IEEE Computer Society Annual Symposium on VLSI, July 2016

Power-Performance Optimization of a Virtualized SMT Vector Processor via Thread Fusion and Lane Configuration

YAOJIE LU, SEYEDAMIN ROOHOLAMIN and SOTIRIOS G. ZIAVRAS
New Jersey Institute of Technology, Dept. of Electrical and Computer Engineering

***Abstract*—Lane-based Vector Processors (VPs) are highly scalable. However, they become less energy efficient as they scale for vector applications having insufficient data-level parallelism (DLP) to keep the extra computation lanes fully occupied. We present a scalable and yet flexible VP that is capable of dynamically deactivating some of its computing lanes in order to reduce static power with minimum performance loss. In addition, our simultaneous multi-threaded (SMT) VP can exploit identical instruction flows that may be present in different vector applications by running in a novel fused mode that increases its utilization. We introduce a power model and two optimization policies for minimizing the consumed energy, or the product of the energy and runtime for a given application. Benchmarking that involves an FPGA prototype shows up to 33.8% energy reduction in addition to 40% runtime improvement, or up to 62.7% reduction in the product of energy and runtime.**

Keywords—vector processor; dynamic configuration; instruction fusion; DLP; performance and energy optimization.

I. INTRODUCTION

VP (co)processors exploit DLP due to their SIMD specialized architecture [1]. The modular design of lane-based VPs empowers scalability [2]. As a VP scales up, however, higher DLP is required to keep its lanes fully utilized. Vector applications optimized for a given VP size would yield poor utilization on its scaled up version with an increased number of vector lanes, so VP sharing among many on-chip cores is recommended [3]. Without proper resource management, scaling will adversely lead to many idling cycles in each lane for an otherwise efficient vector application, and thus unnecessarily drain static power [4].

We propose a flexible lane-based VP design that can dynamically deactivate wisely some of its lanes toward reducing the static power consumption with minimum performance loss. We also propose a novel thread fusion technique that is used by our SMT VP for multiplying its utilization when similar threads coming from different applications are identified in the pending vector task queue. We also derive a highly accurate power dissipation model for the VP that is used toward runtime optimization of the energy and/or performance. The complexities of managing the VP's fusion process and dynamic lane configuration are hidden from application programmers via complete VP virtualization; the VP management kernel sets VP state registers for controlling configurable hardware components, and handling vector instruction synchronization, vector memory (VM) access, and vector register file (VRF) usage. Each vector application is executed as a thread with its own virtual VM address and VRF name space, and does not need to be recompiled to run under a different VP configuration or fusion state.

In [4], a dynamic power-gating technique is proposed to control the VP's width (i.e., number of active lanes) in order to achieve optimized performance and/or energy. Compared to [4], our lane deactivation process is capable of power-gating the entire lane including its dedicated VM bank due to our distributed memory architecture, while the memory crossbar connecting the lanes to the VM and all memory modules always have to stay active in [4]. Thread fusion [5] fuses parallel threads that run on the same SMT processor/core. Instruction fusion [6] fuses identical instruction flows within unrolled loops. While both fusion techniques are applied to general purpose RISC processors mainly for energy reduction, our fusion technique is applied to a VP for boosting VP utilization while saving the host processors' energy.

The rest of this paper is organized as follows. In Section II, we present VM address virtualization to support dynamic lane configuration. The proposed fusion technique, which originates from an SMT VP technology, is covered in Section III. The FPGA-based VP prototype architecture is described in Section IV. Section V discusses benchmarking. A VP power model is introduced in Section VI along with two optimization policies. Conclusions are drawn in Section VII.

II. VIRTUALIZED VECTOR MEMORY (VM) ADDRESS SPACE

A. *Virtualization of the VM's Address Domains*

Each vector lane in our VP contains an ALU (FPU) unit as well as a LDST (load/store) unit that interfaces the VM. Our VP features a distributed VM design, where one of each VM bank's dual ports is assigned exclusively to one VP lane, and yet all VM banks can be accessed by the host processors via a mux connected to the second port of every VM bank. Since the VM is accessed by two heterogeneous types of masters (i.e., the on-chip host cores and the VP), it is assigned two different address domains with regard to each one of its masters. The host-to-VM mux accesses VM banks in low-order interleaved fashion to hide the bank selection details from the hosts; therefore, all VM banks appear as one large memory module

with a continuous address space on the system bus. Each VP lane, on the other hand, can only access and process elements within its dedicated VM bank based on the VP-to-VM issued address and vector length (VL: number of array elements per vector instruction) information, which is within vector instructions sent from the hosts.

