only half of each block RAM's capacity to be used. We avoid this under-utilization by setting MVL to 64 by default for this research, which allows lanes between 1 and 16 with no under-utilization. Note that in our area measurements we count the entire silicon space of the block RAM used regardless of the number of memory bits actually used.

Figure 5 shows the performance of a set of vector processors with varying numbers of lanes and MVL=256, each normalized to an identically-configured vector processor with MVL=32. The benchmarks AUTCOR and FBITAL both have reduction operations and hence show a decrease in performance caused by the extra cycles required to perform vector reductions. The performance degradation is more pronounced for low numbers of lanes: as the number of lanes increase the reduction operations themselves execute more quickly, until finally the amortization of looping overheads dominates and results in an overall benchmark speedup for 16 lanes. The remaining benchmarks do not contain significant reduction operations and hence experience faster execution times for the longer vectors when MVL=256. For CONVEN, which has a very tight vectorized loop, the speedup is quite significant: 43% for 16 lanes. The remaining benchmarks have larger loop bodies which already amortize the loop overhead and hence have only very minor speedups.

## 5.3 Sizing the Memory Crossbar (M)

The parameter M controls the number of lanes that the memory crossbar connects to and hence directly controls the crossbar size and the amount of parallelism possible for memory operations. For example, a 16-lane vector processor with M = 4 can complete 16 add operations in parallel, but can only satisfy up to 4 vector load/store operations provided all 4 are accessing the same cache line. Decreasing M reduces area but also decreases the performance of vector memory instructions. Note that the useful range of M is bounded by the number of lanes and the number of bytes in a cache line divided by the access size. Since our data cache line is 16 bytes wide, a 16-lane vector processor can only make use of an M = 16 crossbar on unit-stride byte accesses, while unit-stride word accesses can only make use of an M = 4 crossbar.

Figure 6 shows the impact on performance of varying the memory crossbar size for a 16-lane vector processor. The results are normalized to the case where M = 1—i.e., with memory operations executed serially. The performance benefit of the larger crossbar is clearly a significant component of overall system performance resulting in up to 2.5x speedup for the M = 16 crossbar, which matches the number of vector lanes in the system. On average, with only a single memory lane, M = 1, the performance of the 16-lane VESPA processor is less than a 4-lane VESPA processor with M = 4, stressing the importance of configuring VESPA with enough memory bandwidth. In some cases the performance benefit is much less significant: for FBITAL and VITERB performance saturates at M = 8 suggesting that the memory crossbar size can be halved from M = 16 for free, resulting in a 2200 LE savings from the 36000 LE design.

The memory crossbar has the potential to severely limit the processor's clock frequency. We have designed our pipeline and clock frequency to accommodate an M = 16 memory crossbar and hence have prevented any clock frequency impact for our range of study (note that even in the 32-lane and 64-lane VESPA processors we limit the crossbar to M = 16 to match the cache line size as discussed previously). Any increases to the cache line size would greatly increase the memory crossbar size and reduce the maximum value of M we can achieve without impacting clock frequency.

## 6. CUSTOMIZING VESPA

In this section we evaluate the customization opportunities in vector width and instruction set support to reduce area without affecting performance. We also combine the two approaches to achieve even greater area savings.

### 6.1 Impact of Vector Width Customization

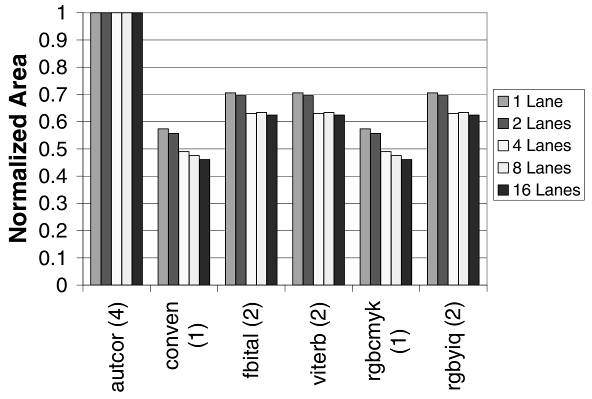
We can leverage the reconfigurability of the FPGA to customize the vector coprocessor to the application. Table 2 shows that with the exception of AUTCOR, none of the benchmarks actually require 32-bit datapaths, and can instead use only 8 or 16-bit vector datapaths. By changing a single parameter entry, our vector coprocessor can automatically implement different vector lane datapath widths. We modify the width to match the largest data type size specified in Table 2 for each benchmark, thereby creating width-optimized application-specific vector processors with reduced area. The area of a conventional processor cannot be reduced once it is fabricated so designers opt instead to dynamically (with some area overhead) reclaim any unused vector lane datapath width to emulate an increased number of lanes, hence gaining performance. Our current benchmarks each operate on a single data width, making support for dynamically configuring vector lane width and number of lanes uninteresting—however for a different set of applications this could be motivated.