All vector instructions from the hosts go through the vector controller (VC) that handles hazard detection and virtualization, and then broadcasts them to the ALU or LDST pipeline interface in each lane. To ensure the correct execution of a vector application under various numbers of active lanes, both address domains of the VM as well as the VL information must be virtualized. This is essential for runtime VP lane configuration, since all address values and the VL for a vector application are determined statically, and therefore the same values must be properly interpreted by the hardware under disparate VP configurations. To facilitate address virtualization, the host-to-VM mux and the VC are designed to be configurable by host requests. Before starting a new vector thread, a host will submit a request to configure state registers based on the optimal number of lanes needed by the thread.

B. *Configurable Components*

Fig. 1 illustrates how a data array with base host-to-VM address of 4N and VL=8 can be accessed by the virtualized VP correctly under different lane configurations. The figure shows the cases of two-lane (Fig. 1a) and four-lane (Fig. 1b) configurations in a VP with four lanes; however, this scheme can be easily adapted for any 2^N active lanes with any VL = 2^M, where N and M are both natural numbers and M ≥ N. The VP's lane state register, which can be configured dynamically by the hosts via a simple control instruction, stores the number of active lanes and determines how the VP behaves in the following cases.

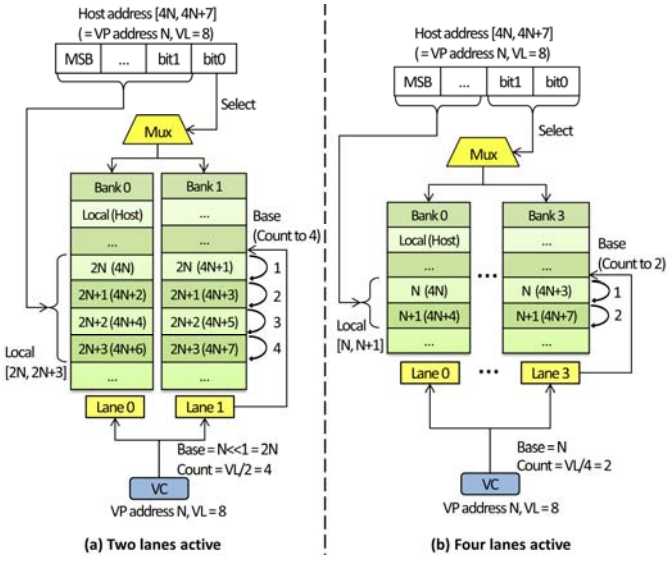

Fig. 1. Mapping of VL, host-to-VM address and VP-to-VM address via virtualization

In the case of all lanes being active (four in this example), the lowest two bits of the host-to-VM address will be used as the select signal for the host-to-VM mux, and the remaining bits will be used as the actual physical address for every VM bank. As shown in Fig. 1b, the data array is mapped to the physical address [N, N+1] of each VM bank, with two elements per bank. Therefore, the array's base VP-to-VM address is compiled to be N, which is the same as its physical address in each bank. The LDST unit within each lane will start accessing the array with base address N, and based on the VL = 8 and four active lanes information passed from the VC, the instruction decoder will set the counter to two so that each lane will access two elements per instruction.

When the host dynamically deactivates two VP lanes and their attached VM banks, only two banks remain and therefore the mux must be configured to take only the LSB of the host-to-VM address as the bank select signal. All remaining bits will be used as each bank's physical address, and since the host-to-VM address is compiled at static time and does not change, under the new configuration the array will be mapped to the physical address [2N, 2N+3] of each remaining VM bank, with four elements per bank. To ensure that the VP can still reach the array with the unchangeable VP-to-VM address of N, the VC's virtualization stage simply has to shift left the address by one bit and pass it to all lanes' LDST units. The new configuration also requires that the VC shift left the VL by one bit to make each lane access four elements per instruction. Since the decoder unit in each lane relies on the register name and VL value to locate the right vector registers, shifting VL also ensures that each lane will use the right location and number of registers under the new configuration.

III. SMT VP AND THREAD FUSION

A. *VRF and VM Virtualization under SMT*

The VP in our prototype [7] was originally designed to support vector-based SMT and sharing among many processors. To achieve true SMT where instructions from multiple threads can coexist inside the VP pipeline without interference, both the VRF name space and the VM space are virtualized on a per instruction basis. With SMT virtualization, one SMT capable VP appears as multiple logical VPs (LVPs) to multiple hosts/cores. Shown in Fig. 2 is a simple example of an SMT VP of degree two. The VP has only one physical instruction input channel; however, the FIFOs and arbitrator structure create two virtual channels. The VP input arbitrator accepts instructions from two different FIFOs in round-robin fashion, and each FIFO can be assigned to a host; in this example, only one host is used and the two LVPs are used to exploit TLP via thread fusion. Each instruction has its LSB as the thread ID that is filled by the arbitrator based on the source FIFO. For ID = 0, all VRF names are unchanged. When ID = 1, the virtualization stage in the VC properly flips a few bits in each register name based on the instruction's VL. The scheme ensures that LVP0 occupies the lower half of the VRF and LVP1 occupies the higher half. The mechanism achieves VRF resource sharing with significant flexibility in that it allows both LVPs to function correctly as long as (a) the total VRF

usage does not exceed the available physical VRF resources, and (b) in the single LVP mode, either LVP0 or LVP1 can occupy the entire VRF space.

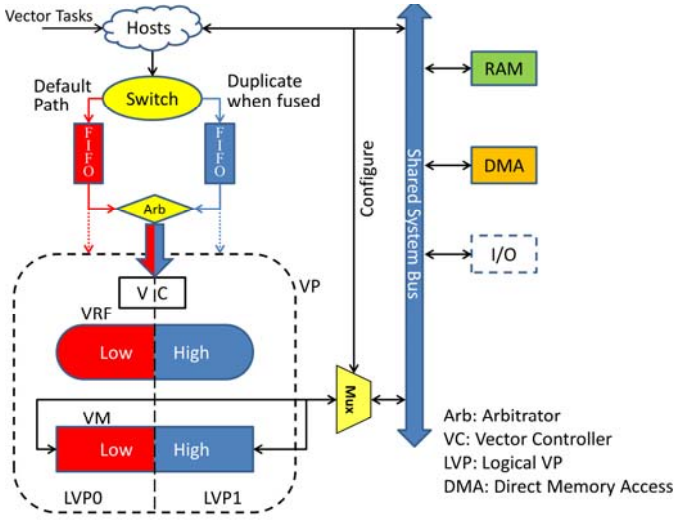

Fig. 2. System architecture of a fusion capable VP of degree two.

As shown in Fig. 2, the host-to-VM mux supports data transfers between the hosts' RAM space and both LVPs' virtual VM spaces. Based on the thread state register, which can be configured by the hosts, part of the host-to-VM address is flipped to map the LVP1's virtual address space to the higher half of the VM banks. The data transfer only happens at the beginning and the end of a vector application, and therefore no per instruction switching between LVP0 and LVP1 is required for data transfers. The thread state register can be configured by the hosts using a simple control instruction which is similar to that used for dynamic lane configuration. The virtualization for SMT capability does not conflict with that for dynamic lane configuration, and therefore the prototype is extremely versatile; without recompilation, any two applications can simultaneously function properly on the VP regardless of their assigned thread ID or the number of active VP lanes.

For simplicity, we built an FPGA-based VP prototype capable of executing two threads simultaneously. However, the max number of simultaneous threads can be easily increased by increasing the number of instruction FIFOs and modifying the arbitrator's state machine. VRF virtualization for more than two threads can be supported by using a VRF renaming algorithm [7], which dynamically maps the threads' virtual VRF names to physical names while minimizing register fragmentation. Virtual VM for multiple threads can be implemented by using a memory management unit.

B. *Fusion of Similar Threads*

For frequently used computation intensive operations, highly optimized VP routines are implemented and stored in a library. When multiple pending tasks are of the same operation, it is possible to fuse these operations thanks to the VP's per thread virtual VM and VRF space. Fig. 3 shows how two Discrete Cosine Transform (DCT) operations are accelerated by fusing the threads. Without fusion (Fig. 3a), the two operations will be executed sequentially. When two threads are fused (Fig. 3b), the major parts of their execution are merged, so that the hosts' domain issues vector instructions only once while the VP receives two copies. The switch in Fig. 2 is set to the fusion state for duplicating each vector instruction from the host domain and sending it to both FIFOs. A scheduler of vector threads decides on fusion (the details are omitted for brevity). Due to the independent virtual nature of each LVP, the two identical instruction flows will perform the same operation but on different input data within each virtual space.