Figure 7(a) shows the area for each of the width-optimized application-specific vector processors normalized to the original full vector processor. The AUTCOR benchmark utilizes 4-byte (32-bit) data types resulting in no area savings for the width-optimized vector coprocessor over the full vector coprocessor. The benchmarks with 2-byte (16-bit) data types experience up to a 37.5% area reduction when 16 lanes are implemented, while those with 1-byte (8-bit) data types experience up to a 54% area reduction. The figure also shows that the width-customization technique has a greater impact as the lanes increase.

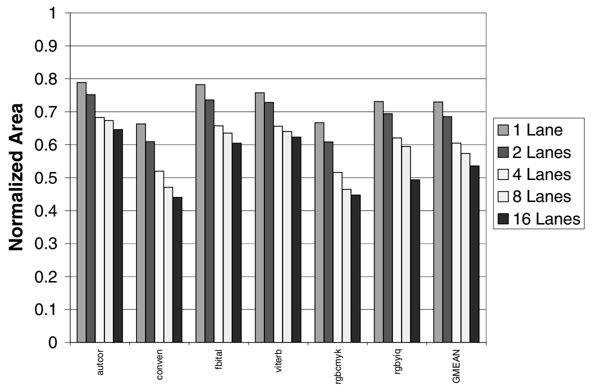
### 6.2 Impact of Instruction Set Subsetting

With VESPA we can also automatically eliminate hardware support for any unused instructions. We parse each benchmark binary and eliminate control logic and datapath hardware for instructions that are not found. The area savings achieved depends not only on the fraction of the VIRAM ISA used, as specified in Table 1, but also on *which* vector instructions are unused—since some functional units are larger than others.

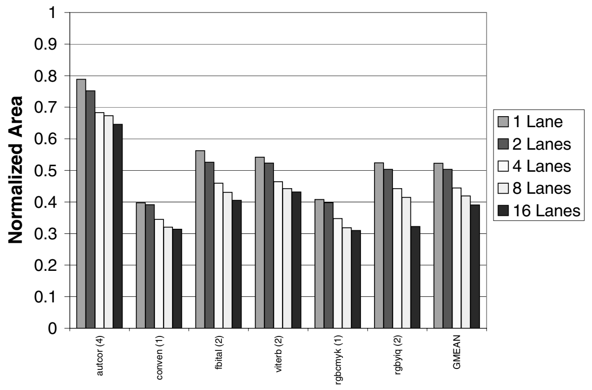
Figure 7(b) shows the area of the resulting subsetted vector coprocessors. Up to 55% of the area is reduced in some cases, but on average a 16-lane vector processor saves 46.5% of its area, and the benefit increases with the number



(a) Area of vector coprocessor after reducing lane width to match the application. The number in parenthesis indicates the number of bytes of the largest data type in the application, and hence indicates the vector lane width implemented.



(b) Area of the vector coprocessor after eliminating hardware support for unused instructions.



(c) Area of the vector coprocessor after eliminating both unused lane-width and hardware support for unused instructions.

**Figure 7: Effect of customizing the VESPA lane width and instruction set support to each benchmark.**

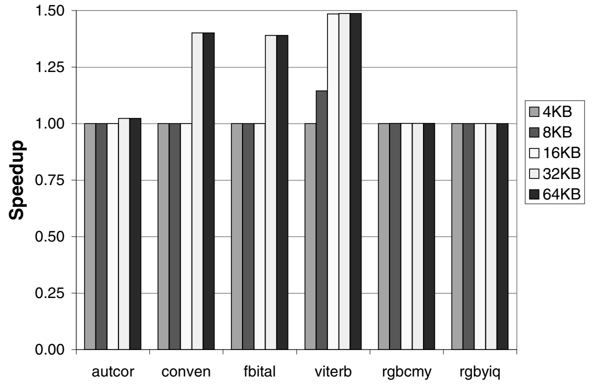


**Figure 8: Cycle speedup achieved for a 16-lane vector processor with varying data cache sizes, normalized to a 4KB data cache.**

of lanes. The cycle behavior of the processor is unaltered with this technique and allows the instruction set to grow without necessarily increasing the vector processor area. For example, the area overhead of supporting fixed-point operations in VESPA can easily be removed by a designer who is not using them.