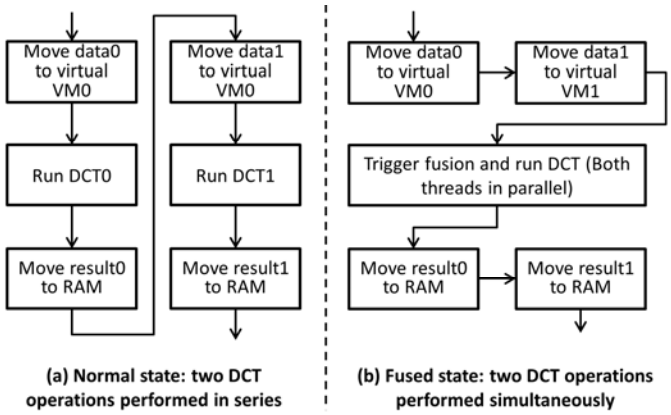

Fig. 3. Fusion of two DCT operations.

Vector thread fusion has many benefits: (a) it significantly increases the vector instruction issue rate for all hosts; (b) the VP utilization is effectively multiplied by the degree of fusion as long as the aggregate utilization does not exceed 100%; (c) it reduces the overall energy consumption since the host domain only has to run the flow control program once to send out vector instructions for fused threads; (d) since the VP's SMT virtualization is compatible with dynamic lane configuration, fusion can be combined with lane configuration to optimize performance and energy figures.

IV. System Architecture and FPGA Implementation

To evaluate the two proposed techniques, we prototyped on a Xilinx XC7Z045-1fbg676 FPGA a dual-threaded VP interfaced with a hosts system. The system architecture is similar to that in Fig. 2, with the hosts system replaced by a MicroBlaze (MB) processor that issues vector threads. MB is a RISC processor with a low latency local memory; we stored various vector kernels in its 16KB local memory. The system RAM and VM are 64KB each. A DMA engine is attached to the system bus for fast data transfers between the system RAM and VM. The mux connecting the VM and system bus is configurable by the MB to support the virtualization for lane configurability and SMT, as per Sections II and III. I/O components on the bus are used for debugging purposes and we implemented an 8-bit LED to show the system status. A cycle accurate timer (not shown in the figure) that measures application runtime can interrupt the MB.

The VP has four lanes and is capable of running with 1, 2, or 4 active lanes. Each lane's dedicated VM bank can be deactivated with its assigned lane. The VRF can store 1024 32-bit elements, with all the registers evenly distributed across the four vector lanes. Without loss of generality, the VP supports three VLs (16, 32, and 64) in the prototype. A vector register of length N contains N register elements, and therefore the number of available vector registers depends on the VL of each register. The VP, the fusion switch, the VM data mux and the vector instruction arbitrator are custom hardware designed in VHDL, and the rest of the system components are Xilinx IPs. The target FPGA has a speed grade of -1. The minimum achievable critical delay is 6.01ns; for simplicity, we implemented a 100MHz system. The resource consumption breakdown for the VP is shown in Table I. The rest of the custom hardware components are not shown as they consume negligible amount of resources compared to the VP (< 1%). RAMB36E1 is a Xilinx block RAM IP that can be configured to various data widths and depths as long as the total memory capacity is within 32Kbits.

TABLE I. RESOURCE CONSUMPTION AND UTILIZATION PERCENTAGE

Entity	Registers U(%)	LUTs U(%)	RAMB36E1s U(%)	DSP48E1s U(%)
One Lane	9571 (2%)	17437 (7%)	0	5 (<1%)
VM	16 (<1%)	272 (<1%)	16 (2%)	0
VC	287 (<1%)	451 (<1%)	0	0
VP	38674 (8%)	70143 (32%)	16 (2%)	20 (2%)

V. BENCHMARKING

We picked four vector applications, namely DCT, Finite Input Response filter (FIR), RGB to YIQ color space conversion (RGB), and Vector Dot Product (VDP). Since our current VP supports three different VLs, we evaluate each picked application using all supported VLs, creating a total of 12 benchmarks. The VP has separate pipelines for ALU and LDST operations; each benchmark is characterized by its ALU utilization (U_{ALU}) and LDST utilization (U_{LDST}). We executed each benchmark under various configurations, measured the corresponding runtime, and calculated the utilizations of the pipelines. The utilization is defined as O_{total}/O_{4lanes}, where O_{total} is the total number of operations for an application and O_{4lanes} is the maximum number of operations that can be performed by the four lanes during the application's runtime. Tables II-IV show the runtime and utilization figures under three VP configurations (**a**. Four lanes active without fusion. **b**. Four lanes active with fusion. **c**. Two lanes active without fusion.)