### 6.3 Impact of Combining Width Reduction and Instruction Set Subsetting

We can additionally customize both the vector width and the supported instruction set of VESPA for a given application, thereby creating highly area-reduced VESPA processors with identical performance to a full VESPA processor. Since these customizations overlap, we expect that the savings will not be perfectly additive: for example, the savings from reducing the width of a hardware adder from 32-bits to 8-bits will disappear if that adder is eliminated by instruction set subsetting. Figure 7(c) shows the area savings of combining both the width reduction and the instruction set subsetting. Such an application-specific vector processor can drastically reduce the area-cost for a 16-lane processor by up to 70%, and 61% on average.

## 7. IMPACT OF DATA CACHE

Previous work has indicated that soft processor systems were largely compute-bound and hence that memory system performance was not a significant limiter of overall system performance [12]. In this work we show that after accelerating the computation through vectorization, that memory system performance indeed becomes significant. Using a 16 lane vector processor and our parameterized direct-mapped cache we measure the area and performance of the system for cache sizes ranging from 4KB to 64KB. This 16-fold increase in cache size can be achieved with only 2000 equivalent LEs of silicon (recall that this measurement also includes all memory bits), making the overall area cost relatively minor with respect to the complete system. This result is somewhat expected since the relative cost of logic is much higher than for memory on an FPGA with dedicated blocks of RAM.

Figure 8 shows the speedup for caches up to 64KB in size. The performance of the streaming benchmarks RGBCMY and RGBYIQ are unaffected due to the lack of temporal locality in those applications. These applications are good candidates for a simple prefetching policy, a topic we will investigate in the future. AUTCOR is also unaffected because of its small data set size which fits in the 4KB cache. However large performance gains between 39% and 49% are evident for the remaining benchmarks. The increase in computational power has allowed data movement to become a larger component of overall system performance, whereas previous work limited performance growth of a perfect cache to only 12% faster than a 4KB direct-mapped cache [12]. In the future, we plan to architect and explore better memory systems for soft vector processors.

**Table 4: Cycle speedup of VESPA (assembly-coded and vectorized benchmarks) versus Scalar MIPS (C-coded benchmarks).**

| | L=1 | L=2 | L=4 | L=8 | L=16 |
|---|---|---|---|---|---|
| autcor | 1.3 | 2.6 | 4.7 | 8.1 | 11.4 |
| conven | 6.9 | 12.5 | 21.4 | 32.9 | 43.6 |
| fbital | 1.2 | 2.2 | 3.7 | 5.5 | 6.9 |
| viterb | 1.0 | 1.8 | 2.8 | 3.6 | 3.9 |
| rgbcmyk | 1.0 | 1.8 | 2.4 | 3.2 | 3.8 |
| rgbyiq | 2.4 | 4.2 | 6.7 | 9.6 | 12.0 |
| GMEAN | 1.8 | 3.1 | 5.1 | 7.4 | 9.2 |

## 8. IMPACT OF ASSEMBLY PROGRAMMING

As mentioned previously, the EEMBC benchmarks, originally written in C, were vectorized using assembly language due to the current lack of availability of a high quality vectorizing compiler for VIRAM. The resulting vectorized benchmarks used in this study have two advantages over the original benchmarks: i) they are vectorized, and ii) they are written in assembly and hence may benefit from manual optimization and instruction scheduling. So far we have focussed only on the former advantage by normalizing against the vectorized benchmarks—in this section we examine and quantify the latter advantage.

Table 4 shows the speedup of the vectorized benchmarks run on VESPA processors with varying lanes against the original scalar C-written benchmarks executed on the SPREE-generated MIPS scalar processor. The VESPA processor with only one vector lane has minimal benefit from vectorization hence exposing the advantage of assembly programming. The VITERB and RGBCMYK benchmarks perform the same on the 1-lane VESPA as originally on the scalar MIPS, suggesting that any assembly-level manual optimization was nullified by the extra vector control instructions inserted into the instruction stream. In the CONVEN benchmark, a significant amount of code responsible for modelling a shift register was eliminated as a result of manual optimization. Hence even with the same number of functional units as in the scalar MIPS processor core, the vectorized assembly version exhibits a 6.9x speedup on VESPA with one lane.