TABLE II. PERFORMANCE PROFILE DATA FOR 4LANE UNFUSED VP

APP	VL	T(µs)	ALU(%)	LDST(%)
DCT	***16***	75	6.8	14.1
	32	75	13.6	28.2
	64	75	27.3	56.4
VDP	***16***	23.7	6.7	11.8
	32	28.3	14.1	25.4
	64	34.4	32.5	51.1
RGB	***16***	243.6	15.8	6.3
	32	123.8	31.0	12.4
	64	64.0	60.0	24.0
FIR	***16***	25.7	10.6	5.5
	32	46.8	22.7	11.5
	64	89.1	47.8	24.0

We call *native utilization (U)* and *native runtime (T)* an application's figures under configuration **a**. Utilization and runtime figures under other configurations are represented by *U'* and *T'*. With two active lanes, the maximum achievable utilization is 50%; it is the average with two active lanes at 100% and the other two lanes at 0%. For benchmarks with ALU and LDST native utilizations below 50%, the runtime and utilizations are not affected due to lane deactivation. For other benchmarks, from U_{ALU} and U_{LDST} the higher will hit the 50% saturation level while the other will decrease proportionally. The runtime increase is related to the higher of U_{ALU} and U_{LDST}. The relation between each benchmark's actual figures for two active lanes and their native figures is shown in Eq. (1). The proof is omitted for brevity.

TABLE III. PERFORMANCE PROFILE DATA FOR 4LANE FUSED VP

APP	VL	T(µs)	ALU(%)	LDST(%)
DCT	***16***	75	13.6	28.2
	32	75	27.2	56.4
	64	86.5	47.36	97.6
VDP	***16***	23.7	13.4	23.6
	32	28.3	28.2	50.8
	64	35.8	62.6	98.4
RGB	***16***	243.7	31.6	12.6
	32	123.5	62.1	24.9
	64	78.3	98.1	39.2
FIR	***16***	25.9	21.1	10.9
	32	46.7	45.5	22.9
	64	89.2	95.7	47.9

TABLE IV. PERFORMANCE PROFILE DATA FOR 2LANE UNFUSED VP

APP	VL	T(µs)	ALU(%)	LDST(%)
DCT	***16***	75	6.8	14.1
	32	75	13.6	28.2
	64	84.9	24.1	49.8
VDP	***16***	23.7	6.7	11.8
	32	28.3	14.1	25.4
	64	35.7	31.3	49.2
RGB	***16***	243.6	15.8	6.3
	32	123.8	31.0	12.4
	64	77.7	49.4	19.8
FIR	***16***	25.7	10.6	5.5
	32	46.8	22.7	11.5
	64	89.03	47.8	24.0

if (U_{ALU}<50 and U_{LDST} < 50) then

$$U'_{ALU\_2lanes} = U_{ALU};\ U'_{LDST\_2lanes} = U_{LDST};\ T'_{2lanes} = T$$

else if ($U_{ALU} > U_{LDST}$) then (1)

$$U'_{ALU\_2lanes} = 50;\ U'_{LDST\_2lanes} = \frac{U_{LDST}}{U_{ALU}}50;\ T'_{2lanes} = \frac{U_{ALU}}{50}T$$

else

$$U'_{ALU\_2lanes} = \frac{U_{ALU}}{U_{LDST}}50;\ U'_{LDST\_2lanes} = 50;\ T'_{2lanes} = \frac{U_{LDST}}{50}T$$

The maximum utilization achievable is 50% when two lanes are deactivated. The equation agrees with the measurements shown in Tables II-IV. U'_{ALU_1lane}, U'_{LDST_1lane} and T'_{1lane} with one active lane can be derived using a similar approach and changing the threshold to 25%. A fused

benchmark can be considered as a new one with new native runtime and utilizations, as shown in Table III. The runtime scheduler will use the utilization information to choose the optimal number of active lanes based on the scheduling policy. We discuss the scheduling policy in Section VI.B.

VI. Power Model and Scheduling Policy

A. The Power Model

A highly accurate VP power consumption model is needed for optimization purposes. We performed simulation on the fully placed and routed VP design. By combining the VP's Native Circuit Description (NCD) with the testbenches of different scenarios, we obtained the detailed Switching Activity Information File (SAIF) for various VP utilizations. The NCD and SAIF file is fed to the Xilinx Power Analyzer (XPA) tool to calculate exact dynamic and static power. By using testbenches that issue instructions to the VP at various rates, we measured the VP's static and dynamic power under various utilizations. Fig. 4 shows dynamic power results.

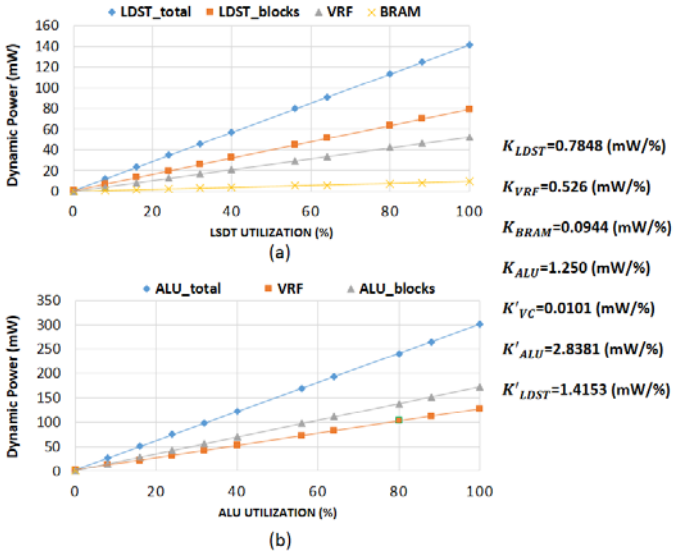

Fig. 4. Dynamic power vs. utilization for both ALU and LDST data paths.

All VP lanes' dynamic power can be broken down into four components corresponding to the: VRF, VM banks, LDST data path (including LDST FIFO and decoder, address generator, and write back unit) and ALU data path (including ALU FIFO and decoder, execution and write back units). Each component's dynamic power is linear to its utilization, and is therefore related to U_{ALU} and U_{LDST}. Each LDST operation involves one memory access and one VRF access, and each ALU operation involves reading two operands from the VRF and writing one result back to the VRF. Therefore, the relation between VP lanes' dynamic power and their utilizations can be described by Eq. (2). Each coefficient K is the power per utilization in mW/% for each corresponding component. On the other hand, VC is a common block that processes both ALU and LDST instructions, and therefore its power consumption is linear to the total issue rate (IR) of both types of instruction, and that can be described by Eq. (3).

$$P_{ALU\_dynamic} = K_{ALU}U_{ALU} + K_{VRF}(3*U_{ALU})$$
$$P_{LDST\_dynamic} = K_{LDST}U_{LDST} + K_{VRF}U_{LDST} + K_{BRAM}U_{LDST} \quad (2)$$

$$P_{VC} = K_{VC} * IR = K_{VC} * (\frac{(U_{ALU} + U_{LDST}) * 4}{VL}) = K'_{VC}(U_{ALU} + U_{LDST}) \quad (3)$$

By adding together the terms in (2) and (3), we obtain in Eq. (4) the VP's dynamic power as a simple linear function of U_{ALU} and U_{LDST}.

$$\begin{aligned} P_{VP\_dynamic} &= K_{LDST}U_{LDST} + K_{ALU}U_{ALU} + K_{VRF}(3*U_{ALU} + U_{LDST}) \\ &\quad + K_{BRAM}U_{LDST} + K'_{VC}(U_{ALU} + U_{LDST}) \\ &= K'_{ALU}U_{ALU} + K'_{LDST}U_{LDST} \end{aligned} \quad (4)$$

This power model matches our measurements of the VP's dynamic power vs. U_{LDST} with idle ALU (Fig. 4a), and power vs. U_{ALU} with idle LDST (Fig. 4b). From the measured data we extract the coefficient K for each component; the most important are for the ALU and LDST units: K'_{ALU} = 2.838mw/%, and K'_{LDST} = 1.415mW/%.