## 9. CONCLUSIONS

Soft vector processors provide a scalable, flexible, and portable framework for trading area for performance for FPGA applications that contain data-level parallelism. We have shown that with VESPA, our parameterized FPGA-based vector processor, we can easily scale the number of vector lanes to exploit the available resources of an FPGA: the Stratix 1S80 that we target accommodates 16 full vector lanes, and on our EEMBC benchmarks achieves an average speedup of 6.3x (versus a single lane vector processor).

A soft vector processor design can be quite portable, as evidenced by VESPA which was designed for Stratix III yet implemented and evaluated on Stratix I FPGAs. Soft vector processors can be implemented on any FPGA device family while maintaining software compatibility, largely due to the abstraction of the VIRAM instruction set. Such portability could allow FPGA vendors to provide vectorized software IP libraries, allowing systems developers to tune the area and performance of their designs without modifying their software. Such a design flow could significantly enhance existing FPGA design flows, which currently struggle with respect to IP-core reuse.

The flexibility of soft vector processors is perhaps their most appealing quality: for VESPA, designers can tune the vector lane length and width, maximum vector length, and the size of the memory crossbar. We demonstrated that vector lane width can be tuned to match application requirements, reducing the area of the vector coprocessor by as much as 54%. Similarly, we showed that eliminating support for unused instructions can reduce the area of the vector coprocessor by up to 55%. While these approaches both trade portability for area savings, when combined for a 16 lane vector coprocessor they can result in an area savings of up to 70% (61% on average).

We are currently migrating VESPA to a hardware platform with a Stratix III FPGA and DDR2-DRAM. We also intend to develop customizable memory systems which can be adapted to the memory access patterns of applications, and to explore the potential for custom vector instructions.

## 10. ACKNOWLEDGEMENTS

We would like to sincerely thank Christos Kozyrakis for his help with the benchmarking infrastructure used in this research. We would also like to acknowledge the funding support we received from both Altera Corporation and NSERC. Finally we would like to thank the anonymous reviewers for their feedback and insightful comments.

## 11. REFERENCES

[1] K. Asanovic, J. Beck, B. Irissou, B. Kingsbury, and N. Morgan. The TO Vector Microprocessor. *Hot Chips*, 7:187–196, 1995.

[2] J. Cho, H. Chang, and W. Sung. An fpga based simd processor with a vector memory unit. *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4 pp.–, 21-24 May 2006.

[3] R. Cliff. Altera Corporation. Private Comm, 2005.

[4] J. Fender, J. Rose, and D. R. Galloway. The transmogrifier-4: An fpga-based hardware development system with multi-gigabyte memory capacity and high host and memory bandwidth. In *IEEE International Conference on Field Programmable Technology*, pages 301–302, 2005.

[5] J. Gebis and D. Patterson. Embracing and Extending 20th-Century Instruction Set Architectures. *Computer*, 40(4):68–75, 2007.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture; A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[7] P. Jamieson and J. Rose. Enhancing the area-efficiency of fpgas with hard circuits using shadow clusters. *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 1–8, Dec. 2006.

[8] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An fpga-based vliw processor with custom hardware execution. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 107–117, New York, NY, USA, 2005. ACM.

[9] C. Kozyrakis and D. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 283–293, 2002.

[10] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. *SIGARCH Comput. Archit. News*, 31(2):399–409, 2003.

[11] C. Kozyrakis and D. Patterson. Scalable, vector processors for embedded systems. *Micro, IEEE*, 23(6):36–45, 2003.

[12] M. Labrecque, P. Yiannacouras, and J. G. Steffan. Scaling Soft Memory Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'08).*, Palo Alto, CA, April 2008.

[13] J. E. Veenstra and R. J. Fowler. MINT: a front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '94).*, pages 201–207, Durham, NC, January 1994.

[14] H. Yang, S. Wang, S. G. Ziavras, and J. Hu. Vector processing support for fpga-oriented high performance applications. In *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 447–448, Washington, DC, USA, 2007. IEEE Computer Society.

[15] P. Yiannacouras. SPREE. http://www.eecg.utoronto.ca/∼yiannac/SPREE/.

[16] P. Yiannacouras, J. Rose, and J. G. Steffan. The Microarchitecture of FPGA Based Soft Processors. In *CASES'05: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 202–212. ACM Press, 2005.

[17] P. Yiannacouras, J. G. Steffan, and J. Rose. Application-specific customization of soft processor microarchitecture. In *FPGA'06: Proceedings of the International Symposium on Field Programmable Gate Arrays*, pages 201–210, New York, NY, USA, 2006. ACM Press.

[18] J. Yu, G. Lemieux, and C. Eagleston. Vector processing as a soft-core cpu accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 222–232, New York, NY, USA, 2008. ACM.