The VP's total power is given by Eq. (5). The measured VC static power is 2.2mW, and each lane's static power is 26.5mW with its dedicated memory bank. Since the FPGA does not support power gating, it is implemented using extra logic to isolate the power signal. Power gated components still dissipate about 15% of their original static power [8]. P_{static} is 108.2mW, 63.15mW and 40.63mW for the 4, 2, and 1 lane configuration, respectively. In Eq. (5), U_{ALU} and U_{LDST} are the applications' actual utilizations under various situations. Combining Eq. (5) with Eq. (1), we obtain Eq. (6) that describes the relation of an application's power consumption with two active lanes and its native utilizations. A similar equation can be derived with one active lane.

$$P_{total} = P_{static} + K'_{ALU}U_{ALU} + K'_{LDST}U_{LDST} \quad (5)$$

$$\begin{aligned} &\text{if } (U_{ALU} < 50 \text{ and } U_{LDST} < 50) \text{ then} \\ &\quad P_{total\_2lanes} = 63.15 + K'_{ALU}U_{ALU} + K'_{LDST}U_{LDST} \\ &\text{elseif } (U_{ALU} > U_{LDST}) \text{ then} \\ &\quad P_{total\_2lanes} = 63.15 + 50K'_{ALU} + 50\frac{U_{LDST}}{U_{ALU}}K'_{LDST} \\ &\text{else} \\ &\quad P_{total\_2lanes} = 63.15 + 50K'_{LDST} + 50\frac{U_{ALU}}{U_{LDST}}K'_{ALU} \end{aligned} \quad (6)$$

B. The Scheduling Policy

We obtain a vector application's P_{4lanes}, P_{2lanes} and P_{1lane} as a function of its native utilizations. The execution times T_{2lanes} and T_{1lane} are also related to T_{4lanes}, and the example for T_{2lanes} is shown in Eq. (1). The set of P and T values form two-dimensional matrices with U_{ALU} and U_{LDST} as indexes. We propose two different scheduling policies using P and T. The first policy is to achieve minimum energy consumption. The energy matrix for each configuration can be calculated by $E_{Nlanes} = P_{Nlanes} * T_{Nlanes}$. By comparing E_{4lanes}, E_{2lanes} and E_{1lane}, we can determine the utilization boundary for optimal

configuration. Fig. 5a shows a generic contour for minimum energy consumption; the actual values depend on the application. All applications whose native utilizations fall into region A consume minimum energy when executed with one active lane, while region B is for two lanes and region C is for four lanes. Using a similar approach, we can obtain the boundary for the second scheduling policy which minimizes the product of an application's execution time and energy consumption; it is shown in Fig. 5b.

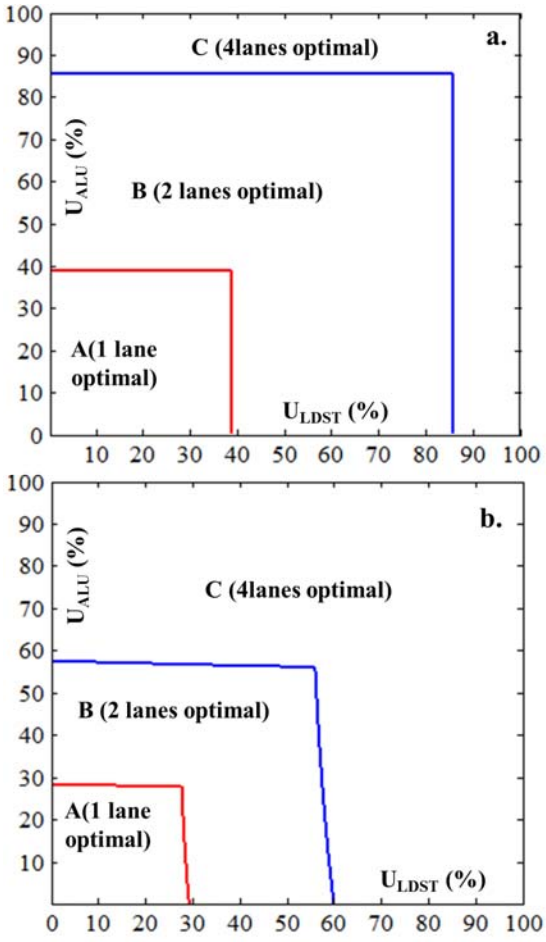

Fig. 5. Optimal utilization boundries for **a.** minimum energy **b.** minimum energy-execution time profuct.

We tested our two scheduling policies (E_{min} and ET_{min}) using an open system model where tasks that arrive within a time slice of size 10ms are scheduled in the following 10ms slice. The arrival of every task follows the Poisson distribution; six arrival rates λ = 1, 3, 5, 7, 9, 11 are tested. Tasks in the queue are ordered by their task type; since similar tasks are adjacent in the queue, the scheduler easily identifies fusable tasks. The tasks are those in Section V. For each optimization policy, every task has two optimal execution configurations: unfused and fused modes. All configurations can be obtained by combining each task's U_{ALU} and U_{LDST} with the results shown in Fig. 5. As mentioned previously, the scheduler will treat a fused task as a new task with its own U_{ALU} and U_{LDST}. We generated the task queues for 1000 time slices using the MATLAB random number generator, and calculated the average parameters for the two scheduling policies and also for the VP without the proposed techniques. As shown in Fig. 6, for the E_{min} policy, our proposed techniques reduce the average energy consumption by up to 33.8% while improving the runtime by 40%. The ET_{min} policy reduces the product of energy and runtime by up to 62.7%. For the VP without fusion and lane configuration, the average execution time at λ = 11 is close to 10ms and the system is about to overflow.

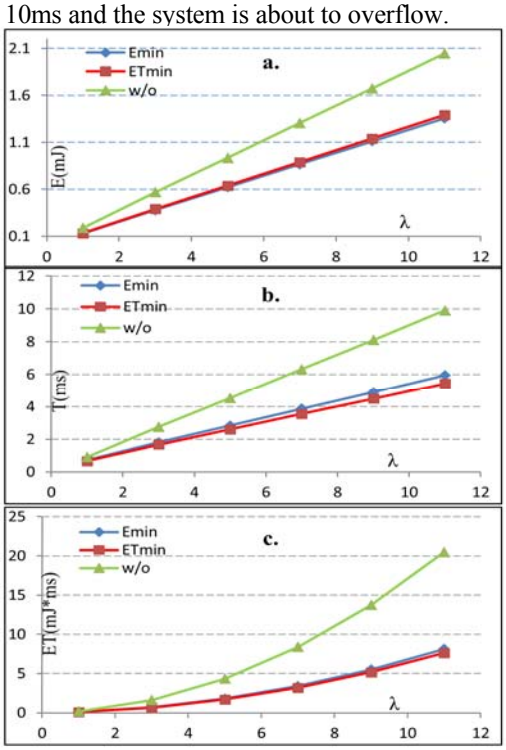

Fig. 6. Comparison of the E_{min}, ET_{min} policies against a VP w/o fusion and lane configuration over the average of 1000 time slices. **a**. energy **b.** runtime **c.** energy-runtime product.

VII. Conclusions

By combining the proposed dynamic lane configuration and fusion techniques in the design of a shared SMT VP, the VP's energy consumption and energy runtime product are improved very substantially under two proposed optimization policies. As VPs scale up in number of vector lanes, fine-grain power management provided by lane configuration becomes more critical. The benefit of our fusion technique will also be amplified when the fusion degree grows above two.

References

[1] C. E. Kozyrakis, and D. A. Patterson, "Scalable vector processors for embedded systems," *Micro, IEEE*, vol. *23*(6), 2003, pp. 36-45.

[2] SA. Rooholamin, and S. G. Ziavras, "Modular vector processor architecture targeting at data-level parallelism, " Micropr. Microsyst., vol. 39(4), July 2015, pp.237-249.

[3] S.F. Beldianu and S.G. Ziavras, "Multicore-based vector coprocessor sharing for performance and energy gains," ACM Trans. Embedded Computing Syst., vol. 13(2), Sep 2013, pp.17:1-17:25.

[4] S. F. Beldianu, and S. G. Ziavras, "Performance-energy optimizations for shared vector accelerators in multicores," IEEE Trans. Computers, vol. 64(3), March 2015, pp.805-817.

[5] R. Rakvic, J. González, Q. Cai, P. Chaparro, G. Magklis, and A. Gonzalez, "Energy efficiency via thread fusion and value reuse.," IET Computers Digital Techn., vol. 4(2), 2010, pp.114-125.

[6] Y. Lu, and S. G. Ziavras, "Instruction fusion for multiscalar and many-core processors," Int. Journal Parallel Programming, 2015, DOI 10.1007/s10766-015-0386-1, pp.1-12.

[7] Y. Lu, S. Rooholamin and S.G. Ziavras, "Vector coprocessor virtualization for simultaneous multithreading," ACM Transactions on Embedded Computing Systems, accepted for publication.

[8] S. Roy, N. Ranganathan, and S. Katkoori, "A framework for power-gating functional units in embedded microprocessors," IEEE Transactions on VLSI Systems, vol. 17(11), Nov. 2009, pp.1640-1649.